Sixteenth Brainstorming Week on Membrane Computing

Sevilla, January 30 – February 2, 2018

David Orellana-Martín
Gheorghe Păun
Agustín Riscos-Núñez
Luis Valencia-Cabrera
Editors

Sixteenth Brainstorming Week on Membrane Computing

Sevilla, January 30 – February 2, 2018

David Orellana-Martín
Gheorghe Păun
Agustín Riscos-Núñez
Luis Valencia-Cabrera
Editors

RGNC REPORT 1/2018

Research Group on Natural Computing
Sevilla University

Sevilla, 2018

Preface

The Sixteenth Brainstorming Week on Membrane Computing (BWMC) was held in Sevilla, from January 30 to February 2, 2018, in the organization of the Research Group on Natural Computing (RGNC) from the Department of Computer Science and Artificial Intelligence of Sevilla University. The first edition of BWMC was organized at the beginning of February 2003 in Rovira i Virgili University, Tarragona, and all the next editions took place in Sevilla at the beginning of February, each year.

In the style of previous meetings in this series, the sixteenth BWMC was conceived as a period of active interaction among the participants, with the emphasis on exchanging ideas and cooperation. Several "provocative" talks were delivered, mainly devoted to open problems, research topics, conjectures waiting for proofs, followed by an intense cooperation among the about 30 participants – see the list in the end of this preface. The efficiency of this type of meetings was again proved to be very high and the present volume is only part of the proof of this assertion.

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

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

Other papers elaborated during the 2018 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. José A. Andreu Guzmán, Universidad de Sevilla, Spain
   andreuguzman36@gmail.com
2. Péter Battyányi, University of Debrecen, Hungary
   battyanyi.peter@zimbra.inf.unideb.hu
3. Lucie Ciencialová, Silesian University in Opava, Czech Republic
   lucie.ciencialova@fpf.slu.cz
4. Erzsébet Csuhaj-Varjú, Eötvös Loránd University, Hungary
   csuhaj@inf.elte.hu
5. Andrés Doncel Ramírez, Universidad de Sevilla, Spain
   andrestp95@gmail.com
6. Rudolf Freund, Technological University of Vienna, Austria
   rudi@emcc.at
7. Sandra M. Gómez Canaval, Universidad Politécnica de Madrid, Spain
   sgomez@etsisi.upm.es
8. Carmen Graciani, Universidad de Sevilla, Spain
   cgdiaz@us.es
9. Florentin Ipate, University of Bucharest, Romania
   florentin.ipate@ifsoft.ro
10. Sergiu Ivanov, Université Paris Est Créteil, France
    sivanov@colimite.fr
11. Jiang Jung, Chongqing University of Technology and Business, China
    jiangyun@email.ctbu.edu.cn
12. Kristóf Kántor, University of Debrecen, Hungary
    rapid90x@gmail.com
13. Raluca Lefticaru, University of Bradford, United Kingdom
    R.Lefticaru@bradford.ac.uk
14. Alberto Leporati, University of MilanBicocca, Italy
    leporati@disco.unimib.it
15. Luca Manzoni, University of MilanBicocca, Italy
    luca.manzoni@disco.unimib.it
16. Miguel A. Martínez-Del-Amor, Universidad de Sevilla, Spain
    mdelamor@us.es
17. David Orellana, Universidad de Sevilla, Spain
    dorellana@us.es
18. Georghe Păun, Universidad de Sevilla, Spain, and the Romanian Academy, Bucharest
    gpaun@us.es, curteadelaarges@gmail.com
19. Ignacio Pérez-Hurtado, Universidad de Sevilla, Spain
    perezh@us.es
20. Mario de J. Pérez-Jiménez, Universidad de Sevilla, Spain
    marper@us.es

21. Antonio E. Porreca, University of MilanBicocca, Italy
porreca@disco.unimib.it
22. Agustín Riscos-Núñez, Universidad de Sevilla, Spain
ariscosn@us.es
23. Daniel Rodríguez Chavarría, Universidad Sevilla, Spain
danrodcha@gmail.com
24. Álvaro Romero-Jiménez, Universidad de Sevilla, Spain
romero.alvaro@us.es
25. Zeyi Shang, University Southwest Jiaotong, China
zeyi.shang@lacl.fr
26. Ana Ţurlea, University of Bucharest, Romania
t_anacris@yahoo.com
27. Luis Valencia-Cabrera, Universidad de Sevilla, Spain
lvalencia@us.es
28. 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 Sevilla University (`http://www.gcn.us.es`)– and all the members of this group were enthusiastically involved in this (not always easy) work.

The meeting was supported from various sources: (i) Proyecto de Excelencia de la Junta de Andalucía, grant TIC 581, (ii) Proyecto de Excelencia con investigador de reconocida valía, de la Junta de Andalucía, grant P08 – TIC 04200, (iii) Proyecto del Ministerio de Educación y Ciencia, grant TIN2006 – 13425, (iv) Instituto de Matemáticas de la Universidad de Sevilla (IMUS), (v) Consejería de Innovación, Ciencia y Empresa de la Junta de Andalucía, as well as by the Department of Computer Science and Artificial Intelligence from Sevilla University.

The Editors
(May 2018)

Contents

Introducing the Concept of Activation and Blocking of Rules in the General Framework for Regulated Rewriting in Sequential Grammars*

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

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

[2] Faculty of Informatics, TU Wien
Favoritenstraße 9–11, 1040 Vienna, Austria
rudi@emcc.at

[3] IBISC, Université Évry, Université Paris-Saclay
23 Boulevard de France, 91025, Évry, France
sergiu.ivanov@univ-evry.fr

Summary. We introduce new possibilities to control the application of rules based on the preceding application of rules which can be defined for a general model of sequential grammars and we show some similarities to other control mechanisms as graph-controlled grammars and matrix grammars with and without applicability checking as well as grammars with random context conditions and ordered grammars. Using both activation and blocking of rules, in the string and in the multiset case we can show computational completeness of context-free grammars equipped with the control mechanism of activation and blocking of rules even when using only two nonterminal symbols.

1 Introduction

Nearly thirty years ago, the monograph on regulated rewriting by Jürgen Dassow and Gheorghe Păun [2] already gave a first comprehensive overview on many concepts of regulated rewriting, especially for the string case. Yet as it turned out later, many of the mechanisms considered there for guiding the application of productions/rules can also be applied to other objects than strings, e.g., to n-dimensional arrays [4]. Even in the emerging field of P systems [10, 14] where mostly multisets are considered, such regulating mechanisms were used [1]. As exhibited in [6], for comparing the generating power of grammars working in the sequential derivation

* The work is supported by National Natural Science Foundation of China (61320106005, 61033003, and 61772214) and the Innovation Scientists and Technicians Troop Construction Projects of Henan Province (154200510012).

mode, many relations between various regulating mechanisms can be established in a very general setting without any reference to the underlying objects the rules are working on, using a general model for graph-controlled, programmed, random-context, and ordered grammars of arbitrary type based on the applicability of rules.

In the second section, we recall some notions from formal language theory as well as the main definitions of the general framework for sequential grammars elaborated in [6]. Then we define the new concept of activation and blocking of rules based on the applicability of rules within this general framework for regulated rewriting. In Section 3 some general results for sequential grammars using the control mechanism of activation or activation and blocking of rules are established. Specific results on computational completeness for strings, multisets, and arrays as underlying objects then are shown in Section 4. In Section 5 we establish our main results for strings and multisets showing that context-free (string and multiset) grammars with activation and blocking of rules are computationally complete even when only two non-terminal symbols are used, which establishes a sharp border as one non-terminal symbol is not sufficient. Finally, a summary of the results obtained in this paper and some future research topics extending the notions and results obtained in this paper are given in Section 6.

2 Definitions

After some preliminaries from formal language theory, we define our general model for grammars and recall some notions for string, array, and multiset grammars and languages in the general setting of this paper. Then we formulate the models of graph-controlled, programmed, matrix grammars with and without appearance checking, as well as random-context grammars, based on the applicability of rules.

2.1 Preliminaries

The set of integers is denoted by $\mathbb{Z}$, the set of non-negative integers by $\mathbb{N}_0$, and the set of positive integers (natural numbers) by $\mathbb{N}$. An *alphabet* V is a finite non-empty set of abstract *symbols*. Given V, the free monoid generated by V under the operation of concatenation is denoted by V^*; the elements of V^* are called strings, and the *empty string* is denoted by λ; $V^* \setminus \{\lambda\}$ is denoted by V^+. Let $\{a_1, ..., a_n\}$ be an arbitrary alphabet; the number of occurrences of a symbol a_i in x is denoted by $|x|_{a_i}$; the *Parikh vector* associated with x with respect to $a_1, ..., a_n$ is $\left(|x|_{a_1}, ..., |x|_{a_n}\right)$. The *Parikh image* of a language L over $\{a_1, ..., a_n\}$ is the set of all Parikh vectors of strings in L, and we denote it by $Ps\,(L)$. For a family of languages FL, the family of Parikh images of languages in FL is denoted by $PsFL$.

A (finite) multiset over the (finite) alphabet V, $V = \{a_1, ..., a_n\}$, is a mapping $f : V \longrightarrow \mathbb{N}_0$ and represented by $\langle f\,(a_1)\,, a_1\rangle \ldots \langle f\,(a_n)\,, a_n\rangle$ or by any string x the

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

For more details of formal language theory the reader is referred to the monographs and handbooks in this area [2, 12].

2.2 A General Model for Sequential Grammars

We first recall the main definitions of the general model for sequential grammars as established in [6], grammars generating a set of terminal objects by derivations where in each derivation step exactly one rule is applied to exactly one object. This does not cover rules involving more than one object – as, for example, splicing rules – or other derivation modes – as, for example, the maximally parallel mode considered in many variants of P systems [10].

A *(sequential) grammar* G is a construct $(O, O_T, w, P, \Longrightarrow_G)$ where

- O is a set of *objects*;
- $O_T \subseteq O$ is a set of *terminal objects*;
- $w \in O$ is the *axiom (start object)*;
- P is a finite set of *rules*;
- $\Longrightarrow_G \subseteq O \times O$ is the *derivation relation* of G.
  We assume that each of the rules $p \in P$ induces a relation $\Longrightarrow_p \subseteq O \times O$ with respect to $\Longrightarrow_G$ fulfilling at least the following conditions: (i) for each object $x \in O$, $(x, y) \in \Longrightarrow_p$ for only finitely many objects $y \in O$; (ii) there exists a finitely described mechanism as, for example, a Turing machine, which, given an object $x \in O$, computes all objects $y \in O$ such that $(x, y) \in \Longrightarrow_p$. A rule $p \in P$ is called *applicable* to an object $x \in O$ if and only if there exists at least one object $y \in O$ such that $(x, y) \in \Longrightarrow_p$; we also write $x \Longrightarrow_p y$. The derivation relation $\Longrightarrow_G$ is the union of all $\Longrightarrow_p$, i.e., $\Longrightarrow_G = \cup_{p\in P} \Longrightarrow_p$. The reflexive and transitive closure of $\Longrightarrow_G$ is denoted by $\overset{*}{\Longrightarrow}_G$.

In the following we shall consider different types of grammars depending on the components of G (where the set of objects O is infinite, e.g., V^*, the set of strings over the alphabet V), especially with respect to different types of rules (e.g., context-free string rules). Some specific conditions on the elements of G, especially on the rules in P, may define a special type X of grammars which then will be called *grammars of type X*.

The *language generated by G* is the set of all terminal objects (we also assume $v \in O_T$ to be decidable for every $v \in O$) derivable from the axiom, i.e.,

$$L(G) = \left\{ v \in O_T \mid w \overset{*}{\Longrightarrow}_G v \right\}.$$

The family of languages generated by grammars of type X is denoted by $\mathcal{L}(X)$.

Let $G = (O, O_T, w, P, \Longrightarrow_G)$ be a grammar of type X. If for every G of type X we have $O_T = O$, then X is called a *pure* type, otherwise it is called *extended*; X is called *strictly extended* if for any grammar G of type X, $w \notin O_T$ and for all $x \in O_T$, no rule from P can be applied to x.

In many cases, the type X of the grammar allows for (one or even both of) the following features:

A type X of grammars is called a *type with unit rules* if for every grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ of type X a grammar $G' = \left(O, O_T, w, P \cup P^{(+)}, \Longrightarrow_{G'}\right)$ of type X exists such that $\Longrightarrow_G \ \subseteq \ \Longrightarrow_{G'}$ and

- $P^{(+)} = \left\{p^{(+)} \mid p \in P\right\}$,
- for all $x \in O$, $p^{(+)}$ is applicable to x if and only if p is applicable to x, and
- for all $x \in O$, if $p^{(+)}$ is applicable to x, the application of $p^{(+)}$ to x yields x back again.

A type X of grammars is called a *type with trap rules* if for every grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ of type X a grammar $G' = \left(O, O_T, w, P \cup P^{(-)}, \Longrightarrow_{G'}\right)$ of type X exists such that $\Longrightarrow_G \ \subseteq \ \Longrightarrow_{G'}$ and

- $P^{(-)} = \left\{p^{(-)} \mid p \in P\right\}$,
- for all $x \in O$, $p^{(-)}$ is applicable to x if and only if p is applicable to x, and
- for all $x \in O$, if $p^{(-)}$ is applicable to x, the application of $p^{(-)}$ to x yields an object y from which no terminal object can be derived anymore.

2.3 Specific Types of Objects

String grammars

In the general notion as defined above, a *string grammar* G_S is represented as

$$\left((N \cup T)^*, T^*, w, P, \Longrightarrow_P\right)$$

where N is the alphabet of *non-terminal symbols*, T is the alphabet of *terminal symbols*, $N \cap T = \emptyset$, $w \in (N \cup T)^+$, P is a finite set of *rules* of the form $u \to v$ with $u \in V^*$ (for generating grammars, $u \in V^+$) and $v \in V^*$ (for accepting grammars, $v \in V^+$), with $V := N \cup T$; the derivation relation for $u \to v \in P$ is defined by $xuy \Longrightarrow_{u \to v} xvy$ for all $x, y \in V^*$, thus yielding the well-known derivation relation $\Longrightarrow_{G_S}$ for the string grammar G_S. In the following, we shall also use the common notation $G_S = (N, T, w, P)$ instead, too. We remark that, usually, the axiom w is supposed to be a non-terminal symbol, i.e., $w \in V \setminus T$, and is called the *start symbol*.

As special types of string grammars we consider string grammars with arbitrary rules and context-free rules of the form $A \to v$ with $A \in N$ and $v \in V^*$. The corresponding types of grammars are denoted by ARB an CF, thus yielding the families of languages $\mathcal{L}(ARB)$, i.e., the family of recursively enumerable languages (also denoted by RE), as well as $\mathcal{L}(CF)$, i.e., the familiy of context-free languages, respectively.

Observe that the types ARB and CF are types with unit rules and trap rules (for $p = w \rightarrow v \in P$, we can take $p^{(+)} = w \rightarrow w$ and $p^{(-)} = w \rightarrow F$ where $F \notin T$ is a new symbol – the trap symbol).

We refer to [6] where some examples for string grammars of specific types illustrating the expressive power of this general framework are given.

Array grammars

We now introduce the basic notions for n-dimensional arrays and array grammars, for example, see [4, 11, 13].

Let $d \in \mathbb{N}$. Then a *d-dimensional array* $\mathcal{A}$ over an alphabet V is a function $\mathcal{A} : \mathbb{Z}^d \rightarrow V \cup \{\#\}$, where $shape(\mathcal{A}) = \left\{v \in \mathbb{Z}^d \mid \mathcal{A}(v) \neq \#\right\}$ is finite and $\# \notin V$ is called the *background* or *blank symbol*. We usually write $\mathcal{A} = \{(v, \mathcal{A}(v)) \mid v \in shape(\mathcal{A})\}$.

The set of all d-dimensional arrays over V is denoted by V^{*d}. The *empty array* in V^{*d} with empty shape is denoted by Λ_d. Moreover, we define $V^{+d} = V^{*d} \setminus \{\Lambda_d\}$.

Let $v \in \mathbb{Z}^d$, $v = (v_1, \ldots, v_d)$. The *translation* $\tau_v : \mathbb{Z}^d \rightarrow \mathbb{Z}^d$ is defined by $\tau_v(w) = w + v$ for all $w \in \mathbb{Z}^d$, and for any array $\mathcal{A} \in V^{*d}$ we define $\tau_v(\mathcal{A})$, the corresponding d-dimensional array translated by v, by $(\tau_v(\mathcal{A}))(w) = \mathcal{A}(w - v)$ for all $w \in \mathbb{Z}^d$. The vector $(0, \ldots, 0) \in \mathbb{Z}^d$ is denoted by Ω_d.

A *d-dimensional array rule* p over V is a triple $(W, \mathcal{A}_1, \mathcal{A}_2)$, where $W \subseteq \mathbb{Z}^d$ is a finite set and $\mathcal{A}_1$ and $\mathcal{A}_2$ are mappings from W to $V \cup \{\#\}$ such that $shape(\mathcal{A}_1) \neq \emptyset$. We say that the array $\mathcal{B}_2 \in V^{*d}$ is *directly derivable* from the array $\mathcal{B}_1 \in V^{*d}$ by the d-dimensional array rule $(W, \mathcal{A}_1, \mathcal{A}_2)$, i.e., $\mathcal{B}_1 \Longrightarrow_p \mathcal{B}_2$, if and only if there exists a vector $v \in \mathbb{Z}^d$ such that $\mathcal{B}_1(w) = \mathcal{B}_2(w)$ for all $w \in \mathbb{Z}^d \setminus \tau_v(W)$ as well as $\mathcal{B}_1(w) = \mathcal{A}_1(\tau_{-v}(w))$ and $\mathcal{B}_2(w) = \mathcal{A}_2(\tau_{-v}(w))$ for all $w \in \tau_v(W)$, i.e., the subarray of $\mathcal{B}_1$ corresponding to $\mathcal{A}_1$ is replaced by $\mathcal{A}_2$, thus yielding $\mathcal{B}_2$. In the following, we shall also write $\mathcal{A}_1 \rightarrow \mathcal{A}_2$, because W is implicitly given by the finite arrays $\mathcal{A}_1, \mathcal{A}_2$.

A *d-dimensional array grammar* G_A is represented as

$$\left((N \cup T)^{*d}, T^{*d}, \{(v_0, S)\}, P, \Longrightarrow_{G_A}\right) \text{ where}$$

- N is the alphabet of *non-terminal symbols*;
- T is the alphabet of *terminal symbols*, $N \cap T = \emptyset$;
- $\{(v_0, S)\}$ is the *start array (axiom)* with $S \in N$ and $v_0 \in \mathbb{Z}^d$*;*
- P is a finite set of *d-dimensional array rules* over V, $V := N \cup T$;
- $\Longrightarrow_{G_A}$ is the derivation relation induced by the array rules in P according to the explanations given above, i.e., for arbitrary $\mathcal{B}_1, \mathcal{B}_2 \in V^{*d}$, $\mathcal{B}_1 \Longrightarrow_{G_A} \mathcal{B}_2$ if

and only if there exists a d-dimensional array rule $p = (W, \mathcal{A}_1, \mathcal{A}_2)$ in P such that $\mathcal{B}_1 \Longrightarrow_p \mathcal{B}_2$.

A d-dimensional array rule $p = (W, \mathcal{A}_1, \mathcal{A}_2)$ in P is called *#-context-free*, if *shape* $(\mathcal{A}_1) = \{\Omega_d\}$. A d-dimensional array grammar is said to be of type d-$ARBA$, d-#-CFA if every array rule in P is of the corresponding type, i.e., an arbitrary and #-context-free d-dimensional array rule, respectively. The corresponding families of d-dimensional array languages of type X are denoted by $\mathcal{L}(X)$, i.e., $\mathcal{L}(d\text{-}ARBA)$ and $\mathcal{L}(d\text{-}\#\text{-}CFA)$ are the families of recursively enumerable and #-context-free d-dimensional array languages, respectively.

Observe that the types d-$ARBA$ and d-#-CFA are types with unit rules and trap rules – for $p = (W, \mathcal{A}_1, \mathcal{A}_2)$, we can take $p^{(+)} = (W, \mathcal{A}_1, \mathcal{A}_1)$ and $p^{(-)} = (W, \mathcal{A}_1, \mathcal{A}_F)$ with $\mathcal{A}_F(v) = F$ for $v \in W$, where F is a new non-terminal symbol – the trap symbol.

Multiset grammars

$G_m = \left((N \cup T)^\circ, T^\circ, w, P, \Longrightarrow_{G_m}\right)$ is called a *multiset grammar*; N is the alphabet of *non-terminal symbols*, T is the alphabet of *terminal symbols*, $N \cap T = \emptyset$, w is a non-empty multiset over V, $V := N \cup T$, and P is a (finite) set of multiset rules yielding a derivation relation $\Longrightarrow_{G_m}$ on the multisets over V; the application of the rule $u \to v$ to a multiset x has the effect of replacing the multiset u contained in x by the multiset v. For the multiset grammar G_m we also write $(N, T, w, P, \Longrightarrow_{G_m})$.

As special types of multiset grammars we consider multiset grammars with *arbitrary* rules as well as *context-free* (*non-cooperative*) rules of the form $A \to v$ with $A \in N$ and $v \in V^\circ$; the corresponding types X of multiset grammars are denoted by $mARB$ and mCF, thus yielding the families of multiset languages $\mathcal{L}(X)$. Observe that $mARB$ and mCF are types with unit rules and trap rules (for $p = w \to v \in P$, we can take $p^{(+)} = w \to w$ and $p^{(-)} = w \to F$ where F is a new symbol – the trap symbol). Even with arbitrary multiset rules, it is not possible to get $Ps(\mathcal{L}(ARB))$ [8]:

$$\mathcal{L}(mCF) = Ps(\mathcal{L}(CF)) \subsetneqq \mathcal{L}(mARB) \subsetneqq Ps(\mathcal{L}(ARB)).$$

2.4 Register Machines

As a computationally complete model able to generate/accept all sets in $PsRE = Ps(\mathcal{L}(ARB))$ we use register machines/deterministic register machines:

A *register machine* is a construct $M = (n, L_M, R_M, p_0, h)$ where n, $n \geq 1$, is the number of registers, L_M is the set of instruction labels, p_0 is the start label, h is the halting label (only used for the `HALT` instruction), and R_M is a set of (labeled) instructions being of one of the following forms:

- $p : (\texttt{ADD}(r), q, s)$ increments the value in register r and continues with the instruction labeled by q or s,

- $p : (\texttt{SUB}(r), q, s)$ decrements the value in register r and continues the computation with the instruction labeled by q if the register was non-empty, otherwise it continues with the instruction labeled by s;
- $h : \texttt{HALT}$ halts the machine.

M is called deterministic if in all ADD-instructions $p : (\texttt{ADD}(r), q, s)$ $q = s$; in this case we write $p : (\texttt{ADD}(r), q)$. Deterministic register machines can accept all recursively enumerable sets of vectors of natural numbers with k components using $k + 2$ registers, for instance, see [9].

2.5 Graph-controlled and Programmed Grammars

A *graph-controlled grammar* (with applicability checking) of type X is a construct

$$G_{GC} = (G, g, H_i, H_f, \Longrightarrow_{GC})$$

where $G = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X; $g = (H, E, K)$ is a labeled graph where H is the set of node labels identifying the nodes of the graph in a one-to-one manner, $E \subseteq H \times \{Y, N\} \times H$ is the set of edges labeled by Y or N, $K : H \to 2^P$ is a function assigning a subset of P to each node of g; $H_i \subseteq H$ is the set of initial labels, and $H_f \subseteq H$ is the set of final labels. The derivation relation $\Longrightarrow_{GC}$ is defined based on $\Longrightarrow_G$ and the control graph g as follows: For any $i, j \in H$ and any $u, v \in O$, $(u, i) \Longrightarrow_{GC} (v, j)$ if and only if

- $u \Longrightarrow_p v$ by some rule $p \in K(i)$ and $(i, Y, j) \in E$ *(success case)*, **or**
- $u = v$, no $p \in K(i)$ is applicable to u, and $(i, N, j) \in E$ *(failure case)*.

The language generated by G_{GC} is defined by

$$L(G_{GC}) = \left\{ v \in O_T \mid (w, i) \Longrightarrow_{G_{GC}}^{*} (v, j),\ i \in H_i, j \in H_f \right\}.$$

If $H_i = H_f = H$, then G_{GC} is called a *programmed grammar*. The families of languages generated by graph-controlled and programmed grammars of type X are denoted by $\mathcal{L}(X\text{-}GC_{ac})$ and $\mathcal{L}(X\text{-}P_{ac})$, respectively. If the set E contains no edges of the form (i, N, j), then the graph-controlled grammar is said to be *without applicability checking*; the corresponding families of languages are denoted by $\mathcal{L}(X\text{-}GC)$ and $\mathcal{L}(X\text{-}P)$, respectively.

As a special variant of graph-controlled grammars we consider those where all labels are final; the corresponding family of languages generated by graph-controlled grammars of type X is abbreviated by $\mathcal{L}\left(X\text{-}GC_{ac}^{allfinal}\right)$. By definition, programmed grammars are just a subvariant where in addition all labels are also initial.

The notions and concepts *with/without applicability checking* were introduced as *with/without appearance checking* in the original definition for string grammars because the appearance of the non-terminal symbol on the left-hand side of a context-free rule was checked, which coincides with checking for the applicability of this rule in our general model; in both cases – applicability checking and appearance checking – we can use the abbreviation *ac*.

2.6 Matrix Grammars

A *matrix grammar* (with applicability checking) of type X is a construct

$$G_M = (G, M, F, \Longrightarrow_{G_M})$$

where $G = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X, M is a finite set of sequences of the form $(p_1, \dots, p_n)$, $n \geq 1$, of rules in P, and $F \subseteq P$. For $w, z \in O$ we write $w \Longrightarrow_{G_M} z$ if there are a matrix $(p_1, \dots, p_n)$ in M and objects $w_i \in O$, $1 \leq i \leq n+1$, such that $w = w_1$, $z = w_{n+1}$, and, for all $1 \leq i \leq n$, either

- $w_i \Longrightarrow_G w_{i+1}$ or
- $w_i = w_{i+1}$, p_i is not applicable to w_i, and $p_i \in F$.

$L(G_M) = \left\{ v \in O_T \mid w \Longrightarrow_{G_M}^* v \right\}$ is the language generated by G_M. The family of languages generated by matrix grammars of type X is denoted by $\mathcal{L}\left(X\text{-}MAT_{ac}\right)$. If the set F is empty, then the grammar is said to be *without applicability checking*; the corresponding family of languages is denoted by $\mathcal{L}\left(X\text{-}MAT\right)$.

We mention that in this paper we choose the definition where the sequential application of the rules of the final matrix may stop at any moment.

2.7 Random-Context Grammars

The following general notion of a random context-grammar had already been introduced in [7, 1] in a similar way before it was formulated in [6].

A *random-context grammar* G_{RC} *of type* X is a construct $(G, P', \Longrightarrow_{G_{RC}})$ where

- $G = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X;
- P' is a set of rules of the form (q, R, Q) where $q \in P$, $R \cup Q \subseteq P$;
- $\Longrightarrow_{G_{RC}}$ is the derivation relation assigned to G_{RC} such that for any $x, y \in O$, $x \Longrightarrow_{G_{RC}} y$ if and only if for some rule $(q, R, Q) \in P'$, $x \Longrightarrow_q y$ and, moreover, all rules from R are applicable to x as well as no rule from Q is applicable to x.

A random-context grammar $G_{RC} = (G, P', \Longrightarrow_{G_{RC}})$ of type X is called a *grammar with permitting contexts of type* X if for all rules (q, R, Q) in P' we have $Q = \emptyset$, i.e., we only check for the applicability of the rules in R.

A random-context grammar $G_{RC} = (G, P', \Longrightarrow_{G_{RC}})$ of type X is called a *grammar with forbidden contexts of type* X if for all rules (q, R, Q) in P' we have $R = \emptyset$, i.e., we only check for the non-applicability of the rules in Q.

$L(G_{RC}) = \left\{ v \in O_T \mid w \Longrightarrow_{G_{RC}}^* v \right\}$ is the language generated by G_{RC}. The families of languages generated by random context grammars, grammars with permitting contexts, and grammars with forbidden contexts of type X are denoted by $\mathcal{L}\left(X\text{-}RC\right)$, $\mathcal{L}\left(X\text{-}pC\right)$, and $\mathcal{L}\left(X\text{-}fC\right)$, respectively.

2.8 Ordered Grammars

An *ordered grammar* G_O *of type* X is a construct $(G_s, \prec, \Longrightarrow_{G_O})$ where

- $G_s = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X;
- $\prec$ is a partial order relation on the rules in P;
- $\Longrightarrow_{G_O}$ is the derivation relation assigned to G_O such that for any $x, y \in O$, $x \Longrightarrow_{G_O} y$ if and only if for some rule $q \in P$ $x \Longrightarrow_q y$ and, moreover, no rule p from P with $q \prec p$ is applicable to x.

$L(G_O) = \left\{ v \in O_T \mid w \Longrightarrow_{G_O}^* v \right\}$ is the language generated by G_O. The family of languages generated by ordered grammars of type X is denoted by $\mathcal{L}(X\text{-}O)$.

2.9 Grammars with Activation and Blocking of Rules

We now define our new concept of regulating the application of rules at a specific moment by activation and blocking relations.

A *grammar with activation and blocking of rules* (an *AB-grammar* for short) of type X is a construct

$$G_M = (G, L, f_L, A, B, L_0, \Longrightarrow_{G_{AB}})$$

where $G = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X, L is a finite set of labels with each label having assigned one rule from P by the function f_L, A, B are finite subsets of $L \times L \times \mathbb{N}$, and L_0 is a finite set of tuples of the form $\left(q, Q, \bar{Q}\right)$, $q \in L$, with the elements of $Q, \bar{Q}$ being of the form (l, t), where $l \in L$ and $t \in \mathbb{N}$, $t > 1$.

A derivation in G_M starts with one element $\left(q, Q, \bar{Q}\right)$ from L_0 which means that the rule labeled by q has to be applied to the initial object w in the first step and for the following derivation steps the conditions given by Q as activations of rules and $\bar{Q}$ as blockings of rules have to be taken into account in addition to the activations and blockings coming along with the application of the rule labeled by q. The role of L_0 is to get a derivation started by activating some rule for the first step although no rule has been applied so far, but probably also providing additional activations and blockings for further derivation steps.

A configuration of G_M in general can be described by the object derived so far and the activations Q and blockings $\bar{Q}$ for the next steps. In that sense, the starting tuple $\left(q, Q, \bar{Q}\right)$ can be interpreted as $\left(\{(q,1)\} \cup Q, \bar{Q}\right)$, and we may also simply write $\left(Q', \bar{Q}\right)$ with $Q' = \{(q,1)\} \cup Q$. We mostly will assume Q and $\bar{Q}$ to be non-conflicting, i.e., $Q \cap \bar{Q} = \emptyset$; otherwise, we interpret $\left(Q', \bar{Q}\right)$ as $\left(Q' \setminus \bar{Q}, \bar{Q}\right)$.

Given a configuration $\left(u, Q, \bar{Q}\right)$, in one step we can derive $\left(v, R, \bar{R}\right)$, and we also write

$$\left(u, Q, \bar{Q}\right) \Longrightarrow_{G_{AB}} \left(v, R, \bar{R}\right),$$

if and only if

- $u \Longrightarrow_G v$ using the rule r such that $(q,1) \in Q$ and $(q,r) \in f_L$, i.e., we apply the rule labeled by q activated for this next derivation step to u; the new sets of activations and blockings are defined by

$$\begin{aligned}\bar{R} &= \left\{(x,i) \mid (x,i+1) \in \bar{Q},\ i>0\right\} \cup \{(x,i) \mid (q,x,i) \in B\}, \\ R &= (\{(x,i) \mid (x,i+1) \in Q,\ i>0\} \cup \{(x,i) \mid (q,x,i) \in A\}) \\ &\quad \setminus \ \left\{(x,i) \mid (x,i) \in \bar{R}\right\}\end{aligned}$$

  (observe that R and $\bar{R}$ are made non-conflicting by eliminating rule labels which are activated and blocked at the same time);
  or
- no rule r is activated to be applied in the next derivation step; in this case we take $v=u$ and continue with $\left(v,R,\bar{R}\right)$ constructed as before provided R is not empty, i.e., there are rules activated in some further derivation steps; otherwise the derivation stops.

The language generated by G_{AB} is defined by

$$L(G_{AB}) = \left\{v \in O_T \mid \left(w,Q,\bar{Q}\right) \Longrightarrow^*_{G_{AB}} \left(v,R,\bar{R}\right) \text{ for some } \left(Q,\bar{Q}\right) \in L_0\right\}.$$

The family of languages generated by AB-grammars of type X is denoted by $\mathcal{L}(X\text{-}AB)$. If the set B of blocking relations is empty, then the grammar is said to be a *grammar with activation of rules* (an *A-grammar* for short) of type X; the corresponding family of languages is denoted by $\mathcal{L}(X\text{-}A)$. In this case we might not allow the second case in a derivation of the A-grammar that in a derivation step no rule is activated to be applied. Moreover, an A-grammar is called an *A*1-*grammar* if for all $(p,q,t) \in A$ we have $t=1$, which means that only the rule applied in one derivation step activates the rules which can be applied in the next step; in this case we may only write (p,q) instead of $(p,q,1)$. Moreover, in L_0 we may simply list the labels of the rules to be applied in the first step.

Example 1. Consider the string grammar $G_S = \left((N \cup T)^*, T^*, w, P, \Longrightarrow_P\right)$ with $N=\{A,B,C\}$, $T=\{a,b,c\}$, $w=ABC$, and the set of rules
$P=\{A \to aA, B \to bB, C \to cC, A \to \lambda, B \to \lambda, C \to \lambda\}$,
as well as the A1-grammar
$G_A = (G, L, f_L, A, L_0, \Longrightarrow_{G_A})$ with
$L=\{p_a,p_b,p_c,p_A,p_B,p_C\}$, and, writing $p:r$ for the pairs (p,r) in f_L,

$$\begin{aligned} f_L &= \{p_a : A \to aA, p_b : B \to bB, p_c : C \to cC\} \\ &\cup \{p_A : A \to \lambda, p_B : B \to \lambda, p_C : C \to \lambda\} \\ A &= \{(p_a,p_b),(p_b,p_c),(p_c,p_a),(p_c,p_A),(p_A,p_B),(p_B,p_C)\}, \text{ and} \\ P_0 &= \{p_a,p_A\}. \end{aligned}$$

The underlying string grammar generates the regular set $\{a\}^*\{b\}^*\{c\}^*$, whereas the A1-grammar G_A generates $\{a^nb^nc^n \mid n \in \mathbb{N}_0\}$: starting with the rule labeled by p_a from L_0, the rules corresponding to the sequence of labels $p_ap_bp_c$ is applied $n \geq 1$ times, and finally we switch to the sequence of rules given by $p_Ap_Bp_C$ whereafter no rule can be applied any more. Starting with p_A yields the empty string.

Only allowing blocking of rules would not make sense, but if we implicitly have all rules activated in every derivation step, then blocking some of the rules with the application of a rule in a derivation step for the next derivation step(s) allows us to speak of a *grammar with blocking of rules* (a *B-grammar* for short) of type X; the corresponding family of languages is denoted by $\mathcal{L}(X\text{-}B)$. Moreover, a B-grammar is called a *$B1$-grammar* if for all $(p,q,t) \in B$ we have $t=1$, which means that the rule applied in one derivation step can only block the rules to be applied in the next step; in this case we again only write (p,q) instead of $(p,q,1)$. Moreover, in L_0 we may simply list the labels of the rules to be applied in the first step.

Example 2. We consider the same underlying string grammar as in Example 1, $G_S = \left((N \cup T)^*, T^*, w, P, \Longrightarrow_P\right)$ with $N = \{A,B,C\}$, $T = \{a,b,c\}$, $w = ABC$, and the set of rules
$P = \{A \to aA, B \to bB, C \to cC, A \to \lambda, B \to \lambda, C \to \lambda\}$. From the A1-grammar as constructed in Example 1, we construct an equvalent B1-grammar
$G_B = (G, L, f_L, B, L_0, \Longrightarrow_{G_B})$ with
$L = \{p_a, p_b, p_c, p_A, p_B, p_C\}$, and, writing $p : r$ for the pairs (p,r) in f_L,

$$\begin{aligned}
f_L &= \{p_a : A \to aA, p_b : B \to bB, p_c : C \to cC\} \\
&\cup \{p_A : A \to \lambda, p_B : B \to \lambda, p_C : C \to \lambda\} \\
B &= \{(p_a, L \setminus \{p_b\}), (p_b, L \setminus \{p_c\}), (p_c, L \setminus \{p_a, p_A\})\} \\
&\cup \{(p_A, L \setminus \{p_B\}), (p_B, L \setminus \{p_C\})\}, \text{ and} \\
P_0 &= \{p_a, p_A\}.
\end{aligned}$$

This B1-grammar G_B generates the same language as the A1-grammar G_A constructed in Example 1, i.e., $\{a^n b^n c^n \mid n \in \mathbb{N}_0\}$: instead of activating the next rules to be applied, we block all the other rules.

3 General Results

In this section, we elaborate some general results holding true for many types of grammars, some even holding for any type X, whereas some of them rely on specific conditions on X.

3.1 General Results for Standard Control mechanisms

The main results elaborated for the relations between the specific regulating mechanisms in [6] and in [5] (not including the new mechanism of activation and blocking of rules) are depicted in Figure 1; most of these relations even hold for arbitrary types X.

Theorem 1. *The inclusions indicated by vectors as depicted in Figure 1 hold, the additionally needed features of having unit and/or trap rules indicated by u and t, respectively, aside the vector.*

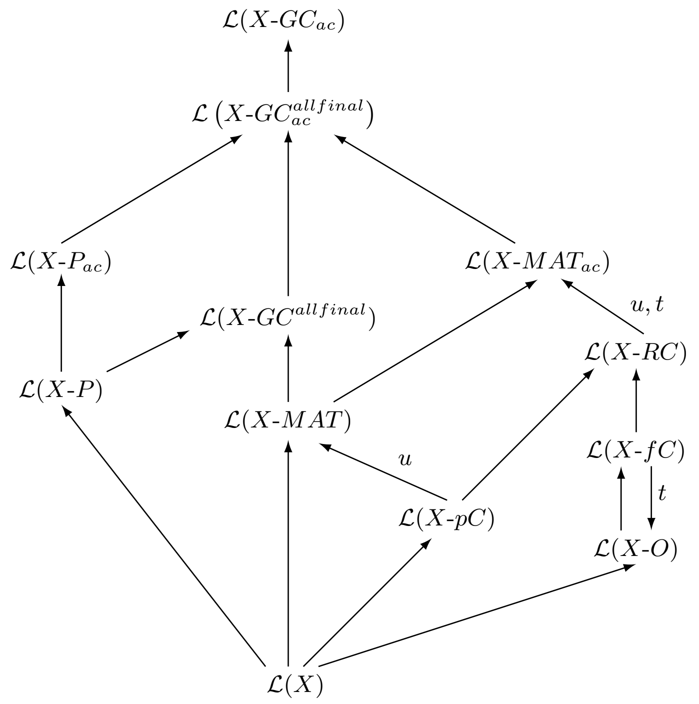

$\mathcal{L}(X\text{-}GC_{ac})$
$\mathcal{L}\left(X\text{-}GC_{ac}^{allfinal}\right)$
$\mathcal{L}(X\text{-}P_{ac})$
$\mathcal{L}(X\text{-}MAT_{ac})$
u, t
$\mathcal{L}(X\text{-}GC^{allfinal})$
$\mathcal{L}(X\text{-}RC)$
$\mathcal{L}(X\text{-}P)$
$\mathcal{L}(X\text{-}MAT)$
$\mathcal{L}(X\text{-}fC)$
u
t
$\mathcal{L}(X\text{-}pC)$
$\mathcal{L}(X\text{-}O)$
$\mathcal{L}(X)$

Fig. 1. Hierarchy of control mechanisms for grammars of type X.

3.2 A1-Grammars and B1-Grammars

There is an interesting relation between A1-Grammars and B1-Grammars which is quite surprising as usually forbidding rules to be applied does not yield the same computational power as prescribing the rules to be applied in the next step as, for example, in matrix grammars without *ac*. The conceptual reason behind this result is that in B-grammars, by default, all rules are activated for every derivation step.

Theorem 2. *For any type X, $\mathcal{L}(X\text{-}A1) = \mathcal{L}(X\text{-}B1)$.*

Proof. We first show $\mathcal{L}(X\text{-}A1) \subseteq \mathcal{L}(X\text{-}B1)$.

Let $G_A = (G, L, f_L, A, L_0, \Longrightarrow_{G_A})$ be an A1-grammar where the underlying grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ is of type X, L is a finite set of labels with each label having assigned one rule from P by the function f_L, A is a finite subset of $L \times L$, and $L_0 \subseteq L$ is the set of initial rule labels.

Then we define the equivalent B1-grammar of type X G_B as follows:

$$\begin{aligned} G_B &= (G, L, f_L, B, L_0, \Longrightarrow_{G_B}), \\ B &= \{(l, L \setminus \{m \mid (l, m) \in A\}) \mid l \in L\}, \end{aligned}$$

i.e., B is constructed in such a way that instead of activating the rules to be applied in the next derivation step, we block all the other rules – observe that by default in B-grammars all rules are activated for every derivation step (compare this with the construction of the B1-grammar in Example 2 from the A1-grammar given in Example 1).

We now show the other direction, $\mathcal{L}(X\text{-}A1) \supseteq \mathcal{L}(X\text{-}B1)$.

Let $G_B = (G, L, f_L, B, L_0, \Longrightarrow_{G_B})$ be a B1-grammar where the underlying grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ is of type X, L is a finite set of labels with each label having assigned one rule from P by the function f_L, B is a finite subset of $L \times L$, and $L_0 \subseteq L$ is the set of initial rule labels.

Then we define the equivalent A1-grammar of type X G_A as follows:

$$\begin{aligned} G_A &= (G, L, f_L, A, L_0, \Longrightarrow_{G_A}), \\ A &= \{(l, L \setminus \{m \mid (l,m) \in B\}) \mid l \in L\}, \end{aligned}$$

i.e., A is constructed from B in such a way that only those rules are activated to be applied in the next derivation step which are not blocked according to B. □

It remains as an open question if a similar result also holds for arbitrary A- and B-grammars.

3.3 Matrix Grammars and A1-Grammars

Our first result shows a close connection between matrix grammars without appearance checking and A1-grammars:

Theorem 3. *For any type X, $\mathcal{L}(X\text{-}MAT) \subseteq \mathcal{L}(X\text{-}A1)$.*

Proof. Let $G_M = (G, M, F, \Longrightarrow_{G_M})$ be a matrix grammar with the underlying grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ being a grammar of type X; let $M = \{(p_{i,1}, \dots, p_{i,n_i}) \mid 1 \le i \le n\}$ with $p_{i,j} \in P$, $1 \le j \le n_i$, $1 \le i \le n$.

We construct the equivalent A1-grammar

$$\begin{aligned} G_A &= (G, L, f_L, A, L_0, \Longrightarrow_{G_A}), \\ L &= \{l_{i,j} \mid 1 \le j \le n_i,\ 1 \le i \le n\}, \\ f_L &= \{(l_{i,j}, p_{i,j}) \mid 1 \le j \le n_i,\ 1 \le i \le n\}, \\ A &= \{(l_{i,j}, l_{i,j+1}) \mid 1 \le j < n_i, 1 \le i \le n\} \\ &\quad \cup \{(l_{i,n_i}, l_{j,1}) \mid 1 \le j \le n, 1 \le i \le n\}, \\ L_0 &= \{l_{i,1} \mid 1 \le i \le n\}. \end{aligned}$$

We mention that according to our definitions the sequential application of the rules of the chosen matrix may stop at any moment if the next rule cannot be applied, in which case also the simulation in the A1-grammar stops. □

We immediately infer the following for the special cases of strings, multisets, and arrays as underlying objects:

Corollary 1. *For $X \in \{CF, mCF\} \cup \{d\text{-}\#\text{-}CFA \mid d \in \mathbb{N}\}$,*

$$\mathcal{L}(X\text{-}MAT) \subseteq \mathcal{L}(X\text{-}A1).$$

3.4 Random Context Grammars and AB-Grammars

For any type X with unit rules, random context grammars of type X can be simulated by AB-grammars of type X.

Remark 1. In order to keep proofs shorter, in the following, instead of specifying the set of rules P, the set of labels L, and the function f_L assigning rules to the labels separately, we will only specify the corresponding labeled rules of the form $l : r$ with $l \in L$, $r \in P$, and $(l, r) \in f_L$. Moreover, for $X \in \{A, B\}$, instead of $(p, q, t) \in X$, we write $(p, q, t)_X$.

Theorem 4. *For any type X with unit rules, $\mathcal{L}(X\text{-}RC) \subseteq \mathcal{L}(X\text{-}AB)$.*

Proof. Let $(G, R, \Longrightarrow_{G_{RC}})$ be a random context grammar with the underlying grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ being of a type X with unit rules, where

$$\begin{aligned} R &= \{(r_i, P_i, Q_i) \mid 1 \le i \le n\}, r_i \in P,\ 1 \le i \le n, \\ P_i &= \{p_{i,j} \mid 1 \le j \le m_i,\ 1 \le i \le n\}, m_i \ge 0,\ 1 \le i \le n, \\ Q_i &= \{q_{i,j} \mid 1 \le j \le n_i,\ 1 \le i \le n\}, n_i \ge 0,\ 1 \le i \le n. \end{aligned}$$

Then we construct an AB-grammar G_{AB} of type X as follows:

$$\begin{aligned} G_{AB} &= (G', L, f_L, A, B, L_0, \Longrightarrow_{G_A}), \\ G' &= (O, O_T, w, P', \Longrightarrow_{G'}), \\ P' &= P \cup \{r^+ \mid r \in P\}; \\ L_0 &= \{l_{r_i} \mid 1 \le i \le n\}; \end{aligned}$$

the application of a random context rule (r_i, P_i, Q_i) is simulated by the following sequence of labeled rules together with suitable activations and blockings of rules:

- $l_{r_i} : {r_i}^+$, $(l_{r_i}, l_{r_i,1})_A$, $(l_{r_i}, \bar{l}_{r_i,j}, m_i{+}j)_A$, $1 \le j \le n_i$; at the beginning, the checking of all rules which should not be applicable is initiated, and the sequence of applicability checkings for the rules in P_i is started;
- $l_{r_i,j} : {p_{i,j}}^+$, $(l_{r_i,j}, l_{r_i,j+1})_A$, $1 \le j < m_i$;
- $l_{r_i,m_i} : {p_{i,m_i}}^+$, $(l_{r_i,m_i}, \hat{l}_{r_i}, n_i + 1)_A$; when all rules in P_i have been checked to be applicable, the application of rule r_i after further n_i steps is activated; yet if any of the rules in Q_i is applicable, then this application of rule r_i is blocked;
- $\bar{l}_{r_i,j} : {q_{i,j}}^+$, $(\bar{l}_{r_i,j}, \hat{l}_{r_i}, n_i - j + 1)_B$, $1 \le j \le n_i$;
- $\hat{l}_{r_i} : r_i$, $(\hat{l}_{r_i}, l_{r_k})$, $1 \le k \le n$; after the successful application of rule r we may continue with trying to apply any random context rule from R.

We finally observe that only unit rules and no trap rules as in other simulations known from [6] are needed to obtain this result. □

3.5 AB-Grammars and Graph-Controlled Grammars

Already in [6] graph-controlled grammars have been shown to be the most powerful control mechanism, and they can also simulate AB-grammars with the underlying grammar being of any arbitrary type X.

Theorem 5. *For any type X, $\mathcal{L}(X\text{-}AB) \subseteq \mathcal{L}(X\text{-}GC_{ac})$.*

Proof. Let $G_{AB} = (G, L, f_L, A, B, L_0, \Longrightarrow_{G_A})$ be an AB-grammar with the underlying grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ being of any type X. Then we construct a graph-controlled grammar

$$G_{GC} = (G, g, H_i, H_f, \Longrightarrow_{GC})$$

with the same underlying grammar G. The simulation power is captured by the structure of the control graph $g = (H, E, K)$. The node labels in H, identifying the nodes of the graph in a one-to-one manner, are obtained from G_{AB} as all possible triples of the forms $\left(q, Q, \bar{Q}\right)$ or $\left(\bar{q}, Q, \bar{Q}\right)$ with $q \in L$ and the elements of $Q, \bar{Q}$ being of the form (r, t), $r \in L$ and $t \in \mathbb{N}$ such that t does not exceed the maximum time occurring in the relations in A and B, hence, this in total is a bounded number. We also need a special node labeled $\emptyset$, where a computation in G_{GC} ends in any case when this node is reached.

All nodes can be chosen to be final, i.e., $H_f = H$. $H_i = L_0$ is the set of initial labels, i.e., we start with one of the initial conditions as in the AB-grammar.

The idea behind the node $\left(q, Q, \bar{Q}\right)$ is to describe the situation of a configuration derived in the AB-grammar where q is the label of the rule to be applied and $Q, \bar{Q}$ describe the activated and blocked rules for the further derivation steps in the AB-grammar. Hence, as already in the definition of an AB-grammar, we therefore assume $Q \cap \bar{Q} = \emptyset$.

Now let $g(l)$ denote the rule r assigned to label l, i.e., $(l, r) \in f_L$. Then, the set of rules assigned to $\left(q, Q, \bar{Q}\right)$ is taken to be $\{g(q)\}$. The set of rules assigned to $\emptyset$ is taken to be $\emptyset$.

As it will become clear later in the proof why, the nodes $\left(\bar{q}, Q, \bar{Q}\right)$ are assigned the set of rules $\{g(l) \mid (l, 1) \in Q,\ l \neq q\}$; we only take those nodes where this set is not empty.

When being in node $\left(q, Q, \bar{Q}\right)$, we have to distinguish between two possibilities:

- If $g(q)$ is applicable to the object derived so far, a Y-edge has to go to every node which describes a situation corresponding to what would have been the next configuration in the AB-grammar. We then compute

$$\begin{aligned}\bar{R} &= \left\{(x, i) \mid (x, i+1) \in \bar{Q},\ i > 0\right\} \cup \{(x, i) \mid (q, x, i) \in B\}, \\ R &= (\{(x, i) \mid (x, i+1) \in Q,\ i > 0\} \cup \{(x, i) \mid (q, x, i) \in A\}) \\ &\quad \setminus \ \left\{(x, i) \mid (x, i) \in \bar{R}\right\}\end{aligned}$$

  (observe that R and $\bar{R}$ are made non-conflicting) as well as – if it exists – $t_0 := min\{t \mid (x, t) \in R\}$, i.e., the next time step when the derivation in the AB-grammar could continue. Hence, we take a Y-edge to every node $\left(p, P, \bar{P}\right)$ where $p \in \{x \mid (x, t_0) \in R\}$ and

$$\begin{aligned}\bar{P} &= \left\{(x, i) \mid (x, i + t_0 - 1) \in \bar{R},\ i > 0\right\}, \\ P &= \{(x, i) \mid (x, i + t_0 - 1) \in R\}.\end{aligned}$$

If $t_0 := min\{t \mid (x,t) \in R\}$ does not exist, this means that R is empty and we have to make a Y-edge to the node $\emptyset$.

- If $g(q)$ is not applicable to the object derived so far, we first have to check that none of the other rules activated at this step could have been applied, i.e., we check for the applicability of the rules in the set of rules

$$\bar{U} := \{g(l) \mid (l,1) \in Q,\ l \neq q\}$$

by going to the node $(\bar{q}, Q, \bar{Q})$ with a N-edge; from there no Y-edge leaves, as this would indicate the unwanted case of the applicability of one of the rules in $\bar{U}$, but with a N-edge we continue the computation in any node $(p, P, \bar{P})$ with p, P, $\bar{P}$ computed as above in the first case. We observe that in case R is empty, we can omit the path through the node $(\bar{q}, Q, \bar{Q})$ and directly go to the nodes $(p, P, \bar{P})$ which are obtained as follows: we first check whether $t_0 := min\{t \mid (x,t) \in Q,\ t > 1\}$ exists or not; if not, then the computation has to end with a N-edge to node $\emptyset$. Otherwise, a N-edge goes to every node $(p, P, \bar{P})$ with $p \in \{x \mid (x, t_0) \in Q\}$ and

$$\begin{aligned}\bar{P} &= \left\{(x,i) \mid (x, i + t_0 - 1) \in \bar{Q},\ i > 0\right\},\\ P &= \{(x,i) \mid (x, i + t_0 - 1) \in Q\}.\end{aligned}$$

where the simulation may continue.

In this way, every computation in the AB-grammar can be simulated by the graph-controlled grammar with taking a correct path through the control graph and finally ending in node $\emptyset$; due to this fact, we could also choose the node $\emptyset$ to be the only final node, i.e., $H_f = \{\emptyset\}$. On the other hand, if we have made a wrong choice and wanted to apply a rule which is not applicable, although another rule activated at the same moment would have been applicable, we get stuck, but the derivation simulated in this way still is a valid one in the AB-grammar, although in most standard types X, which usually are strictly extended ones, such a derivation does not yield a terminal object. Having taken $H_f = \{\emptyset\}$, such paths would not even lead to successful computations in G_{GC}.

In any case, we conclude that the graph-controlled grammar G_{GC} generates the same language as the AB-grammar G_{AB}, which observation concludes the proof. □

The power of rule activation is really strong and in most cases the additional power of blocking is not needed. As a special variant of graph-controlled grammars we consider those where all labels are final; the corresponding family of languages generated by graph-controlled grammars of type X is abbreviated by $\mathcal{L}\left(X\text{-}GC_{ac}^{allfinal}\right)$.

Theorem 6. *For any strictly extended type X with unit rules and trap rules,*

$$\mathcal{L}\left(X\text{-}GC_{ac}^{allfinal}\right) \subseteq \mathcal{L}\left(X\text{-}A\right).$$

Proof. Let

$$G_{GC} = (G, g, H_i, H_f, \Longrightarrow_{GC})$$

be a graph-controlled grammar where $G = (O, O_T, w, P, \Longrightarrow_G)$ is a strictly extended grammar of type X; $g = (H, E, K)$, $E \subseteq H \times \{Y, N\} \times H$ is the set of edges labeled by Y or N, $K : H \to 2^P$ is a function assigning a subset of P to each node of g; $H_i \subseteq H$ is the set of initial labels, and H_f is the set of final labels coinciding with the whole set H, i.e., $H_f = H$.

Then we construct an equivalent A-grammar

$$G_A = (G', L, f_L, A, L_0, \Longrightarrow_{G_A})$$

as follows:

The underlying grammar G' is obtained from G by adding all unit and trap rules, i.e., $G' = (O, O_T, w, P', \Longrightarrow_{G'})$ with $P' = P \cup \{p^+, p^- \mid p \in P\}$. G' again is strictly extended and $w \notin O_T$, hence, also in G_A rules have to be applied before terminal objects are obtained. For any node in g labeled by l with the assigned set of rules P_l we assume it to be described by $P_l = \{p_{l,i} \mid 1 \leq i \leq n_l\}$. Moreover, for $p_{l,i}$ we take a label (l, i) into L and $((l, i), p_{l,i})$ into f_L.

We now sketch how the transitions from a node in g labeled by l with the assigned set of rules P_l can be simulated:

For each rule $p_{l,i}$ in P_l, $1 \leq i \leq n_l$, $(l, Y, k) \in E$ and $p_{k,j} \in P_k$, we take $((l, i), (k, j))$ into A.

If no rule in P_l can be applied, a trickier construction is needed: as long as we assume that at some moment when going through the control graph a rule will be applicable, we guess in which node k this will happen as well as a path $h_0 = l - h_1 - \cdots - h_n = k$ in g following only N-edges from node l to node k which does not contain a loop. For any such path we introduce a label $(\bar{l}, h_1, \ldots, (k, j))$ in L and $(\bar{l}, h_1, \ldots, (k, j)) : {p_{k,j}}^+$ in f_L. Moreover, we use the following activations in A:

- $((\bar{l}, h_1, \ldots, (k, j)), \{q^- \mid q \in \bigcup_{0 \leq i \leq k-1} P_{h_i}, 1)$ is used to check in the next step that no rule along the path from node l to node k is applicable, whereas in the second next step only the designated rule $p_{k,j}$ can be applied, i.e., we take
- $((\bar{l}, h_1, \ldots, (k, j)), p_{k,j}, 2)$ into A.

What remains to be settled is how a derivation in the A-grammar starts:

As $w \notin O_T$, at least one rule must be applied to obtain a terminal object; hence, we check all possibilities that a rule in an initial node in H_i or along a path in g following only N-edges from such an initial node can be applied; for each such rule $p_{k,j}$ in node k we introduce an initial label $\overline{(k, j)}$ in L and also take it into L_0 as well as $\overline{(k, j)} : {p_{k,j}}^+$ into f_L which allows for starting with $p_{k,j}$ using the activation $(\overline{(k, j)}, (k, j))$ in A. As by construction $p_{k,j}$ is applicable it is guaranteed that any continuation of the computation will follow a Y-edge in g and thus the simulation in G_A will also follow the simple simulation of an applicable rule and its continuation with a direct activation of rule in a set assigned to a node directly reachable from node k by a Y-edge.

In total, the construction given above guarantees that the simulation of a computation in G_{GC} by a computation in G_A starts correctly and continues until no rule can be applied any more. As we have assumed all nodes in g to be final and X to be a strictly extended type, i.e., no rules can be applied to a terminal object any more, the only condition to get a result is to obtain a terminal object at the end of a computation. This observation completes our proof. □

As programmed grammars are just a special case of graph-controlled grammars with all labels being final, we immediately infer the following result:

Corollary 2. *For any strictly extended type X with unit rules and trap rules,*

$$\mathcal{L}\left(X\text{-}P_{ac}\right) \subseteq \mathcal{L}\left(X\text{-}A\right).$$

4 Special Results for Specific Objects

In this section we show computational completeness results for AB-grammars based on corresponding well-known computational completeness for other control mechanisms.

4.1 Special Results for Arrays

In both the one- and the two-dimensional case, it has been shown, see [4], that even matrix grammars without ac are sufficient to generate any recursively enumerable array language, i.e., for $d \in \{1,2\}$, $\mathcal{L}\left(d\text{-}\#\text{-}CFA\text{-}MAT\right) = \mathcal{L}\left(d\text{-}ARBA\right)$ (the main reason for such a result is the "#-sensing" ability of the rules of type d-#-CFA). Based on Theorem 3, we immediately infer the following result:

Theorem 7. *For $d \in \{1,2\}$,*

$$\mathcal{L}\left(d\text{-}\#\text{-}CFA\text{-}A1\right) = \mathcal{L}\left(d\text{-}\#\text{-}CFA\text{-}MAT\right) = \mathcal{L}\left(d\text{-}ARBA\right).$$

For arbitrary dimensions $d \in \mathbb{N}$, we have (see [4])

$$\mathcal{L}\left(d\text{-}\#\text{-}CFA\text{-}O\right) = \mathcal{L}\left(d\text{-}ARBA\right).$$

Hence, based on Corollary 2 and Theorem 1 we obtain the following result:

Theorem 8. *For any $d \in \mathbb{N}$ and for any control mechanism Y,*
$Y \in \left\{O, fC, RC, MAT_{ac}, P_{ac}, GC_{ac}^{allfinal}, GC_{ac}, A, AB\right\}$,

$$\mathcal{L}\left(d\text{-}ARBA\right) = \mathcal{L}\left(d\text{-}\#\text{-}CFA\text{-}Y\right).$$

4.2 Special Results for Strings

It is well-known, for example see [2], that $\mathcal{L}(CF\text{-}RC) = \mathcal{L}(ARB)$. Based on Theorem 4, we immediately infer the following computational completeness result:

Theorem 9. $\mathcal{L}(CF\text{-}AB) = \mathcal{L}(CF\text{-}RC) = \mathcal{L}(ARB) = RE$.

Based on Corollary 2, we even obtain the following stronger result:

Theorem 10. $\mathcal{L}(CF\text{-}A) = \mathcal{L}(CF\text{-}P_{ac}) = \mathcal{L}(CF\text{-}GC_{ac}) = \mathcal{L}(CF\text{-}RC) = RE$.

4.3 Special Results for Multisets

As in the case of multisets the structural information contained in the sequence of symbols cannot be used, arbitrary multiset rules are not sufficient for obtaining all sets in $Ps(\mathcal{L}(ARB))$. Yet we can show that even with A-grammars we obtain the following:

Theorem 11. $PsRE = Ps(\mathcal{L}(ARB)) = \mathcal{L}(mARB\text{-}A)$.

Proof. It is folklore, for example see [8] and [6], that

$$PsRE = Ps(\mathcal{L}(ARB)) = \mathcal{L}(mARB\text{-}fC) = \mathcal{L}(mARB\text{-}RC),$$

hence, by Theorem 4, we also obtain $PsRE = \mathcal{L}(mARB\text{-}AB)$. Based on Corollary 2, we even obtain $PsRE = \mathcal{L}(mARB\text{-}P_{ac}) = \mathcal{L}(mARB\text{-}A)$. $\square$

5 Computational Completeness for Context-Free AB-Grammars with Two Non-Terminal Symbols

In this section, we state our main results for context-free string and multiset grammars showing that computational completeness can already be obtained with two non-terminal symbols, which result is optimal with respect to the number of non-terminal symbols.

Theorem 12. *Any recursively enumerable set of strings can be generated by a context-free AB-grammar using only two non-terminal symbols.*

Proof. (Sketch) The main technical details of how to use only two non-terminal symbols A and B for generating a given recursively enumerable language follow the construction given in [6] for graph-controlled grammars. The most important to be shown here is how to simulate the ADD- and SUB-instructions of a deterministic register machine with the contents of the two working registers being given by the number of symbols A and B; only at the end, both numbers are zero, whereas in between, during the whole computation, at least one symbol A or B is present.

The initial string is A, and one A is also the last symbol to be erased at the end in order to obtain a terminal string.

In the following, we use X to specify one of the two non-terminal symbols A and B, and Y then stands for the other one. For any label p of the register machine we use two labels p and p'. The simulations in the AB-grammar work as follows:

- $p : (ADD(X), q)$ is simulated by $p : X \to XX$ and $p' : Y \to YX$ with $(p, p', 1)_B$ as well as $(p, q, 2)_A$, $(p, q', 3)_A$, and $(p', q, 1)_A$, $(p', q', 2)_A$;
- $p : (SUB(X), q, s)$ is simulated by $p : X \to \lambda$ and $p' : Y \to Y$ with $(p, p', 1)_B$ as well as $(p, q, 2)_A$, $(p, q', 3)_A$, and $(p', s, 1)_A$, $(p', s', 2)_A$;

in both cases, the application of the rule labeled by p blocks the rule labeled by p'; in any case, for the next rule labeled r to be simulated, both r and r' are activated, again r' following r one step later.

For the halting label h, only the labeled rule $h : A \to \lambda$ is to be activated. □

This result is optimal with respect to the number of non-terminal symbols: as it has been shown in [3], even for graph-controlled context-free grammars one non-terminal symbol is not enough, hence, the statement immediately follows from Theorem 5.

We now show a similar result for multset grammars.

Theorem 13. *Any recursively enumerable set of multisets can be generated by an AB-grammar using context-free multiset rules and only two non-terminal symbols.*

Proof. Given a recursively enumerable set of multisets L over the terminal alphabet $T = \{a_1, \dots, a_k\}$, we can construct a register machine M_L generating L in the following way: instead of speaking of a number n in register r we use the notation ${a_r}^n$, i.e., a configuration of M_L is represented as a string over the alphabet $V = T \cup \{a_{k+1}, a_{k+2}\}$ with the two non-terminal symbols a_{k+1}, a_{k+2}.

We start with one a_{k+1} and first generate an arbitrary multiset over T step by step adding one element a_m from T and at the same time multiply the number of symbols a_{k+1} by p_m, where p_m is the m-th prime number. At the end of this procedure, for the multiset ${a_1}^{n_1} \dots {a_k}^{n_k}$ we have obtained ${a_m}^{n_m}$ in each register m, $1 \le m \le k$, and ${a_{k+1}}^{{p_1}^{n_1} \dots {p_k}^{n_k}}$ in register $k+1$. As for example, already shown in [9], only using registers $k+1$ and $k+2$, a deterministic register machine M_L' simulating any number of registers by this prime number encoding can compute starting with ${a_{k+1}}^{{p_1}^{n_1} \dots {p_k}^{n_k}}$ and halt if and only if ${a_1}^{n_1} \dots {a_k}^{n_k} \in L$. Only with halting, all registers of M_L' are cleared to zero, i.e., we end up with only one a_{k+1} in M_L when this deterministic register machine M_L' has reached its halting label h. So the last step of M_L before halting is just to eliminate this last a_{k+1}. During the whole computation of M_L, the sum of symbols a_{k+1} and a_{k+2} is greater than zero. Hence, it only remains to show how to simulate the instructions of a register machine, which is done in a similar way as in the preceding proof; we use X to specify one of the two non-terminal symbols a_{k+1} and a_{k+2}, and Y then stands for the other one, i.e., $X, Y \in \{a_{k+1}, a_{k+2}\}$. For any label p of the register machine we use two labels p and p'. The simulations in the AB-grammar work as follows:

- a non-deterministic ADD-instruction $p : (ADD(X), q, s)$ is simulated by branching into two deterministic ADD-instructions even twice:
  $p : X \to X$ and $p' : Y \to Y$ with $(p, p', 1)_B$ as well as
  $(p, (p, X, q), 2)_A$, $(p, (p, X, s), 2)_A$, and $(p', (p, Y, q), 1)_A$, $(p', (p, Y, s), 1)_A$;

  in the third step of the simulation, we already know whether X is present or else we have to use Y; this now allows us to simulate the four deterministic ADD-instructions $(p, \alpha, \beta) : (ADD(X), \beta)$, $\alpha \in \{X, Y\}$, $\beta \in \{q, s\}$, in a simpler way by using the rules
  $(p, \alpha, \beta) : \alpha \to \alpha X$
  and the activations
  $((p, \alpha, \beta), \beta, 1)_A$, $((p, \alpha, \beta), \beta', 2)_A$;
- $p : (ADD(X), q)$ is simulated by $p : X \to XX$ and $p' : Y \to YX$ with $(p, p', 1)_B$ as well as $(p, q, 2)_A$, $(p, q', 3)_A$, and $(p', q, 1)_A$, $(p', q', 2)_A$;
- $p : (SUB(X), q, s)$ is simulated by $p : X \to \lambda$ and $p' : Y \to Y$ with $(p, p', 1)_B$ as well as $(p, q, 2)_A$, $(p, q', 3)_A$, and $(p', s, 1)_A$, $(p', s', 2)_A$;

  in both cases, the application of the rule labeled by p blocks the rule labeled by p'; in any case, for the next rule labeled r to be simulated, both r and r' are activated, again r' following r one step later;
- for the halting label h, only the labeled rule $h : a_{r+1} \to \lambda$ is to be activated.

When the final rule $h : a_{r+1} \to \lambda$ is applied, no further rule is activated, thus the derivation ends yielding the multiset ${a_1}^{n_1} \dots {a_k}^{n_k} \in L$ as terminal result. $\square$

6 Conclusion

We have considered the concept of regulating the applicability of rules based on the application of rules in the preceding step(s) within a very general model for sequential grammars and compared the resulting computational power in relation to various other control mechanisms based on the applicability of rules in the underlying grammar, especially for graph-controlled and matrix grammars as well as random context grammars. Even only using the structural features of the sequences of applied rules, yet not taking into account the features of the underlying objects (e.g., strings, multisets, arrays), general simulation results are obtained. Then we also established some special computational completeness results for string, array, and multiset grammars only using activation of rules. Using both activation and blocking of rules in the case of string and multiset grammars with context-free rules, computational completeness can already be obtained with only two non-terminal symbols, which is a sharp result, as only one non-terminal symbol is not sufficient.

The concept of activation and blocking of rules can also be used when rules are applied in parallel, which is an attractive idea for the area of P systems where several variants of parallel derivation modes are common.

References

1. Cavaliere, M., Freund, R., Oswald, M., Sburlan, D.: Multiset random context grammars, checkers, and transducers. Theoretical Computer Science **372**(2–3), 136–151 (2007)
2. Dassow, J., Păun, Gh.: Regulated Rewriting in Formal Language Theory. Springer-Verlag, Berlin (1989)
3. Fernau, H., Freund, R., Oswald, M., Reinhardt, K.: Refining the nonterminal complexity of graph-controlled, programmed, and matrix grammars. Journal of Automata, Languages and Combinatorics **12**(1–2), 117–138 (2007)
4. Freund, R.: Control mechanisms on #-context-free array grammars. In: Păun, Gh. (ed.) Mathematical Aspects of Natural and Formal Languages, pp. 97–137. World Scientific Publ., Singapore (1994)
5. Freund, R.: Control mechanisms for array grammars on cayley grids. In: Durand-Lose, J., Verlan, S. (eds.) Proceedings of MCU 2018. Lecture Notes in Computer Science, Springer (2018)
6. Freund, R., Kogler, M., Oswald, M.: A general framework for regulated rewriting based on the applicability of rules. In: Kelemen, J., Kelemenová, A. (eds.) Computation, Cooperation, and Life - Essays Dedicated to Gheorghe Păun on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 6610, pp. 35–53. Springer (2011)
7. Freund, R., Oswald, M.: Modelling grammar systems by tissue P systems working in the sequential mode. Fundamenta Informaticae **76**(3), 305–323 (2007)
8. Kudlek, M., Martín-Vide, C., Păun, Gh.: Toward a formal macroset theory. In: Calude, C.S., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) Multiset Processing – Mathematical, Computer Science and Molecular Computing Points of View, Lecture Notes in Computer Science, vol. 2235, pp. 123–134. Springer-Verlag, Berlin (2001)
9. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1967)
10. Păun, Gh., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)
11. Rosenfeld, A.: Picture Languages. AcademicPress, Reading, MA (1979)
12. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, 3 volumes. Springer, New York, NY, USA (1997)
13. Wang, P.S.P. (ed.): Array Grammars, Patterns and Recognizers, World Scientific Series in Computer Science, vol. 18. World Scientific Publ., Singapore (1989)
14. The P Systems Website. `http://ppage.psystems.eu/`

Input-Driven Tissue P Automata

Artiom Alhazov[1], Rudolf Freund[2], Sergiu Ivanov[3],
Marion Oswald[2], and Sergey Verlan[4]

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

[2] Faculty of Informatics, TU Wien
Favoritenstraße 9–11, 1040 Vienna, Austria
{rudi,marion}@emcc.at

[3] IBISC, Université Évry, Université Paris-Saclay
23 Boulevard de France, 91025, Évry, France
sergiu.ivanov@univ-evry.fr

[4] Laboratoire d'Algorithmique, Complexité et Logique,
Université Paris Est Créteil,
61 Avenue du Général de Gaulle, 94010 Créteil, France
verlan@u-pec.fr

Summary. We introduce several variants of input-driven tissue P automata where the rules to be applied only depend on the input symbol. Both strings and multisets are considered as input objects; the strings are either read from an input tape or defined by the sequence of symbols taken in, and the multisets are given in an input cell at the beginning of a computation, enclosed in a vesicle. Additional symbols generated during a computation are stored in this vesicle, too. An input is accepted when the vesicle reaches a final cell and it is empty. The computational power of some variants of input-driven tissue P automata is illustrated by examples and compared with the power of the input-driven variants of other automata as register machines and counter automata.

1 Introduction

In the basic model of membrane systems as introduced at the end of the last century by Gheorghe Păun, e.g., see [9] and [30], the membranes are organized in a hierarchical membrane structure (i.e., the connection structure between the compartments/regions within the membranes being representable as a tree), and the multisets of objects in the membrane regions evolve in a maximally parallel way, with the resulting objects also being able to pass through the surrounding membrane to the parent membrane region or to enter an inner membrane. Many variants of membrane systems, for obvious reasons mostly called *P systems*, have been investigated during nearly two decades, most of them being computationally complete, i.e., being able to simulate the computations of register machines. If an arbitrary graph is used as the connection structure between the cells/membranes, the systems are called *tissue P systems*, see [21].

Instead of multisets of plain symbols coming from a finite alphabet, P systems quite often operate on more complex objects (e.g., strings, arrays), too. A comprehensive overview of different variants of (tissue) P systems and their expressive power is given in the handbook which appeared in 2010, see [31]. For a short view on the state of the art on the domain, we refer the reader to the P systems website [34] as well as to the Bulletin series of the International Membrane Computing Society [33].

The notion and concept of input-driven push-down automata goes back to the seminal paper [22] as well as the papers [6] and [10] improving the complexity measures shown in [22]. The main idea of input-driven push-down automata is that the input letters uniquely determine whether the automaton pushes a symbol, pops a symbol, or leaves the pushdown unchanged. Input-driven push-down automata have been rediscovered at the beginning of this century under the name of visibly pushdown automata, see [3] and [4]. Since then, variants of input-driven push-down automata have gained growing interest, especially because closure properties and decidable questions of the language classes defined by these devices turn out to be similar to those of regular languages. Several new variants of input-driven automata have been developed, for example, using stacks or queues, see [5], [19], and [20]. For complexity issues of input-driven push-down automata, the reader is referred to [24, 25, 26, 27].

The so-called *point mutations*, i.e., *insertion*, *deletion*, and *substitution*, which mean inserting or deleting one symbol or replacing one symbol by another one in a string or multiset are very simple biologically motivated operations. For example, on strings graph-controlled insertion-deletion systems have been investigated in [13], and P systems using these operations at the left or right end of string objects were introduced in [16], where also a short history of using these point mutations in formal language theory can be found.

The operations of insertion and deletion in multisets show a close relation with the increment and decrement instructions in register machines. The power of changing states in connection with the increment and decrement instructions then can be mimicked by moving the whole multiset representing the configuration of a register machine from one cell to another one in the corresponding tissue system after the application of an insertion or deletion rule. Yet usually moving the whole multiset of objects in a cell to another one, besides maximal parallelism, requires *target agreement* between all applied rules, i.e., that all results are moved to the same target cell, e.g., see [15].

A different approach has been introduced in [2]: in order to guarantee that the whole multiset is moved even if only one point mutation is applied, the multiset is enclosed in a vesicle, and this vesicle is moved from one cell to another one as a whole, no matter if a rule has been applied or not. Requiring that one rule has to be applied in every derivation step, a characterization of the family of sets of (vectors of) natural numbers defined by partially blind register machines, which itself corresponds with the family of sets of (vectors of) natural numbers obtained as number (Parikh) sets of string languages generated by graph-controlled or matrix grammars without appearance checking, is obtained.

The idea of using vesicles of multisets has already been used in variants of P systems using the operations drip and mate, corresponding with the operations cut and paste well-known from the area of DNA computing, see [14]. Yet in that case, always two vesicles (one of them possibly an axiom available in an unbounded number) have to interact. In the model as introduced in [2] and also to be adapted in this paper, the rules are always applied to the same vesicle. The *point mutations*, i.e., *insertion*, *deletion*, and *substitution*, well-known from biology as operations on DNA, have also widely been used in the variants of *networks of evolutionary processors (NEPs)*, which consist of cells (processors) each of them allowing for specific operations on strings, and in each derivation step, after the application of a rule, allow the resulting string to be sent to another cell provided specific conditions (for example, random context output and input filters). A short overview on NEPs is given in [2], too.

In this paper, we now introduce input-driven tissue P automata where the rules to be applied only depend on the input symbol. Taking strings as input objects, these are either read from an input tape or defined by the sequence of symbols taken in, and as a kind of additional storage we use a multiset of different symbols enclosed in a vesicle which moves from one cell of the tissue P system to another one depending on the input symbol; the input symbol at the same time also determines whether (one or more) symbols are added to the multiset in the vesicle or removed from there. The given input is accepted if the whole input has been read and the vesicle has reached a final cell and is empty at this moment. When using multisets as input objects, these are enclosed in the vesicle in the input cell at the beginning of a computation, which vesicle then will also carry the additional symbols. The given input multiset is accepted if no input symbols are present any more and the vesicle has reached a final cell and is empty at this moment.

As rules operating on the multiset enclosed in the vesicle when reading/consuming an input symbol we use insertion, deletion, and substitution of multisets, applied in the sequential derivation mode. As restricted variants, we consider systems without allowing substitution of multisets and systems only allowing symbols to be inserted or deleted (or substituted) as it is common when using point mutation rules.

Multiset automata have already been considered in [7], where models for finite automata, linear bounded automata, and Turing machines working on multisets are discussed. When dealing with multisets only, the tissue P automata considered in this paper can be seen as one of the variants of multiset pushdown automata as investigated in [18], where no checking for the emptiness of the multiset *memory* during the computation is possible. Various lemmas proved there then can immediately be adapted for our model. Moreover, also the input-driven variants can be defined in a similar manner, although input-driven multiset pushdown automata have not yet been considered in that paper.

We should also like to mention that the control given by the underlying communication structure of the tissue P system could also be interpreted as having a P system with only one membrane but using states instead. For a discussion on how to use and interpret features of (tissue) P systems as states we refer to [1], where also an example only using the point mutation rules insertion and deletion is given. Moreover, we will also consider another alternative model very common in the P systems area, i.e., P systems with antiport and symport rules, which were introduced in [29]; for an overview, we refer to [31], Chapter 5. One-membrane P systems using antiport rules in a sequential manner and with specific restrictions on the rules then are an adequate model for (input-driven) P automata, yet the restrictions are less visible than in the model of input-driven tissue P automata. On the other hand, when dealing with strings instead of multisets, the way how to read or define the input string in P systems with antiport rules has already been investigated thoroughly, e.g., see [8], [28], and [11] for an overview.

The rest of the paper now is structured as follows: In Section 2 we recall some well-known definitions from formal language theory. The main definitions for the model of (input-driven) tissue P automata as well as its variants to be considered in this paper are given in Section 3, and there we also present the definition of the alternative model of (input-driven) one-membrane P automata with (restricted) antiport rules; moreover we also give some first examples and results. Further illustrative examples and some more results, especially for input-driven tissue P automata are exhibited in Section 4. As upper bound for the family of sets of vectors of natural numbers accepted by input-driven tissue P automata we get the family of sets of vectors of natural numbers generated by partially blind register machines, and as upper bound for the family of sets of strings accepted by input-driven tissue P automata we get the family of sets of strings accepted by partially blind counter automata. A summary of the results obtained in this paper and an outlook to future research are presented in Section 5.

2 Prerequisites

We start by recalling some basic notions of formal language theory. An alphabet is a non-empty finite set. A finite sequence of symbols from an alphabet V is called a *string* over V. The set of all strings over V is denoted by V^*; the *empty string* is denoted by λ; moreover, we define $V^+ = V^* \setminus \{\lambda\}$. The *length* of a string x is denoted by $|x|$, and by $|x|_a$ we denote the number of occurrences of a letter a in a string x.

A *multiset* M with underlying set A is a pair (A, f) where $f : A \to \mathbb{N}$ is a mapping, with $\mathbb{N}$ denoting the set of natural numbers (non-negative integers). If $M = (A, f)$ is a multiset then its *support* is defined as $supp(M) = \{x \in A \mid f(x) > 0\}$. A multiset is empty (respectively finite) if its support is the empty set (respectively a finite set). If $M = (A, f)$ is a finite multiset over A and $supp(M) = \{a_1, \ldots, a_k\}$, then it can also be represented by the string $a_1^{f(a_1)} \ldots a_k^{f(a_k)}$ over the alphabet $\{a_1, \ldots, a_k\}$ (the corresponding vector $(f(a_1), \ldots, f(a_k))$ of natural numbers is called Parikh vector of the string $a_1^{f(a_1)} \ldots a_k^{f(a_k)}$), and, moreover, all permutations of this string precisely identify the same multiset M (they have the same Parikh vector). The set of all multisets over the alphabet V is denoted by V°.

The family of all recursively enumerable sets of strings is denoted by RE, the corresponding family of recursively enumerable sets of Parikh vectors is denoted by $PsRE$. For more details of formal language theory the reader is referred to the monographs and handbooks in this area, such as [32].

2.1 Insertion, Deletion, and Substitution

For an alphabet V, let $a \to b$ be a rewriting rule with $a, b \in V \cup \{\lambda\}$, and $ab \neq \lambda$; we call such a rule a *substitution rule* if both a and b are different from λ and we also write $S(a, b)$; such a rule is called a *deletion rule* if $a \neq \lambda$ and $b = \lambda$, and it is also written as $D(a)$; $a \to b$ is called an *insertion rule* if $a = \lambda$ and $b \neq \lambda$, and we also write $I(b)$. The sets of all insertion rules, deletion rules, and substitution rules over an alphabet V are denoted by Ins_V, Del_V, and Sub_V, respectively. Whereas an insertion rule is always applicable, the applicability of a deletion and a substitution rule depends on the presence of the symbol a. We remark that insertion rules, deletion rules, and substitution rules can be applied to strings as well as to multisets. Whereas in the string case, the position of the inserted, deleted, and substituted symbol matters, in the case of a multiset this only means incrementing the number of symbols b, decrementing the number of symbols a, or decrementing the number of symbols a and at the same time incrementing the number of symbols b.

These types of rules and the corresponding notations can be extended by allowing more than one symbol on the left-hand and/or the right-hand side, i.e., $a, b \in V^*$, and $ab \neq \lambda$. The corresponding sets of all extended insertion rules, deletion rules, and substitution rules over an alphabet V are denoted by Ins_V^*, Del_V^*, and Sub_V^*, respectively.

2.2 Register Machines

Register machines are well-known universal devices for computing (generating or accepting) sets of vectors of natural numbers.

Definition 1. *A* register machine *is a construct* $M = (m, B, I, h, P)$ *where*

- m *is the number of registers,*

- *B is a set of labels bijectively labeling the instructions in the set P,*
- *$I \subseteq B$ is the set of initial labels, and*
- *$h \in B$ is the final label.*

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

- *$p : (ADD\,(r)\,, K)$, with $p \in B \setminus \{l_h\}$, $K \subseteq B$, $1 \leq r \leq m$.*
  Increase the value of register r by one, and non-deterministically jump to one of the instructions in K.
- *$p : (SUB\,(r)\,, K, F)$, with $p \in B \setminus \{l_h\}$, $K, F \subseteq B$, $1 \leq r \leq m$.*
  *If the value of register r is not zero then decrease the value of register r by one (*decrement *case) and jump to one of the instructions in K, otherwise jump to one of the instructions in F (*zero-test *case).*
- *$h : HALT$.*
  Stop the execution of the register machine.

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

In the accepting case, a computation starts with the input of a k-vector of natural numbers in its first k registers and by executing one of the initial instructions of P (labeled with $l \in I$); it terminates with reaching the $HALT$-instruction. Without loss of generality, we may assume all registers to be empty at the end of the computation.

By $\mathcal{L}(RM)$ we denote the family of sets of vectors of natural numbers accepted by register machines. It is folklore (e.g., see [23]) that $PsRE = \mathcal{L}(RM)$.

Partially blind register machines

In the case when a register machine cannot check whether a register is empty we say that it is partially blind: the registers are increased and decreased by one as usual, but if the machine tries to subtract from an empty register, then the computation aborts without producing any result (that is we may say that the subtract instructions are of the form $p : (SUB\,(r)\,, K, abort)$; instead, we simply will write $p : (SUB\,(r)\,, K)$.

Moreover, acceptance now by definition also requires all registers to be empty at the end of the computation, i.e., there is an implicit test for zero at the end of a (successful) computation, that is why we say that the device is partially blind. By $\mathcal{L}(PBRM)$ we denote the family of sets of vectors of natural numbers accepted by partially blind register machines. It is known (e.g., see [12]) that partially blind register machines are strictly less powerful than general register machines (hence, than Turing machines); moreover, $\mathcal{L}(PBRM)$ characterizes the Parikh sets of languages generated by graph-controlled or matrix grammars without appearance checking.

2.3 Counter Automata

Register machines can also be equipped with an input tape to be able to process strings, and the registers then are only used as auxiliary storage. We then call the registers *counters* and the automaton a *counter automaton* (we mention that in the literature slightly different definitions with respect to the instructions may be found). The additional instruction needed then is a *read instruction* reading one symbol from the input tape:

$$p : (read(a), K), \text{ with } p \in B \setminus \{h\},\ K \subseteq B, \text{ and } a \in T.$$

T is the input alphabet, i.e., in sum we obtain a counter automaton as a construct

$$M = (m, B, I, h, P, T).$$

A counter automaton accepts an input $w \in T^*$ if and only if it starts in some initial state and with w on its input tape, and finally M reaches h having read the whole input string w. Without loss of generality, we again may assume all registers to be empty at the end of the computation.

It is folklore (e.g., see [23]) that the family of string languages accepted by counter automata equals RE (in fact, only two counters are needed).

Partially blind counter automata

As in the case of register machines, a counter automaton is called partially blind if it cannot check whether a register is empty, and acceptance by definition requires the whole input to be read and all counters to be empty at the end of the computation. For basic results on partially blind counter automata we refer to the seminal paper [17]. The family of string languages accepted by partially blind counter automata is denoted by $\mathcal{L}(PBCA)$.

2.4 Input-Driven Register Machines and Counter Automata

An input-driven register machine / counter automaton (an $IDRM^*$ and $IDCA^*$, respectively, for short) can be defined in the following way: any decrement of an input register r / any reading of a terminal symbol a is followed by *fixed* sequences of instructions on the working registers / counters only depending on the input register r / the terminal symbol a. If each such sequence is of length exactly one, then we speak of a real-time input-driven register machine / counter automaton (an $IDRM$ and $IDCA$, respectively, for short).

In the case of an $IDCA$, these sequences are of the form

$$p : (read(a), K) \to q : (\alpha(r), K_q),\ \ q \in K,$$

with $\alpha \in \{ADD, SUB\}$, $1 \leq r \leq m$, and they could be written as *one* extended instruction

$$p : (read(a), \alpha(r), \textstyle\bigcup_{q \in K} K_q).$$

In a similar way, for an $IDCA^*$ we replace $\alpha(r)$ by the whole sequence of instructions following the reading of the input symbol a. A similar notation can be adapted for the case of a SUB-instruction on an input register instead of $read(a)$. Moreover, analogous definitions and notations hold for the partially blind variants of input-driven register machines / counter automata.

Remark 1. We emphasize that we have chosen a very restricted variant of what it means that the actions on the working registers only depend on the input symbol just read: no matter which label the read instruction $read(a)$ has, it must always be followed by the same sequence $\alpha(r)$; only the branching to labels from $\bigcup_{q\in K} K_q)$ allows for taking different actions afterwards. $\square$

Remark 2. Allowing a set of initial labels as well as sets of labels in the ADD- and SUB-instructions may look quite unusual, but especially for the input-driven automata this feature turns out to be essential:

Assume we had allowed only one initial label i in any input-driven counter automaton. Now consider the finite multiset language $\{a,b\}$: assume there is an input-driven partially blind counter automaton accepting $\{a,b\}$. By definition, the instruction assigned to the initial label i must be a read instruction. With the initial label i, only *one* of the read instructions $read(a)$ or $read(b)$ can be assigned, hence, only a or only b can be accepted, a contradiction.

A similar argument holds for partially blind register machines taking the input set of two-dimensional vectors $\{(1,0),(0,1)\}$: the instruction assigned to i must be a SUB-instruction either on register 1 or on register 2, again leading to a contradiction.

On the other hand, with our more general definition, we get closure under union for free for $\mathcal{L}(X)$, $X \in \{IDRM, IDCA, IDRM^*, IDCA^*\}$. $\square$

3 Tissue P Automata as Multiset Pushdown Automata

We now define a model of a tissue P automaton and its input-driven variants, first for the case of working with multisets as input objects:

Definition 2. *A* tissue P automaton *(a* tPA* *for short) is a tuple*

$$\Pi = (L, V, \Sigma, \Gamma, R, g, I, F)$$

where

- *L is a* set of labels *identifying in a one-to-one manner the $|L|$ cells of the tissue P system Π;*
- *V is the* alphabet *of the system;*
- *$\Sigma \subseteq V$ is the (non-empty)* input alphabet *of the system;*
- *$\Gamma \subseteq V$ is the (possibly empty)* memory alphabet *of the system, $\Gamma \cap \Sigma = \emptyset$;*

- *R is a* set of rules *of the form (i,p) where $i \in L$ and $p \in Ins_V^* \cup Del_V^* \cup Sub_V^*$, i.e., p is an extended insertion, deletion or substitution rule over the alphabet V; we may collect all rules from cell i in one set and then write $R_i = \{(i,p) \mid (i,p) \in R\}$, so that $R = \bigcup_{i \in L} R_i$; moreover, for the sake of conciseness, we may simply write $R_i = \{p \mid (i,p) \in R\}$, too;*
- *g is a directed graph describing the underlying communication structure of Π, $g = (N,E)$ with $N = L$ being the set of nodes of the graph g and the set of edges $E \subseteq L \times L$;*
- *$I \subseteq L$ is the set of labels of* initial cells *one of them containing the input multiset w at the beginning of a computation;*
- *$f \subseteq L$ is the set of labels of* final cells *for acceptance.*

If in the definition above we take $p \in Ins_V \cup Del_V \cup Sub_V$ instead of $p \in Ins_V^* \cup Del_V^* \cup Sub_V^*$, then we speak of a tPA instead of a tPA^*.

A tPA^* Π now works as follows: The computation of Π starts with a vesicle containing the input multiset w in one of the initial cells $i \in I$, and the computation proceeds with derivation steps until a specific output condition is fulfilled.

In each derivation step, with the vesicle enclosing the multiset w being in cell k, one rule from R_k is applied to w and the resulting multiset in its vesicle is moved to a cell m such that $(k,m) \in E$.

As we are dealing with membrane systems, the classic output condition is to only consider halting computations; yet in case of automata, the standard acceptance condition is reaching a final state, which in our case means reaching a final cell h, and, moreover, the vesicle to be empty. We will take these two conditions as our mode of acceptance in this paper, as with the vesicle being empty no decrement rule can be applied any more and, moreover, it is guaranteed that we have "read the whole input". Only requiring the vesicle to be empty or else requiring to have reached a final cell with the vesicle containing no input symbol any more, are two other variants of acceptance.

The set of multisets accepted by Π is denoted by $Ps_{acc}(\Pi)$. The families of sets of vectors of natural numbers accepted by tPA^* and tPA with at most n cells are denoted by $\mathcal{L}_n(tPA^*)$ and $\mathcal{L}_n(tPA)$, respectively. If n is not bounded, we simply omit the subscript in these notations. In order to specify which rules are allowed in the tPA^* and tPA, we may explicitly specify I^*, D^*, S^* and I, D, S, respectively, to indicate the use of (extended) insertion, deletion, and substitution rules. For example, $\mathcal{L}(tPA, ID)$ then indicates that only insertion and deletion rules are used.

Remark 3. The model of a tPA^* comes very close to the model of a multiset pushdown automaton as introduced in [18]; in fact, the family of sets of vectors of natural numbers accepted by these multiset pushdown automata equals $\mathcal{L}(tPA^*)$. A formal proof would go far beyond the scope of this paper, but the basic similarity of these two models becomes obvious when identifying the cells in the tPA^* with the states in the multiset pushdown automaton; moving the vesicle from one cell to another one corresponds to changing the states. As shown for the states of the multiset pushdown automata in [18], we could also restrict ourselves to only one initial as well as only one final cell in the general case, as this does not restrict the computational power of a tPA^*. On the other hand, for any of the following restricted variants this need not be true any more, especially for the input-driven variants defined later; in this context we also remind the arguments given in Remark 2. $\square$

The following result shows that having more than one rule in a cell is not necessary:

Lemma 1. *For any tPA^* Π there exists an equivalent tPA^* Π' such that every cell contains at most one rule.*

Proof. (*Sketch*) Let $\Pi = (L, V, \Sigma, \Gamma, R, g, I, F)$ be a tPA^*. The equivalent tPA^* $\Pi' = (L', V, \Sigma, \Gamma, R', g', I', F')$ then is constructed as follows:

For every cell k with R_k containing n_k rules, instead of cell k we take n_k copies of that cell, cells $(k, 1), \dots, (k, n_k)$, into Π', each of it containing one of the rules from R_k, say $p_{k,l}$, $1 \leq l \leq n_k$. The connection graph g then has to be enlarged to a graph g' containing all the edges

$$\{((k,l),(j,m)) \mid (k,j) \in g, 1 \leq l \leq n_k, 1 \leq m \leq n_j\}.$$

If cell k contains no rule, we rename it to cell $(k, 1)$, and no rule is contained in this cell, too.

The new sets of labels of initial and final cells are obtained by taking all copies of the original cell labels, i.e., we take

$$I' = \{(k,l) \mid (k \in I, 1 \leq l \leq n_k\},$$
$$F' = \{(k,l) \mid (k \in F, 1 \leq l \leq n_k\}.$$

We now immediately infer $Ps(\Pi) = Ps(\Pi')$. $\square$

Remark 4. It is easy to avoid having more than one final cell: based one the preceding proof, we introduce a new final cell f', i.e., we take $F' = \{f'\}$, with this new cell not containing any rule; moreover, we add all edges

$$\{((k,l), f') \mid ((k,l),(j,m)) \in g', j \in F\}.$$

This new cell corresponds to the label of the final HALT instruction in a register machine or a counter automaton. $\square$

Remark 5. Having only one initial cell cannot be shown by only using a new structure: we may add a new single initial cell i' containing a substitution rule $S(a, a)$ for some $a \in V$, and add all edges

$$\{(i', (k,l)) \mid (k,l) \in I'\}.$$

If we want to avoid substitution rules, we may add two new cells containing the rules $I(a)$ and $D(a)$, respectively, use the first one as the only new initial cell only having an arc to the second one from where to branch as described above.

Continuing the discussions from Remark 2 and Remark 3 we mention that both constructions are not feasable for the input-driven variants to be defined in Subsection 3.2. $\square$

The following result is based on the fact that the insertion, deletion, or substitution of a multiset over V can easily be simulated by a sequence of insertions and deletions:

Lemma 2. *For any tPA^* Π there exists an equivalent tPA Π' even not using substitution rules.*

Proof. Let $\Pi = (L, V, \Sigma, \Gamma, R, g, I, F)$ be a tPA^*. According to Lemma 2, without loss of generality, we may assume every cell to contain only one rule. The equivalent tPA Π' then is constructed as follows:

Let $u \to v$ be a substitution rule with $u = u_1 \dots u_k$ and $v = v_1 \dots v_m$. Then the following sequence of deletion and insertion rules has the same effect as $u \to v$:

$$D(u_1) \to \dots D(u_k) \to I(v_1) \to \dots I(v_m)$$

Taking cells for each of these rules and the corresponding connections into Π', it is easy to see that following this path in Π' has the same effect as the application of the original rule in Π. Similar arguments hold true if $u = \lambda$ or $v = \lambda$, too, i.e., in case of an insertion or a deletion rule, respectively. In sum, we conclude $Ps(\Pi) = Ps(\Pi')$. $\square$

Now let $\mathcal{L}(mARB)$ denote the family of sets of multisets generated by arbitrary multiset grammars.

Corollary 1. *$\mathcal{L}(tPA^*, IDS) = \mathcal{L}(tPA, ID) = \mathcal{L}(mARB) = \mathcal{L}(PBRM)$.*

Proof. (*Sketch*) The equality $\mathcal{L}(tPA^*, IDS) = \mathcal{L}(tPA, ID)$ follows from the definitions and Lemma 2.

The equality $\mathcal{L}(tPA^*, IDS) = \mathcal{L}(mARB)$ is a consequence of the observation discussed above in Remark 3 that $\mathcal{L}(tPA^*, IDS)$ corresponds to the family of sets of multisets accepted by multiset pushdwon automata as defined in [18]. In a similar way, interpreting the cells in a tissue P automaton as the states of a partially blind register machine and seeing the correspondence of the acceptance conditions, we also infer the equality $\mathcal{L}(tPA^*, IDS) = \mathcal{L}(PBRM)$. The details are left to the reader. $\square$

3.1 Accepting Strings

The tissue P automata defined above can also be used to accept sets of strings by assuming the input string to be given on a separate input tape, from where the symbols of the input string are read from left to right. As when going from register machines to counter automata, we use the additional instruction (*read instruction*) $read(a)$ with $a \in \Sigma$, Σ being the input alphabet. The corresponding automata then are defined as follows:

Definition 3. *A* tissue P automaton for strings *(a* tPAL* *for short) is a tuple*

$$\Pi = (L, V, \Sigma, \Gamma, R, g, I, F)$$

where L, V, Σ, Γ, R, g, I, F are defined as for a tPA^*, except that besides insertion, deletion, and substitution rules we also allow rules of the form $read(a)$ with $a \in T$, i.e., read instructions.

If we only take rules from $Ins_V \cup Del_V \cup Sub_V$ instead of $Ins_V^* \cup Del_V^* \cup Sub_V^*$, then we speak of a $tPAL$ instead of a $tPAL^*$.

A $tPAL^*$ Π now works as follows: The computation of Π starts with the input string on the input tape as well as an empty vesicle in one of the initial cells $i \in I$, and the computation proceeds with derivation steps until the whole input string has been read and the vesicle has reached a final cell, again being empty at the end of the computation.

In each derivation step, with the vesicle enclosing the multiset w being in cell k, one rule from R_k is applied, either reading a symbol from the input tape or affecting w, and the resulting multiset in its vesicle then is moved to a cell m such that $(k, m) \in E$.

The set of strings accepted by Π is denoted by $L(\Pi)$. The families of sets of strings accepted by $tPAL^*$ and $tPAL$ with at most n cells are denoted by $\mathcal{L}_n(tPAL^*)$ and $\mathcal{L}_n(tPAL)$, respectively. If n is not bounded, we simply omit the subscript in these notations. In order to specify which rules are allowed in the tPA^* and tPA, we again may explicitly specify I^*, D^*, S^* and I, D, S, respectively, to indicate the use of (extended) insertion, deletion, and substitution rules.

As for tissue P automata accepting multisets, also for the ones accepting strings we obtain some similar results as shown above:

Lemma 3. *For any $tPAL^*$ Π there exists an equivalent $tPAL^*$ Π' such that every cell contains at most one rule.*

Lemma 4. *For any $tPAL^*$ Π there exists an equivalent $tPAL$ Π' even not using substitution rules.*

Corollary 2. $\mathcal{L}(tPAL^*, IDS) = \mathcal{L}(tPAL, ID) = \mathcal{L}(PBCA)$.

3.2 Input-Driven Tissue P Automata

We now define the input-driven variants of tPA^* and tPA as well as $tPAL^*$ and $tPAL$:

Definition 4. *A* tPA^* $\Pi = (L, V, \Sigma, \Gamma, R, g, I, F)$ *is called* input-driven *(and called a* tIDPA* *for short) if the following conditions hold true:*

- *to each cell, (at most) one rule is assigned;*
- *any decrement of an input register* r *is followed by some* fixed *sequence of instructions on the working registers only depending on the input register* r *before a cell with the next decrement instruction on an input register is reached. Such a sequence of instructions may even be of length zero.*

If each such sequence is of length exactly one, then we speak of a real-time input-driven $tIDPAL^*$ *(a* $tIDPArt^*$ *for short).*

Definition 5. *A* $tPAL^*$ $\Pi = (L, V, \Sigma, \Gamma, R, g, I, F)$ *is called* input-driven *(and called a* tIDPAL* *for short) if the following conditions hold true:*

- *to each cell, (at most) one rule is assigned;*
- *any reading of a terminal symbol* a *by a read instruction* $read(a)$ *is followed by some* fixed *sequence of instructions on the working registers only depending on the the terminal symbol* a *before a cell with the next read instruction is reached. Such a sequence of instructions may even be of length zero.*

If each such sequence is of length exactly one, then we speak of a real-time input-driven $tPAL^*$ *(a* $tPALrt^*$ *for short).*

The corresponding families of sets of vectors of natural numbers and of sets of strings accepted by tissue P automata of type X with X being one of the types $tIDPA^*$, $tIDPA$, $tIDPA^*rt$, $tIDPArt$ as well as $tIDPAL^*$, $tIDPAL$, $tIDPAL^*rt$, $tIDPALrt$, are denoted by $\mathcal{L}(X)$.

Remark 6. As already discussed in Remark 1 for input-driven register machines and counter automata, we emphasize that we have chosen a very restricted variant of what it means that the actions on the multiset in the vesicle only depend on the input symbol just read: no matter in which cell we have the read instruction $read(a)$, it must always be followed by the same finite sequence of instructions not including read instructions. $\square$

Remark 7. If we only have SUB-instructions on input registers / read instructions, i.e., if the tPA^* / $tPAL^*$ does not use the vesicle at all for storing any intermediate information, then such a tPA^* / $tPAL^*$ can be interpreted as a finite automaton accepting a regular multiset / string language. In this case, the condition of not having rules on the vesicle already subsumes the condition of the automaton being input-driven. $\square$

3.3 One-Membrane Antiport P Automata

The idea of using states instead of cells can also be "implemented" by using a well-investigated model of membrane systems using antiport rules:

Definition 6. *A* one-membrane antiport P automaton *(a* 1APA* *for short) is a tuple* $\Pi = (V, \Sigma, \Gamma, Q, R, I, F)$ *where*

- V *is the* alphabet *of the system;*
- $\Sigma \subseteq V$ *is the (non-empty)* input alphabet *of the system;*
- $\Gamma \subseteq V$ *is the (possibly empty)* memory alphabet *of the system,* $\Gamma \cap \Sigma = \emptyset$*;*
- $Q \subseteq V$*,* $Q \cap (\Gamma \cup \Sigma) = \emptyset$*, is the set of* states*;*
- R *is a* set of rules *of the form* $pu \to qv$*,* $p, q \in Q$*,* $u \in (\Gamma \cup \Sigma)^*$*,* $v \in \Sigma^*$*;*
- $I \subseteq Q$ *is the set of* initial states*;*
- $F \subseteq Q$ *is the set of* final states.

The $1APA^*$ can be seen as a membrane system consisting of only one membrane with the rules $pu \to qv$ interpreted as antiport rules $(pu, out; qv, in)$, i.e., the multiset pu leaves the membrane region and the multiset qv enters the membrane region.

Π starts with an input multiset w_0 together with one of the initial states p_0, i.e., with $w_0 p_0$ in its single membrane region, and then applies rules from R until a configuration with only $p_f \in F$ in the membrane region is reached, thus accepting the input multiset w_0.

For antiport P automata the acceptance of strings can be defined without needing an input tape as follows, e.g., see [28]: the rules in R now are of the form $pu \to qv$, $p, q \in Q$, $u \in \Gamma^*$ and $v \in (\Gamma \cup \Sigma)^*$, i.e., the input symbols are now taken from outside the membrane (from the environment); the sequence how the input symbols are taken in defines the input string (we may assume v to contain only one symbol from Σ; otherwise, we have to take any permutation of the symbols taken in in one step for defining several input strings).

Using such rules and the interpretation of the input string as defined above, we obtain the model of a *one-membrane antiport P automaton for strings* (a $1APAL^*$ for short).

As in the preceding subsections we now can define specific variants of $1APA^*$ and $1APAL^*$, e.g., the corresponding input-driven automata. Yet as we have introduced these models especially to show the correspondence with an automaton model well-known in the area of P systems, we leave the technical details to the interested reader.

4 Examples and Results

The concepts of $tIDPA^*$ and $IDPBRM^*$ are closely related:

Theorem 1. $\mathcal{L}(tIDPA^*) \subseteq \mathcal{L}(PBRM^*)$ *and*
$\mathcal{L}(tIDPA^*) = \mathcal{L}(tIDPA^*, ID) = \mathcal{L}(IDPBRM^*)$.

Proof. (*Sketch*) The inclusion $\mathcal{L}(IDPBRM^*) \subseteq \mathcal{L}(PBRM^*)$ is obvious from the definitions.

The equality $\mathcal{L}(tIDPA, ID^*) = \mathcal{L}(IDPBRM^*)$ follows from the definitions of these types of input-driven automata: as already mentioned earlier, the cells in a tPA^* correspond to the states in a $PBRM$. The acceptance conditions – the vesicle being empty in a final cell in a tPA^* and all registers being empty in a $PBRM$ when reaching the final label – directly correspond to each other, too. Moreover, insertion and deletion rules directly correspond to ADD- and SUB-instructions. Finally, the conditions for the input-driven variants requiring the same actions for a consumed input symbol and the decrement of the corresponding register are equivalent, too.

The equality $\mathcal{L}(tIDPA^*) = \mathcal{L}(tIDPA^*, ID)$ follows from the possibility to simulate substitution rules by a sequence of insertion and deletion rules. This observation completes the proof. □

Using similar arguments as in the preceding proof, now considering read instructions instead of decrements on input registers, we obtain the corresponding result for the string case:

Theorem 2. $\mathcal{L}(tIDPAL^*) \subseteq \mathcal{L}(PBCA^*)$ *and*
$\mathcal{L}(tIDPAL^*) = \mathcal{L}(tIDPAL^*, ID) = \mathcal{L}(IDPBCA^*)$.

In the real-time variants, we cannot use substitution rules in the input-driven tissue P automata, as the simulation by deletion and insertion rules takes more than one step:

Theorem 3. $\mathcal{L}(tIDPArt, ID) = \mathcal{L}(IDPBRMrt)$ *and*
$\mathcal{L}(tIDPALrt, ID) = \mathcal{L}(IDPBCArt)$.

We now illustrate the computational power of input-driven tissue P automata accepting strings by showing how well-known string languages can be accepted. We remark that in all cases the automaton has only one initial label and one final label.

Example 1. The Dyck language L_D over the alphabet of brackets $\{\,[\,,\,]\,\}$ can easily be accepted by the $tIDPBCArt$ M_D:

$$\begin{aligned} M_D &= (1, B = \{1,2,3,4,5\}, l_0 = 1, l_h = 5, P, T = \{\,[\,,\,]\,\})\,, \\ P &= \{1 : (read\,(\,[\,)\,, \{2\})\,, 2 : (ADD\,(1)\,, \{1,3\})\,, \\ &\qquad 3 : (read\,(\,]\,)\,, \{4\})\,, 4 : (SUB\,(1)\,, \{1,3,5\})\; 5 : HALT)\}. \end{aligned}$$

L_D can also be accepted by the corresponding $tIDPALrt$ Π_D:

$$\begin{aligned}
\mathit{\Pi}_D &= (L = \{1,2,3,4,5\}, V, \mathit{\Sigma}, \mathit{\Gamma}, R, g = (L,E), I = \{1\}, F = \{5\})\,,\\
V &= \{a_1,\, [\,,\,]\,\},\\
\mathit{\Sigma} &= \{\, [\,,\,]\,\},\\
\mathit{\Gamma} &= \{a_1\},\\
R &= \{(1, \mathit{read}\,(\,[\,)), (2, I\,(a_1)), (3, \mathit{read}\,(\,]\,)), (4, D\,(a_1))\},\\
E &= \{(1,2),(2,1),(2,3),(3,4),(4,1),(4,3),(4,5)\}.
\end{aligned}$$

The two constructions elaborated above implement the following definition of a well-formed bracket expression w over the alphabet of brackets $\{\,[\,,\,]\,\}$:

- for every prefix of w, the number of closing brackets $]$ must not exceed the number of opening brackets $[\,$;
- the number of closing brackets $]$ in w equals the number of opening brackets $[\,$.

Hence, during the whole computation, the (non-negative) difference between the number of opening and the number of closing brackets is stored as the number of symbols a_1; at the end, this number must be zero, which is guaranteed by the acceptance conditions. $\square$

$\mathcal{L}(IDPBCArt)$ even contains a non-context-free language:

Example 2. The language $L_{il} = \{a^n b^m c^n d^m \mid m, n \geq 1\}$ is not context-free, but accepted by the following $tIDPALrt\ \mathit{\Pi}_{il}$:

$$\begin{aligned}
\mathit{\Pi}_{il} &= (L = \{1,\ldots,9\}, V, \mathit{\Sigma}, \mathit{\Gamma}, R, g = (L,E), I = \{1\}, F = \{9\})\,,\\
V &= \{a_1, a_2, a, b, c, d\},\\
\mathit{\Sigma} &= \{a, b, c, d\},\\
\mathit{\Gamma} &= \{a_1, a_2\},\\
R &= \{(1, \mathit{read}\,(\,a\,)), (2, I\,(a_1)), (3, \mathit{read}\,(\,b\,)), (4, I\,(a_2)),\\
&\qquad (5, \mathit{read}\,(\,c\,)), (6, D\,(a_1)), (7, \mathit{read}\,(\,d\,)), (8, D\,(a_2))\},\\
E &= \{(1,2),(2,1),(2,3),(3,4),(4,3),\\
&\qquad (4,5),(5,6),(6,5),(6,7),(7,8),(8,7),(8,9)\}.
\end{aligned}$$

By this construction, we conclude $L_{il} \in \mathcal{L}(tIDPALrt, ID)$. $\square$

For the language considered in the next example we show that it is in $\mathcal{L}(tIDPAL^{*}rt)$, but we claim that it is not in $\mathcal{L}(tIDPALrt)$:

Example 3. Let $k > 2$ and consider the string language $L_k = \{{b_1}^n \ldots {b_k}^n \mid n \geq 1\}$, which is not context-free, but accepted by the following $tIDPAL^{*}rt\ \mathit{\Pi}$:

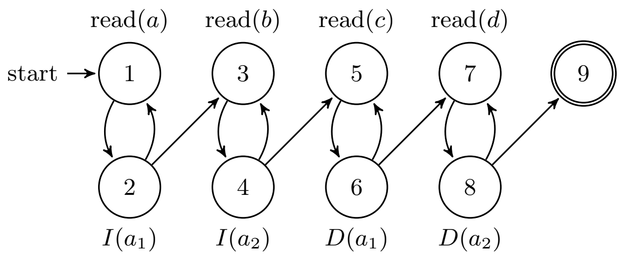


Fig. 1. Graphic representation of the *tIDPALrt* Π_{il}.

$$\begin{aligned}
\Pi_k &= (L = \{1, \ldots, 2k+1\}, V, \Sigma, \Gamma, R, g = (L,E), I = \{1\}, F = \{2k+1\}),\\
V &= \{a_i, b_i \mid 1 \le i \le k\},\\
\Sigma &= \{b_i \mid 1 \le i \le k\},\\
\Gamma &= \{a_i \mid 1 \le i \le k\},\\
R &= \{(1, read\,(\,b_1\,)), (2, I\,(a_2 \ldots a_k))\}\\
&\quad \cup \{(2j-1, read\,(\,b_j\,)), (2j, D\,(a_j)) \mid 1 < j \le k\},\\
E &= \{(2j-1, 2j), (2j, 2j-1), (2j, 2j+1) \mid 1 \le j \le k\}.
\end{aligned}$$

Without proof we claim that $L_k \notin \mathcal{L}(tIDPALrt)$. □

5 Conclusion and Future Research

In this paper, we have introduced tissue P automata as a specific model of multiset automata as well as input-driven tissue P automata where the rules to be applied depend on the input symbol. Taking strings as input objects, these are either read from an input tape or defined by the sequence of symbols taken in, and as an additional storage of a multiset of different symbols we use a vesicle which moves from one cell of the tissue P system to another one depending on the input symbol; the input symbol at the same time determines whether (one or more) symbols are added to the multiset in the vesicle or removed from there and where the vesicle moves afterwards. The given input is accepted if the whole input has been read and the vesicle has reached a final cell and/or is empty at this moment. When using multisets as input objects, these are enclosed in the vesicle in the input cell at the beginning of a computation, which vesicle then will also take the additional symbols. The given input multiset is accepted if no input symbols are present any more and the vesicle has reached a final cell and is empty at this moment.

As rules operating on the multiset enclosed in the vesicle when reading/consuming an input symbol we have used insertion, deletion, and substitution of multisets, working in the sequential derivation mode. As restricted variants, we have considered systems without allowing substitution of multisets and systems only allowing symbols to be inserted or deleted (or substituted).

We have shown how input-driven tissue P automata with multisets and strings can be characterized by input-driven register machines and input-driven counter automata, respectively. Moreover, we have exhibited some illustrative examples, for example, how the Dyck language or even some non-contextfree languages can be accepted by simple variants of input-driven tissue P automata.

Several challenging topics remain for future research: for example, a characterization of the language classes accepted by several variants of tissue P automata accepting multisets or strings, especially for the input-driven variants, introduced in this paper is still open.

As acceptance condition we have only considered reaching the final cell h with an empty vesicle. The other variants of acceptance, i.e., only requiring the vesicle to be empty or else requiring to have reached the final cell with the vesicle containing no input symbol any more, are to be investigated in the future in more detail.

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., Ivanov, S., Verlan, S.: (tissue) P systems with vesicles of multisets. In: Csuhaj-Varjú, E., Dömösi, P., Vaszil, Gy. (eds.) Proceedings 15th International Conference on Automata and Formal Languages, AFL 2017, Debrecen, Hungary, September 4-6, 2017. EPTCS, vol. 252, pp. 11–25 (2017). https://doi.org/10.4204/EPTCS.252.6
3. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Babai, L. (ed.) Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004. pp. 202–211. ACM (2004). https://doi.org/10.1145/1007352.1007390
4. Alur, R., Madhusudan, P.: Adding nesting structure to words. J. ACM **56**(3), 16:1–16:43 (2009). https://doi.org/10.1145/1516512.1516518
5. Bensch, S., Holzer, M., Kutrib, M., Malcher, A.: Input-driven stack automata. In: Baeten, J.C.M., Ball, T., de Boer, F.S. (eds.) Theoretical Computer Science - 7th IFIP TC 1/WG 2.2 International Conference, TCS 2012, Amsterdam, The Netherlands, September 26-28, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7604, pp. 28–42. Springer (2012). https://doi.org/10.1007/978-3-642-33475-7_3
6. von Braunmühl, B., Verbeek, R.: Input-driven languages are recognized in log n space. In: Karpinski, M. (ed.) Foundations of Computation Theory. pp. 40–51. Springer, Berlin, Heidelberg (1983)
7. Csuhaj-Varjú, E., Martín-Vide, C., Mitrana, V.: Multiset automata. In: Calude, C.S., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) Multiset Processing. pp. 69–83. Springer, Berlin, Heidelberg (2001)

8. Csuhaj-Varjú, E., Vaszil, Gy.: P automata or purely communicating accepting p systems. In: Păun, Gh., Rozenberg, G., Salomaa, A., Zandron, C. (eds.) Membrane Computing. pp. 219–233. Springer, Berlin, Heidelberg (2003)
9. Dassow, J., Păun, Gh.: On the power of membrane computing. J. UCS **5**(2), 33–49 (1999). https://doi.org/10.3217/jucs-005-02-0033
10. Dymond, P.W.: Input-driven languages are in log n depth. Information Processing Letters **26**(5), 247–250 (1988). https://doi.org/10.1016/0020-0190(88)90148-2
11. Freund, R.: P automata: New ideas and results. In: Bordihn, H., Freund, R., Nagy, B., Vaszil, Gy. (eds.) Eighth Workshop on Non-Classical Models of Automata and Applications, NCMA 2016, Debrecen, Hungary, August 29-30, 2016. Proceedings. books@ocg.at, vol. 321, pp. 13–40. Österreichische Computer Gesellschaft (2016)
12. Freund, R., Ibarra, O., Păun, Gh., Yen, H.C.: Matrix languages, register machines, vector addition systems. Third Brainstorming Week on Membrane Computing pp. 155–167 (2005)
13. Freund, R., Kogler, M., Rogozhin, Yu., Verlan, S.: Graph-controlled insertion-deletion systems. In: Proceedings Twelfth Annual Workshop on Descriptional Complexity of Formal Systems, DCFS 2010, Saskatoon, Canada, 8-10th August 2010. pp. 88–98 (2010). https://doi.org/10.4204/EPTCS.31.11
14. Freund, R., Oswald, M.: Tissue P systems and (mem)brane systems with mate and drip operations working on strings. Electr. Notes Theor. Comput. Sci. **171**(2), 105–115 (2007). https://doi.org/10.1016/j.entcs.2007.05.011
15. Freund, R., Păun, Gh.: How to obtain computational completeness in P systems with one catalyst. In: Proceedings Machines, Computations and Universality 2013, MCU 2013, Zürich, Switzerland, September 9-11, 2013. pp. 47–61 (2013). https://doi.org/10.4204/EPTCS.128.13
16. Freund, R., Rogozhin, Yu., Verlan, S.: Generating and accepting P systems with minimal left and right insertion and deletion. Natural Computing **13**(2), 257–268 (2014). https://doi.org/10.1007/s11047-013-9396-3
17. Greibach, S.A.: Remarks on blind and partially blind one-way multicounter machines. Theoretical Computer Science **7**, 311–324 (1978). https://doi.org/10.1016/0304-3975(78)90020-8
18. Kudlek, M., Totzke, P., Zetzsche, G.: Multiset pushdown automata. Fundam. Inform. **93**(1-3), 221–233 (2009). https://doi.org/10.3233/FI-2009-0098
19. Kutrib, M., Malcher, A., Wendlandt, M.: Tinput-driven pushdown, counter, and stack automata. Fundamenta Informaticae **155**(1-2), 59–88 (2017). https://doi.org/10.3233/FI-2017-1576
20. Kutrib, M., Malcher, A., Wendlandt, M.: Queue automata: Foundations andÂădevelopments. In: Adamatzky, A. (ed.) Reversibility and Universality: Essays Presented to Kenichi Morita on the Occasion of his 70th Birthday. pp. 385–431. Springer (2018). https://doi.org/10.1007/978-3-319-73216-9_19
21. Martín-Vide, C., Pazos, J., Păun, Gh., Rodríguez-Patón, A.: A new class of symbolic abstract neural nets: Tissue P systems. In: Computing and Combinatorics, pp. 290–299. Springer (2002). https://doi.org/10.1007/3-540-45655-4_32
22. Mehlhorn, K.: Pebbling mountain ranges and its application to dcfl-recognition. In: de Bakker, J., van Leeuwen, J. (eds.) Automata, Languages and Programming. pp. 422–435. Springer, Berlin, Heidelberg (1980)
23. Minsky, M.L.: Computation. Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, NJ (1967)

24. Okhotin, A., Salomaa, K.: Input-driven pushdown automata: nondeterminism and unambiguity. 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. 31–33. Österreichische Computer Gesellschaft (2013)
25. Okhotin, A., Salomaa, K.: Input-driven pushdown automata with limited nondeterminism - (invited paper). In: Shur, A.M., Volkov, M.V. (eds.) Developments in Language Theory - 18th International Conference, DLT 2014, Ekaterinburg, Russia, August 26-29, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8633, pp. 84–102. Springer (2014). https://doi.org/10.1007/978-3-319-09698-8_9
26. Okhotin, A., Salomaa, K.: Descriptional complexity of unambiguous input-driven pushdown automata. Theoretical Computer Science **566**, 1–11 (2015). https://doi.org/10.1016/j.tcs.2014.11.015
27. Okhotin, A., Salomaa, K.: State complexity of operations on input-driven pushdown automata. J. Comput. Syst. Sci. **86**, 207–228 (2017). https://doi.org/10.1016/j.jcss.2017.02.001
28. Oswald, M.: P Automata. Ph.D. thesis, Faculty of Computer Science, TU Wien (2003)
29. Paun, A., Păun, Gh.: The power of communication: P systems with symport/antiport. New Generation Comput. **20**(3), 295–306 (2002). https://doi.org/10.1007/BF03037362
30. Păun, Gh.: Computing with Membranes. Journal of Computer and System Sciences **61**(1), 108–143 (2000). https://doi.org/10.1006/jcss.1999.1693
31. Păun, Gh., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, Oxford, England (2010)
32. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, vol. 1-3. Springer (1997)
33. Bulletin of the International Membrane Computing Society (IMCS). `http://membranecomputing.net/IMCSBulletin/index.php`
34. The P Systems Website. `http://ppage.psystems.eu/`

A Note on a New Class of APCol Systems

Lucie Ciencialová[1], Erzsébet Csuhaj-Varjú[2], and György Vaszil[3]

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

Summary. We introduce a new acceptance mode for APCol systems (Automaton-like P colonies), variants of P colonies where the environment of the agents is given by a string and during functioning the agents change their own states and process the string similarly to automata. In case of the standard variant, the string is accepted if it can be reduced to the empty word. In this paper, we define APCol systems where the agents verify their environment, a model resembling multihead finite automata. In this case, a string of length n is accepted if during every halting computation the length of the environmental string in the configurations does not change and in the course of the computation every agent applies a rule to a symbol on position i of some of the environmental strings for every i, $1 \leq i \leq n$ at least once. We show that these verifying APCol systems simulate one-way multihead finite automata.

1 Introduction

Automaton-like P colonies (APCol systems, for short), introduced in [1], are variants of of P colonies (introduced in [9]) - very simple membrane systems inspired by colonies of formal grammars. The interested reader is referred to [12] for detailed information on P systems (membrane systems) and to [10] and [5] for more information to grammar systems theory; for more details on P colonies consult [8] and [4].

An APCol system consists of a finite number of agents - finite collections of objects in a cell - and a 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 environmental string 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. The reader may easily observe that APCol systems resembling automata as well, the current configuration of the system (the objects inside the agents) and the current environmental string correspond to the current state of an automaton and the currently processed input string.

The agents may perform rewriting, communication or checking rules [9]. A rewriting rule $a \to b$ allows the agent to rewrite (evolve) one object a to object b. 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 this paper, we define a new variant of APCol systems, a model resembling multihead finite automata, where the agents verify their environment. In this case, a string of length n is accepted if during every halting computation the length of the environmental string in the configurations does not change and in the course of the computation every agent applies a rule to a symbol on position i of some of the environmental strings for every i, $1 \leq i \leq n$ at least once. We show that these verifying APCol systems simulate one-way multihead finite automata.

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 [13, 12].

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 $|w|_a$.

For a string $x \in \Sigma^*$, $x[i]$ denotes the symbol at ith position of x, i.e., if $x = x_1 \dots x_n$, $x_i \in \Sigma$, then $x[i] = x_i$. For every string $x \in \Sigma^*$, $x[0] = \varepsilon$.

For every string $x \in \Sigma^*$, $perm(x)$ denotes the set of all permutations of x and $pref(x)$ denotes the set of prefixes of x.

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 One-way Multihead Finite Automata

We recall some basic notions concerning multi-head finite automata based on [7]. A non-deterministic one-way k-head finite automaton (a 1NFA(k), for short) is a construct $M = (Q, \Sigma, k, \delta, \triangleright, \triangleleft, q_0, F)$, where Q is the finite set of states, Σ is the set of input symbols, $k \geq 1$ is the number of heads, $\triangleright \notin \Sigma$ and $\triangleleft \notin \Sigma$ are the left and the right endmarkers, respectively, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states, and δ is the partial transition function which maps $Q \times (\Sigma \cup \{\triangleright, \triangleleft\})^k$ into subsets of $Q \times \{0, 1\}^k$, where 1 means that the head moves one tape cell to the right and 0 means that it remains at the same position. We note that the heads can never move to the left of the left endmarker and to the right of the right endmarker.

A configuration of a 1NFA(k) $M = (Q, \Sigma, k, \delta, \triangleright, \triangleleft, q_0, F)$ is a triplet $c = (w, q, p)$, where $w \in \Sigma^*$ is the input, $q \in Q$ is the current state, and $p = (p_1, \dots, p_k) \in \{0, 1, \dots, |w| + 1\}^k$ gives the head positions. If a position p_i is 0, then head i is scanning the symbol $\triangleright$, if $1 \leq p_i \leq |w|$, then head i scans the p_ith letter of w, and if $p_i = |w| + 1$, then the ith head is scanning the symbol $\triangleleft$.

The initial configuration for an input $w \in \Sigma^*$ is $(w, q_0, (1, \dots, 1))$, that is, a 1NFA(k) starts processing a nonempty input word with all of its heads positioned on the first symbol of w.

In the course of the computation, M performs direct changes of its configurations. Let $w = a_1 \dots a_n$, be the input, $a_0 = \triangleright$, $a_{n+1} = \triangleleft$. For two configurations, $c_1 = (w, q, (p_1, \dots, p_k))$ and $c_2 = (w, q', (p'_1, \dots, p'_k))$, we say that c_2 directly follows c_1, denoted by $c_1 \vdash c_2$, if $(q', (d_1, \dots, d_k)) \in \delta(q, (a_{p_1}, \dots, a_{p_k}))$ and $p'_i = p_i + d_i$, $1 \leq i \leq k$. The reflexive transitive closure of $\vdash$ is denoted by $\vdash^*$. Note that due to the restriction of the transition function, the heads cannot move beyond the endmarkers.

The language $L(M)$ accepted by a 1NFA(k) $M = (Q, \Sigma, k, \delta, \triangleright, \triangleleft, q_0, F)$ is the set of words w such that there is a computation which starts with $\triangleright w \triangleleft$ on the input tape and ends when M reaches an accepting state, i.e.,

$$L(M) = \{w \in \Sigma^* \mid (w, q_0, (1, \dots, 1)) \vdash^* (w, q_f, (p_1, \dots, p_k)),\ q_f \in F\}.$$

The class of languages accepted by 1NFA(k), for $k \geq 1$, is denoted by $\mathcal{L}$(1NFA(k))

According to the definition of a 1NFA(k) $M = (Q, \Sigma, k, \delta, \triangleright, \triangleleft, q_0, F)$, the heads do not need to move away from the scanned tape cell after reading the input symbol. For technical reasons, we use a modified but equally powerful version of this definition in such a way that the automaton reads the input symbols only in the case when the head moves away from the tape cell containing the symbol. Otherwise, it "scans" the empty word ε, that is, the symbol does not have any role in the determination of the next configuration of the machine. For details and the proof of the equivalence the reader is referred to [6].

Thus, we can simplify the notation for the elements of the transition relation of one-way k-head finite automata, since we can assume that if $(q', (d_1, \ldots, d_k)) \in \delta(q, a_1, \ldots, a_k)$, then $d_j = 0$ if and only if $a_j = \varepsilon$, $1 \leq j \leq k$. As $(d_1, \ldots, d_k) \in \{0, 1\}^k$, we can simply denote the above transition as $q' \in \delta(q, a_1, \ldots, a_k)$: If $a_j \neq \varepsilon$ for some $j,\ 1 \leq j \leq k$, then the jth reading head is moved to the right, otherwise, if $a_j = \varepsilon$ it remains in its current position.

2.2 APCol Systems

In the following we recall the notion of an APCol system (an automaton-like P colony) where the environment of the agents is given in the form of a string [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, is of the form $a \rightarrow b$. This means that object a inside of the agent is rewritten to object b. The second type, 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 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.

An APCol system, for 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 one 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 $\langle e \to e; c \leftrightarrow d\rangle$.

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 $c = (\omega; \omega_1, \dots, \omega_n)$ 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 $(w; w_1, \dots, w_n)$, where $|w_i| = 2,\ 1 \le i \le n$, 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 $1 \le i \le n$.

In [1] it was shown that the family of languages accepted by jumping finite automata (introduced in [11]) is properly included in the family of languages accepted by APCol systems with one agent, and it was proved that any recursively enumerable language can be obtained as a projection of a language accepted by an APCol system with two agents.

3 Verifying APCol Systems

In this section we introduce a new variant of acceptance for APCol systems, motivated by the behaviour of multihead finite automata. In case of standard APCol systems acceptance means identifying and erasing symbols of the current environmental string. In case of verifying APCol systems, the agents only indicate that they "visit" a certain position in the current environmental string, i.e., rewrite the symbol at that position to some symbol but not to the empty word. A string is verified if the computation process is halting and for every i, $1 \le i \le n$ - supposed that the length of the input string is n -, each agent rewrites a symbol at position i in some of the environmental strings occurring in the computation process. This means that the agents "visit" each position (of the input string or that of its descendant), i.e., they verify that the environment. To perform a transition, the APCol systems work with the maximally parallel mode.

Definition 1. *Let $\Pi = (O, e, A_1, \ldots, A_n)$, $n \ge 1$, be and APCol system working with the maximally parallel mode. We say that Π verifies input string ω if the following conditions hold.*

- *There exists a halting computation c in Π where* $c = (\omega; \omega_1, \ldots, \omega_n) \implies \left(\omega^{(1)}; \omega_1^{(1)}, \ldots, \omega_n^{(1)}\right) \implies \ldots \left(\omega^{(s)}; w_1^{(s)}, \ldots, w_n^{(s)}\right)$, *$s \ge 1$, such that $|\omega| = |\omega^{(j)}|$ for $1 \le j \le s$.*
- *Computation c satisfies the following property. Let $\omega^{(0)} = \omega$. For every agent A_k, $1 \le k \le n$ and for every i, $1 \le i \le m$ where $|\omega| = m$, there exists j, $1 \le j \le s$ such that the symbol at the ith position of $\omega^{(j-1)}$ is letter b and A_k applies a rule $a \leftrightarrow b$ to this position of $\omega^{(j-1)}$.*
  Computation c is called a verification or a verifying computation.

The set of all words that can be verified by Π is called the language verified by Π. We call a word, resp. a language strongly verified if every computation of the word, resp. of every word in the language is verifying.

In the following we show that verifying APCol systems simulate one-way multihead finite automata.

Theorem 1. *Let $M = (Q, \Sigma, n, \delta, \triangleright, \triangleleft, q_0, F)$, $n \geq 1$, be an n-head finite automaton. Then we can construct an APCol system Π with $n+2$ agents such that any word w that can be accepted by M can strongly be verified by Π.*

Proof. To prove the statement, we construct an APCol system $\Pi = (O, e, A_{ini}, A_1, \dots, A_n, A_{fin})$ such that each agent A_j, $1 \leq j \leq n$ simulates the work of the jth reading head of M and only that. Agent A_{ini} serves for initializing the simulation and agent A_{fin} checks whether the agents visited every position in the environmental string. The verifying process in Π corresponds to an accepting process in M: if a symbol a was scanned by reading heads $\{i_1, \dots, i_r\} \subseteq \{1, \dots, n\}$, then this fact will be indicated by a symbol $a^{(x)}$ in the environmental string of Π, where $x \in perm(i_1 \dots \imath_r)$; $i_1, \dots, i_r$ are the numbers of reading heads that scanned symbol a. Every computation step in M is simulated by a sequence of computation steps performed by agents $A_1, \dots A_n$.

The input word for Π is of the form $\triangleright w$.

To help the easier reading, we will present only the agents together with their programs.

Π has agent A_{ini} for initializing the simulation and also for assisting the checking whether every position has been visited or not.

It has the following programs:

(1) $\langle e \to q_{0,1}, e \to e \rangle$,
(2) $\langle q_{0,1} \leftrightarrow \triangleright, e \to e \rangle$,
(3) $\left\langle e \to e, \bar{a}^{(x)} \to a'^{(x)} \right\rangle$,

where q_0 is the initial state and $x \in perm(1 \dots n)$, $a \in \Sigma$.

Programs (1) and (2) are for initializing the simulation, programs of type (3) are for checking whether or not all positions have been visited.

Every agent A_j simulates the work of the reading head j, $1 \leq j \leq n$.

For every transition relation

$$s \in \delta(q, a_1, \dots, a_n)$$

of M, where $a_1, \dots, a_n \in \Sigma \cup \{\varepsilon\}$, agent A_j, $1 \leq j \leq n-1$ has the following programs.

(We recall that if $a_j \neq \varepsilon$ for some j, $1 \leq j \leq k$, then the jth reading head is moved to the right, otherwise, if $a_j = \varepsilon$ it remains in its current position.)

(0) $\langle e \to \triangleright_j, e \to \triangleright'_j \rangle$,

(1) $\langle \triangleright'_j \leftrightarrow s_j, e \to e \rangle$,
(1a) $\langle \triangleright'_j \leftrightarrow s_j, \triangleright_j \leftrightarrow e \rangle$,
(1b) $\langle \triangleright'_1 \leftrightarrow q_1, e \to e \rangle$,
(1c) $\langle \triangleright'_1 \leftrightarrow q_1, \triangleright_1 \leftrightarrow e \rangle$,

(2) $\left\langle s_j \to s'_j, e \to a_j^{(xj)} \right\rangle$,
(2a) $\left\langle s_j \to s'_j, e \to a_j^{(x)} \right\rangle$,
(2b) $\left\langle q_1 \to s'_j, e \to a_j^{(xj)} \right\rangle$,
(2c) $\left\langle q_1 \to s'_j, e \to a_j^{(x)} \right\rangle$,
(3) $\left\langle a_j^{xj} \leftrightarrow \rhd_j, s'_j \leftrightarrow a_j^{(x)} \right\rangle$,
(3a) $\left\langle s'_j \leftrightarrow \rhd_j, a_j^{x} \leftrightarrow a_j^{(x)} \right\rangle$,

(4) $\left\langle \rhd_j \leftrightarrow s'_j, a_j^{(x)} \to s_{j+1} \right\rangle$,
(4a) $\left\langle \rhd_j \leftrightarrow s'_j, a_j^{(xj)} \to s_{j+1} \right\rangle$,

(5) $\langle s_{j+1} \leftrightarrow \rhd'_j, s'_j \to e \rangle$,

where $x \in pref(perm(1 \dots n))$, $|x|_j = 0$, if $a_j \neq \varepsilon$ and $x \in pref(perm(1 \dots n))$ and $|x|_j = 1$ if $a_j = \varepsilon$. Furthermore, $y \in pref(perm(1 \dots n))$ and $|y|_j = 1$.

For $j = n$ and $s = (q', p_1, \dots, p_n)$, agent A_j has the same programs (0)-(3a), and programs (4),(4a) and (5) are changed as follows:

(5) $\langle \rhd_n \to p'_1, e \to e \rangle$,
(6) $\langle p'_1 \leftrightarrow \rhd_n, e \to e \rangle$,

We present a brief explanation of the programs. The first program, (0) is used for initialization. it generates two symbols of $\rhd_j$ - the first to mark the position of reading head and the second to exchange for symbol of the simulated transition. The simulation of the move of the jth reading head starts with program (1). If it is simulation of the first step and the first head, then program (1c) is used. For the first use of other heads the program (1a) is used. The program (1b) is executed when the system simulates not the first step of computation. The input string has the form $q_j\alpha$, where q is the state in transition relation $s \in \delta(q, a_1, \dots, a_n)$, its subscript j refers to the jth reading head, and $\alpha \in \Sigma^*$ such that $|\alpha| = |w|$ (w is the input word). Then q_j is changed for $\rhd_j$ in the environmental string, implying that no action of some other agent can be performed. Meantime, A_j changes s_j for s'_j and makes a guess whether symbol a_j is to be scanned or the reading head will remain at the same position. This is done by introducing $a_j^{(xj)}$ or $a_j^{(x)}$, programs (2) or (2a). Superscript xj refers to that letter a_j has not been scanned by reading head j (x is the sequence of the number of reading heads that already scanned this symbol). When agent A_1 does the same thing (uses one of programs (2b), (2c)), it also nondeterministically chooses transition s from the possible transitions $\delta(q, a_1, \dots, a_n)$ for given $q \in Q$ and arbitrary sequence of symbols $a_1, \dots, a_n$. After then, programs (3) or (3a) perform the corresponding action, namely change $\rhd_j a_j^{(x)}$ to $a_j^{(xj)} s'_j$ or leave the two letters unchanged. (Notice that the order of rules in this type of programs is important). In the first case we simulate that the jth reading head scanned a_j, in the second case it remained at the same position. Note that if the reading head is in the position of cell with symbol a, the symbol a is not marked as read in this moment. The symbol a is read when head is leaving the cell with this symbol. After then, by programs (4), (5), agent A_j will return to state $(\rhd'_j, e)$ and the simulation of the move of the next reading head in transition s starts, i.e., the environmental string will have s_{j+1} as first letter. If $j = n$, then the first letter in the environmental string is changed to q'_1, meaning that M entered state q' and the simulation of the first reading head starts.

The computation is successful if it is halting and all positions have been visited by each agents. This is checked by agent A_{fin} and then A_{init}. The programs of A_{fin} are as follows.

(0) $\langle e \to e, e \to h \rangle$,
(1) $\langle e \leftrightarrow q_{f,1}, e \to h \rangle$,
(2) $\left\langle h \leftrightarrow c^{(z)}, e \to e \right\rangle$,
(3) $\left\langle c^{(x)} \to \bar{c}^{(x)}, e \to e \right\rangle$,
(4) $\left\langle \bar{c}^{(x)} \leftrightarrow h, e \to e \right\rangle$,
(5) $\left\langle c^{(y)} \leftrightarrow \#, e \to e \right\rangle$,
(6) $\langle h \leftrightarrow \rhd_j, e \to e \rangle$,
(7) $\langle \rhd_j \to e, e \leftrightarrow h \rangle$,
(8) $\langle \# \to \#, e \to e, \rangle$,

where $1 \le j \le n$, $c \in \Sigma$, $z = pref(perm(1 \dots n))$, $x = perm(1 \dots n)$, and $y \ne perm(1 \dots n)$.

Agent A_{fin} nondeterministically consumes all symbols in the environmental string. It checks if each of them has been visited by every agent. This is done by introducing symbol h in the environmental string. Then, programs of the form (3) and (4), indicates that the symbol was visited by all agents. If there is a symbol which does not satisfy this property (program (5)), then symbol # is introduced which implies that the system will never stop (program (8)). Suppose that this is not the case, then after a while A_{fin} will not be able to perform any of its programs. It is easy to see that A_{fin} visited all letters in the environmental word. To complete the proof, A_{init} has to visit all symbols as well. This is done by its programs of type (3). Since A_{fin} visited all symbols, A_{init} will do that too, thus, the computation halts.

The reader my observe that the agents simulate the transitions of M and that no agent can work simultaneously. Furthermore, programs of different agents cannot interfere. Thus, the language accepted by M can be verified by Π. Furthermore, every accepting computation of a word in Π is a verifying computation, thus Π strongly verifies the language accepted by Π.

4 Conclusions

In this paper we demonstrated a further connection between APCol systems and automata, as we proved that verifying APCol systems simulate one-way multihead finite automata. The new concept, the verifying computation opens further research directions: describing two-way multihead finite automata, jumping multihead finite automata in terms of APCol systems. We plan investigations in these topics in the future.

Acknowledgments.

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science - LQ1602, by SGS/13/2016 and by Grant No. 120558 of the National Research, Development, and Innovation Office, Hungary.

References

1. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E.: Towards P Colonies Processing Strings. In: Proc. BWMC 2014, Sevilla, 2014. pp. 103–118. 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. 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)
9. 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)
10. Kelemen, J., Kelemenová, A.: A Grammar-Theoretic Treatment of Multiagent Systems. Cybern. Syst. 23(6), 621–633 (1992),
11. Meduna, A., Zemek, P.: Jumping Finite Automata. Int. J. Found. Comput. Sci. 23(7), 1555–1578 (2012)
12. Păun, Gh., Rozenberg, G., Salomaa, A.(eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)
13. Rozenberg, G., Salomaa, A.(eds.): Handbook of Formal Languages I-III. Springer Verlag., Berin-Heidelberg-New York (1997)

P Colony Automata with LL(k)-like Conditions [⋆]

Erzsébet Csuhaj-Varjú[1], Kristóf Kántor[2], and György Vaszil[2]

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

[2] Department of Computer Science, Faculty of Informatics
University of Debrecen
Kassai út 26, 4028 Debrecen, Hungary
{`kantor.kristof, vaszil.gyorgy`}`@inf.unideb.hu`

Summary. We investigate the possibility of the deterministic parsing (that is, parsing without backtracking) of languages characterized by (generalized) P colony automata. We define a class of P colony automata satisfying a property which resembles the LL(k) property of context-free grammars, and study the possibility of parsing the characterized languages using a k symbol lookahead, as in the LL(k) parsing method for context-free languages.

1 Introduction

The computational model called P colony is similar to tissue-like membrane systems, where multisets of objects are used to describe the contents of the cells and environment and then are processed by the cells in the corresponding colony using rules which enable the evolution of the objects present in the cells and the exchange of objects between the environment and the cells. These computing agents have a very confined functionality: they can store a restricted amount of objects at a given time (this is called the capacity of the system) and they can process a restricted amount of information. The way the information processing is done is really simple: The rules are either of the form $a \to b$ (for changing an object a into an object b inside the cell), or $a \leftrightarrow b$ (for exchanging an object a inside a cell with an object b in the environment). A rule set is called a program, it consists of exactly the same number of rules as the capacity of the system. When a program is executed, the k (the capacity of the system) rules that it contains are applied to

[⋆] Supported in part by project no. K 120558, implemented by the National Research, Development and Innovation Fund of Hungary, financed under the K_16 funding scheme. Also supported by the construction EFOP-3.6.3-VEKOP-16-2017-00002, a project financed by the European Union, co-financed by the European Social Fund.

the k objects simultaneously. During a computational step, every colony member cell execute one of their programs in parallel. A computation ends when the system reaches one of its the final configurations (usually given as the set of halting configurations, that is, those situations when no programs can be applied by any of the cells).

There are many theoretical results concerning P colonies. Despite the fact that they are extremely simple computing systems, they are computationally complete, even with very restricted size parameters and other syntactic or functioning restrictions. For these, and more topics, results, see [5, 6, 4, 3, 8, 9, 11, 12].

P colony automata were introduced in [2]. They are called automata, because they accept string languages by assuming an initial input tape and an input string in the environment. The available types of rules are extended by so called tape rules. These types of rules in addition to manipulating the objects as their non-tape counterparts, also read the processed objects from the input tape.

To overcome the difficulty that different tape rules can read different symbols in the same computational step, generalized P colony automata were introduced in [13] and studied further in [15, 14]. The main idea of this computational model was to get the process of input reading closer to other kinds of membrane systems, especially to antiport P systems and P automata. The latter, introduced in [10] (see also [7]) are P systems using symport and antiport rules (see [16]), characterizing string languages.

This generality is used in the generalized P colony automata theory, that is, the idea of characterizing strings through the sequences of multisets processed during computations. A computation in this model defines accepted multiset sequences, which are transformed into accepted symbol sequences / strings. In this model there is no input string, but there are tape and non-tape rules equally for evolution and communication rules. In a single computational step, this system is able to read more than one symbol, thus reading a multiset. This way generalized P colony automata are able to avoid the conflicts present in P Colony automata, where simultaneous usage of tape rules in a single computational step can arise problems. After getting the result of a computation, that is, the accepted sequence of multisets, it is possible to map them to strings in a similar way as shown in P automata.

In [13], some basic variants of the model were introduced and studied from the point of view of their computational power. In [15, 14] we continued the investigations structuring our results around the capacity of the systems, and different types of restrictions imposed on the use of tape rules in the programs of the systems. In the present paper we study the possibility of deterministically parsing the languages characterized by these devices. We define the so called LL(k) condition for these types of automata, which enables deterministic parsing with a one symbol lookahead, as in the case of context-free LL(k) languages, and present an initial result showing that using P colony automta we can deterministically parse context-free languages that are not LL(k) in the “original” sense.

2 Preliminaries and Definitions

Let V be a finite alphabet, let the set of all words over V be denoted by V^*, and let ε be the empty word. We denote the number of occurrences of a symbol $a \in V$ in w by $|w|_a$.

A multiset over a set V is a mapping $M : V \to \mathbb{N}$ where $\mathbb{N}$ denotes the set of non-negative integers. This mapping assigns to each object $a \in V$ its multiplicity $M(a)$ in M. The set $supp(M) = \{a \mid M(a) \geq 1\}$ is the support of M. If V is a finite set, then M is called a finite multiset. A multiset M is empty if its support is empty, $supp(M) = \emptyset$. The set of finite multisets over the alphabet V is denoted by $\mathcal{M}(V)$. A finite multiset M over V will also be represented by a string w over the alphabet V with $|w|_a = M(a)$, $a \in V$, the empty multiset will be denoted by $\emptyset$.

We say that $a \in M$ if $M(a) \geq 1$, and the cardinality of M, $card(M)$ is defined as $card(M) = \Sigma_{a \in M} M(a)$. For two multisets $M_1, M_2 \in \mathcal{M}(V)$, $M_1 \subseteq M_2$ holds, if for all $a \in V$, $M_1(a) \leq M_2(a)$. The union of M_1 and M_2 is defined as $(M_1 \cup M_2) : V \to \mathbb{N}$ with $(M_1 \cup M_2)(a) = M_1(a) + M_2(a)$ for all $a \in V$, the difference is defined for $M_2 \subseteq M_1$ as $(M_1 - M_2) : V \to \mathbb{N}$ with $(M_1 - M_2)(a) = M_1(a) - M_2(a)$ for all $a \in V$.

A *genPCol automaton* of capacity k and with n cells, $k, n \geq 1$, is a construct

$$\Pi = (V, e, w_E, (w_1, P_1), \ldots, (w_n, P_n), F)$$

where

- V is an *alphabet*, the alphabet of the automaton, its elements are called *objects*;
- $e \in V$ is the *environmental object* of the automaton, the only object which is assumed to be available in an arbitrary, unbounded number of copies in the environment;
- $w_E \in (V - \{e\})^*$ is a string representing a multiset from $\mathcal{M}(V - \{e\})$, the multiset of objects different from e which is found in the environment initially;
- $(w_i, P_i), 1 \leq i \leq n$, specifies the i-th *cell* where w_i is (the representation of) a multiset over V, it determines the initial contents of the cell, and its cardinality $|w_i| = k$ is called the *capacity* of the system. P_i is a set of *programs*, each program is formed from k rules of the following types (where $a, b \in V$):
  - *tape rules* of the form $a \overset{T}{\to} b$, or $a \overset{T}{\leftrightarrow} b$, called rewriting tape rules and communication tape rules, respectively; or
  - *nontape rules* of the form $a \to b$, or $a \leftrightarrow b$, called rewriting (nontape) rules and communication (nontape) rules, respectively.

  A program is called a *tape program* if it contains at least one tape rule.
- F is a set of *accepting configurations* of the automaton which we will specify in more detail below.

A genPCol automaton reads an input word during a computation. A part of the input (possibly consisting of more than one symbols) is read during each configuration change: the processed part of the input corresponds to the multiset of symbols introduced by the tape rules of the system.

A *configuration* of a genPCol automaton is an $(n+1)$-tuple $(u_E, u_1, \ldots, u_n)$, where $u_E \in \mathcal{M}(V - \{e\})$ is the multiset of objects different from e in the environment, and $u_i \in \mathcal{M}(V),\ 1 \leq i \leq n$, are the contents of the i-th cell. The *initial configuration* is given by $(w_E, w_1, \ldots, w_n)$, the initial contents of the environment and the cells. The elements of the set F of *accepting configurations* are given as configurations of the form $(v_E, v_1, \ldots, v_n)$, where

- $v_E \in \mathcal{M}(V - \{e\})$ denotes a multiset of objects different from e being in the environment, and
- $v_i \in \mathcal{M}(V),\ 1 \leq i \leq n$, is the contents of the i-th cell.

In order to describe the functioning of genPCol automata, let us define the following multisets. Let r be a rewriting or a communication rule (tape or nontape), and let us denote by $left(r)$ and $right(r)$ the objects on the left and on the right side of r, respectively.

Let also, for $\alpha \in \{left, right\}$ and for any program p, $\alpha(p) = \bigcup_{r\in p} \alpha(r)$ where the union denotes multiset union (as defined above), and for a rule r and program $p = \langle r_1, \ldots, r_k\rangle$, the notation $r \in p$ denotes the fact that r is one of the rules of the program, that is, $r = r_j$ for some $j,\ 1 \leq j \leq k$.

Moreover, for any tape program p we also define $read(p)$ as the multiset of symbols (different from e) on the right side of rewriting tape rules and on the left side of communication tape rules, that is, $read(p) = \bigcup_{r\in p, r=a\overset{T}{\to}b, b\neq e} right(r) \cup \bigcup_{r\in p, r=a\overset{T}{\leftrightarrow}b, a\neq e} left(r)$. If p is not a tape program, that is, p contains no tape rules, then $read(p) = \emptyset$.

Let us also denote by $export(p)$ and $import(p)$ the multiset of objects that are sent out to the environment and brought inside the cell when applying the program p, respectively, that is, $export(p) = \bigcup_{r\in p} left(r)$, $import(p) = \bigcup_{r\in p} right(r)$ for all communication rules r of the program p. Moreover, by $create(p)$ we denote the multiset of symbols produced by the rewriting rules of program p, thus, $create(p) = \bigcup_{r\in p} right(p)$ for the rewriting rules r of p.

Let $c = (u_E, u_1, \ldots, u_n)$ be a configuration of a genPCol automaton Π, and let $U_E = u_E \cup \{e, e, \ldots\}$, thus, the multiset of objects found in the environment (together with the infinite number of es which are always present). The *set of programs*

$$(p_1, \ldots, p_n) \in (P_1 \cup \{\#\}) \times \ldots \times (P_n \cup \{\#\})$$

is *applicable in configuration* c, if the following conditions hold.

- The selected programs are applicable in the cells (the left sides of the rules contain the same symbols that are present in the cell), that is, for each $1 \leq i \leq n$, if $p_i \in P_i$ then $left(p_i) = u_i$;
- the symbols to be brought inside the cells by the programs are present in the environment, that is, $\bigcup_{p_i\neq\#, 1\leq i\leq n} import(p_i) \subseteq U_E$;

- the set of selected programs is maximal, that is, if any $p_i = \#$ is replaced by some $p'_i \in P_i$, $1 \leq i \leq n$, then the above conditions are not satisfied any more.

Let us denote by App_c be the set of all applicable sets of programs in the configuration $c = (u_E, u_1, \ldots, u_n)$, that is,

$$App_c = \{P_c = (p_1, \ldots, p_n) \in (P_1 \cup \{\#\}) \times \ldots \times (P_n \cup \{\#\}) \mid \text{ where } P_c \\ \text{is a set of applicable programs in the configuration } c\}.$$

Let $c = (u_E, u_1, \ldots, u_n)$ be a configuration of the genPCol automaton. By applying a set of applicable programs $P_c \in App_c$, the configuration c is *changed* to a configuration $c' = (u'_E, u'_1, \ldots, u'_n)$, denoted by $c \overset{P_c}{\Longrightarrow} c'$, if the following properties hold:

- If $(p_1, \ldots, p_n) = P_c$ and $p_i \in P_i$, then $u'_i = create(p_i) \cup import(p_i)$, otherwise, if $p_i = \#$, then $u'_i = u_i$, $1 \leq i \leq n$. Moreover,
- $U'_E = U_E - \bigcup_{p_i \neq \#, 1 \leq i \leq n} import(p_i) \cup \bigcup_{p_i \neq \#, 1 \leq i \leq n} export(p_i)$ (where U'_E again denotes $u'_E \cup \{e, e, \ldots\}$ with an infinite number of es).

Thus, in genPCol automata, we apply the programs in the maximally parallel way, that is, in each computational step, every component cell nondeterministically applies one of its applicable programs. Then we collect all the symbols that the tape rules "read" (these multisets are denoted by $read(p)$ for a program p above): this is the multiset read by the system in the given computational step. For any P_c, a set of applicable programs in a configuration c, let us denote by $read(P_c)$ the multiset of objects read by the tape rules of the programs of P_c, that is,

$$read(P_c) = \bigcup_{p_i \neq \#,\ (p_1, \ldots, p_n) = P_c} read(p_i).$$

Then we can also define the set of multisets which can be read in any configuration of the genPCol automaton Π as

$$in(\Pi) = \{read(P_c) \mid P_c \in App_c\}.$$

Remark 1. Although the set of configurations of a genPCol automaton Π is infinite (because the multiset corresponding to the contents of the environment is not necessarily finite), the set $in(\Pi)$ is finite. To see this, note that the applicability of a program by a component cell also depends on the contents of the particular component. Since at most one program can be applied in a component in one computational step, and the number of programs associated to each component is finite, the number of different sets of applicable programs in any configuration, that is, the set App_c.

A successful computation defines this way an accepted sequence of multisets: $u_1 u_2 \ldots u_s,\ u_i \in in(\Pi)$, for $1 \leq i \leq s$, that is, the sequence of multisets entering the system during the steps of the computation.

Let $\Pi = (V, e, w_E, (w_1, P_1), \ldots, (w_n, P_n), F)$ be a genPCol automaton. The *set of input sequences accepted by* Π is defined as

$$A(\Pi) = \{u_1 u_2 \ldots u_s \mid u_i \in in(\Pi),\ 1 \le i \le s,\ \text{and there is a configuration sequence } c_0, \ldots, c_s,\ \text{with } c_0 = (w_E, w_1, \ldots, w_n),\ c_s \in F,\ \text{and } c_i \overset{P_{c_i}}{\Longrightarrow} c_{i+1} \text{ with } u_{i+1} = read(P_{c_i}) \text{ for all } 0 \le i \le s-1\}.$$

Let Π be a genPCol automaton, and let $f : in(\Pi) \to 2^{\Sigma^*}$ be a mapping, such that $f(u) = \{\varepsilon\}$ if and only if u is the empty multiset.

The *language accepted by* Π with respect to f is defined as

$$L(\Pi, f) = \{f(u_1) f(u_2) \ldots f(u_s) \in \Sigma^* \mid\ u_1 u_2 \ldots u_s \in A(\Pi)\}.$$

Let us denote the class of languages accepted by generalized PCol automata with capacity l and with mappings from the class $\mathcal{F}$

- by $\mathcal{L}(\text{genPCol}, \mathcal{F}, \text{com-tape}(l))$ when all the communication rules are tape rules,
- by $\mathcal{L}(\text{genPCol}, \mathcal{F}, \text{all-tape}(l))$ when all the programs must have at least one tape rule, and
- by $\mathcal{L}(\text{genPCol}, \mathcal{F}, *(l))$ when programs with any kinds of rules are allowed.

Let V and Σ be two alphabets, and let $\mathcal{M}_{FIN}(V) \subseteq \mathcal{M}(V)$ denote the set of finite subsets of the set of finite multisets over an alphabet V. Consider a mapping $f : D \to 2^{\Sigma^*}$ for some $D \in \mathcal{M}_{FIN}(V)$. We say that $f \in \mathcal{F}_{\text{TRANS}}$, if for any $v \in D$, we have $|f(v)| = 1$, and we can obtain $f(v) = \{w\}$, $w \in \Sigma^*$ by applying a deterministic finite transducer to any string representation of the multiset v, (as w is unique, the transducer must be constructed in such a way that all string representations of the multiset v as input result in the same $w \in \Sigma^*$ as output, and moreover, as f should be nonerasing, the transducer produces a result with $w \ne \varepsilon$ for any nonempty input).

Besides the above defined class of mappings, we also use the so called permutation mapping. Let $f_{perm} : \mathcal{M}(V) \to 2^{\Sigma^*}$ where $V = \Sigma$ be defined as follows. For all $v \in \mathcal{M}(V)$, we have

$$f(v) = \{a_1 a_2 \ldots a_s \mid |v| = s, a_1 a_2 \ldots a_s \text{ is a permutation of the elements of } v\}.$$

We denote the language classes that can be characterized with these types of input mappings as $\mathcal{L}_X(\text{genPCol}, Y(k))$, where $X \in \{f_{perm}, \text{TRANS}\}$, $Y \in \{\text{com-tape}, \text{all-tape}, *\}$.

Now we recall an example from [14] to demonstrate the above defined notions.

Example 1. Let $\Pi = (\{a, b, c\}, e, \emptyset, (ea, P), F)$ be a genPCol automaton where

$$P = \{\langle e \to a, a \overset{T}{\leftrightarrow} e\rangle, \langle e \to b, a \overset{T}{\leftrightarrow} e\rangle, \langle e \to b, b \overset{T}{\leftrightarrow} a\rangle, \langle e \to c, b \overset{T}{\leftrightarrow} a\rangle, \langle a \to b, b \overset{T}{\leftrightarrow} a\rangle, \langle a \to c, b \overset{T}{\leftrightarrow} a\rangle\}$$

with all the communication rules being tape rules. Let also $F = \{(v, ca) \mid a \notin v\}$ be the set of final configurations.

A possible computation of this system is the following:

$$(\emptyset, ea) \Rightarrow (a, ea) \Rightarrow (aa, ea) \Rightarrow (aaa, eb) \Rightarrow (aab, ba) \Rightarrow (bba, ba) \Rightarrow (bbb, ac)$$

where the first three computational steps read the multiset containing an a, the last three steps read a multiset containing a b, thus the accepted multiset sequence of this computation is $(a)(a)(a)(b)(b)(b)$.

It is not difficult to see that similarly to the one above, the computations which end in a final configuration (a configuration which does not contain the object a in the environment) accept the set of multiset sequences

$$A(\Pi) = \{(a)^n(b)^n \mid n \geq 1\}.$$

The set of multisets which can be read by Π is $in(\Pi) = \{a, b\}$ (where a and b denote the multisets containing one copy of the object a and b, respectively).

If we consider f_{perm} as the input mapping, we have

$$L(\Pi, f_{perm}) = \{a^n b^n \mid n \geq 1\}.$$

On the other hand, if we consider the mapping $f_1 \in \mathcal{F}_{\mathrm{TRANS}}$ where $f_1 : in(\Pi) \to 2^{\Sigma^*}$ with $\Sigma = \{c, d, e, f\}$ and $f_1(a) = \{cd\}$, $f_1(b) = \{ef\}$, we get the language

$$L(\Pi, f_1) = \{(cd)^n(ef)^n \mid n \geq 1\}.$$

The computational capacity of genPCol automata was investigated in [13, 15, 14]. It was shown that with unrestricted programs systems of capacity *one* generate any recursively enumerable language, that is,

$$\mathcal{L}_X(\text{genPCol}, *(k)) = \mathcal{L}(\text{RE}),\ k \geq 1,\ X \in \{perm, TRANS\}.$$

A similar result holds for all-tape systems with capacity at least two.

$$\mathcal{L}_X(\text{genPCol}, \text{all-tape}(k)) = \mathcal{L}(\text{RE}) \text{ for } k \geq 2,\ X \in \{perm, TRANS\}.$$

.

3 P Colony Automata and the LL(k) Condition

Let $U \subset \Sigma^*$ be a finite set of strings over some alphabet Σ. Let us denote by $\text{FIRST}_k(U)$ for some $k \geq 1$, the set of length k prefixes of the elements of U, that is, let

$$\text{FIRST}_k(U) = \{pref_k(u) \in \Sigma^* \mid u \in U\}$$

where $pref_k(u)$ denotes the string of the first k symbols of u if $|u| \geq k$, or $pref_k(u) = u$ otherwise.

Definition 1. Let $\Pi = (V, e, w_E, (w_1, P_1), \ldots, (w_n, P_n), F)$ be a genPCol automaton, let $f : in(\Pi) \to 2^{\Sigma^*}$ be a mapping as above, and let $c_0, c_1, \ldots, c_s$ be a sequence of configurations with $c_i \Longrightarrow c_{i+1}$ for all $0 \le i \le s-1$.

We say that the P colony Π is LL(k) for some $k \ge 1$ with respect to the mapping f, if for any two distinct sets of programs applicable in configuration c_s, $P_1, P_2 \in Acc_{c_s}$ with $P_1 \neq P_2$, if $u_1 = read(P_1)$ and $u_2 = read(P_2)$, then $\mathrm{FIRST}_k(f(u_1)) \cap \mathrm{FIRST}_k(f(u_2)) = \emptyset$.

The class of context-free LL(k) languages will be denoted by $\mathcal{L}$(CF,LL(k)) (see for example the monograph [1] for more details), while the the languages characterized by genPCol automata satisfying the above defined condition, with input mapping of type f_{perm} or $f \in TRANS$, will be denoted by $\mathcal{L}_X$(genPCol,LL(k)), $X \in \{perm, TRANS\}$.

Let us illustrate the above definition with an example.

Example 2. Let $\Pi = (\{a, b, c, d, f, g, e\}, e, \emptyset, (ea, P_1), F)$ where

$$\begin{aligned} P_1 = \{&\langle e \to b, a \overset{T}{\leftrightarrow} e\rangle, \langle e \to e, b \overset{T}{\leftrightarrow} a\rangle, \langle e \to c, a \overset{T}{\leftrightarrow} e\rangle, \langle e \to f, a \overset{T}{\leftrightarrow} e\rangle, \\ &\langle e \to d, c \overset{T}{\leftrightarrow} b\rangle, \langle b \to c, d \overset{T}{\leftrightarrow} e\rangle, \langle e \to g, f \overset{T}{\leftrightarrow} b\rangle, \langle b \to f, g \overset{T}{\leftrightarrow} e\rangle\} \text{ and} \\ F = \{&(v, ce), (v, fe) \mid v \in V^*, b \notin v\}. \end{aligned}$$

The language characterized by Π is

$$L(\Pi, f_{perm}) = \{a\} \cup \{(ab)^n a(cd)^n \mid n \ge 1\} \cup \{(ab)^n a(fg)^n \mid n \ge 1\}.$$

To see this, consider the possible computations of Π. The initial configuration is $(\emptyset, ea)$ and there are three possible configurations that can be reached, namely (we denote by $\Rightarrow_u$ a configuration change during which the multiset of symbols u was read by the automaton)

1. $(\emptyset, ea) \Rightarrow_a (a, ce)$,
2. $(\emptyset, ea) \Rightarrow_a (a, fe)$,
3. $(\emptyset, ea) \Rightarrow_a (a, be)$.

The first two cases are non-accepting states, but the derivations cannot be continued, so let us consider the third one.

$$(a, be) \Rightarrow_b (b, ea) \Rightarrow_a (ba, be) \Rightarrow_b (bb, ea) \Rightarrow_a \ldots \Rightarrow_b (b^i, ea).$$

At this point, the computation can follow two different paths again, either

$$(b^i, ae) \Rightarrow_a (b^i a, ec) \Rightarrow_c (b^{i-1}ac, db) \Rightarrow_d (b^{i-1}acd, ce) \Rightarrow_c \ldots \Rightarrow_d (ac^i d^i, ce),$$

or

$$(b^i, ae) \Rightarrow_a (b^i a, ef) \Rightarrow_f (b^{i-1}af, gb) \Rightarrow_g (b^{i-1}afg, fe) \Rightarrow_f \ldots \Rightarrow_g (af^i g^i, fe).$$

In the first phase of the computation, the system produces bs and sends them to the environment, then in the second phase these bs are exchanged to cds or fgs. The system can reach an accepting state when all the bs are used, that is, when an equal number of abs and either cds or fgs were produced.

Note that the system satisfies the LL(1) property, the symbol that has to be read, in order to accept a desired input word, determines the set of programs that has to be used in the next computational step.

As a consequence of the above example, we can state the following.

Theorem 1. *There are context-free languages in $\mathcal{L}_X$ (genPCol,LL(1)), $X \in \{perm, TRANS\}$, which are not in $\mathcal{L}$(CF,LL(k)) for any $k \geq 1$.*

Proof. The language $L(\Pi, f_{perm}) \in \mathcal{L}_{perm}$(genPCol,LL(1)) from Example 2 is not in $\mathcal{L}$(CF,LL(k)) for any $k \geq 1$. If we consider the mapping $f_1 \in TRANS$, $f_1 : \{a, b, c, d, f, g\} \to \{a, b, c, d, f, g\}$ with $f_1(x) = x$ for all $x \in \{a, b, c, d, f, g\}$, then $L(\Pi, f_1) = L(\Pi, f_{perm})$, thus, $\mathcal{L}_{TRANS}$(genPCol,LL(1)) also contains the non-LL(k) context-free language.

4 Conclusions

We have investigated the possibility of deterministically parsing languages characterized by P colony automata. We have given the definition of an LL(k)-like property for (generalized) P colony automata, and shown that languages which are not LL(k) in the "original" context-free sense for any $k \geq 1$ can be characterized by LL(1) P colony automata with different types of input mappings.

The properties of these language classes for different ks and different types of input mappings are open to further investigations.

References

1. Aho, A.V., Ulmann, J.D.: The Theory of Parsing, Translation, and Compiling, vol. 1. Prentice-Hall, Englewood Cliffs, N.J. (1973)
2. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E., Vaszil, G.: Pcol automata: Recognizing strings with P colonies. In: Martínez del Amor, M.A., Păun, G., Pérez Hurtado, I., Riscos Núñez, A. (eds.) Eighth Brainstorming Week on Membrane Computing, Sevilla, February 1-5, 2010, pp. 65–76. Fénix Editora (2010)
3. Cienciala, L., Ciencialová, L., Kelemenová, A.: On the number of agents in P colonies. In: Eleftherakis, G., Kefalas, P., Păun, G., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing, 8th International Workshop, WMC 2007, Thessaloniki, Greece, June 25-28, 2007 Revised Selected and Invited Papers. Lecture Notes in Computer Science, vol. 4860, pp. 193–208. Springer (2007)
4. Cienciala, L., Ciencialová, L., Kelemenová, A.: Homogeneous P colonies. Computing and Informatics 27(3+), 481–496 (2008)

5. Ciencialová, L., Cienciala, L.: Variation on the theme: P colonies. In: Kolăr, D., Meduna, A. (eds.) Proc. 1st Intern. Workshop on Formal Models. pp. 27–34. Ostrava (2006)
6. Ciencialová, L., Csuhaj-Varjú, E., Kelemenová, A., Vaszil, G.: Variants of P colonies with very simple cell structure. International Journal of Computers, Communication and Control 4(3), 224–233 (2009)
7. Csuhaj-Varjú, E., Oswald, M., Vaszil, G.: P automata. In: Păun, G., Rozenberg, G., Salomaa, A. (eds.) The Oxford Handbook of Membrane Computing. Oxford University Press, Inc. (2010)
8. Csuhaj-Varjú, E., Kelemen, J., Kelemenová, A.: Computing with cells in environment: P colonies. Multiple-Valued Logic and Soft Computing 12(3-4), 201–215 (2006)
9. Csuhaj-Varjú, E., Margenstern, M., Vaszil, G.: P colonies with a bounded number of cells and programs. In: Hoogeboom, H.J., Păun, G., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing, 7th International Workshop, WMC 2006, Leiden, The Netherlands, July 17-21, 2006, Revised, Selected, and Invited Papers. Lecture Notes in Computer Science, vol. 4361, pp. 352–366. Springer (2006)
10. Csuhaj-Varjú, E., Vaszil, G.: P automata or purely communicating accepting P systems. In: Păun, G., Rozenberg, G., Salomaa, A., Zandron, C. (eds.) Membrane Computing, International Workshop, WMC-CdeA 2002, Curtea de Arges, Romania, August 19-23, 2002, Revised Papers. Lecture Notes in Computer Science, vol. 2597, pp. 219–233. Springer (2002)
11. Freund, R., Oswald, M.: P colonies working in the maximally parallel and in the sequential mode. In: Zaharie, D., Petcu, D., Negru, V., Jebelean, T., Ciobanu, G., Cicortas, A., Abraham, A., Paprzycki, M. (eds.) Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2005), 25-29 September 2005, Timisoara, Romania. pp. 419–426. IEEE Computer Society (2005)
12. Freund, R., Oswald, M.: P colonies and prescribed teams. Int. J. Comput. Math. 83(7), 569–592 (2006)
13. Kántor, K., Vaszil, G.: Generalized P colony automata. Journal of Automata, Languages and Combinatorics 19(1-4), 145–156 (2014)
14. Kántor, K., Vaszil, G.: Generalized P colony automata and their relation to P automata. In: Gheorghe, M., Rozenberg, G., Salomaa, A., Zandron, C. (eds.) Membrane Computing - 18th International Conference, CMC 2017, Bradford, UK, July 25-28, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10725, pp. 167–182. Springer (2017)
15. Kántor, K., Vaszil, G.: On the classes of languages characterized by generalized P colony automata. Theor. Comput. Sci. 724, 35–44 (2018)
16. Păun, A., Păun, G.: The power of communication: P systems with symport/antiport. New Generation Comput. 20(3), 295–306 (2002)

Testing Identifiable Kernel P Systems Using an X-machine Approach

Marian Gheorghe[1], Florentin Ipate[2], Raluca Lefticaru[1,2], Ana Ţurlea[2]

[1] School of Electrical Engineering and Computer Science,
University of Bradford, West Yorkshire, Bradford BD7 1DP, UK
{m.gheorghe,r.lefticaru}@bradford.ac.uk

[2] Department of Computer Science,
Faculty of Mathematics and Computer Science and ICUB
University of Bucharest,
Str. Academiei nr. 14, 010014, Bucharest, Romania
florentin.ipate@ifsoft.ro,ana.turlea@fmi.unibuc.com

Summary. This paper presents a testing approach for kernel P systems (*kP systems*), based on the X-machine testing framework and the concept of cover automaton. The testing methodology ensures that the implementation conforms the specifications, under certain conditions, such as the *identifiably* concept in the context of kernel P systems.

Keywords: membrane computing; kernel P systems; X-machines; cover automata; testing.

1 Introduction

Membrane computing [19] is a research field initiated twenty years ago [17, 18] by Gheorghe Păun. Initially inspired by the structure and functioning of the living cells, the field has known a fast development, different types of membrane systems (or *P systems*) being investigated.

Having so many computational models (cell-like, tissue-like P systems, P colonies, kernel P systems) and also different software implementations for these models, it is important to devise testing methodologies that ensure that the implementation conforms with the 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 [13], stream X-machine based testing [14], mutation testing for evaluating the efficiency of the test sets [16], model-checking based testing [15].

In this paper we will present a testing approach for *kernel P systems*, which is based on the X-machine testing approach and has as core concept the *identifiability* of multisets of rules. Kernel P systems are a model introduced in [9], which can be simulated using a software framework, called kPWorkbench [5] or some earlier variants (so called *simple kP systems*) using P-Lingua and the MeCoSim simulator [11].

This paper is structured as follows: Section 2 presents the preliminaries regarding kP systems and theoretical background regarding automata and X-machine based testing. Section 3 introduces the concept of *identifiable* kernel P systems, while Section 4 illustrates our testing approach for kP systems. Finally, conclusions are presented in Section 5.

2 Preliminaries

This section briefly presents the notations used, then gives the basic definitions regarding kernel P systems [9] and presents the previous testing approaches for automata and X-machines, that have been applied also for testing simple cell-like P systems.

In the following we introduce the notations used in the paper. For a finite alphabet $A = \{a_1, ..., a_p\}$, A^* represents the set of all strings (sequences) over A. The empty string is denoted by λ and $A^+ = A^* \setminus \{\lambda\}$ denotes the set of non-empty strings. A^n denotes the set of all strings of length n, $n \geq 0$, with members in the alphabet A, and $A[n] = \bigcup_{0 \leq i \leq n} A^i$ denotes the set of all strings of length at most n.

For a string $u \in A^*$, $|u|_a$ denotes the number of occurrences of a in u, where $a \in A$. For a subset $S \subseteq A$, $|u|_S$ denotes the number of occurrences of the symbols from S in u. The length of a string u is given by $\sum_{a_i \in A} |u|_{a_i}$. The length of the empty string is 0, i.e. $|\lambda| = 0$.

A multiset over A is a mapping $f : A \rightarrow \mathbb{N}$. Considering only the elements from the support of f (where $f(a_{i_j}) > 0$, for some j, $1 \leq j \leq p$), the multiset is represented as a string $a_{i_1}^{f(a_{i_1})} \ldots a_{i_p}^{f(a_{i_p})}$, where the order is not important. In the sequel multisets will be represented by such strings.

2.1 Kernel P systems

In the following we will give a formal definition of kernel P systems (or kP systems) [9]. We start by introducing the concept of a *compartment type* utilised later in defining the compartments of a kernel P system (kP 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.*

Kernel P systems have features inspired by object-oriented programming, for example one *compartment type* can have one or more *instances*. These instances share the same set of rules and execution strategies (so will deliver the same functionality), but they may contain different multisets of objects and different neighbours according to the graph relation specified.

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.*

In this paper we will only deal with a simplified version of kP systems having *one single compartment* as this does not affect the general method introduced here and makes the presentation easier to follow. For details regarding the ways of flattening an arbitrary P system, including the kP system discussed in this paper, we refer mainly to [7], but similar approaches are also presented in other papers ([20], [1]). The kP system will be denoted $k\Pi = (A, \mu_1, C_1, 1)$, where μ_1 denotes the graph with one node.

Within the general kP systems framework, the following types of evolution rules have been considered so far:

- *rewriting and communication* rule: $x \to y\{g\}$, where g represents a **guard** (will be formally explained in Def. 4), $x \in A^+$ 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, $y = (a_1, t_1) \ldots (a_h, t_h)$, 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 [9]. However, this type of rules will not be considered in the following discussion.

Remark 1. In the context of *one compartment kP systems*, there will be no need to specify the target compartment, so the rules will be simple communication rules, which in addition can have *guards*. Each rule occurring in the following discussion has the form $r : x \to y\{g\}$, where r identifies the rule and is called **label**, $x \to y$ is the rule itself and g is its **guard**. The part $x \to y$ is also called the **body** of the rule, denoted also $b(r)$. The guards are constructed using multisets over A, as operands, and relational or Boolean operators. The definition of the guards is now introduced. We start with some notations.

For a multiset w over A and an element $a \in A$, we denote by $|w|_a$ the number of objects a occurring in w. Let us denote $Rel = \{<, \leq, =, \neq, \geq, >\}$, the set of relational operators, $\gamma \in Rel$, a relational operator, and a^n a multiset, 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.

Example 1. 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.

In addition to its evolution rules, each compartment type in a kP system has an associated *execution strategy.* The rules corresponding to a compartment can be grouped in blocks, each having one of the following strategies:

In kP systems the way in which rules are executed is defined for each compartment type t from T – see Def. 1. As in Def. 1, $Lab(R)$ is the set of labels of the rules R.

Definition 5. *For a compartment type $t = (R, \sigma)$ from T and $r \in Lab(R)$, $r_1, \ldots, r_s \in Lab(R)$, the* execution strategy*, σ, is defined by the following*

- *$\sigma = \lambda$, means no rule from the current compartment will be executed;*
- *$\sigma = \{r\}$ – the rule r is executed;*
- *$\sigma = \{r_1, \ldots, r_s\}$ – one of the rules labelled $r_1, \ldots, r_s$ will be non-deterministically chosen and executed; if none is applicable then nothing is executed; this is called* alternative *or* choice*;*
- *$\sigma = \{r_1, \ldots, r_s\}^*$ – the rules are applied an arbitrary number of times (*arbitrary parallelism*);*

- $\sigma = \{r_1, \ldots, r_s\}^\top$ – *the rules are executed according to the* maximal parallelism *strategy;*
- $\sigma = \sigma_1 \& \ldots \& \sigma_s$, *means executing sequentially* $\sigma_1, \ldots, \sigma_s$, *where* σ_i, $1 \leq i \leq s$, *describes any of the above cases; if one of* σ_i *fails to be executed then the rest is no longer executed.*

These execution strategies and the fact that in any compartment several blocks with different strategies can be composed and executed offer a lot of flexibility to the kP system designer, similarly to procedural programming.

Definition 6. *A* configuration *of a kP system, kΠ, with n compartments, is a tuple* $c = (c_1, \ldots, c_n)$, *where* $c_i \in A^*$, $1 \leq i \leq n$, *is the multiset from compartment i. The* initial configuration *is* $(w_1, \ldots, w_n)$, *where* $w_i \in A^*$ *is the initial multiset of the compartment i,* $1 \leq i \leq n$.

A *transition* (or *computation step*), introduced by the next definition, is the process of passing from one configuration to another.

Definition 7. *Given two configurations* $c = (c_1, \ldots, c_n)$ *and* $c' = (c'_1, \ldots, c'_n)$ *of a kP system, kΠ, with n compartments, where for any* $i, 1 \leq i \leq n$, $u_i \in A^*$, *and a multiset of rules* $M_i = r_{1,i}^{n_{1,i}} \ldots r_{k_i,i}^{n_{k_i,i}}$, $n_{j,i} \geq 0$, $1 \leq j \leq k_i, k_i \geq 0$, *a* transition *or a* computation step *is the process of obtaining* c' *from c by using the multisets of rules* M_i, $1 \leq i \leq n$, *denoted by* $c \Longrightarrow^{(M_1,\ldots,M_n)} c'$, *such that for each i,* $1 \leq i \leq n$, c'_i *is the multiset obtained from* c_i *by first extracting all the objects that are in the left-hand side of each rule of* M_i *from* c_i *and then adding all the objects a that are in the right-hand side of each rule of* M_i *represented as* (a, t_i) *and all the objects b that are in the right-hand side of each rule of* M_j, $j \neq i$, *such that b is represented as* (b, t_i).

In the theory of kP systems, each compartment might have its own execution strategy. In the sequel we focus on three such execution strategies, namely maximal parallelism, arbitrary parallelism (also called asynchronous execution) and sequential execution. These will be denoted by *max*, *async* and *seq*, respectively. When in a transition from c to c' using $(M_1, \ldots, M_m)$, we intend to refer to a specific transition mode tm, $tm \in \{max, async, seq\}$, then this will be denoted by $c \Longrightarrow_{tm}^{(M_1,\ldots,M_m)} c'$.

A *computation* in a P system is a sequence of transitions (computation steps).

A configuration is called *final configuration*, if no rule can be applied to it. In a final configuration the computation stops.

As usual in P systems, we only consider terminal computations, i.e., those arriving in a final configuration and using one of the above mentioned transition modes. We are now ready to define the result of a computation.

Definition 8. *For a kP system kΠ using the transition mode tm,* $tm \in \{max, async, seq\}$, *in each compartment, we denote by* $N_{tm}(\Pi)$ *the number of objects appearing in the output compartment of a final configuration.*

Two kP systems $k\Pi$ and $k\Pi'$ are called *equivalent* with respect to the transition mode tm, $tm \in \{max, async, seq\}$, if $N_{tm}(k\Pi) = N_{tm}(k\Pi')$.

In this paper we will only deal with kP systems having *one single compartment* as this does not affect the general method introduced here and makes the presentation easier to follow. Indeed, limiting the investigation to one compartment kP systems does not affect the generality of it due to the fact that there are ways of flattening an arbitrary P system, including the kP system discussed in this paper, into a P system with one single compartment. For details regarding the flattening of a P system we refer mainly to [7], but similar approaches are also presented in other papers ([20], [1]). Such a kP system will be denoted $k\Pi = (A, \mu_1, C_1, 1)$, where μ_1 denotes the graph with one node. The rules on the right-hand side will have multisets over A, as in the case of one single compartment there is no need to indicate where objects are sent to.

2.2 The *W*-method for testing finite cover automata

In the following subsection we introduce the basic finite cover automata concepts [3, 12] and the W-method for generating test suites from finite cover automata [13]. We will consider only deterministic finite automata.

Finite Cover Automata

Definition 9. *A finite automaton (abbreviated FA) is a tuple $A = (V, Q, q_0, F, h)$, where:*

- V *is the finite input alphabet;*
- Q *is the finite set of states;*
- $q_0 \in Q$ *is the initial state;*
- $F \subseteq Q$ *is the set of final states;*
- $h : Q \times V \to Q$ *is the next-state function.*

Definition 10. *Let $A = (V, Q, q_0, F, h)$ be a FA, $U \subseteq V^*$ a finite language and l the length of the longest sequence(s) in U. Then A is called a* deterministic finite cover automaton *(*DFCA*) of U if $L_A \cap V[l] = U$. A* minimal *DFCA for U is a DFCA for U having the least number of states.*

The concept of DFCA was introduced by Câmpeanu et al. [2], [3]. A minimal DFCA have considerably fewer states than the minimal FA that accepts U.

The *W*-method

In *conformance testing* there is a formal specification of the system (for example a FA) and the aim is to generate a test suite such that whenever the implementation under test (IUT) passes all tests, it is guaranteed to conform to the specification. The IUT is unknown but it is assumed to behave like some element from a set of models, called *fault model.* In the case of the W-method, the fault model consists of all FAs A' with the same input alphabet V as the specification A, whose number of states m' does not exceed the number of states m of A by more than k ($m'-m \leq k$), where $k \geq 0$ is a predetermined integer that must be estimated by the tester.

The W-method was originally devised for when the conformance relation is automata equivalence [4], but in this paper we are interested in conformance for bounded sequences. This problem is described in [10] as follows: given an FA specification A and an integer $l \geq 1$ (the upper bound) such that L_A contains at least one sequence of length l, we want to construct a set of sequences of length less than or equal to l that can establish whether the implementation behaves as specified for all sequences in $V[l]$. Since L_A contains at least one sequence of length l, A is a $DFCA$ for $L_A \cap V[l]$ and so the test suite will check whether the IUT model A' is also a DFCA for $L_A \cap V[l]$.

A *test suite* will be a finite set $Y_k \subseteq V[l]$ of input sequences that, for every A' in the fault model that is not $V[l]$-equivalent to A, will produce at least one erroneous output. That is, A and A' are $V[l]$-equivalent whenever A and A' are Y_k-equivalent.

Suppose the specification A used for test generation is a minimal DFCA for $L_A \cap V[l]$. The W-method for bounded sequences, as developed in [12], involves the selection of two sets of input sequences, S and W, as follows:

Definition 11. *$S \subseteq V^*$ is called a proper state cover of A if for every state q of A there exists $s \in S$ such that $h(q_0, s) = q$ and $|s| = level(q)$.*

Definition 12. *$W \subseteq V^*$ is called a strong characterisation set of A if for every two states q_1 and q_2 of A and every $j \geq 0$, if q_1 and q_2 are $V[j]$-distinguishable then q_1 and q_2 are $(W \cap V[j])$-distinguishable.*

Naturally, in the above definition, it is sufficient for q_1 and q_2 to be $(W \cap V[j])$-distinguishable when j is the length of the shortest sequences that distinguish between q_1 and q_2.

Once S and W have been selected, the test suite is obtained using the formula: $Y_k = SV[k+1](W \cup \{\lambda\}) \cap V[l] \setminus \{\lambda\}$ [12].

2.3 X-machine based testing

This subsection presents the X-machine based testing methodology, giving the formal definitions for X-machines, the test transformation of an X-machine and l-bounded conformance test suites. For more details and complete proofs [10] can be consulted, here only the main results are given.

An X-machine is a finite automaton in which transitions are labelled by partial functions on a data set X instead of mere symbols [6].

Definition 13. *An* X-machine *(XM) is a tuple $Z = (Q, X, \Phi, H, q_0, x_0)$ where:*

- *Q is a finite set of states;*

- *X is the (possible infinite) data set;*
- *Φ is a finite set of distinct* processing functions*; a processing function is a non-empty (partial) function of type $X \to X$;*
- *H is the (partial)* next-state *function, $H : Q \times \Phi \to Q$;*
- *$q_0 \in Q$ is the initial state;*
- *$x_0 \in X$ is the initial data value.*

We regard an X-machine as a finite automaton with the arcs labelled by functions from the set Φ, which is often called the *type* of Z. The automaton $A_Z = (\Phi, Q, H, q_0)$ over the alphabet Φ is called the *associated finite automaton* (FA) of Z. The language accepted by the automaton is denoted by L_{A_Z}.

Definition 14. *A* computation *of Z is a sequence $x_0, \dots x_n$, with $x_i \in X, 1 \le i \le n$, such that there exist $\phi_1, \dots, \phi_n \in \Phi$ with $\phi_i(x_{i-1}) = x_i, 1 \le i \le n$ and $\phi_1 \dots \phi_n \in L_{A_Z}$. The set of computations of Z is denoted by $Comp(Z)$.*

A sequence of processing functions that can be applied in the initial data value x_0 is said to be controllable.

Definition 15. *A sequence $\phi_1, \dots, \phi_n \in \Phi^*$, with $\phi_i \in \Phi, 1 \le i \le n$, is said to be controllable if there exist $x_1, \dots x_n \in X$ such that $\phi_i(x_{i-1}) = x_i, 1 \le i \le n$. A set $P \subseteq \Phi^*$ is called controllable if for every $p \in P$, p is controllable.*

Let us assume we have an X-machine specification Z and an (unknown) IUT that behaves like an element Z' of a fault model. In this case, the fault model will be a set of X-machines with the same data set X, type Φ and initial data value x_0 as the specification. The idea of test generation from an X-machine is to reduce checking that the IUT Z' conforms to the specification Z to checking that the associated automaton of the IUT conforms to the associated automaton of the X-machine specification.

Definition 16. *The test transformation of Z is the (partial) function $t : \Phi^* \to X^*$ defined by:*

- *$t(\lambda) = x_0$.* (1)
- *Let $p \in \Phi^*$ and $\phi \in \Phi$.*
  - *Suppose $t(p)$ is defined. Let $t(p) = x_0 \dots x_n$.*
    - *If $x_n \in dom\phi$ then:*
      - *If $p \in L_{A_Z}$ then $t(p\phi) = t(p)\phi(x_n)$.* (2)
      - *Else $t(p\phi) = t(p)$.* (3)
    - *Else $t(p\phi)$ is undefined.* (4)
  - *Otherwise, $t(p\phi)$ is undefined.* (5)

Lemma 1. *Let t be a test transformation of Z and $p = \phi_1 \dots \phi_n$, with $\phi_1, \dots, \phi_n \in \Phi$.*

- *Suppose p is controllable and let $x_1, \dots, x_n \in X$ such that $\phi_i(x_{i-1}) = x_i, 1 \le i \le n$.*

  - *If $p \in L_{A_Z}$, then $t(p) = x_0 \dots x_n$.*
  - *If $p \notin L_{A_Z}$, then $t(p) = x_0 \dots x_{k+1}$, where $0 \leq k \leq n-1$, is such that $\phi_1 \dots \phi_k \in L_{A_Z}$ and $\phi_1 \dots \phi_k \phi_{k+1} \notin L_{A_Z}$.*
- *If p is not controllable, then $t(p)$ is not defined.*

In order to establish that the associated automaton of the IUT Z' conforms to the associated automaton of the X-machine specification Z, we have to be able to identify the processing functions that are applied when the computations of Z and Z' are examined.

Definition 17. *Φ is called* identifiable *if for all $\phi_1, \phi_2 \in \Phi$, whenever there exists $x \in X$ such that $\phi_1(x) = \phi_2(x)$, $\phi_1 = \phi_2$.*

If Φ is identifiable, then we are able to establish if a controllable sequence of processing functions is correctly implemented by examining the computations of the specification Z and the implementation Z', as shown by the following lemma.

Lemma 2. *Let Z and Z' be XMs with type Φ. Suppose Φ is identifiable. Let $p = \phi_1 \dots \phi_n \in \Phi^*$, with $\phi_i \in \Phi$, $1 \leq i \leq n$, be a controllable sequence. Suppose $t(p)$ is a computation of Z if and only if $t(p)$ is a computation of Z'. Then $p \in L_{A_Z}$ if and only if $p \in L_{A'_Z}$.*

Definition 18. *Let Z be an X-machine and C a fault model for Z. An* l-bounded conformance test suite *for Z w.r.t. C, $l > 0$, is a set $T \subseteq X[l+1]$ such that for every $Z' \in C$ the following holds: if $T \cap Comp(Z) = T \cap Comp(Z')$ then $Comp(Z) \cap X[l+1] = Comp(Z') \cap X[l+1]$.*

That is, whenever any element of T is a computation of Z if and only if it is a computation of Z', Z' conforms to Z for sequences of length up to l. The following theorem shows that the test transformation defined earlier provides a mechanism for converting test suites for finite automata into set suites for X-machines.

Theorem 1. *Let Z be an XM with type Φ, data set X and initial data value x_0. Suppose Φ is identifiable and $L_{A_Z} \cup \Phi[l]$ is controllable. Let C be a set of XMs such that for every $Z' \in C$, $L_{A'_Z} \cap \Phi[l]$ is controllable. Let $P \subseteq \Phi[l]$, such that, for every $Z' \in C$, whenever $P \cap L_{A_Z} = P \cap L_{A'_Z}$ we have $L_{A_Z} \cap \Phi[l] = L_{A'_Z} \cap \Phi[l]$. Then $t(P)$ is an* l-bounded conformance test suite *for Z w.r.t. C.*

Let $l > 0$ be a predefined upper bound. We assume that Φ is identifiable and $L_{A_Z} \cap \Phi[l]$ is controllable. We assume that A_Z, the associated automaton of Z, is a minimal DFCA for $L_{A_Z} \cup \Phi[l]$ (if not, this is minimised [3]). Suppose the fault model C is the set of X-machines Z' with the same data set X, type Φ and initial data value x_0 as Z such that $L_{A_{Z'}} \cap \Phi[l]$ is controllable, whose number of states m' does not exceed the number of states m of Z by more than k ($m' - m \leq k$), $k \leq 0$. Then an l-bounded conformance test suite for Z w.r.t. C is

[3] The minimisation preserves the controlability requirements as the set $L_{A_Z} \cap \Phi[l]$ remains unchanged.

$$T_k = t(S\Phi[k+1](W \cup \{\lambda\}) \cap \Phi[l] \setminus \{\lambda\}),$$

where S is a proper state cover of A_Z, W is a strong characterisation set of A_Z and t is a test transformation of Z.

3 Identifiable transitions in kernel P systems

The concept of identifiable transitions in cell-like P systems was first introduced in [10] and then extended to kernel P systems in [8]. We now aim to present the *identifiability* concept in the context of kP systems and then illustrate how it is used as basis for kP systems testing. The identifiability concept is first introduced for simple rules and then is generalised for multisets of rules.

Definition 19. *Two rules $r_1 : x_1 \to y_1\{g_1\}$ and $r_2 : x_2 \to y_2\{g_2\}$ from R_1, are said to be* identifiable *in configuration c, if they are applicable to c and if $c \Longrightarrow^{r_1} c'$ and $c \Longrightarrow^{r_2} c'$ then $b(r_1) = b(r_2)$.*

According to the above definition the rules r_1 and r_2 are identifiable in c if when the result of applying them to c is the same then their bodies, $x_1 \to y_1$ and $x_2 \to y_2$, are identical. The rules are not identifiable when the condition from Definition 19 is not satisfied.

A multiset or rules $M = r_1^{n_1} \dots r_k^{n_k}, M \in R_1^*$, where $r_i : x_i \to y_i\{g_i\}$, $1 \le i \le k$, is applicable to the multiset c iff $x_1^{n_1} \dots x_k^{n_k} \subseteq c$ and g_i is true in c for $1 \le i \le k$.

Definition 20. *The multisets of rules $M', M'' \in R_1^*$, are said to be* identifiable*, if there is a configuration c where M' and M'' are applicable and if $c \Longrightarrow^{M'} c'$ and $c \Longrightarrow^{M''} c'$ then $M' = M''$.*

Example 2. Considering the rules $r_1 : a \to x\{\ge a\}$, $r_2 : b \to y\{\ge b\}$, $r_3 : a \to y\{\ge a\}$, $r_4 : b \to x\{\ge b\}$, and the configuration ab it is clear that the multisets of rules $M' = r_1r_2$ and $M'' = r_3r_4$ are not identifiable in the configuration $c = ab$, as $c = ab \Longrightarrow^{M'} c' = xy$ and $c = ab \Longrightarrow^{M''} c' = xy$, but $M' \neq M''$.

A kP system $k\Pi$ has its *rules identifiable* if any two multisets of rules, $M', M'' \in R_1^*$, are identifiable.

Given a multiset of rules $M = r_1^{n_1} \dots r_k^{n_k}$, where $r_i : x_i \to y_i\{g_i\}$, $1 \le i \le k$, we denote by r_M the rule $x_1^{n_1} \dots x_k^{n_k} \to y_1^{n_1} \dots y_k^{n_k}\{g_1 \wedge \dots \wedge g_k\}$, i.e., the concatenation of all the rules in M. One can observe that the applicability of the multiset of rules M to a certain configuration is equivalent to the applicability of the rule r_M to that configuration. It follows that one can study first the usage of simple rules.

Remark 2. For any two rules $r_i : x_i \to y_i$, $1 \le i \le 2$, when we check whether they are identifiable or not one can write them as $r_i : uv_i \to wz_i\{g_i\}$, $1 \le i \le 2$, where for any $a \in V$, a appears in at most one of the v_1 or v_2, i.e., all the common symbols on the left-hand side of the rules are in u. Let us denote by c_{r_1,r_2}, the configuration uv_1v_2. Obviously this is the smallest configuration in which r_1 and r_2 are applicable, given that g_1 and g_2 are true in uv_1v_2.

Remark 3. If $r_i : x_i \to y_i \ \{g_i\}$, $1 \leq i \leq 2$, are applicable in a configuration c and $c \subseteq c'$ then they are not always applicable to c'. They are applicable to c' when all g_i, $1 \leq i \leq 2$, are true in c'.

Remark 4. If the rules r_1, r_2 are not applicable to c_{r_1,r_2} then there must be minimal configurations c where the rules are applicable and they are minimal, i.e., there is no c_1, $c_1 \subset c$ where the rules are applicable. Such minimal configurations where r_1, r_2 are applicable are of the form tc_{r_1,r_2}, where $t \in A^*$, $t \neq \lambda$.

In the following we introduce some theoretical results, characterising the identifiability or non-identifiability or rules and multisets of rules under certain conditions. The complete proofs for these the results are given in [8].

Lemma 3. *Two rules which are identifiable in a configuration c are identifiable in any configuration containing c in which they are applicable.*

Lemma 4. *Two rules which are identifiable in a minimal configuration c are identifiable in any other minimal configuration c' where they are applicable.*

Corollary 1. *Two rules r_1 and r_2 identifiable in a minimal configuration tc_{r_1,r_2}, $t \in A^*$, are identifiable in any configuration in which they are applicable.*

Corollary 2. *Two multisets of rules M_1 and M_2 identifiable in $tc_{r_{M_1},r_{M_2}}$, $t \in A^*$, are identifiable in any configuration in which they are applicable.*

From now on, we will always verify the identifiability (or non identifiability) only for the smallest configurations associated with rules or multisets of rules and will not mention these configurations anymore in the results to follow.

The applicability of two rules (multisets of rules) to a certain configuration depends not only on the fact that there left hand sides (the concatenation of the left hand sides) must be contained in the configuration and the guards must be true, but takes into account the execution strategy.

Remark 5. For the *async* transition mode two multisets of rules (and two rules) applicable in a configuration are also applicable in any other bigger configuration, when the corresponding guards are true. For the *seq* mode this is true only for multisets with one single element and obviously for simple rules. In the case of the *max* mode the applicability of the multisets of rules (or rules) to various configurations depends on the contents of the configurations and other available rules. For instance if we consider a P system containing the rules $r_1 : a \to a \ \{\geq a\}; r_2 : ab \to abb \ \{\leq b^{100}\}; r_3 : bb \to c \ \{\geq b^2\}$ and the configuration $c = ab$ then in c only r_1 and r_2 are applicable and identifiable, but in $c_1 = abb$, containing c, r_1 is no longer applicable, but instead we have r_2 and the multiset r_1r_3 applicable. In ab^{101} r_2 and any multiset containing it are not applicable due to the guard being false; also r_1 is no longer applicable, but $r_1r_3^{55}$ is now applicable, due to maximal parallelism.

Remark 6. In the following results whenever we refer to arbitrary rules or multisets of rules they are always meant to be applicable with respect to the transition mode.

Theorem 2. *The rules* $r_1 : x_1 \to y_1 \ \{g_1\}$ *and* $r_2 : x_2 \to y_2 \ \{g_2\}$*, are not identifiable if and only if they have the form* $r_1 : uv_1 \to wv_1 \ \{g_1\}$ *and* $r_2 : uv_2 \to wv_2 \ \{g_2\}$ *and for any* $a \in A$*,* a *appears in at most one of* v_1 *or* v_2*.*

Corollary 3. *The rules* $r_1 : uv_1 \to wz_1 \ \{g_1\}$ *and* $r_2 : uv_2 \to wz_2 \ \{g_2\}$*, such that for any* $a \in A$*,* a *appears in at most one of* v_1 *or* v_2*, are identifiable if and only if* $v_1 \neq z_1$ *or* $v_2 \neq z_2$*.*

Theorem 3. *If* r_1 *and* r_2 *are identifiable then* r_1^n *and* r_2^n *are identifiable, for any* $n \geq 1$*.*

4 Testing identifiable kernel P systems

In order to generate test suites for a kernel P system using the X-machine testing method, first a corresponding X-machine model needs to be constructed. As discussed in Section 2, multi-compartment P systems can be flattened into one membrane P systems and there are different ways to realise this [1, 7, 20]. Consequently, we will illustrate the testing approach using an one-membrane kP system model $k\Pi = (V, T, \mu_1, w_1, R_1, 1)$. The main idea is to construct an X-machine $Z^t = (Q^t, X, \Phi, H^t, q_0^t, x_0)$, corresponding to the computation tree of $k\Pi$. As the computation tree of the kP system might be infinite, we will consider only computations of maximum l steps, where $l > 0$ is a predefined integer. Let $R_1 = \{r_1, \ldots, r_n\}$ be the set of rules of $k\Pi$. As only finite computations are considered, for every rule $r_i \in R_1$ there will be some N_i such that, in any step, r_i can be applied at most N_i times, $1 \leq i \leq n$. Thus the X-machine $Z^t = (Q^t, X, \Phi, H^t, q_0^t, x_0)$ is defined as follows:

- Q^t is the set of nodes of the computation tree of maximum l steps;
- q_0^t is the root node;
- X is the set of multisets with elements in V;
- x_0 is the initial multiset w_1;
- Φ is the set of (partial) functions induced by the application of multisets of rules $r_1^{i_1} \ldots r_n^{i_n}$, $0 \leq i_1 \leq N_1, \ldots, 0 \leq i_n \leq N_n$, $i_1 + \ldots i_n > 0$;
- H^t is the next-state function determined by the computation tree.

Remark 7. Note that, by definition, L_{A_Z} is controllable, i.e. any sequence of processing functions from the associated automaton A_Z can be applied in the initial data x_0 (corresponding to the initial multiset w_1). Intuitively a path in the DFCA corresponds to a path in the computation tree of the kP system.

Remark 8. The set of (partial) functions, Φ, from the above definition is identifiable (according to Definition 17) if and only if the corresponding multisets of rules are pairwise identifiable (according to Definition 20).

Example 3. Let us consider one compartment kP system $k\Pi_1 = (V, V, \mu_1, w_1, R_1, 1)$, where $V = \{a, b, c\}$, $w_1 = ab$, and

$$R_1 = \left\{ \begin{array}{ll} r_1 : a \to b\{\geq a \wedge \geq b\} & r_2 : ab \to bc\{\leq a \wedge \geq b\} \\ r_3 : c \to b\{\geq b \wedge \leq c^{100}\} & r_4 : c \to cc\{\leq c^{100}\} \end{array} \right\}$$

Let us build the computation tree considering that rules are applied in the maximally parallel mode. The initial configuration $w_1 = ab$ is at the root of the tree (level 0 of the tree). Two computation steps are possible from the root: $ab \Longrightarrow^{r_1} b^2$ and $ab \Longrightarrow^{r_2} bc$. Then the configurations b^2 and bc are at the first level of the tree. No rule can be applied in b^2 (this is a terminal configuration of $k\Pi_1$), but two computation steps exist form bc: $bc \Longrightarrow^{r_3} b^2$ and $bc \Longrightarrow^{r_4} bc^2$. The new configurations produced represent the second level of the tree. Again, no rule can be applied in b^2, but there are three computation steps from bc^2: $bc^2 \Longrightarrow^{r_3^2} b^3$, $bc^2 \Longrightarrow^{r_3 r_4} b^2c^2$ and $bc^2 \Longrightarrow^{r_4^2} bc^4$. No rule can be applied in b^3, but there are three computation steps from b^2c^2 and five from bc^4: $b^2c^2 \Longrightarrow^{r_3^2} b^4$, $b^2c^2 \Longrightarrow^{r_3 r_4} b^3c^2$ and $b^2c^2 \Longrightarrow^{r_4^2} b^2c^4$; $bc^4 \Longrightarrow^{r_3^4} b^5$, $bc^4 \Longrightarrow^{r_3^3 r_4} b^4c^2$, $bc^4 \Longrightarrow^{r_3^2 r_4^2} b^3c^4$, $bc^4 \Longrightarrow^{r_3 r_4^3} b^2c^6$ and $bc^4 \Longrightarrow^{r_4^4} bc^8$. The configurations produced by these eight multisets of rules represent the fourth level of the tree.

It can be easily checked that any two of the above multisets of rules are identifiable, according to Corollary 3, and consequently they produce different results when applied to the same configuration – see above.

Let the upper bound on the number of computation steps considered be $l = 4$. For this value of l, the rules r_1 and r_2 have been applied at most once, so $N_1 = 1$ and $N_2 = 1$, whereas rules r_3 and r_4 have been applied at most four times, so $N_3 = 4$ and $N_4 = 4$. Therefore the type Φ of the X-machine Z^t corresponding to the computation tree is the set of partial functions induced by the multisets $r_1^{i_1} r_2^{i_2} r_3^{i_3} r_4^{i_4}$, $0 \leq i_1 \leq 1, 0 \leq i_2 \leq 2, 0 \leq i_3 \leq 4, 0 \leq i_4 \leq 4$, $i_1 + i_2 + i_3 + i_4 > 0$. The associated automaton A_{Z^t} is as represented in Figure 1.

Let $L_{A_{Z^t}} \subseteq \Phi^*$ be the language accepted by the associated automaton A_{Z^t}. In order to apply the test generation method presented in Section 2.3, an X-machine Z whose associated automaton A_Z is a DFCA for $L_{A_{Z^t}}$ needs to be constructed first.

Let $\leq$ be a total order on Q^t such that $q_1 \leq q_2$ whenever $level(q_1) \leq level(q_2)$ and denote $q_1 < q_2$ if $q_1 \leq q_2$ and $q_1 \neq q_2$. In other words, the node at the superior level in the tree is before the node at the inferior level; if the nodes are at the same level then their order is arbitrarily chosen. Define $P^t = \{q \in Q^t \mid \neg\exists q' \in Q^t \cdot q' \sim q, q' < q\}$ and $[q] = \{q' \in Q^t \mid q' \sim q \wedge \neg\exists q'' \in P^t \cdot q'' \sim q', q'' < q\}$ for every $q \in P^t$ (i.e. $[q]$ denotes the set of all states q' for which q is the minimum state similar to q'). Then we have the following result (the proof is given in [10]).

Theorem 4. *Let $Z = (Q, X, \Phi, H, q_0, x_0)$, where $Q = \{[q] \mid q \in P^t\}$, $q_0 = [q_0^t]$, $H([q], \phi) = [H^t(q, \phi)]$ for all $q \in P^t$ and $\phi \in \Phi$. Then A_Z is a minimal DFCA for $L_{A_{Z^t}}$.*

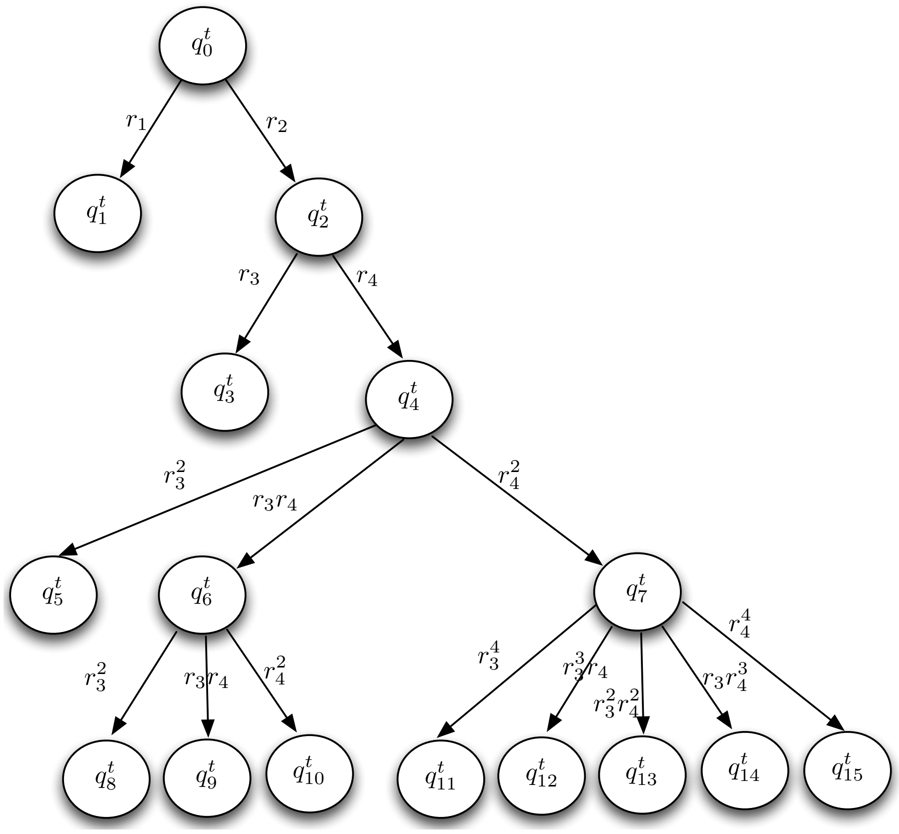


Fig. 1. The associated automaton A_{Z^t} corresponding to the computation tree for $k\Pi_1$ and $l = 4$

Remark 9. Consider Z^t as in the previous example. $P^t = \{q_0^t, q_1^t, q_2^t, q_4^t, q_7^t\}$; $[q_0^t] = \{q_0^t, q_8^t, q_9^t, q_{10}^t, q_{11}^t, q_{12}^t, q_{13}^t, q_{14}^t, q_{15}^t\}$, $[q_1^t] = \{q_1^t, q_3^t, q_5^t\}$, $[q_2^t] = \{q_2^t\}$, $[q_4^t] = \{q_4^t, q_6^t\}$, $[q_7^t] = \{q_7^t\}$. Then $Z = (Q, X, \Phi, H, q_0, x_0)$, where $Q = \{[q_0^t], [q_1^t], [q_2^t], [q_4^t], [q_7^t]\}$ and $q_0 = [q_0^t]$. The associated automaton of Z is a minimal DFCA for $L_{A_{Z^t}}$ and is as represented in Figure 2.

Once the X-machine Z has been constructed the test generation process entails the following steps:

1. **Construct the sets S and W (proper state cover and characterisation sets, respectively).**
   It can be easily remarked from Fig. 2. that λ, r_1, r_2, $r_2\ r_4$, $r_2\ r_4\ r_4^2$ are the sequences of minimum length[4] that reach $[q_0^t]$, $[q_1^t]$, $[q_2^t]$, $[q_4^t]$ and $[q_7^t]$, respectively.

[4] Notation: for rules r and r', rr' denotes the application of rules r and r' in one single step, whereas $r\ r'$ (separated by space) denotes the application of rule r in one step

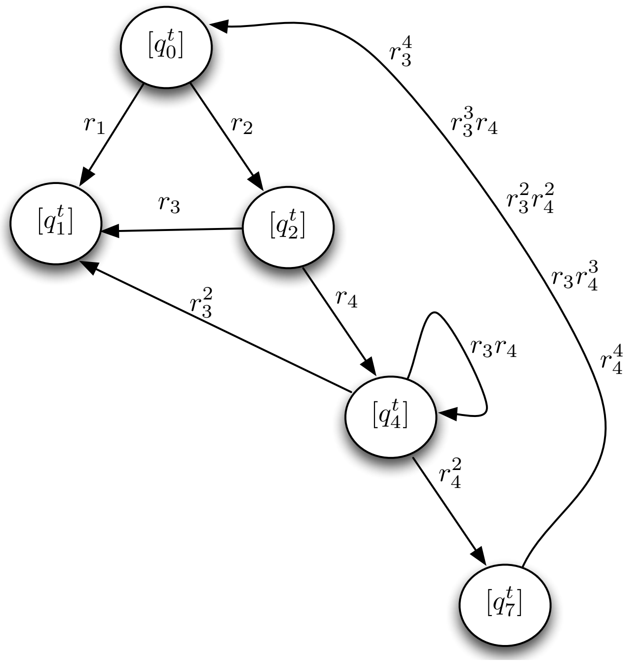


Fig. 2. The DFCA for $L_{A_{Z^t}}$

Consequently $S = \{r_1, r_2, r_2\ r_4, r_2\ r_4\ r_4^2\}$ is a proper state cover of Z. Furthermore, since r_1 distinguishes $[q_0^t]$ from all remaining states and r_3, $r_3 r_4$ and r_3^4 hold the same property for $[q_2^t]$, $[q_4^t]$ and $[q_7^t]$, respectively, $W = \{r_1, r_3, r_3 r_4, r_3^4\}$ is a strong characterisation set of Z.

2. **Determine the fault model of the IUT.**
This entails establishing the transitions that a (possibly faulty) implementation is capable to perform. For example, when the correct application of rules (in the P system specification) is in the maximally parallel mode *max*, one fault that we may consider is when the rules are applied in a less restrictive mode such that the asynchronous mode *async*. Hence the notion of controllability for P systems is defined by considering this, less restrictive, application mode.

Definition 21. *A sequence of multisets of rules* $p = M_1 ... M_m$*, with* $M_i \in R_1^*$*,* $1 \leq i \leq m$*, is said to be* controllable *if there exist configurations* $u_0 = w_1, u_1, \ldots, u_m$*,* $u_i \in V^*$*,* $0 \leq i \leq m$*, such that* $u_{i-1} \Longrightarrow_{FM}^{M_i} u_i$*,* $1 \leq i \leq m$*, where* $u \Longrightarrow_{FM}^{M} u'$ *denotes a computation step in the* fault model *from configuration* u *to configuration* u' *by applying the multiset of rules* M*.*

followed by the application of rule r' in the following step; the second notation is also used for multisets of rules.

Consider again the P system $k\Pi_1$ as in Example 3. Then $ab \Longrightarrow^{r_2} bc$ and $bc \Longrightarrow^{r_4} bc^2$, but $bc^2 \Longrightarrow^{r_4} bc^3$ does not hold since the rules of $k\Pi_1$ must be applied in the maximally parallel mode. However, if we consider that in the fault model of the IUT rules may be applied in the asynchronous mode, the sequence $r_2\ r_4\ r_4$ is controllable. The fault model is also determined by the maximum number of states $m + k$ that the IUT may have, where m is the number of states of the X-machine Z and $k \geq 0$ is a non-negative integer estimated by the tester.

3. **Construct an l-bounded conformance test suite.**
This is $T_k = t(Y_k)$, where $Y_k = S\Phi[k+1](W \cup \{\lambda\}) \cap \Phi[l] \setminus \{\lambda\}$ and t is a test transformation of Z.
According to [4], the upper bound for the number of sequences in $S\Phi[k+1]W$ is $m^2 \cdot r^{k+1}$ and the total length of all sequences is not greater that $m^2 \cdot (m + k) \cdot r^{k+1}$, where r is the number of elements of Φ. In particular, for $k = 0$, the respective bounds are $m^2 \cdot r$ and $m^3 \cdot r$. The increase in size produced by replacing W with $W \cup \{\lambda\}$ in the above formula is negligible. Note that these bounds refer to the worst case; in an average case, the size of Y_k is much lower. Furthermore, the size of $t(Y_k)$ is normally significantly lower than the size of Y_k since only the controllable sequences are in the domain of t.
The construction of Y_k is straightforward, so we illustrate only the construction of the test transformation t with an example. Consider again rule application mode is maximal parallelism for $k\Phi$ and the asynchronous mode for the fault model. Consider the sequences $s_0 = \lambda$, $s_1 = r_2$, $s_2 = s_1\ r_4$, $s_3 = s_2\ r_4$, $s_4 = s_3\ r_4$, $s_5 = s_4\ r_1$ and $s_6 = s_5\ r_1$. By rule (1) of Definition 16, $t(s_0) = x_0 = ab$. As $ab \Longrightarrow^{r_2} bc$, by rule (2) $t(s_1) = ab\ bc$. Similarly, as $bc \Longrightarrow^{r_4} bc^2$, by rule (2) $t(s_2) = ab\ bc\ bc^2$. On the other hand r_4 cannot be applied in configuration bc^2 in the maximally parallel mode, but $bc^2 \Longrightarrow^{r_4}_{FM} bc^3$ (in the asynchronous mode) and so, by rule (2), $t(s_3) = ab\ bc\ bc^2\ bc^3$. Furthermore, $bc^3 \Longrightarrow^{r_4}_{FM} bc^4$ and so, by rule (3) of Definition 16, $t(s_4) = t(s_3) = ab\ bc\ bc^2\ bc^3$. As r_1 cannot be applied in bc^4, by rule (4) $t(s_5)$ is undefined. Furthermore, by rule (5), $t(s_6)$ is also undefined, so no test sequences will be generated for s_5 and s_6.

5 Conclusions

This paper presents a testing approach for kernel P systems that, under certain conditions, ensures that the implementation conforms to the specification. The methodology is based on the *identifiable kernel P systems* concept, which is essential for testing, and has been introduced for one-compartment kP systems with rewriting rules, but could be extended.

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. Agrigoroaiei, O., Ciobanu, G.: Flattening the transition P systems with dissolution. In: Gheorghe, M., Hinze, T., Paun, G., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing - 11th International Conference, CMC 2010, Jena, Germany, August 24-27, 2010. Revised Selected Papers. Lecture Notes in Computer Science, vol. 6501, pp. 53–64. Springer (2010), https://doi.org/10.1007/978-3-642-18123-8_7
2. Câmpeanu, C., Sântean, N., Yu, S.: Minimal cover-automata for finite languages. In: International Workshop on Implementing Automata. pp. 43–56. Springer (1998), https://doi.org/10.1007/3-540-48057-9_4
3. Câmpeanu, C., Santean, N., Yu, S.: Minimal cover-automata for finite languages. Theoretical Computer Science 267(1-2), 3–16 (2001), https://doi.org/10.1016/S0304-3975(00)00292-9
4. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering 4(3), 178–187 (1978), https://doi.org/10.1109/TSE.1978.231496
5. 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), https://doi.org/10.1007/978-3-642-54239-8_12
6. Eilenberg, S.: Automata, languages, and machines. Academic press (1974)
7. 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, Y., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing. Lecture Notes in Computer Science, vol. 8340, pp. 173–188. Springer Berlin Heidelberg (2014), https://doi.org/10.1007/978-3-642-54239-8_13
8. Gheorghe, M., Ipate, F.: Identifiable kernel P systems. Submitted (2018)
9. 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), http://www.gcn.us.es/files/11bwmc/097_gheorghe_ipate.pdf
10. Gheorghe, M., Ipate, F., Konur, S.: Testing based on identifiable P systems using cover automata and X-machines. Information Sciences 372, 565–578 (2016), https://doi.org/10.1016/j.ins.2016.08.028
11. 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
12. Ipate, F.: Bounded sequence testing from deterministic finite state machines. Theoretical Computer Science 411(16-18), 1770–1784 (2010), https://doi.org/10.1016/j.tcs.2010.01.030

13. 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
14. 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
15. 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
16. 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
17. Păun, G.: Computing with membranes. Tech. rep., Turku Centre for Computer Science (1998), http://tucs.fi/publications/view/?pub_id=tPaun98a
18. Păun, G.: Computing with membranes. Journal of Computer and System Sciences 61(1), 108–143 (2000), https://doi.org/10.1006/jcss.1999.1693
19. The P systems website. http://ppage.psystems.eu, [Online; accessed 12/05/2018]
20. Verlan, S.: Using the formal framework for 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. 56–79. Springer Berlin Heidelberg (2014), https://doi.org/10.1007/978-3-642-54239-8_6

Spiking Neural P Systems with Addition/Subtraction Computing on Synapses

Yun Jiang[1,2] ⋆ and Zhiqiang Chen[1,2]

[1] Chongqing Engineering Laboratory for Detection, Control and Integrated Systems
Chongqing Technology and Business University, Chongqing 400067, China
[2] School of Computer Science and Information Engineering,
Chongqing Technology and Business University, Chongqing 400067, China
jiangyun@email.ctbu.edu.cn

Summary. Spiking neural P systems (SN P systems, for short) are a class of distributed and parallel computing models inspired from biological spiking neurons. In this paper, we introduce a variant called SN P systems with addition/subtraction computing on synapses (CSSN P systems). CSSN P systems are inspired and motivated by the shunting inhibition of biological synapses, while incorporating ideas from dynamic graphs and networks. We consider addition and subtraction operations on synapses, and prove that CSSN P systems are computationally universal as number generators, under a normal form (i.e. a simplifying set of restrictions).

Key words: Membrane computing, Spiking neural P system, Computing on synapse, Computationally universal

1 Introduction

Brain is a rich source of inspiration for informatics. Specifically, it has provided plenty of ideas to construct high performance computing models, as well as to design efficient algorithm. Inspired from the biological phenomenon that neurons cooperate in the brain by exchanging spikes via synapses, various neural-like computing models have been proposed. In the framework of membrane computing, a kind of distributed and parallel neural-like computing model were proposed in 2006 [1], which is called spiking neural P systems (SN P systems for short).

SN P systems have neurons that process only one type of symbols, the spike, based on the indistinct signal used by biological neurons. Neurons are placed on nodes of a directed graph, and the edges between neurons are called synapses, again based on synapses of biological neurons. SN P systems processes spikes by applying rules, and two of the most common types are firing rules and forgetting rules: the former rules produce one or more spike, which is/are sent from the source neuron

⋆ Corresponding author.

to every neuron connected by a synpase, while the latter rules remove spikes from the neuron.

Since the human brain and biological neurons are rich sources of computing ideas, many variants of SN P systems have been introduced, taking inspiration from biological phenomena, e.g. synapse weight, neuron division, astrocytes, inhibitory synapses, as in [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. Investigation on the theoretical and practical usefulness have also been applied to these variants: their computing power in relation to well-known models of computation, e.g. finite automata, register machines, grammars, computing numbers or strings as in [17, 18, 19, 20, 21, 22, 23, 24, 25, 26]; computing efficiency in solving hard problems, as in [27, 28].

Moreover, practical applications and software for simulations have been developed for SN P systems and their variants: to design logic gates, logic circuits [29] and databases [30], to perform basic arithmetic operations [31][32], to represent knowledge [33], to diagnose fault [34, 35, 36], to approximately solve combinatorial optimization problems [37].

In this work, we introduce a variant of SN P systems which we refer to as SN P systems with addition/subtraction computing on synapses (CSSN P systems, for short). CSSN P systems take inspirations and motivations from biological, mathematical and computing sources.

Biologically, it is known that not only neurons but also synapses can process spikes, as in [38]. Synapses monitor the spikes go through it and change the value of spikes according to their excitatory or inhibitory. Hence, it is natural to consider addition/subtraction computing of two consecutive spikes on synapses.

The mathematical and computing inspirations are taken from the study of dynamic graphs. Since SN P systems are in essence static graphs, it is natural to consider them for dynamic graphs as well.

In the survey of dynamic graphs, two main kinds of structural evolutions of the graphs are identified: node-centric evolutions, i.e. nodes or vertices are the focus, and edge-centric evolutions, i.e. edges are the focus. In the framework of SN P systems, several works have focused on dynamism for the neuron, as in [28][13]. More recently, SN P systems with structural plasticity were introduced in [39], with subsequent works in [40, 41, 42]. In these systems, synapses can be created or removed by plasticity rules of neurons, hence, structural evolution of the systems are more edge-centric.

For CSSN P systems however, we further focus on synapse dynamism, but this time we add an addition or subtraction computing to each synapse in the system. Specifically, for two consecutive spikes get through the synapse, if excitatory spike come first, the synapse does addition, otherwise, subtraction. Furthermore, we show that CSSN P systems are computationally universal, under a normal form (m. ore details below), for generating numbers

This work is organized as follows: Section 2 provides the definition of CSSN P systems and their semantics; Section 3 provides an example of CSSN P systems; Section 4 provides universality result on CSSN P systems; At last, Section 5 concludes this work and provides further directions for research.

2 Spiking Neural P Systems with Computing on Synapses

In this section we define our proposed variant, and provides the semantics. A spiking neural P system with computing on synapses, CSSN P system for short, of degree $m \geq 1$ is a construct of the form

$$\Pi = (O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, in, out),$$

where:

1. $O = \{a\}$ is a singleton alphabet, and a is called spike;
2. $\sigma_1, \sigma_2, \ldots, \sigma_m$ are neurons of the form $\sigma_i = (n_i, R_i), 1 \leq i \leq m$; $n_i \geq 0$ is the initial number of spikes contained in the neuron σ_i; R_i is a finite set of rules of the following two forms:
   (a) Firing rule: $E/a^c \rightarrow a^p; d$, where E is a a regular expression over $\{a\}$, $c \geq 1$, $d \geq 0$, with the restriction $c \geq p$. Specifically, when $d = 0$, it can be omitted;
   (b) Forgetting rule: $a^s \rightarrow \lambda$, for some $s \geq 1$, with the restriction that for each rule $E/a^c \rightarrow a^p; d$ of type (1) from R_i, we have $a^s \notin L(E)$;
3. $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ is the set of synapses between neurons, with restriction $(i, i) \notin syn$ for $1 \leq i \leq m$, which means no σ_i has a synapse to itself;
4. $in, out \in \{1, 2, \ldots, m\}$ indicate the input and output neuron, respectively.

The *in* or *out* elements in the construction can be omitted, depending on whether they exist in the system or not. For a given neuron σ_i (also called neuron i or simply σ_i), we illustrate σ_i as an oval, and synapses between neurons as arcs. The input neuron has a synapse or arc from the environment, i.e. everything outside or not part of the system, while the output neuron has a synapse to the environment.

The firing rule of the form $E/a^c \rightarrow a^p; d$ with $c \geq p \geq 1$ is called an *extended rule*; if $p = 1$, the rule is called a *standard rule*. As a convention, if $L(E) = \{a^c\}$, the rule can be simply written as $a^c \rightarrow a^p; d$. Specifically, if $d = 0$, it can be omitted and the rule can be simply written as $a^c \rightarrow a^p$.

The semantics of applying firing rules are as follows: if the neuron σ_i contains k spikes, $a^k \in L(E)$ and $k \geq c$, then the firing rule $E/a^c \rightarrow a^p; d \in R_i$ can be applied, i.e., the number of spikes k in σ_i satisfies the requirement for applying the rule. Applying such a rule means consuming c spikes from σ_i and producing p spikes after d time units, thus $k-c$ spikes remains in σ_i. If $d = 0$, then the produced spikes are released immediately, and if $d = 1$, then the spikes are emitted in the next step, and so on. In the case $d \geq 1$, if the rule is applied at step t, neuron σ_i becomes closed in the interval $[t, t + d)$, i.e., it cannot receive spikes and spikes sent to it while it is closed are lost. At step $t+d$, neuron σ_i becomes open, i.e., it can receive spikes again, and at the same time it sends p spikes to come across all the synapses such that $(i,j) \in syn$.

The semantics of applying forgetting rules is as follows: if the neuron contains exactly s spikes, then the forgetting rule $a^s \to \lambda$ can be used, and this means that all s spikes are removed from the neuron.

The semantics of computing synapse is as follows: each synapse from syn monitors the flow of spikes come across it, and do the computation according to the dynamic status of these spikes. Specifically, synapses will do addition, subtraction or nothing according to the comparison result of two consecutive flows of spikes, and the spikes sent to the receiving neuron (neuron σ_j such that $(i,j) \in syn$) will be the results of these computation.

For each synapse $(i,j) \in syn$, there are several flows of spikes comes across the synapse during the computation of the system, i.e. there is a spike train on the synapse. In this case, synapse will compare the number of spikes come across it consecutively, and get excitatory, inhibitory, or normal accordingly. We say that two flows of spikes a^p and a^q come across the synapse consecutively, i.e. spikes a^p at step t and spikes a^q at step $t+1$, then there are three kinds of relationship between a^p and a^q.

case 1: $p < q$, the synapse gets excited and do addition, and the receiving neuron σ_j will get spikes a^{p+q};
case 2: $p = q$, the synapse gets normal and do nothing, and the receiving neuron σ_j will get spikes a^p;
case 3: $p > q$, the synapse gets inhibited and do subtraction, and the receiving neuron σ_j will get spikes a^{p-q}.

Specially, it is possible that during the computation of the system, there is only one flow of spikes comes across the synapse, for instance, spikes a^s come across the synapse in one step. In this case, there is no comparison and the synapse does not compute, and the receiving neuron σ_j will get spikes a^s.

A *configuration* of the system at a given step is the contribution of spikes among neurons, and the status of each neuron, whether closed or open. The *initial configuration* is given by n_i of each neuron σ_i. The system reaches a *halting configuration* when there is no rule can be applied and all neurons are open. A *computation* is defined as a sequence of configuration transitions, from an initial configuration, and following rule application semantics and synapse computing semantics. A *computation halts* if the system reaches a halting configuration.

The result of the computation can be defined in various ways in SN P systems. In this work we use the following definition: when a computation halts, the number of spikes present in the output neuron is said to be computed by an CSSN P system Π. We denote the set of all number computed in this way by Π as $N_{gen}(\Pi)$. In $N_{gen}(\Pi)$ we have Π able to *generate numbers* (we also say that Π works in the *generative mode*).

CSSN P systems can also accept numbers i.e. Π works in the *acceptive mode*. When working in the *acceptive mode*, the output neuron is ignored, and Π work as follows: a number is introduced into Π as the number of spikes present in the input neuron in the initial configuration, and the number is accepted by Π if the computation halts. The set of numbers accepted this way by Π is denoted as $N_{acc}(\Pi)$.

The families of all sets of $N_\alpha(\Pi)$, with $\alpha \in \{gen, acc\}$ are denoted as $N_\alpha SNPCOS_m(rule_k, cons_r, forg_q)$, with at most $m \geq 1$ neurons in the system, at most $k \geq 1$ rules in each neuron, consuming at most $r \geq 1$ spikes in any firing rule of any neuron, and forgetting at most $q \geq 1$ spikes in any forgetting rule of any neuron. We note that the parameter for the delay for the firing rules are specified in other literature, e.g. in [], but here we do not use it so the parameter is omitted.

3 An Example

In this section we provide an example Π_1 to further clarify the semantics of CSSN P systems, which is shown in Fig. 1.

The system Π_1 is composed of two neurons, labeled with 1 and 2, and they are the input and output neuron, respectively. Formally, system Π_1 is a structure of the form $\Pi = (O, \sigma_1, \sigma_2, syn, 1, 2)$, where:

- $O = \{a\}$;
- $\sigma_1 = (5, R_1)$, with $R_1 = \{a^5 \to a^4, a^5/a^2 \to a^2, a^5/a^3 \to a^3, a^3 \to a^3, a^3 \to a^2, a^2 \to a^2\}$;
- $\sigma_2 = (0, R_2)$, with $R_2 = \emptyset$;
- $syn = \{(1, 2)\}$.

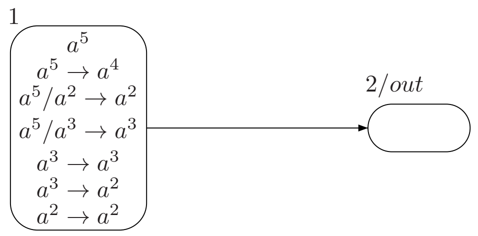


Fig. 1. A simple example of an SN P system with computing synapse

Neuron σ_1 fires at the first step of the computation. As shown in the table below, there are four sets of rules to apply. Also, the flow of spikes on synapse $(1, 2)$, computing on synapse, and spikes received by σ_2 are present in detail.

rules to apply	flow of spikes on synapse	computing on synapse	spikes received by σ_2
$a^5 \to a^4$	a^4	none	a^4
$a^5/a^2 \to a^2$, $a^3 \to a^3$	a^2a^3	addition	a^5
$a^5/a^2 \to a^2$, $a^3 \to a^2$	a^2a^2	none	a^2
$a^5/a^3 \to a^3$, $a^2 \to a^2$	a^3a^2	subtraction	a

From the table we can see that: when working in the generative mode, the set of all number computed by Π is $\{1, 2, 4, 5\}$, i.e. $\Pi_{gen} = \{1, 2, 4, 5\}$.

4 Universality Results

In this section, we present universality results for SN P systems with computing on synapses, for the generating modes.

Theorem 1. $N_{gen}SNPCOS_*(rule_3, cons_5, forg_5) = NRE$.

Proof. In order to prove Theorem, it is enough to simulate a register machine M with an CSSN P system Π with restrictions in the Theorem statement. Before constructing Π, we provide a general overview of the computation as follows: each register r in M corresponds to a σ_r in Π. If r stores the number n, then σ_r stores $2n$ spikes. If M applies an instruction l_i that performs some operation $OP \in \{ADD, SUB, HALT\}$, this means that a corresponding neuron σ_{l_i} becomes activated in order to simulate OP. Without loss of generality, register 1 of M is the output register and this register is never subjected to a SUB instruction. In this way, the spikes in σ_1 are never decremented.

In what follows, we provide the modules in most cases in a graphical manner for easier reference. The initial configuration of Π is such that all neurons are empty, except for σ_{l_0} which contains three spikes. The three spikes in σ_{l_0} begins the computation of Π, corresponding to simulating the initial instruction l_0 of M.

Module ADD: The module simulating $l_i : (ADD(r), l_j, l_k)$ is given in Fig. 2.

Once σ_{l_i} is activated, both neurons l_{i_1} and l_{i_2} receive three spikes.

With three spikes inside, neuron l_{i_2} applies the rule $a^3/a^2 \to a^2$ first, and then the rule $a \to \lambda$. As a result, the spike trains on synapses (l_{i_2}, r) and (l_{i_2}, l_{i_4}) are $a^2\lambda$, and the computing on these synapses is subtraction. In this way, σ_r receive two spikes, corresponding to incrementing the register r by one, and $\sigma_{l_{i_4}}$ also gets two spikes.

With three spikes inside, neuron l_{i_1} applies the rule $a^3/a \to a$ first, and then it must nondeterministically choose to apply either $a^2 \to a^2$ or $a^2 \to \lambda$. If the former rule is chosen, then the spike trains on synapses (l_{i_1}, l_{i_3}) and (l_{i_1}, l_{i_4}) are aa^2, and the synapses does addition. In this way, $\sigma_{l_{i_3}}$ receives three spikes, gets activated, and sends three spikes to σ_{l_j}, which makes neuron l_j activated; $\sigma_{l_{i_4}}$ receives five spikes (three from synapse (l_{i_1}, l_{i_4}) and two from synapse (l_{i_2}, l_{i_4})), which are forgotten according to the rule $a^5 \to \lambda$. If the latter rule is applied, then the spike trains on synapses (l_{i_1}, l_{i_3}) and (l_{i_1}, l_{i_4}) are $a\lambda$, and the synapses

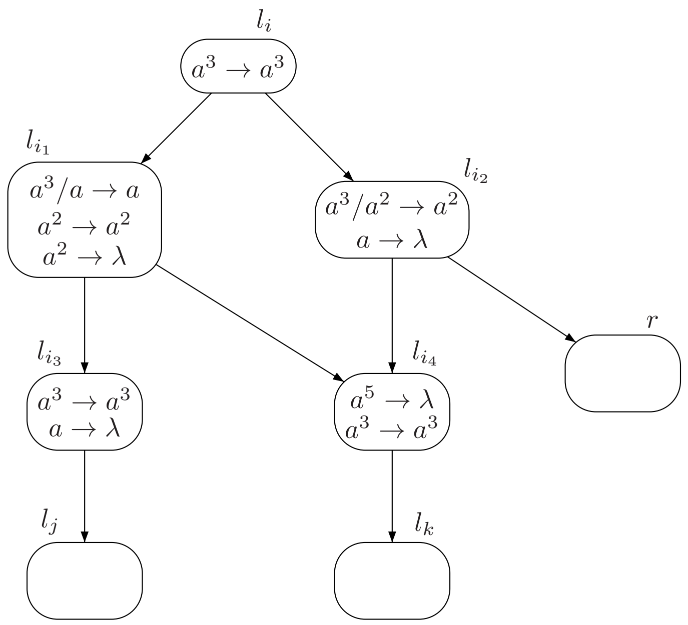


Fig. 2. Module ADD

does subtraction. In this way, $\sigma_{l_{i_3}}$ receives one spikes, which is forgotten according to the rule $a \to \lambda$; $\sigma_{l_{i_4}}$ receives three spikes (one from synapse (l_{i_1}, l_{i_4}) and two from synapse (l_{i_2}, l_{i_4})), gets activated, and sends three spikes to σ_{l_k}, which makes neuron l_k activated.

The functioning of the ADD module correctly simulates $l_i : (ADD(r), l_j, l_k)$ by incrementing σ_r with two spikes, followed by nondeterministically activating either σ_{l_j} or σ_{l_k}.

Module SUB: The module for simulating $l_i : (SUB(r), l_j, l_k)$ is given in Fig. 3. In the instruction l_i of M, we have two case, depending if r stores an empty or nonempty value.

Once σ_{l_i} is activated, both neurons r and l_{i_1} receive three spikes.

With three spikes inside, neuron l_{i_1} applies the rule $a^3/a \to a$ first, and then the rule $a^2 \to a^2$. As a result, the spike trains on synapses (l_{i_1}, l_{i_3}) and (l_{i_1}, l_{i_4}) are aa^2, and the computing on these synapses is addition. In this way, both $\sigma_{l_{i_3}}$ and $\sigma_{l_{i_4}}$ receive three spikes.

Now, let's take a look at neuron σ_r.

On one hand, if register r stores a nonempty value $n \geq 1$, this means that initially there are at least two spikes in σ_r. After receiving three spikes from σ_{l_i}, σ_r contains at least five spikes (in general it contains $2n + 3$, $n \geq 1$ spikes). Only the rule $a^5(a^2)^*/a^5 \to a^2$ can be applied by σ_r. Five spikes are removed from σ_r (hence, only $2(n-1)$ spikes remain in σ_r) and two spikes is produced. The removal of five spikes corresponds to decrementing register r by one. Through the neuron

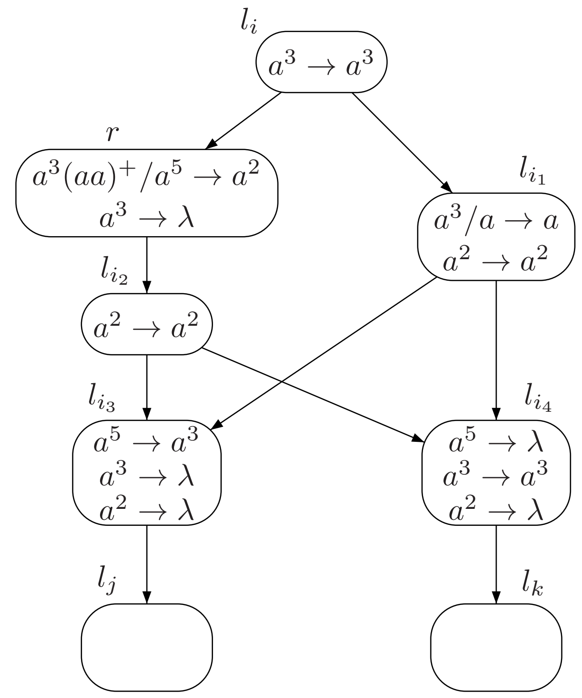


Fig. 3. Module SUB

l_{i_2}, the two spikes arrives at $\sigma_{l_{i_3}}$ and $\sigma_{l_{i_4}}$. Together with the three spikes from $\sigma_{l_{i_1}}$, there are both five spikes in $\sigma_{l_{i_3}}$ and $\sigma_{l_{i_4}}$. The rule $a^5 \to a^3$ is applied by $\sigma_{l_{i_3}}$, and the rule $a^5 \to \lambda$ is applied by $\sigma_{l_{i_4}}$. In this way, three spikes arrives at σ_{l_j}, but not σ_{l_k}. At this point, σ_{l_j} becomes activated.

On the other hand, if register r stores an empty value, this means that initially there is no spike in σ_r. When the three spikes from σ_{l_i} are available in σ_r, only the rule $a^3 \to \lambda$ can be applied. The three spikes are removed (hence, no spike remains in σ_r) and no spike is produced. As a result, there are both three spikes in $\sigma_{l_{i_3}}$ and $\sigma_{l_{i_4}}$. The rule $a^3 \to \lambda$ is applied by $\sigma_{l_{i_3}}$, and the rule $a^3 \to a^3$ is applied by $\sigma_{l_{i_4}}$. In this way, three spikes arrives at σ_{l_k}, but not σ_{l_j}. At this point, σ_{l_k} becomes activated.

We also need to check if there is interference among several *SUB* modules operating on the same σ_r, i.e. when more than one *SUB* instruction operates on register r. However, due to the forgetting rule $a^2 \to \lambda$ in neurons l_{i_3} and l_{i_4}, there is no problem or interference. As shown in Fig. 3, each neuron r sends two spikes to all neurons with label l_{i_2}, then to all neurons with label l_{i_3} and l_{i_4} in the *SUB* module, but all these neurons will forget the two spikes immediately, except for the neurons $\sigma_{l_{i_3}}$ and $\sigma_{l_{i_4}}$ from the module of the *SUB* instruction whose simulation proceeds correctly and which also receives three spikes from the corresponding neuron l_{i_1}.

The functioning of the *SUB* module correctly simulates the $l_i : (SUB(r), l_j, l_k)$ operation by either decrementing σ_r and activating σ_{l_j}, otherwise by activating σ_{l_k}. What remains now is to output the result of the computation.

Module OUTPUT: The module for halting the computation and producing the output is given in Fig. 4. Since the output register 1 is never decremented, this means that no *SUB* module operates on σ_1, and the number of spikes in σ_1 is always of the form $2n$. Once neuron l_h becomes activated, it produces a single spike so that σ_1 becomes activated at the next step.

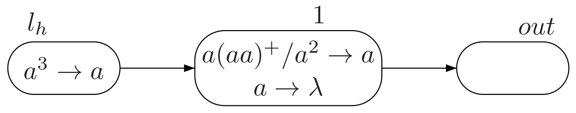


Fig. 4. Module OUTPUT

If register 1 is nonempty, rule $a(aa)^+/a^2 \to a$ in σ_1 is applied since σ_1 now has $2n + 1$ spikes. This rule consumes two spikes, and one spike is sent to σ_{out}. The rule will continue to be applied and consume two spikes each step, stopping only when σ_1 has exactly one spike, which is removed by the rule $a \to \lambda$. In this way, the spike train on synapse $(1, out)$ is of the form $aa \ldots a\lambda$, which begins with n number of spikes and is ended with λ, i.e. $n - 1$ pairs of consecutive aa, and then one pair of $a\lambda$. For the $n-1$ pairs of consecutive aa, synapse $(1, out)$ will do nothing, so σ_{out} receives one spike for each computing on synapse, i.e. $n-1$ spikes together. For the last pair of $a\lambda$, synapse will do subtraction, so σ_{out} receives one spike. Hence, the spikes stored in σ_{out} is the generated number, i.e. $(n-1)+1 = n$, which is exactly the number stored in output register 1 of M.

We note that all modules make use of at most three rules in each neuron, with any rule consuming at most five spikes, and forgetting at most five spike. Hence, the parameters of the Theorem are satisfied, and this completes the proof.

5 Final Remarks

In this work, we introduced spiking neural P systems with computing synapses (in short, CSSN P systems). Such systems incorporate not only biological inspiration, e.g. the shunting inhibition, but also computational and mathematical inspirations, e.g. dynamic graphs or time-varying networks.

Our result in this work show that CSSN P systems are computationally universal, even with a normal form. The normal form, as given by parameters in the theorem, includes: at most three rules in each neuron, with any rule consuming at most five spikes, and forgetting at most five spike, and all these rules have no delay.

We suspect that the result provided in this work could still be improved, i.e. improve the normal form. It is likely we can reduce $cons_5$ to $cons_4$ and $forg_5$ to $forg_4$. How to reduce these and other parameters in the system, using CSSN P semantics, remains open.

Another open problem is to consider more complicated synapse, e.g. synapse computes by multiplication or division. It is also worth considering computing synapses for SN P systems with anti-spikes.

Acknowledgments.

This work was supported by National Natural Science Foundation of China (61502063) and Chongqing Social Science Planning Project (2017YBGL142).

References

1. Ionescu, M., Păun, Gh., Yokomori, T.: Spiking Neural P Systems. Fund. Inform. 71, 279–308 (2006)
2. Cavaliere, M., Ibarra, O.H., Păun, Gh., Egecioglu, O., Ionescu, M., Woodworth, S.: Asynchronous Spiking Neural P Systems. Theor. Comput. Sci. 410, 2352–2364 (2009)
3. Pan, L., Păun, Gh.: Spiking Neural P Systems with Anti-spikes. Int. J. Comput. Commun. IV, 273–282 (2009)
4. Ionescu, M., Păun, Gh., Pérez-Jiménez, M.J., Yokomori, T.: Spiking Neural dP Systems. Fund. Inform. 111, 423–436 (2011)
5. Pan, L., Wang, J., Hoogeboom, H.J.: Spiking Neural P Systems with Astrocytes. Neural. Comput. 24, 805–825 (2012)
6. Pan, L., Zeng, X., Zhang, X., Jiang, Y.: Spiking Neural P Systems with Weighted Synapses. Neural. Process. Lett. 35, 13–27 (2012)
7. Song, T., Pan, L., Păun, Gh.: Spiking Neural P Systems with Rules on Synapses. Theor. Comput. Sci. 529, 82–95 (2014)
8. Wang, J., Hoogeboom, H.J., Pan, L., Păun, Gh., Pérez-Jiménez, M.J.: Spiking Neural P Systems with Weights. Neural. Comput. 22, 2615–2646 (2014)
9. Song, T., Liu, X., Zeng, X.: Asynchronous Spiking Neural P Systems with Anti-Spikes. Neural. Process. Lett. 42, 633–647 (2015)
10. Song, T., Pan, L.: Spiking Neural P Systems with Rules on Synapses Working in Maximum Spiking Strategy. IEEE. T. Nanobiosci. 14, 465–477 (2015)
11. Song, T., Pan, L.: Spiking Neural P Systems with Rules on Synapses Working in Maximum Spikes Consumption Strategy. IEEE. T. Nanobiosci. 14, 38–44 (2015)
12. Song, T., Gong, F., Liu, X., Zhao, Y., Zhang, X.: Spiking Neural P Systems With White Hole Neurons. IEEE. T. Nanobiosci. 15, 666-673 (2016)
13. Zhao, Y., Liu, X., Wang, W., Adamatzky, A.: Spiking Neural P Systems with Neuron Division and Dissolution. Plos. One. 11, e0162882 (2016)
14. Wu, T., Zhang, Z., Păun, Gh., Pan, L.: Cell-like Spiking Neural P Systems. Theor. Comput. Sci. 623, 180–189 (2016)
15. Jiang, K., Chen, W., Zhang, Y., Pan, L.: Spiking Neural P Systems with Homogeneous Neurons and Synapses. Neurocomputing. 171, 1548–1555 (2016)

16. Song, T., Pan, L.: Spiking Neural P Systems with Request Rules. Neurocomputing. 193, 193–200 (2016)
17. Ibarra, O.H., Păun, A., Rodríguez-Patón, A.: Sequential SNP Systems Based on min/max spike number. Theor. Comput. Sci. 410, 2982–2991 (2009)
18. Neary, T.: A Boundary Between Universality and Non-Universality in Extended Spiking Neural P Systems. Lect. Notes. Comput. Sc. 6031, 475–487 (2009)
19. Song, T., Pan, L., Jiang, K., Song, B., Chen, W.: Normal Forms for Some Classes of Sequential Spiking Neural P Systems. IEEE. T. Nanobiosci. 12, 255–264 (2013)
20. Zhang, X., Zeng, X., Luo, B., Pan, L.: On Some Classes of Sequential Spiking Neural P Systems. Neural. Comput. 26, 974–997 (2014)
21. Wang, X., Song, T., Gong, F., Zheng, P.: On the Computational Power of Spiking Neural P Systems with Self-Organization. Sci. Rep. 6: 27624 (2016)
22. Chen, H., Freund, R., Ionescu, M.: On String Languages Generated by Spiking Neural P Systems. Fund. Inform. 75, 141–162 (2007)
23. Krithivasan, K., Metta, V.P., Garg, D.: On String Languages Generated by Spiking Neural P Systems with Anti-spikes. Int. J. Found. Comput. S. 22, 15–27 (2011)
24. Zeng, X., Xu, L., Liu, X.: On String Languages Generated by Spiking Neural P Systems with Weights. Inform. Sciences. 278, 423–433 (2014)
25. Song, T., Xu, J., Pan, L.: On the Universality and Non-Universality of Spiking Neural P Systems with Rules on Synapses. IEEE. T. Nanobiosci. 14, 960–966 (2015)
26. Wu, T., Zhang, Z., Pan, L.: On Languages Generated by Cell-like Spiking Neural P Systems. IEEE. T. Nanobiosci. 15, 455–467 (2016)
27. Ishdorj, T.-O., Leporati, A., Pan, L., Zeng, X., Zhang, X.: Deterministic Solutions to QSAT and Q3SAT by Spiking Neural P Systems with Pre-computed Resources. Theor. Comput. Sci. 411, 2345–2358 (2010)
28. Pan, L., Păun, Gh., Pérez-Jiménez, M.J.: Spiking Neural P Systems with Neuron Division and Budding. Sci. China. Inform. Sci. 54, 1596–1607 (2011)
29. Song, T., Zheng, P., Wong, M.L., Wang, X.: Design of Logic Gates Using Spiking Neural P Systems with Homogeneous Neurons and Astrocytes-like Control. Inform. Sciences. 372, 380–391 (2016)
30. Díaz-Pernil, D., Gutiérrez-Naranjo, M.J.: Semantics of Deductive Databases with Spiking Neural P Systems. Neurocomputing. DOI:10.1016/j.neucom.2017.07.007
31. Zeng, X., Song, T., Zhang, X., Pan, L.: Performing Four Basic Arithmetic Operations with Spiking Neural P Systems. IEEE. T. Nanobiosci. 11, 366–374 (2012)
32. Liu, X., Li, Z., Liu, J., Liu, L., Zeng, X.: Implementation of Arithmetic Operations with Time-free Spiking Neural P Systems. IEEE. T. Nanobiosci. 14, 617–624 (2015)
33. Wang, J., Shi, P., Peng, H., Pérez-Jiménez, M.J., Wang, T. Weighted Fuzzy Spiking Neural P Systems. IEEE. T. Fuzzy. Syst. 21, 209–220, (2013)
34. Peng, H., Wang, J., Pérez-Jiménez, M.J., Wang, H., Shao, J., Wang, T.: Fuzzy Reasoning Spiking Neural P Systems for Fault Diagnosis. Inform. Sciences. 235, 106–116 (2013)
35. Wang, J., Peng, H.: Adaptive Fuzzy Spiking Neural P Systems for Fuzzy Inference and Learning. Int. J. Comput. Math. 90, 857–868 (2013)
36. Wang, T., Zhang, G., Zhao, J., He, Z., Wang, J., Pérez-Jiménez, M.J.: Fault Diagnosis of Electric Power Systems Based on Fuzzy Reasoning Spiking Neural P Systems. IEEE. T. Power. Syst. 30, 1182–1194 (2015)
37. Zhang, G., Rong, H., Neri, F., Pérez-Jiménez, M.J.: An Optimization Spiking Neural P System for Approximately Solving Combinatorial Optimization Problems. Int. J. Neur. Syst. 24, 1440006 (2014)

38. Mitchell, S.J., Silver, R.J.: Shunting Inhibition Modulates Neuronal Gain during Synaptic Excitation. Neuron. 38, 433–445 (2003)
39. Cabarle, F.G.C., Adorna, H.N., F., Pérez-Jiménez, M.J., Song, T.: Spiking Neural P Systems with Structural Plasticity. Neural. Comput. Appl. 26, 1905–1917 (2015)
40. Song, T., Pan, L.: A Normal Form of Spiking Neural P Systems with Structural Plasticity. Int. J. Swarm Intell. 1, 344–357 (2015)
41. Cabarle, F.G.C., Adorna, H.N., F., Pérez-Jiménez, M.J.: Sequential Spiking Neural P Systems with Structural Plasticity Based on Max/min Spike Number. Neural. Comput. Appl. 27, 1337–1347 (2016)
42. Cabarle, F.G.C., Adorna, H.N., F., Pérez-Jiménez, M.J.: Asynchronous Spiking Neural P Systems with Structural Plasticity. In: Proc. UCNC 2015, pp. 132–143 (2015)

Characterizing PSPACE with Shallow Non-Confluent P Systems

Alberto Leporati, Luca Manzoni, Giancarlo Mauri, Antonio E. Porreca, and Claudio Zandron

Dipartimento di informatica, Sistemistica e Comunicazione
Universit degli Studi di Milano-Bicocca,
Viale Sarca 336, 20126, Milan, Italy
`{leporati, luca.manzoni, mauri, porreca, zandron}@disco.unimib.it`

Summary. In P systems with active membranes, the question of understanding the power of non-confluence within a polynomial time bound is still an open problem. It is known that, for shallow P systems, that is, with only one level of nesting, non-confluence allows them to solve conjecturally harder problems than confluent P systems, thus reaching **PSPACE**. Here we show that **PSPACE** is not only a bound, but actually an exact characterization. Therefore, the power endowed by non-confluence to shallow P systems is equal to the power gained by *confluent* P systems when non-elementary membrane division and polynomial depth are allowed, thus suggesting a connection between the roles of non-confluence and nesting depth.

1 Introduction

While families of *confluent* recognizer P systems with active membranes with charges are known to characterize the complexity class **PSPACE** when working in polynomial time [9, 10], their computational power when the nesting level is constrained to one (i.e., only one level of membranes inside the outermost membrane, usually called *shallow* P systems) is reduced to the class $\mathbf{P}^{\#\mathbf{P}}$, which is conjecturally smaller [1]. While confluent P systems can make use of nondeterminism, they are constrained in returning the same result for all computations starting from the same initial configuration. However, by accepting when at least one computation accepts, like nondeterministic Turing Machines (TM) traditionally do, P systems can make use of the entire power of nondeterminism: uniform families of *non-confluent* recognizer P systems with active membranes with charges can solve **PSPACE**-complete problems even in the shallow case and even when send-in rules are disallowed (i.e., for monodirectional systems) [4]. Here we show that, in fact, **PSPACE** is a characterization of this kind of shallow non-confluent P systems when they work in polynomial time. This result shows that the complex relation between computational power, nesting depth, and monodirectionality present for confluent P systems is absent in the non-confluent case. In particular, in the confluent case, systems with no nesting characterize **P** [11] whereas, additional nesting gives additional power [2] until reaching **PSPACE** when unlimited nesting is allowed [9, 10]. In the monodirectional case even unlimited nesting cannot escape $\mathbf{P}^{\mathbf{NP}}$, which is conjecturally smaller [3]. Non-confluent systems, on the other hand, characterize **NP** when there are no internal membranes [8], and immediately gain the full power of **PSPACE** with only one level of nesting. Furthermore, at least for shallow systems, this provides an exact characterization. It is therefore natural to ask what is the relation between the mechanisms that empower confluent P systems and the full power of non-confluence. Are the former ones only a way to simulate the latter?

2 Basic Notions

For an introduction to membrane computing and the related notions of formal language theory, we refer the reader to *The Oxford Handbook of Membrane Computing* [6]. Here we recall the formal definition of P systems with active membranes using only elementary division rules.

Definition 1. *A* P system with active membranes with elementary division rules *of initial degree $d \geq 1$ is a tuple*

$$\Pi = (\Gamma, \Lambda, \mu, w_{h_1}, \ldots, w_{h_d}, R)$$

where:

- *Γ is an alphabet, i.e., a finite non-empty set of symbols, usually called* objects*;*
- *Λ is a finite set of labels for the membranes;*
- *μ is a membrane structure (i.e., a rooted* unordered *tree, usually represented by nested brackets) consisting of d membranes labelled by elements of Λ in a one-to-one way;*
- *$w_{h_1}, \ldots, w_{h_d}$, with $h_1, \ldots, h_d \in \Lambda$, are multisets (finite sets whose elements have a multiplicity) of objects in Γ, describing the initial contents of the d regions of μ;*
- *R is a finite set of rules.*

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

The rules in R are of the following types:

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

(b) *Send-in communication rules*, of the form $a\,[\;]_h^\alpha \to [b]_h^\beta$
They can be applied to a membrane labelled by h, having charge α and such that the external region contains an occurrence of the object a; the object a is sent into h becoming b and, simultaneously, the charge of h is changed to β.
(c) *Send-out communication rules*, of the form $[a]_h^\alpha \to [\;]_h^\beta\, b$
They can be applied to a membrane labelled by h, having charge α and containing an occurrence of the object a; the object a is sent out from h to the outside region becoming b and, simultaneously, the charge of h becomes β.
(e) *Elementary division rules*, of the form $[a]_h^\alpha \to [b]_h^\beta\, [c]_h^\gamma$
They can be applied to a membrane labelled by h, having charge α, containing an occurrence of the object a but having no other membrane inside (an *elementary membrane*); the membrane is divided into two membranes having label h and charges β and γ; the object a is replaced, respectively, by b and c, while the other objects of the multiset are replicated in both membranes.

The instantaneous *configuration* of a membrane of label h consists of its charge α and the multiset w of objects it contains at a given time. It is denoted by $[w]_h^\alpha$. The *(full) configuration* $\mathcal{C}$ of a P system Π at a given time is a rooted, unordered tree. The root is a node corresponding to the external environment of Π, and has a single subtree corresponding to the current membrane structure of Π. Furthermore, the root is labelled by the multiset located in the environment, and the remaining nodes by the configurations $[w]_h^\alpha$ of the corresponding membranes. In the *initial configuration* of Π, the configurations of the membranes are $[w_{h_1}]_{h_1}^0, \ldots, [w_{h_d}]_{h_d}^0$.

A P system is *shallow* if it contains at most one level of membranes inside the outermost membrane. This means that all the membranes contained in the outermost membrane are elementary, i.e., they contain no other nested membrane.

A computation step changes the current configuration according to the following set of principles:

- Each object and membrane can be subject to at most one rule per step, except for object evolution rules: inside each membrane, several evolution rules can be applied simultaneously.
- The application of rules is *maximally parallel*: each object appearing on the left-hand side of evolution, communication, or division rules must be subject to exactly one of them (unless the current charge of the membrane prohibits it). Analogously, each membrane can only be subject to one communication or division rule (types (b)–(e)) per computation step; these rules will be called *blocking rules* in the rest of the paper. In other words, the only objects and membranes that do not evolve are those associated with no rule, or only to rules that are not applicable due to the electrical charges.
- When several conflicting rules can be applied at the same time, a nondeterministic choice is performed; this implies that, in general, multiple possible configurations can be reached after a computation step.
- In each computation step, all the chosen rules are applied simultaneously (in an atomic way). However, in order to clarify the operational semantics, each com-

putation step is conventionally described as a sequence of micro-steps whereby each membrane evolves only after their internal configuration (including, recursively, the configurations of the membrane substructures it contains) has been updated. In particular, before a membrane division occurs, all chosen object evolution rules must be applied inside it; this way, the objects that are duplicated during the division are already the final ones.
- The outermost membrane cannot be divided, and any object sent out from it cannot re-enter the system again.

A *halting computation* of the P system Π is a finite sequence $\boldsymbol{\mathcal{C}} = (\mathcal{C}_0, \ldots, \mathcal{C}_k)$ of configurations, where $\mathcal{C}_0$ is the initial configuration, every $\mathcal{C}_{i+1}$ is reachable from $\mathcal{C}_i$ via a single computation step, and no rules of Π are applicable in $\mathcal{C}_k$. A *non-halting* computation $\boldsymbol{\mathcal{C}} = (\mathcal{C}_i : i \in \mathbb{N})$ consists of infinitely many configurations, again starting from the initial one and generated by successive computation steps, where the applicable rules are never exhausted.

P systems can be used as language *recognisers* by employing two distinguished objects yes and no: in this case we assume that all computations are halting, and that either one copy of object yes or one of object no is sent out from the outermost membrane, and only in the last computation step, in order to signal acceptance or rejection, respectively. If all computations starting from the same initial configuration are accepting, or all are rejecting, the P system is said to be *confluent*. In this paper we deal, however, with *non-confluent* P systems, where multiple computations can have different results and the overall result is established as for nondeterministic TM: it is acceptance iff an accepting computation exists [7].

In order to solve decision problems (or, equivalently, decide languages) over an alphabet Σ, we use *families* of recogniser P systems $\boldsymbol{\Pi} = \{\Pi_x : x \in \Sigma^\star\}$. Each input x is associated with a P system Π_x deciding the membership of x in a language $L \subseteq \Sigma^\star$ by accepting or rejecting. The mapping $x \mapsto \Pi_x$ must be efficiently computable for inputs of any length, as discussed in detail in [5].

Definition 2. *A family of P systems $\boldsymbol{\Pi} = \{\Pi_x : x \in \Sigma^\star\}$ is* (polynomial-time) uniform *if the mapping $x \mapsto \Pi_x$ can be computed by two polynomial-time deterministic Turing machines E and F as follows:*

- *$F(1^n) = \Pi_n$, where n is the length of the input x and Π_n is a common P system for all inputs of length n, with a distinguished input membrane.*
- *$E(x) = w_x$, where w_x is a multiset encoding the specific input x.*
- *Finally, Π_x is simply Π_n with w_x added to its input membrane.*

The family $\boldsymbol{\Pi}$ is said to be (polynomial-time) semi-uniform if there exists a single deterministic polynomial-time Turing machine H such that $H(x) = \Pi_x$ for each $x \in \Sigma^\star$.

Any explicit encoding of Π_x is allowed as output of the construction, as long as the number of membranes and objects represented by it does not exceed the length of the whole description, and the rules are listed one by one. This restriction is enforced in order to mimic a (hypothetical) realistic process of construction of the P systems, where membranes and objects are presumably placed in a constant amount during each construction step, and require actual physical space proportional to their number; see also [5] for further details on the encoding of P systems.

In the following, we denote the class of problems solvable by polynomial-time uniform or semi-uniform families of non-confluent shallow P systems with active membranes with charges by $\mathbf{NPMC}^{[\star]}_{\mathcal{AM}(\text{depth-1},-\text{d},-\text{ne})}$, where $[\star]$ denotes optional semi-uniformity. If no restriction on the depth of the membrane structure is present, but both non-elementary division and dissolution rules are forbidden, then the corresponding class of problems is denoted by $\mathbf{NPMC}^{[\star]}_{\mathcal{AM}(-\text{d},-\text{ne})}$.

3 Nondeterministic Simulation with Oracles

Let $\boldsymbol{\Pi}$ be a semi-uniform family of non-confluent shallow recognizer P systems with active membranes with charges, and let H be the TM of the semi-uniformity condition of $\boldsymbol{\Pi}$. We are going to define a machine M working in polynomial *space* such that on input H and x Turing machine M accepts iff the P system $H(x) = \Pi_x$ of $\boldsymbol{\Pi}$ accepts in polynomial *time*. Notice that a single machine M suffices for all families of P systems. The machine associated with a specific family $\boldsymbol{\Pi}$ of P systems can be obtained by "hard-coding" the input H to M.

First of all, on input H and x, machine M simulates machine H with x as input to obtain a polynomial-size description of Π_x. To simplify the description of the procedure used by machine M to simulate Π_x, we will assume M to work as a nondeterministic polynomial-time TM with access to an oracle for a problem in $\mathbf{NPSPACE} = \mathbf{PSPACE}$. As the following result shows, both this nondeterministic behaviour and the oracle queries can still all be simulated using a polynomial-space deterministic TM.

Proposition 1. $\mathbf{NP}^{\mathbf{NPSPACE}} = \mathbf{PSPACE}$.

Proof. Clearly $\mathbf{NP}^{\mathbf{NPSPACE}} \supseteq \mathbf{PSPACE}$, hence only the opposite inclusion needs to be proved. Let N be a polynomial-time nondeterministic TM with access to an oracle for a language $L \in \mathbf{NPSPACE}$. Let D be a deterministic polynomial space TM built in the following way:

- D simulates N until a query is performed. This simulation, including the nondeterministic choices of N, can be performed in polynomial space by D, since $\mathbf{NP} \subseteq \mathbf{PSPACE}$.
- Since $L \in \mathbf{NPSPACE}$ and $\mathbf{NPSPACE} = \mathbf{PSPACE}$, there exists a deterministic polynomial space TM deciding L that can be simulated by D to answer any query performed by N while still using only a polynomial amount of space. Once a query has been answered, D can resume the simulation of N.

Since D can faithfully simulate N and its oracle queries, D can recognize the same language as N, thus showing that $\mathbf{NP}^{\mathbf{NPSPACE}} \subseteq \mathbf{PSPACE}$, as desired. □

We can now describe how the simulation of Π_x is carried on by M. In the following, we assume that the size of the input x is n, and that each computation of Π_x requires at most T time steps before halting and producing a result. By hypothesis T is polynomial with respect to n.

3.1 Simulation of the Outermost Membrane

The main idea of this construction is to simulate the evolution of the outermost membrane directly by means of a nondeterministic polynomial-time TM. All interactions with the internal membranes are performed via nondeterministic guesses. That is, for each communication rule and for each time step, the number of rules that are applied between the outermost and the inner membranes is guessed in a nondeterministic way. If yes has been sent out by the simulation of the outermost membrane, an oracle query is performed to check whether all performed interactions with the inner membranes were *consistent* with this information, that is, if a computation of the inner membranes able to perform the guessed interactions actually exists. If the query returns a positive answer, then a computation of the entire system actually producing yes exists. In any other case, the simulating machine rejects (since either an invalid simulation of the outermost membrane – and of the P system – was produced, or the simulation itself was correct but the simulated computation was a rejecting one).

To perform this construction we build a table $\mathcal{T}$ indexed by pairs of the form (r, t), where $r \in R$ is either a send-in rule from the outermost membrane to one of the internal membranes or a send-out rule from one of the internal membranes to the outermost membrane, and $t \in \{0, \ldots, T-1\}$ is a time step. The entry $\mathcal{T}(r, t)$ represent the number of times rule r has been applied at the time step t. It is important to notice that table $\mathcal{T}$ can be stored using a polynomial amount of space. In fact, the number of entries is limited by the size of R (which, by uniformity condition, is polynomial in the input size n), and by the number T of time steps needed for the P system to halt. We only need to prove that each entry $\mathcal{T}(r, t)$ can be stored in a polynomial amount of space.

Let $m \in \mathbb{N}$ be number of internal membranes in the initial configuration of Π_x. By the semantics of the rules of P systems, the number of objects sent in to internal membranes or sent out from them after t time steps cannot be greater than $m \times 2^t$, where the second multiplicative factor is the maximum number of membranes per label that can be obtained by membrane division in t time steps. Since this value is exponential in t, it can be represented by a polynomial number of bits with respect to $t \leq T$. Thus, each entry of $\mathcal{T}$ requires at most a polynomial amount of space with respect to n. We denote the maximum value attainable by an entry of $\mathcal{T}$ by K.

Apart from keeping track of the communication rules applied between the outermost and the internal membranes, we also need to assure that all rules are applied in a maximally parallel way. To do so, we define another table $\mathcal{U}$ indexed by pairs of the form (a, t) where $a \in \Gamma$ is an object type and $t \in \{0, \ldots, T-1\}$ is, as before, a time step. The entry $\mathcal{U}(a, t)$ represents the number of objects of type a in the outermost membrane that had no rule applied to them at time t. Table $\mathcal{U}$ can, too, be stored in a polynomial amount of space.

The simulation procedure of the outermost membrane is detailed as Algorithm 1. There, label h always indicates the outermost membrane and the label k an internal membrane label, while $|w|_a$ denotes the number of instances of the object a inside the multiset w. The *applicability* of a rule refers, in the algorithm, to the fact that the indicated membrane must have the correct charge α and, if the rule is *blocking*, that the membrane has not already been used by another blocking rule in the same time step. For example, the condition on line 14 of Algorithm 1 is never verified once another send-out rule has been simulated in a previous iteration of the loop for the current time step.

Lines 1–3 perform the initialization of the algorithm, setting the initial content and charge of the outermost membrane and declaring the environment initially empty. The main simulation loop is performed in lines 4–29. Since the maximum number of time steps needed for Π_x to produce a result is T, the simulation loop is repeated at most T times. If the loop ends without having produced either yes on no in the environment while simultaneously halting, the simulation performed did not correspond to any actual computation of Π_x, thus a negative answer must be produced (line 30).

Lines 5–7 deal with the send-in rules from the outermost membrane to the inner membranes. Since the number of internal membranes where the rule r can be applied is not known, the number is nondeterministically chosen and is bounded by the maximum number of inner membranes and the number of objects of type a in the outermost membrane (line 6). The guessed number of internal membranes saved in table $\mathcal{T}$ and the effect of the rules on the multiset w is scheduled for application (line 7). Notice that, since the state of the internal membranes is not stored, this amounts to the removal of $\mathcal{T}(r, t)$ instances of objects of type a from w.

Lines 8–10 deal with send-out rules from the internal membranes to the outermost membrane. As before, since the configuration and number of the internal membranes is not known, the number of times this rule is applied is nondeterministically guessed (line 9), saved in table $\mathcal{T}$, and the appearance of the corresponding objects of type b in w is scheduled (line 10).

Lines 11–13 perform the simulation of the evolution rules inside the outermost membrane. Since the simulated system is non-confluent, the actual number of applications of each rule is guessed (line 12) before the actual effect of the rule applications are scheduled (line 13).

Lines 14–19 deal with the application of send-out rules from the outermost membrane to the environment. First of all, a nondeterministic guess is performed to decide whether the rule is actually applied (line 15). If so, then the actual effects of the rules are scheduled for application (lines 16–19).

The table $\mathcal{U}$ is then updated to memorize the number of objects that were not subjected to any rule (lines 20–21). This will be used during the query process

```
1  $w \leftarrow$ initial multiset of the outermost membrane;
2  $\mathsf{env} \leftarrow \varnothing$;
3  $\mathsf{charge} \leftarrow 0$;
4  for $t \leftarrow 0$ to $T-1$ do
5      for all applicable $r = a\,[\;]_k^\alpha \to [b]_k^\beta$ do
6          $\mathcal{T}(r,t) \leftarrow$ guess (0, $\min(|w|_a, \mathsf{K})$);
7          mark $\mathcal{T}(r,t)$ instances of $a$ for removal from $w$;
8      for $r = [a]_k^\alpha \to [\;]_k^\beta\, b$ do
9          $\mathcal{T}(r,t) \leftarrow$ guess (0, $\mathsf{K}$);
10         mark $\mathcal{T}(r,t)$ instances of $b$ for insertion in $w$;
11     for all applicable $r = [a \to u]_h^\alpha$ do
12         $m \leftarrow$ guess (0, $|w|_a$);
13         mark $m$ copies of $u$ for addition to $w$ and $m$ copies of $a$ for removal;
14     for all applicable $r = [a]_h^\alpha \to [\;]_h^\beta\, b$ do
15         $m \leftarrow$ guess (0, 1);
16         if $m = 1$ then
17             mark one copy of $a$ for removal from $w$;
18             mark one copy of $b$ for addition in $\mathsf{env}$;
19             mark $\mathsf{charge}$ to be changed from $\alpha$ to $\beta$;
20     for $a \in \Gamma$ do
21         $\mathcal{U}(a,t) \leftarrow$ number of instances of $a$ in $w$ not marked;
22     Apply modifications to $w$, $\mathsf{env}$, and $\mathsf{charge}$ according to the markings;
23     if rule application was not maximally parallel then
24         reject;
25     if $\mathsf{yes}$ or $\mathsf{no}$ has been sent out in the environment then
26         if query $(\mathcal{T}, \mathcal{U}, t)$ answer is positive and no further rules are applicable
             in the next time step then
27             accept or reject accordingly;
28         else
29             reject;
30 reject;
```

Algorithm 1: The nondeterministic algorithm that performs the simulation of the outermost membrane of Π_x.

to ensure that the send-in rules from the outermost membrane to the internal membranes were actually applied in a maximally parallel way.

All the scheduled modifications to the content and charge of the outermost membrane and to the environment are now executed (line 22). If there are irreconcilable problems in the maximally parallel application of the rules then a rejection is performed (lines 23–24). This happens when there were objects in the outermost membrane that were not selected to be sent-in into the internal membranes (this can be checked by looking at table $\mathcal{U}$), nor were they subject to applicable send-out or evolution rules.

Finally, if either yes or no appears in the environment (lines 25–29) then it is necessary to check whenever the guesses performed for the interaction with the internal membranes were accurate and no further rules are applicable in the next time step in the outermost membrane (lines 26–29). If the answer to the query is positive and no further rules were actually applicable, then the simulation can either accept or reject accordingly (line 27). Otherwise, the simulation performed did not correspond to any actual computation of Π_x and we must reject (line 29).

Algorithm 1 can be executed in polynomial time by a nondeterministic TM with access to an oracle to perform the query procedure. In fact, both the outer loop and the inner loops are executed only a polynomial amount of times (either bounded by the time needed for Π_x to halt or by the number or rules in the system). All other operations, including checking the applicability of rules, can be performed in polynomial time given an efficient description of the configuration of the outermost membrane (in which the number of objects is stored in binary). Furthermore, all nondeterministic guesses are of a polynomial amount of bits.

3.2 Simulation of the Oracle

The query that is simulated by means of a nondeterministic machine working in polynomial space is the following one:

> Is there an halting computation of length t of the internal membranes consistent with the rule applications guessed?

To be able to answer this query in nondeterministic polynomial space the main idea is to simulate each membrane sequentially and keep track of the communication rules that are applied while comparing them with the ones guessed by the simulation of the outermost membrane. If division is applied then only the simulation of one of the dividing membranes is immediately carried out (as performing them all at the same time might require exponential – instead of polynomial – space) while the other membrane is pushed into a stack, thus performing a *depth-first simulation* of the membrane hierarchy. This ensures that a polynomial amount of space suffices: it the space needed to simulate one membrane, plus a stack in which the number of elements is at most T, one for each time step. This algorithm is similar to the deterministic one presented in [10], although with an explicit stack instead of a recursive definition, and the further difference that their algorithm was able to work for unbounded-depth system. The actual algorithm implemented to answer the query is presented in Algorithm 2.

Lines 1–3 perform the initial set-up, where a new stack S is filled with the configuration of all internal membranes at the initial time step, i.e., $t = 0$. In particular, for each membrane the multiset of objects contained, label, charge, and time step of the simulation are all pushed as an single record into S.

In the main loop of lines 4–32 the simulation of all internal membranes is performed one at a time. This loop is executed until the stack of membranes to be simulated is not empty, which might require an exponential amount of time.

Once a new membrane to be simulated *starting at time* t_{push} has been extracted (line 5) the simulation of the membrane proceeds up to time step t, which is given

```
$S \leftarrow \varnothing$ ;
for all internal membrane $[w]_k^\alpha$ in the initial configuration do
    push$_S$ $(w, k, \alpha, 0)$ ;
while $S$ is not empty do
    $(w, k, \mathsf{charge}, t_{\text{push}}) \leftarrow$ pop $S$ ;
    for $t' \leftarrow t_{push}$ to $t$ do
        for $r = [a]_k^\alpha \to [b]_k^\beta\ [c]_k^\gamma$ applicable do
            $m \leftarrow$ guess $(0, 1)$;
            if $m = 1$ then
                mark a copy of $a$ for removal, a copy of $b$ for addition to $w$;
                mark charge to be changed to $\beta$;
        for $r = a\ [\ ]_k^\alpha \to [b]_k^\beta$ applicable do
            $m \leftarrow$ guess $(0, 1)$;
            if $m = 1$ then
                $\mathcal{T}(r, t') \leftarrow \mathcal{T}(r, t') - 1$;
                mark a copy of $b$ for addition to $w$;
                mark charge to be changed to $\beta$;
        for $r = [a]_k^\alpha \to [\ ]_k^\beta\ b$ applicable do
            $m \leftarrow$ guess $(0, 1)$;
            if $m = 1$ then
                $\mathcal{T}(r, t') \leftarrow \mathcal{T}(r, t') - 1$;
                mark a copy of $a$ for removal from $w$;
                mark charge to be changed to $\beta$;
        for $r = [a \to u]_k^\alpha$ applicable do
            $m \leftarrow$ guess $(0, |w|_a)$;
            mark $m$ copies of $a$ for removal, $m$ copies of $u$ for addition to $w$;
        apply marked modifications to $w$ and charge;
        if rule application was not maximally parallel then
            reject;
        if division was applied, push$_S$ $(w - \{b\} + \{c\}, k, \gamma, t')$;
    if the current membrane has further applicable rules then
        reject;
if each entry of $\mathcal{T}$ is $0$ then
    accept;
else
    reject;
```

Algorithm 2: The nondeterministic polynomial space algorithm simulating the inner membranes of Π_x.

in input as part the query (loop of lines 6–30) and represents the time at which the simulation of the outermost membrane has suspended in order to perform the query.

In lines 7–11, for each applicable division rule, i.e., the correct object and charge are present and the membrane has not already been used by a blocking rule in this time step, a nondeterministic choice is performed (line 8) to decide if the rule is actually applied. If so (lines 7–11), then the modifications described by the first half of the right-hand-side of the rule are performed, while the other membrane resulting from the division will be pushed on the stack S at the end of the simulation of the current time step (line 30). This cannot be performed earlier since the rewriting rules are applied, by the semantics of rule application in P systems, before the division actually takes place.

The simulation of both send-in and send-out rules (lines 12–17 and lines 18–23, respectively) is performed similarly. Since we are working in a situation of non-confluence, even if a rule is applicable, in order to actually decide whether to apply it, a nondeterministic guess is performed (line 13 and line 19, respectively). In both cases the modifications to be performed to the membrane configuration are scheduled for later execution (lines 16–17 and lines 22–23, respectively). Since send-in and send-out are communication rules between the outermost membrane and the internal membranes, each time one of them is applied the value of $\mathcal{T}(r, t')$ is decremented. If, at the end of the simulation, the number of guessed applications and the real number of applications of the communication rules coincides, all entries $\mathcal{T}(r, t')$ should be 0 (at line 15 and line 21, respectively).

The application of evolution rules (lines 24–26), their effect being limited to the internal state of the membrane, is simpler. As usual, which rules are actually applied is determined by a nondeterministic choice (line 25).

Once all rule applications have been decided, the actual modifications to the state of the membrane are applied (line 27) and, if the rule application was not maximally parallel then the computation rejects (lines 28–29). This can be verified by checking if there still exist objects inside the membrane with applicable rules but no rule was applied to them, or if $\mathcal{U}(a, t')$ is positive for some $a \in \Gamma$ with an applicable send-in rule to the currently simulated membrane. Since $\mathcal{U}(a, t')$ indicates the number of objects that were available for the application of send-in from the outermost membrane but no internal membrane was available, such an inconsistency would denote that the simulation of the internal membranes had no correspondence to the already performed simulation of the outermost membrane.

If a division rule was applied, then the configuration of the second membrane resulting from division is pushed to the stack S (line 30). Here, an instance of the object b has been replaced by an instance of object c and the charge has been changed from β to γ to obtain from the current membrane a copy corresponding to the other one obtained by division.

Before proceeding with the simulation of another membrane, we check that after t steps the computation in this membrane has actually halted (lines 31–32). Otherwise the current computation must reject (line 32).

After the simulation of all internal membranes is finished, i.e., the stack was emptied, a check on the entries of $\mathcal{T}$ is performed. If all and every communication rule application guessed during the simulation of the outermost membrane was actually executed then all entries of $\mathcal{T}$ should be 0. A positive (resp., negative) value for $\mathcal{T}(r,t)$ denotes that less (resp., more) applications of rule r at time t were performed than the number that was guessed.

If at least one accepting computation of the machine simulating the oracle query exists then the answer to the query is positive. Furthermore, if there is a way to "glue" the simulation of the outermost membrane and of the internal membranes, then the result produced by Algorithm 1 was correct. Combining this simulation with the inverse simulation presented in [4], we can then state the main result of the paper.

Theorem 1. $\mathbf{PSPACE} = \mathbf{NPMC}^{[\star]}_{\mathcal{AM}(\mathrm{depth}\text{-}1,-\mathrm{d},-\mathrm{ne})}$. □

As long as no dissolution is allowed, the property of being elementary is a static one and, if no non-elementary division is present, the simulation of the outermost membrane can be extended to include all non-elementary membranes, allowing us to state the following result.

Corollary 1. $\mathbf{PSPACE} = \mathbf{NPMC}^{[\star]}_{\mathcal{AM}(-\mathrm{d},-\mathrm{ne})}$. □

4 Conclusions

We have shown that, differently from confluent P systems, monodirectionality and a restriction on the depth of the system to 1 (or, equivalently, the absence of both dissolution and non-elementary division) do not prevent non-confluent P systems from reaching **PSPACE** in polynomial time. It remains open to establish if this upper bound can be extended to membrane structures of higher (non-constant) depth where non-elementary division is allowed. Since both monodirectionality and nesting depth have a huge influence in the computational power of confluent systems, it would be worthwhile to understand why they do not provide an analogous increase to non-confluent systems. These features are usually employed by algorithms designed for confluent P systems to simulate the power of nondeterminism, so the question is: are they always useless when non-confluence is already present?

References

1. 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)

2. 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)
3. Leporati, A., Manzoni, L., Mauri, G., Porreca, A.E., Zandron, C.: Monodirectional P systems. Natural Computing 15(4), 551–564 (2016)
4. Leporati, A., Manzoni, L., Mauri, G., Porreca, A.E., Zandron, C.: Shallow non-confluent P systems. In: Leporati, A., Rozenberg, G., Salomaa, A., Zandron, C. (eds.) Membrane Computing, 17th International Conference, CMC 2016. Lecture Notes in Computer Science, vol. 10105, pp. 307–316 (2017)
5. Murphy, N., Woods, D.: The computational power of membrane systems under tight uniformity conditions. Natural Computing 10(1), 613–632 (2011)
6. Păun, Gh., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
7. Pérez-Jiménez, M.J., Romero-Jiménez, A., Sancho-Caparrini, F.: Complexity classes in models of cellular computing with membranes. Natural Computing 2(3), 265–284 (2003)
8. Porreca, A.E., Mauri, G., Zandron, C.: Non-confluence in divisionless P systems with active membranes. Theoretical Computer Science 411(6), 878–887 (2010)
9. Sosík, P.: The computational power of cell division in P systems: Beating down parallel computers? Natural Computing 2(3), 287–298 (2003)
10. 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)
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)

Limits on P Systems with Proteins and Without Division

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, ariscosn, lvalencia, marper}@us.es

Summary. In the field of Membrane Computing, computational complexity theory has been widely studied trying to find frontiers of efficiency by means of syntactic or semantical ingredients. The objective of this is to find two kinds of systems, one non-efficient and another one, at least, *presumably* efficient, that is, that can solve **NP**-complete problems in polynomial time, and adapt a solution of such a problem in the former. If it is possible, then **P** = **NP**. Several borderlines have been defined, and new characterizations of different types of membrane systems have been published.

In this work, a certain type of P system, where proteins act as a supporting element for a rule to be fired, is studied. In particular, while division rules, the abstraction of cellular *mitosis* is forbidden, only problems from class **P** can be solved, in contrast to the result obtained allowing them.

Key Words: Membrane Computing, active membranes, proteins, computational complexity theory

1 Introduction

In the beginning, *Membrane Computing* was developed mainly to study certain fields of theoretical computer science, such as formal language theory and computability theory, from a different perspective [12]. In this framework, several models of *P systems*, the main computational device within this framework, have been demonstrated to be universal. For this purpose, the most used technique is to simulate another computationally complete machine, like Turing machines [8] or register machines [2].

Even if it started in this field, the *Membrane Computing* grew rapidly into another fields, such as computational complexity theory [15], biology [16], ecology [5], electrical networks [13] and a wide range of real-life applications [7, 10, 21]. For each field, different variants of *membrane systems* have been developed.

Computational complexity theory is devoted to classify classes of problems depending on their intrinsic complexity, in contrast with algorithms theory, where the complexity measured is that of the algorithm itself. This classification is based on the *resources needed* to solve *efficiently* a problem. In an informal way, we say that a problem is efficiently solved by a machine if the time that this machine takes to solve an instance of length n of such problem is upper bounded by a polynomial $p(n)$.

One of the seven Millenium Prize Problems is the so-called **P** vs. **NP** problem, whose response, whether positive or negative, would have a major impact in several fields such as cryptography, economics, proof theory, even in biology [3]. That is why it seems interesting to study it from a bio-inspired perspective. On the one hand, to demonstrate that a class of P systems is *presumably* efficient it is enough to give an efficient solution to a **NP**-complete problem. On the other hand, various techniques have been developed to show that a class of such devices can only efficiently solve problems from class **P**, such as the *dependency graph technique*, where a directed graph based on the behavior of the system is created from its definition and its resolution is characterized by the `REACHABILITY` problem [1]; the *algorithmic technique*, where an algorithm $\mathcal{A}$ working in polynomial time has as input a *recognizer* P system Π and a multiset m and reproduces a computation of $\Pi + m$ [2]; and the *simulation technique*, where a *recognizer* membrane system is simulated by means of another kind. By this, if Π is a recognizer P system that can solve efficiently problems from the complexity class $\mathcal{C}$, and Π' is another kind of recognizer P system, usually with less "ingredients" than the former, if Π' can simulate Π, then Π' can solve problems from class $\mathcal{C}$. In this work, we use the algorithmic technique to prove that a certain type of P systems cannot solve **NP**-complete problems (unless, of course, $\mathbf{P} = \mathbf{NP}$).

The paper is organized as follows: first of all, in order to make this work self-contained, we introduce some preliminary concepts. Section 3 is devoted to introduce P systems with proteins on membranes. In Section 4, a solution to the open problem from [18] is given, using the algorithmic technique for this purpose. Finally, some conclusions and future research lines are given.

2 Preliminaries

Here, we introduce some concepts that are going to be used through the work.

2.1 Alphabets and multisets

An *alphabet* Γ is a non-empty set and its elements are called *symbols.* A *string* u over Γ is an ordered finite sequence of symbols, that is, a mapping from a natural

[1] It is a problem from class **P**, so its solution can be found in polynomial time.

[2] Let us recall that a *recognizer* P system with a given input is confluent, that is, all its computations return the same answer, so it is enough to reproduce one of them. For a formal definition of *recognizer* membrane systems we refer to [15, 14]

number $n \in \mathbb{N}$ onto Γ. The number n is called the *length* of the string u and it is denoted by $|u|$. The empty string (with length 0) is denoted by λ. The set of all strings over an alphabet Γ is denoted by Γ^*. A *language* over Γ is a subset of Γ^*.

A *multiset* over an alphabet Γ is an ordered pair (Γ, f) where f is a mapping from Γ onto the set of natural numbers $\mathbb{N}$. The *support* of a multiset $m = (\Gamma, f)$ is defined as $supp(m) = \{x \in \Gamma \mid f(x) > 0\}$. A multiset is finite (respectively, empty) if its support is a finite (respectively, empty) set. We denote by $\emptyset$ the empty multiset. Let $m_1 = (\Gamma, f_1)$, $m_2 = (\Gamma, f_2)$ be multisets over Γ, then the union of m_1 and m_2, denoted by $m_1 + m_2$, is the multiset (Γ, g), where $g(x) = f_1(x) + f_2(x)$ for each $x \in \Gamma$. We denote by $M_f(\Gamma)$ the set of all multisets over Γ.

2.2 Graphs and trees

Let us recall some notions related with graph theory (see [6] for details). An *undirected graph* is an ordered pair (V, E) where V is a set whose elements are called nodes or vertices and $E = \{\{x, y\} \mid x \in V, y \in V, x \neq y\}$ whose elements are called *edges*. A *path* of length $k \geq 1$ from a node u to a node v in a graph (V, E) is a finite sequence $(x_0, x_1, \ldots, x_k)$ of nodes such that $x_0 = u$, $x_k = v$ and $\{x_i, x_{i+1}\} \in E$. If $k \geq 2$ and $x_0 = x_k$ then we say that the path is a *cycle* of the graph. A graph with no cycle is said to be *acyclic*. An undirected graph is *connected* if there exist paths between every pair of nodes.

A *rooted tree* is a a connected, acyclic, undirected graph in which one of the vertices (called *the root of the tree*) is distinguished from the others. Given a node x (different from the root), if the last edge on the (unique) path from the root of the tree to the node x is $\{x, y\}$ (in this case, $x \neq y$), then y is **the** *parent* of node x and x is **a** *child* of node y. The root is the only node in the tree with no parent. A node with no children is called a *leaf*.

3 P systems with Proteins on Membranes

The inspiration comes from the biochemistry of living cells, where proteins take part regulating which reactions occur depending on whether certain proteins are present or not [1]. P systems with proteins on membranes, first introduced in [11], have been demonstrated to be universal devices [4], as well as able to solve computationally hard problems. In fact, in [17], a uniform solution to QSAT is given by means of a family of P systems with proteins on membranes and membrane division. Here, we define the syntax and semantics of these systems by adding the necessary elements to introduce cell separation as a method to create an exponential workspace in terms of cells, as it has been used in other variants of P systems [9, 20].

3.1 Syntax

Definition 1. *A P system with proteins on membranes and membrane division of degree $q \geq 1$ is a tuple*

$$\Pi = (\Gamma, \Gamma_0, \Gamma_1, P, P_0, P_1, \mathcal{E}, \mu, \mathcal{M}_1/\mathcal{Z}_1, \dots, \mathcal{M}_q/\mathcal{Z}_q, \mathcal{R}_1, \dots, \mathcal{R}_q, i_{out})$$

where:

- *Γ and P are finite multisets with $\Gamma \cap P = \emptyset$, and $\mathcal{E} \subseteq \Gamma \setminus P$;*
- *$\{\Gamma_0, \Gamma_1\}$ is a partition of the set Γ, where $\Gamma_0 \cup \Gamma_1 = \Gamma$, $\Gamma_0 \cap \Gamma_1 = \emptyset$ and Γ_0, Γ_1 are non-empty sets if separation rules are used, $\Gamma_0 = \Gamma_1 = \emptyset$ otherwise;*
- *$\{P_0, P_1\}$ is a partition of the set P, where $P_0 \cup P_1 = P$, $P_0 \cap P_1 = \emptyset$ and P_0, P_1 are non-empty sets if separation rules are used, $P_0 = P_1 = \emptyset$ otherwise;*
- *μ is a rooted tree;*
- *$\mathcal{M}_1, \dots, \mathcal{M}_q$ are multisets over Γ;*
- *$\mathcal{Z}_1, \dots, \mathcal{Z}_q$ are multisets over P;*
- *$\mathcal{R}_1, \dots, \mathcal{R}_q$ are finite sets of rules associated with the nodes of the graph of the following forms:*
  (1) $[\,p\,|\,a\,]_i \rightarrow [\,p'\,|\,b\,]_i, p, p' \in P, a, b \in \Gamma, 1 \leq i \leq q$ *(in-membrane object evolution rules);*
  (2) $a\,[\,p\,|\,]_i \rightarrow b\,[\,p'\,|\,]_i, p, p' \in P, a, b \in \Gamma, 1 \leq i \leq q$ *(out-membrane object evolution rules);*
  (3) $[\,p\,|\,a\,]_i \rightarrow b\,[\,p'\,|\,]_i, p, p' \in P, a, b \in \Gamma, 1 \leq i \leq q$ *(send-out communication rules);*
  (4) $a\,[\,p\,|\,]_i \rightarrow [\,p'\,|\,b\,]_i, p, p' \in P, a, b \in \Gamma, 1 \leq i \leq q$ *(send-in communication rules);*
  (5) $a\,[\,p\,|\,b\,]_i \rightarrow c\,[\,p'\,|\,d\,]_i, p, p' \in P, a, b, c, d \in \Gamma, 1 \leq i \leq q$ *(antiport communication rules);*
  (6_p) $[\,p\,|\,]_i \rightarrow [\,p'\,|\,]_i[\,p''\,|\,]_i, p, p', p'' \in P, 1 \leq i \leq q, i \neq i_{out}$ *(protein-based division rules)*
  (6_o) $[\,|\,a\,]_i \rightarrow [\,|\,b\,]_i[\,|\,c\,]_i, a, b, c \in \Gamma, 1 \leq i \leq q, i \neq i_{out}$ *(object-based division rules)*
  ($6'_p$) $[\,p\,|\,]_i \rightarrow [\,p'\,|\,]_i[\,p''\,|\,]_i, p, p', p'' \in P, 1 \leq i \leq q, i \neq i_{out}$ *(protein-based division rules)*
  ($6'_o$) $[\,|\,a\,]_i \rightarrow [\,|\,b\,]_i[\,|\,c\,]_i, a, b, c \in \Gamma, 1 \leq i \leq q, i \neq i_{out}$ *(object-based division rules)*
- *$i_{out} = 0$ is the output membrane.*

A P system with proteins on membranes and membrane division (respectively, membrane separation) of degree $q \geq 1$,

$$\Pi = (\Gamma, \Gamma_0, \Gamma_1, P, P_0, P_1, \mathcal{E}, \mu, \mathcal{M}_1/\mathcal{Z}_1, \dots, \mathcal{M}_q/\mathcal{Z}_q, \mathcal{R}_1, \dots, \mathcal{R}_q, i_{out})\ ^3,$$

[3] Let us note that Γ_0, Γ_1, P_0 and P_1 are usually ommited when separation rules are not used

can be viewed as a set of q membranes, labelled by $1, \ldots, q$ arranged in a hierarchical structure μ given by a rooted tree whose root is called the *skin membrane*, such that: (a) $\mathcal{M}_1, \ldots, \mathcal{M}_q$ represent the multisets of objects initially placed in the q membranes of the system; (b) $\mathcal{Z}_1, \ldots, \mathcal{Z}_q$ represent the multisets of proteins initially placed in the q membranes of the system; (c) $\mathcal{E}$ is the set of objects initially located in the environment of the system, all of them available in an arbitrary number of copies; and (d) i_{out} represent a distinguished *region* which will encode the output of the system. We use the term *region* i $(0 \leq i \leq q)$ to refer to membrane i in the case $1 \leq i \leq q$ and to refer to the environment in the case $i = 0$.

A *configuration* at any instant of such kind of P system is described by the membrane structure of the system, the multisets of objects in each membrane, the multisets of proteins in each membrane and the multiset of objects over $\Gamma \setminus \mathcal{E}$ in the environment at the moment. The initial configuration of $\Pi = (\Gamma, \Gamma_0, \Gamma_1, P, P_0, P_1, \mathcal{E}, \mu, \mathcal{M}_1/\mathcal{Z}_1, \ldots, \mathcal{M}_q/\mathcal{Z}_q, \mathcal{R}_1, \ldots, \mathcal{R}_q, i_{out})$ is $(\mu, \mathcal{M}_1/\mathcal{Z}_1, \ldots, \mathcal{M}_q/\mathcal{Z}_q; \emptyset)$.

3.2 Semantics

An in-membrane object evolution rule $[\,p\,|\,a\,]_i \rightarrow [\,p'\,|\,b\,]_i \in \mathcal{R}_i$ is *applicable* at a configuration $\mathcal{C}_t$ at an instant t if there is a region i from $\mathcal{C}_t$ which contains the object a and the protein p. By applying such rule, object a and protein p in region i from $\mathcal{C}_t$ are consumed and object b and protein p' are generated in region i from $\mathcal{C}_{t+1}$.

An out-membrane object evolution rule $a\,[\,p\,|\,]_i \rightarrow b\,[\,p'\,|\,]_i \in \mathcal{R}_i$ is *applicable* at a configuration $\mathcal{C}_t$ at an instant t if there is a region $p(i)$ from $\mathcal{C}_t$ which contains the object a and a region i from $\mathcal{C}_t$ which contains the protein p. By applying such rule, object a in region $p(i)$ and protein p in region i from $\mathcal{C}_t$ are consumed and object b is generated in region $p(i)$ and protein p' is generated in region i from $\mathcal{C}_{t+1}$.

A send-out communication rule $[\,p\,|\,a\,]_i \rightarrow b\,[\,p'\,|\,]_i \in \mathcal{R}_i$ is *applicable* at a configuration $\mathcal{C}_t$ at an instant t if there is a region i from $\mathcal{C}_t$ which contains the object a and the protein p. By applying such rule, object a and protein p in region i from $\mathcal{C}_t$ are consumed and object b is generated in region $p(i)$ and protein p' is generated in region i from $\mathcal{C}_{t+1}$.

A send-in communication rule $a\,[\,p\,|\,]_i \rightarrow [\,p'\,|\,b\,]_i \in \mathcal{R}_i$ is *applicable* at a configuration $\mathcal{C}_t$ at an instant t if there is a region $p(i)$ from $\mathcal{C}_t$ which contains the object a and a region i from $\mathcal{C}_t$ which contains the protein p. By applying such rule, object a in region $p(i)$ and protein p in region i from $\mathcal{C}_t$ are consumed and object b and protein p' are generated in region i from $\mathcal{C}_{t+1}$.

An antiport communication rule $a\,[\,p\,|\,b\,]_i \rightarrow c\,[\,p'\,|\,d\,]_i \in \mathcal{R}_i$ is *applicable* at a configuration $\mathcal{C}_t$ at an instant t if there is a region $p(i)$ from $\mathcal{C}_t$ which contains the object a and a region i from $\mathcal{C}_t$ which contains the object b and the protein p. By applying such rule, object a in region $p(i)$ and object b and protein p in region i

from $\mathcal{C}_t$ are consumed and object c is generated in region $p(i)$ and object d and protein p' are generated in region i from $\mathcal{C}_{t+1}$.

A protein-based division rule $[\,p\,|\,]_i \rightarrow [\,p'\,|\,]_i[\,p''\,|\,]_i \in \mathcal{R}_i$ is *applicable* at a configuration $\mathcal{C}_t$ at an instant t if there is a region i from $\mathcal{C}_t$ which contains the protein p. By applying such rule, protein p in region i from $\mathcal{C}_t$ is consumed, two new membranes with label i are generated at configuration $\mathcal{C}_{t+1}$ and objects and proteins from the original membrane are duplicated in both new membranes, except protein p that evolves in a protein p' that goes to one of the new membranes, and a protein p'' that goes to the other one.

An object-based division rule $[\,|\,a\,]_i \rightarrow [\,|\,b\,]_i[\,|\,c\,]_i \in \mathcal{R}_i$ is *applicable* at a configuration $\mathcal{C}_t$ at an instant t if there is a region i from $\mathcal{C}_t$ which contains the object a. By applying such rule, object a in region i from $\mathcal{C}_t$ is consumed, two new membranes with label i are generated at configuration $\mathcal{C}_{t+1}$ and objects and proteins from the original membrane are duplicated in both new membranes, except object a that evolves in an object b that goes to one of the new membranes, and an object c that goes to the other one.

A protein-based separation rule $[\,p\,|\,]_i \rightarrow [\,P_0\,|\,\Gamma_0\,]_i[\,P_1\,|\,\Gamma_1\,]_i \in \mathcal{R}_i$ is *applicable* at a configuration $\mathcal{C}_t$ at an instant t if there is a region i from $\mathcal{C}_t$ which contains the protein p. By applying such rule, protein p in region i from $\mathcal{C}_t$ is consumed, two new membranes with label i are generated at configuration $\mathcal{C}_{t+1}$ and objects and proteins from the original membrane are distributed in both new membranes, proteins in P_0 and objects in Γ_0 go to one of the new membranes and proteins in P_1 and objects in Γ_1 go to the other one.

An object-based separation rule $[\,|\,a\,]_i \rightarrow [\,P_0\,|\,\Gamma_0\,]_i[\,P_1\,|\,\Gamma_1\,]_i \in \mathcal{R}_i$ is *applicable* at a configuration $\mathcal{C}_t$ at an instant t if there is a region i from $\mathcal{C}_t$ which contains the object a. By applying such rule, object a in region i from $\mathcal{C}_t$ is consumed, two new membranes with label i are generated at configuration $\mathcal{C}_{t+1}$ and objects and proteins from the original membrane are distributed in both new membranes, proteins in P_0 and objects in Γ_0 go to one of the new membranes and proteins in P_1 and objects in Γ_1 go to the other one.

It makes no sense in this kind of systems to define the concept *length*, because all the rules have a fixed amount of objects involved in them.

The rules of these systems are applied in a maximally parallel manner, and we have the restriction that when a membrane i is divided or separated at one transition step, then no other rules can be applied for that membrane i at that step.

A transition from a configuration $\mathcal{C}_t$ to another configuration $\mathcal{C}_{t+1}$ is obtained by applying rules in a maximally parallel manner following the previous remarks. A *computation* of the system is a (finite or infinite) sequence of transitions starting from the initial configuration, where any term of the sequence other than the first, is obtained from the previous configuration in one transition step, and it is denoted by $\mathcal{C}_t \Rightarrow_\Pi \mathcal{C}_{t+1}$. If the sequence is finite (called *halting computation*) then the last term of the sequence is a *halting configuration*, that is, a configuration where no rule is applicable to it. A computation gives a result only when an halting configuration is reached, and that result is encoded by the multiset of objects present in the output region i_{out}. A natural framework to solve decision problems is to use recognizer P systems.

Definition 2. *A recognizer P system with proteins on membranes and membrane division/separation of degree $q \geq 1$ is a tuple*

$$\Pi = (\Gamma, \Gamma_0, \Gamma_1, P, P_0, P_1, \Sigma, \mathcal{E}, \mu, \mathcal{M}_1/\mathcal{Z}_1, \dots, \mathcal{M}_q/\mathcal{Z}_q, \mathcal{R}_1, \dots, \mathcal{R}_q, i_{in}, i_{out})$$

where:

- *The tuple $\Pi = (\Gamma, \Gamma_0, \Gamma_1, P, P_0, P_1, \mathcal{E}, \mu, \mathcal{M}_1/\mathcal{Z}_1, \dots, \mathcal{M}_q/\mathcal{Z}_q, \mathcal{R}_1, \dots, \mathcal{R}_q, i_{out})$ is a P system with proteins on membranes and membrane division/separation of degree $q \geq 1$, where Γ strictly contains an (input) alphabet Σ and two distinguished objects* `yes` *and* `no`*, and $\mathcal{M}_i$ $(1 \leq i \leq q)$ are multisets over $\Gamma \setminus \Sigma$;*
- *$i_{in} \in \{1, \dots, q\}$ is the input membrane and i_{out} is the label of the environment;*
- *for each multiset m over the input alphabet Σ, any computation of the system Π with input m starts from the configuration $(\mathcal{M}_1, \dots, \mathcal{M}_{i_{in}} + m, \dots, \mathcal{M}_q; \emptyset)$, always halts and either object* `yes` *ir object* `no` *(but not both) must appear in the environment at the last step.*

Next, we define the concept of solving a problem in a uniform way and in polynomial time by a family of recognizer P systems with proteins on membranes and membrane division/separation.

Definition 3. *A decision problem $X = (I_X, \theta_X)$ is solvable in a* uniform way *and in polynomial time by a family $\mathbf{\Pi} = \{\Pi(n) | n \in \mathbb{N}\}$ of recognizer P systems with proteins on membranes and membrane division/separation. if the following conditions hold:*

- *the family $\mathbf{\Pi}$ is polynomially uniform by Turing machines;*
- *there exists a polynomial encoding (cod, s) of I_X such that: (a) for each instance $u \in I_X$, $s(u)$ is a natural number and $cod(u)$ is an* input multiset *of the system $\Pi(s(u))$; (b) for each $n \in \mathbb{N}$, $s^{-1}(n)$ is a finite set; and (c) the family $\mathbf{\Pi}$ is polynomially bounded, sound and complete with regard to (X, cod, s).*

4 Limits of P systems with proteins on membranes when division rules are not allowed

In [18], an open problem in this framework is given:

Open Problem 5: What is the computational power of families of these P systems without membrane division? Do they characterize the class **P***, and what happens under various restrictions on the form of the rules?*

Here, we obtain a stronger result dealing with P systems with proteins on membranes and membrane separation.

4.1 Representation of P systems with proteins on membranes and membrane separation

Let $\Pi = (\Gamma, \Gamma_0, \Gamma_1, P, P_0, P_1, \Sigma, \mathcal{E}, \mu, \mathcal{M}_1/\mathcal{Z}_1, \ldots, \mathcal{M}_q/\mathcal{Z}_q, \mathcal{R}_1, \ldots, \mathcal{R}_q, i_{in}, i_{out})$ be a recognizer P system with proteins on membranes and membrane separation. We denote by $\mathcal{R}_E$ (resp., $\mathcal{R}_S$) the set of evolution and communication rules (resp., separation rules) of Π. We will fix a total order in $\mathcal{R}_E$ and a total order in $\mathcal{R}_S$. Because several membranes with the same label are generated by using separation rules, in order to identify the different membranes with the same label, the following recursive definition is used to modify the labels of the new generated membranes:

- We denote the label of a membrane as a pair (i, σ), where $1 \leq i \leq q$ and $\sigma \in \{0, 1\}^*$ is a binary string.
- If a separation rule is applied to a membrane with label (i, σ), then the new created membranes will be labelled by $(i, \sigma 0)$ and $(i, \sigma 1)$, respectively. We mention that for the system during any computation, we consider a lexicographical order over the set of labels of membranes.

Note that if evolution or communication rules occur between membranes, the labels of these do not change.

A configuration at an instant t of such kind of P system is described by the multisets of objects over Γ contained in each membrane and the multiset of objects over $\Gamma \setminus \mathcal{E}$ in the environment. Hence, a configuration of Π can be described as follows:

$$\{(a, i, \sigma) \mid a \in \Gamma \cup \{\lambda\}, 1 \leq i \leq q, \sigma \in \{0,1\}^*\} \cup \{(a, 0) \mid a \in \Gamma \setminus \mathcal{E}\}$$

We use LHS and RHS to refer to the left-hand side and the right-hand side of a rule. They are defined in a natural way according to the definition of a rule:

- $[\,p\,|\,a\,]_i \to [\,p'\,|\,b\,]_i \in \mathcal{R}_i$, with $p, p' \in P$ and $a, b \in \Gamma$. Then, we denote by $n \cdot LHS(r, (i, \sigma_i)) = (p, i, \sigma_i)^n (a, i, \sigma_i)^n$, and by $n \cdot RHS(r, (i, \sigma_i)) = (p', i, \sigma_i)^n (b, i, \sigma_i)^n$.
- $r \equiv a\,[\,p\,|\,]_i \to b\,[\,p'\,|\,]_i \in \mathcal{R}_i$, with $p, p' \in P$ and $a, b \in \Gamma$. Then, we denote by $n \cdot LHS(r, (i, \sigma_i)) = (p, i, \sigma_i)^n (a, j, \sigma_j)^n$, and by $n \cdot RHS(r, (i, \sigma_i)) = (p', i, \sigma_i)^n (b, j, \sigma_j)^n$, where (j, σ_j) is the parent membrane of the membrane (i, σ_i).
- $r \equiv [\,p\,|\,a\,]_i \to b\,[\,p'\,|\,]_i \in \mathcal{R}_i$, with $p, p' \in P$ and $a, b \in \Gamma$. Then, we denote by $n \cdot LHS(r, (i, \sigma_i)) = (p, i, \sigma_i)^n (a, i, \sigma_i)^n$, and by $n \cdot RHS(r, (i, \sigma_i)) = (p', i, \sigma_i)^n (b, j, \sigma_j)^n$, where (j, σ_j) is the parent membrane of the membrane (i, σ_i).
- $r \equiv a\,[\,p\,|\,]_i \to [\,p'\,|\,b\,]_i \in \mathcal{R}_i$, with $p, p' \in P$ and $a, b \in \Gamma$. Then, we denote by $n \cdot LHS(r, (i, \sigma_i)) = (p, i, \sigma_i)^n (a, j, \sigma_j)^n$, and by $n \cdot RHS(r, (i, \sigma_i)) = (p', i, \sigma_i)^n (b, i, \sigma_i)^n$, where (j, σ_j) is the parent membrane of the membrane (i, σ_i).

- $r \equiv a\,[\,p\,|\,b\,]_i \rightarrow c\,[\,p'\,|\,d\,]_i \in \mathcal{R}_i$, with $p, p' \in P$ and $a, b, c, d \in \Gamma$. Then, we denote by $n \cdot LHS(r, (i, \sigma_i)) = (p, i, \sigma_i)^n (a, j, \sigma_j)^n (b, i, \sigma_i)^n$, and by $n \cdot RHS(r, (i, \sigma_i)) = (p', i, \sigma_i)^n (c, j, \sigma_j)^n (d, i, \sigma_i)^n$, where (j, σ_j) is the parent membrane of the membrane (i, σ_i).
- $r \equiv [\,p\,|\,]_i \rightarrow [\,P_0\,|\,\Gamma_0\,]_i [\,P_1\,|\,\Gamma_1\,]_i \in \mathcal{R}_i$, with $p \in P$. Then, we denote by $LHS(r, (i, \sigma_i)) = (p, i, \sigma_i)$.
- $r \equiv [\,|\,a\,]_i \rightarrow [\,P_0\,|\,\Gamma_0\,]_i [\,P_1\,|\,\Gamma_1\,]_i \in \mathcal{R}_i$, with $a \in \Gamma$. Then, we denote by $LHS(r, (i, \sigma_i)) = (a, i, \sigma_i)$.

If $\mathcal{C}_t$ is a configuration of Π, then the multiset obtained by replacing in $\mathcal{C}_t$ every occurrence of (x, i, σ) by (x, i, σ') is denoted by $\mathcal{C}_t + \{(x, i, \sigma)/\sigma'\}$. Moreover, we denote by $\mathcal{C}_t + m$ (resp., $\mathcal{C}_t \setminus m$) a multiset m of labelled objects addition to (resp., removal from) the configuration $\mathcal{C}_t$.

Next, we show that P systems with proteins on membranes and membrane separation can only solve tractable problems.

If $\mathcal{C} = (\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_n)$ is a halting computation, then we denote by $|\mathcal{C}| = n + 1$ the length of $\mathcal{C}$. For each i $(1 \leq i \leq q)$, the multiset of objects over Γ contained in all membranes labelled by i at configuration $\mathcal{C}_t$ is denoted by $\mathcal{C}_{t,o}(i)$, and the multiset of proteins over P contained in all membranes labelled by i at configuration $\mathcal{C}_t$ by $\mathcal{C}_{t,p}(i)$. We denote by $\mathcal{C}_t(0)$ the multiset of objects over $\Gamma \setminus \mathcal{E}$ contained in the environment at configuration $\mathcal{C}_t$. We define in a natural way $\mathcal{C}^*_{t,o} = \mathcal{C}_t(0) + \mathcal{C}_{t,o}(1) + \cdots + \mathcal{C}_{t,o}(q)$ and $\mathcal{C}^*_{t,p} = \mathcal{C}_{t,p}(1) + \cdots + \mathcal{C}_{t,p}(q)$. Finally, the finite multiset $\mathcal{C}_t(0) + \mathcal{C}_{t,o}(1) + \mathcal{C}_{t,p}(1) + \cdots + \mathcal{C}_{t,o}(q) + \mathcal{C}_{t,p}(q) = \mathcal{C}^*_{t,o} + \mathcal{C}^*_{t,p}$ is denoted by $\mathcal{C}^*_t$.

Lemma 1. *Let Π be a recognizer P system with proteins on membranes and membrane separation. Let $M = |\mathcal{M}_1 + \cdots + \mathcal{M}_q|$, $Z = |\mathcal{Z}_1 + \cdots + \mathcal{Z}_q|$ and let $\mathcal{C} = (\mathcal{C}_0, \ldots, \mathcal{C}_n)$ be a computation of Π. Then, we have*

1. *$|\mathcal{C}^*_0| = M + Z = S$, and for each t, $0 \leq t \leq n - 1$, $|\mathcal{C}^*_{t+1}| \leq |\mathcal{C}^*_t| + Z$;*
2. *for each t, $0 \leq t \leq n$, $|\mathcal{C}^*_t| \leq S + t \cdot Z$; and*
3. *the number of created membranes along the computation $\mathcal{C}$ by the application of membrane separation rules is bounded by $2M + (2 + n)Z$.*

Proof. (1) Let us notice that $|\mathcal{C}^*_0| = |\mathcal{C}_0(0) + \mathcal{C}_0(1) + \cdots + \mathcal{C}_0(q)| = |\mathcal{M}_1 + \cdots + \mathcal{M}_q| + |\mathcal{Z}_1 + \cdots + \mathcal{Z}_q| = M + Z = S$. Let Π be a recognizer P system with proteins on membranes and membrane separation, $\mathcal{R}_1, \ldots, \mathcal{R}_q$ be the sets of rules associated with Π, which contains the following types of communication rules:

- $[\,p\,|\,a\,]_i \rightarrow [\,p'\,|\,b\,]_i \in \mathcal{R}_i, p, p' \in P, a, b \in \Gamma, 1 \leq i \leq q.$
- $a\,[\,p\,|\,]_i \rightarrow b\,[\,p'\,|\,]_i \in \mathcal{R}_i, p, p' \in P, a, b \in \Gamma, 1 \leq i \leq q.$
- $[\,p\,|\,a\,]_i \rightarrow b\,[\,p'\,|\,]_i \in \mathcal{R}_i, p, p' \in P, a, b \in \Gamma, 1 \leq i \leq q.$
- $a\,[\,p\,|\,]_i \rightarrow [\,p'\,|\,b\,]_i \in \mathcal{R}_i, p, p' \in P, a, b \in \Gamma, 1 \leq i \leq q.$
- $a\,[\,p\,|\,b\,]_i \rightarrow c\,[\,p'\,|\,d\,]_i \in \mathcal{R}_i, p, p' \in P, a, b, c, d \in \Gamma, 1 \leq i \leq q.$

For each t, $0 \leq t \leq n-1$, in the transition from configuration $\mathcal{C}_t$ to configuration $\mathcal{C}_{t+1}$, by using any rule, at least one object and one protein from $\mathcal{C}_t$ is consumed and at most one object and one protein is produced in $\mathcal{C}_{t+1}$. Let us note that by the use of out-membrane object evolution rules, send-in communication rules and antiport communication rules, $p \neq p'$ if $i = i_{skin}$. If $a \in \mathcal{E}$, then a new object is created in the system. So the number of objects created in a single computation step is bounded by Z, that is, the number of proteins present in the system in configuration $\mathcal{C}_t$. By means of separation rules, neither new objects nor proteins are going to appear. Hence, in any transition step the number of objects in the system is increased at most by Z new objects.

(2) By induction on t. Let us start analyzing the basic case $t = 0$. The result is trivial because of $|\mathcal{C}_0^*| = S$. By induction hypothesis, let us suppose the result holds for t, $0 \leq t \leq n-1$. Then $|\mathcal{C}_{t+1}^*| \geq |\mathcal{C}_t^*| + Z$, that is true because of (1), and by induction hypothesis we know that $|\mathcal{C}_t^*| \leq S + t \cdot Z$, so $|\mathcal{C}_{t+1}^*| \leq |\mathcal{C}_t^*| + Z \leq S + t \cdot Z + Z = S + (t+1) \cdot Z$. Hence, the result is also true for $t+1$.

(3) According to the fact that the application of a separation rule consumes an object and produces two new cells, result (3) can be obtained from (2) easily, since the maximum number of separation rules that can be performed in this kind of systems comes defined by the initial multisets of elements. If new objects are created as explained in (2), then at most $n \cdot Z$ new objects can be created in n computation steps, therefore at most $n \cdot Z$ separation rules can be applied by means of these objects. □

Next, a deterministic algorithm $\mathcal{A}$ working in polynomial time is presented, which receives as input a P system with proteins on membranes and membrane separation Π and an input multiset m of Π, in such manner that algorithm $\mathcal{A}$ reproduces the behavior of a computation of $\Pi + m$. If the system Π is confluent, then the algorithm $\mathcal{A}$ will provide the same answer of Π. We give the following pseudocode of the algorithm $\mathcal{A}$ to describe the simulation process:

```
  Input: A P system with proteins on membranes and membrane
separation $\Pi$ and an input multiset $m$
  Initialization phase: $\mathcal{C}_0$ is the initial configuration of $\Pi + m$
  $t \leftarrow 0$
  while $\mathcal{C}_t$ is a non-halting configuration do
     Selection phase: Input $\mathcal{C}_t$, Output $(\mathcal{C}'_t, A)$
     Execution phase: Input $(\mathcal{C}'_t, A)$, Output $\mathcal{C}_{t+1}$
     $t \leftarrow t + 1$
  end while
 Output: yes if $\Pi + m$ has an accepting computation, no otherwise
```

The algorithm $\mathcal{A}$ receives a recognizer P system with proteins on membranes and membrane separation

$$\Pi = (\Gamma, \Gamma_0, \Gamma_1, P, P_0, P_1, \mu, \mathcal{M}_1/\mathcal{Z}_1, \ldots, \mathcal{M}_q/\mathcal{Z}_q, \mathcal{E}, \mathcal{R}_1, \ldots, \mathcal{R}_q, i_{out}),$$

where m is an input multiset for this system. Let $M = |\mathcal{M}_1 + \cdots + \mathcal{M}_q|$, $Z = |\mathcal{Z}_1 + \ldots \mathcal{Z}_q|$ and $S = M + Z$. Let any computation of Π perform at most p transition steps, $p \in \mathbb{N}^+$. Hence, from Lemma 1, the number of membranes in the system along any computation is bounded by $2M + (2 + n)Z$.

A transition of a recognizer P system $\Pi + m$ is performed in two phases: selection phase and execution phase.

Selection phase.

```
Input: A configuration $\mathcal{C}_t$ of $\Pi + m$ at an instant $t$
  $\mathcal{C}'_t \leftarrow \mathcal{C}_t$; $A \leftarrow \emptyset$; $B \leftarrow \emptyset$
  for $r \in \mathcal{R}_i \wedge r \in \mathcal{R}_C$, according to the chosen order do
      for each membrane $(i, \sigma_i)$ of $\mathcal{C}'_t$ according
              to the lexicographical order do
         $n_r$ ← maximum number of times that $r$ is applicable to $(i, \sigma_i)$

         if $n_r > 0$ then
            $\mathcal{C}'_t \leftarrow \mathcal{C}'_t - n_r \cdot LHS(r, (i, \sigma_i))$
            $A \leftarrow A \cup \{(r, n_r, (i, \sigma_i))\}$
            $B \leftarrow B \cup \{(i, \sigma_i)\}$
         end if
      end for
  end for
  for $r \in \mathcal{R}_i \wedge r \in \mathcal{R}_S$, according to the chosen order do
      for each $(a, i, \sigma_i)$ according to the lexicographical order,
              and such that $(i, \sigma_i) \notin B$ do
         $\mathcal{C}'_t \leftarrow \mathcal{C}'_t \setminus \{(a, i, \sigma_i)\}$
         $A \leftarrow A \cup \{(r, 1, (i, \sigma_i))\}$
         $B \leftarrow B \cup \{(i, \sigma_i)\}$
      end for
  end for
```

It is easy to check that this algorithm is deterministic and its running time is polynomial in the size of Π because the number of cycles of the first main loop **for** is of the order $O(|\mathcal{R}| \cdot (M^2 + Z^2) \cdot q^2)$; and the number of cycles of the second main loop **for** is of the order $O(|\mathcal{R}| \cdot (M + Z) \cdot q \cdot (|\Gamma| + |P|))$.

Execution phase.

```
Input: The output $(\mathcal{C}'_t, A)$ of the selection phase
  for each $(r, n_r, (i, \sigma_i)) \in A, r \in \mathcal{R}_C$ do
      $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + n_r \cdot RHS(r, (r, \sigma_i))$
  end for
```

```
for each $(r, 1, (i, \sigma_i)) \in A, r \in \mathcal{R}_S$ do
    $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + \{(\lambda, i, \sigma_i)/\sigma_i 0\}$
    $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + \{(\lambda, i, \sigma_i 1)\}$
        for each $(x, i, \sigma_i) \in \mathcal{C}'_t$ according to the lexicographical order do
            if $x \in \Gamma_0$ then
                $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + \{(x, i, \sigma_i)/\sigma_i 0\}$
            else
                $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + \{(x, i, \sigma_i)/\sigma_i 1\}$
            end if
        end for
end for
$\mathcal{C}_{t+1} \leftarrow \mathcal{C}'_t$
```

This algorithm is deterministic and its running time is polynomial in the size of Π because the number of cycles of the first main loop **for** is of the order $O(|\mathcal{R}| \cdot (M^2 + Z^2) \cdot q^2)$; and the number of cycles of the second main loop **for** is of the order $O(|\mathcal{R}| \cdot (M + Z) \cdot q \cdot (|\Gamma| + |P|))$.

Theorem 1. *Only problems from class* **P** *can be solved efficiently by P systems with proteins on membranes and membrane separation.*

Proof. Because the complexity class of recognizer P systems with proteins on membranes and membrane separation is closed under polynomial time reduction and non-empty, **P** is a subset of this class. In what follows, we show the reverse inclusion. Let X a decision problem that can be solved efficiently by recognizer P systems with proteins on membranes and membrane separation and let $\mathbf{\Pi} = \{\Pi(n) \mid n \in \mathbb{N}\}$ be a family of these kinds of P systems solving X according to Definition 3. Let (cod, s) be a polynomial encoding associated with that solution. If $u \in I_X$ is an instance of the problem X, then u will be processed by the system $\Pi(s(u)) + cod(u)$. We consider the following deterministic algorithm $\mathcal{A}'$:

```
Input: An instance $u$ of the problem $X$
  Construct the system $\Pi(s(u)) + cod(u)$
  Run algorithm $\mathcal{A}$ with input $\Pi(s(u)) + cod(u)$
Output: yes if $\Pi(s(u)) + cod(u)$ has an accepting computation,
        no otherwise
```

The algorithm $\mathcal{A}'$ receives an instance u of the decision problem $X = (I_X, \theta_X)$, and working in a polynomial time. The following three assertions are equivalent:

- $\theta_X = 1$, that is, the answer of problem X to instance u is affirmative.
- Every computation of $\Pi(s(u)) + cod(u)$ is an accepting computation.
- The output of the algorithm $\mathcal{A}'$ with input u is yes.

Hence, $X \in \mathbf{P}$. □

5 Conclusions and future work

In this work, P systems with proteins on membranes and separation rules have been studied. While in [19] it has been shown that these systems can solve computationally hard problems using division rules, forbidding them takes from efficiency to non-efficiency. Moreover, even if we add the power of creating an exponential workspace in linear time by means of separation rules, it is shown by the *algorithmic technique* that only problems of the class **P** can be solved efficiently with this kind of systems. This result shows that an exponential workspace in terms of membranes is not enough, but some kind of creation of an exponential workspace of objects [4] is needed, thus a new borderline between efficiency and non-efficiency has been defined.

There are some open problems noted in [18] regarding P systems with proteins on membranes that have not been solved yet, so it seems an interesting research line to work on in order to obtain new frontiers of efficiency. In fact, in the Open Problem 4, the restriction of changing proteins while firing rules is allowed. If not, another frontier of efficiency could be found there.

Acknowledgements

This work was supported by Project TIN2017-89842-P of the Ministerio de Economía y Competitividad of Spain and by Grants No 61320106005 of the National Natural Science Foundation of China.

References

1. B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, P. Walter. Molecular Biology of the Cell. (4th ed.), Garland Science, New York (2002).
2. A. Alhazov, R. Freund, and S. Ivanov. Extended spiking neural P systems with states. *Proceedings of the Fourteenth Brainstorming Week on Membrane Computing*, Seville, Spain, February 1 - 5, 2016. Report RGNC 01/2016, Fénix Editora, 2016, pp. 43–58.
3. J.M. Bahi, W. Bienia, N. Côté, C. Guyeux. Is protein folding problem really a NP-complete one ? First investigations, 2013, `arXiv:1306.1372`.
4. R. Brijder, M. Cavaliere, A. Riscos-Núñez, G. Rozenberg, D. Sburlan, Membrane systems with proteins embedded in membranes. *Theoretical Computer Science*, **404** (2008), Issues 12, 26–39

[4] When division rules are permitted, new objects are created along with the membranes.

5. M. Cardona, M.A. Colomer, M.J. Pérez-Jiménez, 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.
6. T.H. Cormen, C.E. Leiserson, R.L. Rivest. *An Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1994.
7. P. Frisco, M. Gheorghe, M. J. Pérez-Jiménez. 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).
8. M.Á. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez, F.J. Romero-Campero. Characterizing tractability by cell-like membrane systems. *Formal models, languages and applications*, World Scientific, Series in Machine Perception and Artificial Intelligence **66** (2006), chapter 9, pp. 137-154.
9. L. Pan, T.-O. Ishdorj. P systems with active membranes and separation rules. *Journal of Universal Computer Science*, **10**, 5, (2004), 630649.
10. L. Pan, Gh. Păun, M. J. Pérez-Jiménez, 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`).
11. A. Păun, B. Popa. P Systems with Proteins on Membranes and Membrane Division. *Developments in Language Theory*, DLT 2006. Lecture Notes in Computer Science, vol 4036. Springer, Berlin, Heidelberg, 292-303.
12. 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
13. H. Peng, J. Wang, J. Ming, P. Shi, M.J. Pérez-Jiménez, 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`).
14. M.J. Pérez-Jiménez, A. Riscos, A. Romero, D. Woods. Complexity: Membrane division, membrane creation. In Gh. Păun, G. Rozenberg, A. Salomaa (eds.) *The Oxford Handbook of Membrane Computing*, Oxford University Press, Oxford (U.K.), 2009, Chapter 12, pp. 302-336.
15. M.J. Pérez-Jiménez, Á. Romero-Jiménez, F. Sancho-Caparrini. Complexity classes in models of cellular computing with membranes. *Natural Computing*, **2**, 3 (2003), 265–285 (`doi: 10.1023/A:1025449224520`).
16. F.J. Romero-Campero, M.J. Pérez-Jiménez. 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`).
17. B. Song, M.J. Pérez-Jiménez, L. Pan. An efficient time-free solution to QSAT problem using P systems with proteins on membranes. *Information and Computation*, **256** (2017), 287–299.
18. P. Sosík. Attacking Hard Problems beyond NP: A Survey. *Bulletin of the International Membrane Computing Society*, **4**, 89–106.
19. P. Sosík, A. Păun, A. Rodríguez-Patón. P systems with proteins on membranes characterize PSPACE. *Theoretical Computer Science*, **488** (2013), 78–95.

20. L. Valencia-Cabrera, B. Song, L.F. Macías-Ramos, L. Pan, A. Riscos-Núñez, M.J. Pérez-Jiménez. Computational Efficiency of P Systems with Symport/Antiport Rules and Membrane Separation. *Proceedings of the Thirteenth Brainstorming Week on Membrane Computing*, Seville, Spain, February 2 - 6, 2015. Report RGNC 01/2015, Fénix Editora, 2015, pp. 325–370.
21. G. Zhang, M. J. Pérez-Jiménez, 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`).

Narrowing Frontiers of Efficiency with Evolutional Communication Rules and Cell Separation

David Orellana-Martín[1], Luis Valencia-Cabrera[1], Bosheng Song[2], Linqiang Pan[2,3], Mario J. Pérez-Jiménez[1]

[1]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, marper}@us.es
Key [2]Laboratory of Image Information Processing and Intelligent Control of Education Ministry of China, School of Automation, Huazhong University of Science and Technology, Wuhan, Hubei, 430074, China
E-mail: boshengsong@163, lqpan@mail.hust.edu.cn
[3]School of Electric and Information Engineering,
Zhengzhou University of Light Industry, Zhengzhou 450002, China

Summary. In the framework of *Membrane Computing*, several efficient solutions to computationally hard problems have been given. To find new borderlines between families of P systems that can solve them and the ones that cannot is an important way to tackle the **P** versus **NP** problem. Adding syntactic and/or semantic ingredients can mean passing from non-efficiency to *presumably* efficiency. Here, we try to get narrow frontiers, setting the stage to adapt efficient solutions from a family of P systems to another one. In order to do that, a solution to the SAT problem is given by means of a family of tissue P systems with evolutional symport/antiport rules and cell separation with the restriction that both the left-hand side and the right-hand side of the rules have at most two objects.

Key words: Membrane Computing, symport/antiport rules, the **P** versus **NP** problem, SAT problem.

1 Introduction

Membrane Computing is a bio-inspired computing paradigm based on the structure and behavior of living cells. There are several classes of P systems, the computational models of this paradigm. It was first introduced in [7], defining one of the main models, cell-like P systems that abstract the hierarchical arrangement of membranes within a single cell. In [4], the idea of the interactions of networks of cells (placed in the nodes of a directed graph) between cells and between cells and their environment is used to develop tissue-like P systems, named by the ensemble of cells in living beings. Another approach with the same structure are the so-called spiking neural P systems [2], SN P systems for short, inspired by the way that neurons communicate with each other by means of short electrical impulses (spikes).

Within these models, several variants can be defined only by changing syntactic and/or semantic ingredients, such as kinds of rules possible, length of rules, parallelism permitted, number of objects and so on. Computational complexity theory in the framework of Membrane Computing uses special variants of P systems called *recognizer* membrane systems, devices that, given an initial configuration depending on an instance of a decision problem, return *yes* or *no* depending of the answer to such instance. A deep vision of complexity can be seen in [8, 9].

Tissue P systems have been widely investigated from this point of view, giving characterizations for most of their variants. For instance, in [1] and [11], the borderline of efficiency for tissue P systems with symport/antiport rules and cell division by means of the length of communication rules is given, that is, passing from 1 to 2 means passing from non-efficiency to presumably efficiency. In [5] and [10], a similar result is given for tissue P systems with symport/antiport rules and cell separation, but in this case, rules with length at most 3 are needed in order to solve efficiently computationally hard problems. Thus, three frontiers of efficiency can be found here: two described before by means of the length of the rules, and the third one when using rules with length at most 2, between separation and division rules.

In [12], a new variant of these systems is defined. Based on the chemical reactions within cells and how reactives evolve into new components, evolutional communication rules are described as a movement of components between different cells or a cell and the environment but within the reaction objects can change into something new. It is interesting to study these systems from the computational complexity theory point of view, and in [6], an efficient solution to the SAT problem is given by these systems with some restrictions about the length of their rules, but the narrowest borderline is not defined. The purpose of this paper is to tight it.

The paper is organized as follows: first, we recall some concepts that are going to be used through the work. In Section 3 the framework of tissue P systems with evolutional symport/antiport rules is introduced. After that, Sections 4 and 5 are devoted to give a solution to SAT by means of a family of P systems with evolutional symport/antiport rules with cell separation and rules with length at most $(2, 2)$ and a formal verification of a design. Finally, some conclusions and open research lines are exposed.

2 Preliminaries

In order to make this work self-contained, we introduce some notions that are going to be used through the paper.

2.1 Alphabets and sets

An *alphabet* Γ is a non-empty set and their elements are called *symbols*. A *string* u over Γ is an ordered finite sequence of symbols, that is, a mapping from a natural number $n \in \mathbb{N}$ onto Γ. The number n is called the *length* of the string u and it is denoted by $\mid u \mid$. The empty string (with length 0) is denoted by λ. The set of all strings over an alphabet Γ is denoted by Γ^*. A *language* over Γ is a subset of Γ^*.

A *multiset* over an alphabet Γ is an ordered pair (Γ, f) where f is a mapping from Γ onto the set of natural numbers $\mathbb{N}$. The *support* of a multiset $m = (\Gamma, f)$ is defined as $supp(m) = \{x \in \Gamma \mid f(x) > 0\}$. A multiset is finite (resp., empty) if its support is a finite (resp., empty) set. We denote by $\emptyset$ the empty multiset and we denote by $M(\Gamma)$ the set of all multisets over Γ.

Let $m_1 = (\Gamma, f_1)$, $m_2 = (\Gamma, f_2)$ be multisets over Γ, then the union of m_1 and m_2, denoted by $m_1 + m_2$, is the multiset (Γ, g), when $g(x) = f_1(x) + f_2(x)$ for each $x \in \Gamma$.

2.2 Decision problems

A decision problem X can be informally defined as one whose solution is either *yes* or *no*. This can be formally defined by an ordered pair (I_X, θ_X), where I_X is a language over a finite alphabet Σ_X and θ_X is a total Boolean function over I_X. The elements of I_X are called *instances* of the problem X. Each decision problem X has associated a language L_X over the alphabet Σ_X as follows: $L_X = \{u \in E_X \mid \theta_X(u) = 1\}$. Conversely, every language L over an alphabet Σ has associated a decision problem $X_L = (I_{X_L}, \theta_{X_L})$ as follows: $I_{X_L} = \Sigma^*$ and $\theta_{X_L}(u) = 1$ if and only if $u \in L$. Then, given a decision problem X we have $X_{L_X} = X$, and given a language L over an alphabet Σ we have $L_{X_L} = L$.

It is worth pointing out that any Turing machine M (with input alphabet Σ_M) has associated a *decision* problem $X_M = (I_M, \theta_M)$ defined as follows: $I_M = \Sigma_M^*$, and for every $u \in \Sigma_M^*$, $\theta_M(u) = 1$ if and only if M accepts u. Obviously, the decision problem X_M is solvable by the Turing machine.

3 Tissue P systems with evolutional communication rules

Definition 1. *A recognizer tissue P system with evolutional symport/antiport rules and cell separation of degree $q \geq 1$ is a tuple*

$$\Pi = (\Gamma, \Gamma_0, \Gamma_1, \Sigma, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$$

where:

- *Γ and $\mathcal{E}$ are finite alphabets whose elements are called objects;*
- *Γ_0 and Γ_1 is a partition of Γ;*
- *$\mathcal{E} \subseteq \Gamma$;*

- *$\mathcal{M}_q, \dots, \mathcal{M}_q$ are multisets over Γ;*
- *$\mathcal{R}$ is a finite set of rules, of the following forms:*
  1. *Evolutional communication rules:*
     a) *$[\,u\,]_i[\quad]_j \rightarrow [\quad]_i[\,u'\,]_j$, where $1 \leq i, j \leq q$, $i \neq j$, $u \in M_f^+(\Gamma)$ and $u' \in M_f(\Gamma)$ (evolutional symport rules);*
     b) *$[\,u\,]_i[\,v\,]_j \rightarrow [\,v'\,]_i[\,u'\,]_j$, where $1 \leq i, j \leq q$, $i \neq j$, $u, v \in M_f^+(\Gamma)$ and $u', v' \in M_f(\Gamma)$ (evolutional antiport rules);*
  2. *$[\,a\,]_i \rightarrow [\,\Gamma_0\,]_i[\,\Gamma_1\,]_i$, where $i \in \{1, \dots, q\}, i \neq i_{out}$ and $a \in \Gamma$; (separation rules);*
- *$i_{out} \in \{0, 1, \dots, q\}$.*

A recognizer tissue P system with evolutional symport/antiport rules and cell separation of degree $q \geq 1$

$$\Pi = (\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{out})$$

can be viewed as a set of q cells, labelled by $1, \dots, q$ such that (a) $\mathcal{M}_1, \dots, \mathcal{M}_q$ represent the multisets of objects initially placed in the q cells of the system; (b) $\mathcal{E}$ is the set of objects initially located in the environment of the system, all of them available in an arbitrary number of copies; (c) i_{out} represents a distinguished *region* which will encode the output of the system. We use the term *region* i $(0 \leq i \leq q)$ to refer to cell i in the case $1 \leq i \leq q$ and to refer to the environment in the case $i = 0$.

A *configuration* at any instant of a tissue P system with evolutional symport/antiport rules and cell separation is described by the multisets of objects in each cell and the multiset of objects over $\Gamma \setminus \mathcal{E}$ in the environment at that moment. The initial configuration of $\Pi = (\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{out})$ is $\mathcal{M}_1, \dots, \mathcal{M}_q; \emptyset)$.

An evolutional symport rule $[\,u\,]_i[\quad]_j \rightarrow [\quad]_i[\,u'\,]_j$ is applicable at a configuration $\mathcal{C}_t$ at an instant t if there is a region i from $\mathcal{C}_t$ which contains multiset u. By applying an eovlutional symport rule, the multiset of objects in region i from $\mathcal{C}_t$ is consumed and the multiset of objects u' is generated in region j from $\mathcal{C}_{t+1}$.

An evolutional symport rule $[\,u\,]_i[\,v\,]_j \rightarrow [\,v'\,]_i[\,u'\,]_j$ is applicable at a configuration $\mathcal{C}_t$ at an instant t if there is a region i from $\mathcal{C}_t$ which contains multiset u and there is a region j which contains multiset v. By applying an eovlutional symport rule, the multiset of objects u in region i and multiset of objects v in region j from $\mathcal{C}_t$ are consumed and the multiset of objects u' is generated in region j and the multiset of objects v' in region i from $\mathcal{C}_{t+1}$.

A separation rule $[\,a\,]_i \rightarrow [\,\Gamma_0\,]_i[\,\Gamma_1\,]_i$ is applicable at a configuration $\mathcal{C}_t$ at an instant t if there is a cell i from $\mathcal{C}_t$ which contains object a and $i \neq i_{out}$. By applying a separation rule to such a cell i, (a) object a is consumed from such cell; (b) two new cells with label i are generated at configuration $\mathcal{C}_{t+1}$; and (c) objects from Γ_0 from the original cell are placed in one of the new cells, while objects from Γ_1 from the original cell are placed in the other one.

The rules of a tissue P system with evolutional symport/antiport rules and cell separation are applied in a maximally parallel manner, following the previous remarks, and taking into account that when a cell i is being separated at one transition step, no other rules can be applied to that cell i at that step.

A transition from a configuration $\mathcal{C}_t$ to another configuration $\mathcal{C}_{t+1}$ is obtained by applying rules in a maximally parallel manner following the previous remarks. A *computation* of the system is a (finite or infinite) sequence of transitions starting from the initial configuration, where any term of the sequence other than the first one is obtained from the previous configuration in one transition step. If the sequence is finite (called *halting computation*) then the last term of the sequence is a *halting configuration*, that is, a configuration where no rule is applicable to it. A computation gives a result only when a halting configuration is reached, and that result is encoded by the multiset of objects present in the output region i_{out}.

A natural framework to solve decision problems is to use recognizer P systems.

Definition 2. *A recognizer tissue P system with evolutional symport/antiport rules and cell separation of degree $q \geq 1$ is a tuple*

$$\Pi = (\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \Sigma, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{in}, i_{out}),$$

where

- *the tuple $(\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$ is a tissue P system with evolutional symport/antiport rules of degree $q \geq 1$, where Γ strictly contains an (input) alphabet Σ and two distinguished objects* `yes` *and* `no`*, and $\mathcal{M}_i$ $(1 \leq i \leq q)$ are multisets over $\Gamma \setminus \Sigma$;*
- *$i_{in} \in \{1, \ldots, 1\}$ is the input cell and i_{out} is the label of the environment;*
- *for each multiset m over the input alphabet Σ, any computation of the system Π with input m starts from the configuration of the form $(\mathcal{M}_1, \ldots, \mathcal{M}_{i_{in}} + m, \ldots, \mathcal{M}_q; \emptyset)$, it always halts and either object* `yes` *or object* `no` *(but not both) must appear in the environment at the last step.*

For each ordered pair of natural numbers (k_1, k_2) greater or equal to 1, the class of recognizer P systems with evolutional symport/antiport rules and cell separation with evolutional communication rules of length at most (k_1, k_2) is denoted by $\mathbf{TSEC}(k_1, k_2)$. This means that, given an evolutional communication rule $[\,u\,]_i[\,v\,]_j \rightarrow [\,v'\,]_i[\,u'\,]_j$ the LHS (resp., RHS) of any evolutional communication rule in a system from $\mathbf{TSEC}(k_1, k_2)$ involves at most $k_1 = |u| + |v|$ objects (resp., $k_2 = |u'| + |v'|$ objects).

Next, we define the concept of solving a problem in a uniform way and in polynomial time by a family of recognizer tissue P systems with evolutional symport/antiport rules and cell separation.

Definition 3. *A decision problem $X = (I_X, \theta_X)$ is solvable in a uniform way and in polynomial time by a family $\mathbf{\Pi} = \{\Pi(n) \mid n \in \mathbb{N})\}$ of recognizer tissue P systems with evolutional symport/antiport rules and cell separation if the following conditions hold:*

1. *the family $\mathbf{\Pi}$ is polynomially uniform by Turing machines; and*

2. *there exists a polynomial encoding (cod, s) of I_X in $\mathbf{\Pi}$ such that (a) for each instance $u \in I_X$, $s(u)$ is a natural number and $cod(u)$ is an input multiset of the system $\Pi(s(u))$; (b) for each $n \in \mathbb{N}$, $s^{-1}(n)$ is a finite set; and (c) the family $\mathbf{\Pi}$ is polynomially bounded, sound and complete with regard to (X, cod, s).*

The set of all decision problems that can be solved by recognizer tissue P systems with evolutional symport/antiport rules and cell separation with evolutional communication rules of length at most (k_1, k_2) in a uniform way and polynomial time is denoted by $\mathbf{PMC}_{\mathbf{TSEC}(k_1,k_2)}$.

4 Solution to SAT with evolutional communication rules and separation rules

In [6] an efficient solution to the SAT problem is given by means of a family of P systems from $\mathbf{TSEC}(3,2)$. A frontier of efficiency is given, but some open problems remain, as indicate Figure 1 of such work. It shows that the class of problems that can be solved by P systems from $\mathbf{TSEC}(2,k)$ with $k \geq 2$ is unknown. In this work we improve this borderline closing the previous open questions, giving an efficient solution of the SAT problems by means of a family of P systems from $\mathbf{TSEC}(2,2)$.

Let us briefly recall the description of the SAT problem: given a boolean formula in conjunctive normal form (CNF), to determine whether or not there exists an assignment to its variables, called truth assignment, on which it evaluates true.

Theorem 1. SAT $\in \mathbf{PMC}_{\mathbf{TSEC}(2,2)}$

For each $n, p \in \mathbb{N}$, we consider the recognizer P system

$$\Pi(\langle n,p \rangle) = (\Gamma, \Gamma_0, \Gamma_1, \Sigma, \mathcal{E}, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{in}, i_{out})$$

from $\mathbf{TSEC}(2,2)$ defined as follows:

1. Working alphabet Γ:
   $\{\mathtt{yes}, \mathtt{no}, y_1, y_2, n_1, n_2, \#\} \cup$
   $\{a_{i,j} \mid 1 \leq i \leq n, 0 \leq j \leq i\} \cup$
   $\{a'_{i,j} \mid 2 \leq i \leq n, 0 \leq j \leq i-1\} \cup$
   $\{a^L_{i,j}, a^R_{i,j} \mid 2 \leq i \leq n, 1 \leq j \leq i-1\} \cup$
   $\{\alpha_j, \alpha'_j, \alpha^L_j, \alpha^R_j \mid 1 \leq j \leq p+1\} \cup$
   $\{t_i, f_i, t'_i, t''_i f''_i, t^L_i, t^R_i, f^L_i, f^R_i \mid 1 \leq i \leq n\} \cup$
   $\{\beta_{l,k}, \beta'_{l,k}, \beta^L_{l,k}, \beta^R_{l,k} \mid 0 \leq k \leq n, 1 \leq l \leq n\} \cup$
   $\{x_{i,j,k}, \overline{x}_{i,j,k}, x^*_{i,j,k} \mid 1 \leq i \leq n, 1 \leq j \leq p, 1 \leq k \leq n+j-1\} \cup$
   $\{x'_{i,j,k}, \overline{x}'_{i,j,k}, {x^*}'_{i,j,k}, x''_{i,j,k}, \overline{x}''_{i,j,k}, {x^*}''_{i,j,k}, x'''_{i,j,k}, \overline{x}'''_{i,j,k}, {x^*}'''_{i,j,k}, \mid$
   $0 \leq i \leq n, 1 \leq j \leq p, 1 \leq k \leq n\} \cup$
   $\{c_{j,k} \mid 1 \leq j \leq p, j \leq k \leq p\} \cup \{\delta_i \mid 0 \leq i \leq 4n+p+2\} \cup$
   $\{\delta'_i \mid 0 \leq i \leq 4n+p\}$.

2. $\mathit{\Gamma}_1 = \mathit{\Gamma} \setminus \mathit{\Gamma}_0$, $\mathit{\Gamma}_0 = \{a^L_{i,j} \mid 2 \le i \le n, 1 \le j \le i-1\} \cup$
   $\{\alpha^L_j \mid 1 \le j \le p+1\} \cup \{t^L_i, f^L_i \mid 1 \le i \le n\} \cup$
   $\{\beta^L_{l,k} \mid 0 \le k \le n, k+1 \le l \le n\}$
3. Input alphabet $\mathit{\Sigma}$: $\{x_{i,j,0}, \overline{x}_{i,j,0}, x^*_{i,k,0} \mid 1 \le i \le n, 1 \le j \le p\}$.
4. Environment alphabet $\mathcal{E}$: $\{\gamma\}$.
5. $\mathcal{M}_1 = \{\delta_0, \delta'_0\} \cup \{\beta^{n+p+1}_{l,0} \mid 1 \le l \le n\}$,
   $\mathcal{M}_2 = \{a_{i,0} \mid 1 \le i \le n\} \cup \{\alpha_j \mid 1 \le j \le p+1\}$.
6. The set of rules $\mathcal{R}$ consists of the following rules:

1.1 Rules for $(4k+1)$-th steps.

$$[\,a_{i,i-1}\,]_2[\,\gamma\,]_0 \to [\,a'_{i,i-1}\,t'_i\,]_2[\quad]_0 \text{ , for } 1 \le i \le n$$

$$\left.\begin{array}{l}[\,t_i\,]_2[\,\gamma\,]_0 \to [\,t''_i\,]_2[\quad]_0\\ [\,f_i\,]_2[\,\gamma\,]_0 \to [\,f''_i\,]_2[\quad]_0\end{array}\right\}\text{for } 1 \le i \le n$$

$$[\,a_{i,j}\,]_2[\,\gamma\,]_0 \to [\,a'_{i,j}\,]_2[\quad]_0 \text{ , for } 2 \le i \le n, 0 \le j \le i-2$$

$$[\,\alpha_j\,]_2[\,\gamma\,]_0 \to [\,\alpha'_j\,]_2[\quad]_0 \text{ , for } 1 \le j \le p+1$$

$$[\,\beta_{l,k}\,]_1[\,\gamma\,]_0 \to [\,\beta'_{l,k}\,]_1[\quad]_0 \,\}\text{ for } \begin{array}{l}0 \le k \le n,\\ k+1 \le l \le n\end{array}$$

$$\left.\begin{array}{l}[\,x_{i,j,k}\,]_1[\,\gamma\,]_0 \to [\,x'_{i,j,k}\,]_1[\quad]_0\\ [\,\overline{x}_{i,j,k}\,]_1[\,\gamma\,]_0 \to [\,\overline{x}'_{i,j,k}\,]_1[\quad]_0\\ [\,x^*_{i,j,k}\,]_1[\,\gamma\,]_0 \to [\,{x^*}'_{i,j,k}\,]_1[\quad]_0\end{array}\right\}\text{for } \begin{array}{l}1 \le i \le n,\\ 1 \le j \le p,\\ 0 \le k \le n-1\end{array}$$

1.2 Rules for $(4k+2)$-th steps.

$$\left.\begin{array}{l}[\,a'_{i,i-1}\,]_2[\,\gamma\,]_0 \to [\,a_{i,i}\,f^R_i\,]_2[\quad]_0\\ [\,t'_i\,]_2[\,\gamma\,]_0 \to [\,t^L_i\,]_2[\quad]_0\end{array}\right\}\text{for } 1 \le i \le n$$

$$\left.\begin{array}{l}[\,t''_i\,]_2[\,\gamma\,]_0 \to [\,t^L_i\,t^R_i\,]_2[\quad]_0\\ [\,f''_i\,]_2[\,\gamma\,]_0 \to [\,f^L_i\,f^R_i\,]_2[\quad]_0\end{array}\right\}\text{for } 1 \le i \le n$$

$$[\,a'_{i,j}\,]_2[\,\gamma\,]_0 \to [\,a^L_{i,j+1}\,a^R_{i,j+1}\,]_2[\quad]_0 \text{ , for } \begin{array}{l}2 \le i \le n,\\ 0 \le j \le i-1\end{array}$$

$$[\,\alpha'_j\,]_2[\,\gamma\,]_0 \to [\,\alpha^L_j\,\alpha^R_j\,]_2[\quad]_0 \text{ , for } 1 \le j \le p+1$$

$$[\,\beta'_{l,k}\,]_1[\,\gamma\,]_0 \to [\,\beta^L_{l,k+1}\,\beta^R_{l,k+1}\,]_1[\quad]_0 \text{ , for } \begin{array}{l}0 \le k \le n,\\ k+1 \le l \le n\end{array}$$

$$\left.\begin{array}{l}[\,x'_{i,j,k}\,]_1[\,\gamma\,]_0 \to [\,x''^2_{i,j,k+1}\,]_1[\quad]_0\\ [\,\overline{x}'_{i,j,k}\,]_1[\,\gamma\,]_0 \to [\,\overline{x}''^2_{i,j,k+1}\,]_1[\quad]_0\\ [\,{x^*}'_{i,j,k}\,]_1[\,\gamma\,]_0 \to [\,{x^*}''^2_{i,j,k+1}\,]_1[\quad]_0\end{array}\right\}\begin{array}{l}1 \le i \le n,\\ 1 \le j \le p,\\ 0 \le k \le n-1\end{array}$$

1.3 Rules for $(4k+3)$-th steps.

$$[\,a_{i,i}\,]_2 \to [\,\mathit{\Gamma}_0\,]_2[\,\mathit{\Gamma}_1\,]_2 \text{ , for } 1 \le i \le n$$

$$\left.\begin{array}{l}[\,\beta^O_{k,k}\,]_1[\quad]_0 \to [\quad]_1[\,\beta^O_{k,k}\,]_0\\ [\,\beta^O_{l,k}\,]_1[\quad]_0 \to [\quad]_1[\,\beta_{l,k}\,]_0\end{array}\right\}\text{for } \begin{array}{l}O \in \{L, R\},\\ 1 \le k \le n,\\ k+1 \le l \le n\end{array}$$

$$\left.\begin{array}{l}[\,x''_{i,j,k}\,]_1[\,\gamma\,]_0 \to [\,x'''_{i,j,k}\,]_1[\quad]_0\\ [\,\overline{x}''_{i,j,k}\,]_1[\,\gamma\,]_0 \to [\,\overline{x}'''_{i,j,k}\,]_1[\quad]_0\\ [\,{x^*}''_{i,j,k}\,]_1[\,\gamma\,]_0 \to [\,{x^*}'''_{i,j,k}\,]_1[\quad]_0\end{array}\right\}\text{for } \begin{array}{l}1 \le i \le n,\\ 1 \le j \le p,\\ 1 \le k \le n\end{array}$$

1.4 Rules for $(4k)$-th steps.

$$\left.\begin{array}{l}[\,a^O_{i,j}\,]_2[\,\beta^O_{k,k}\,]_0 \to [\,a_{i,j}\,]_2[\;\;]_0\\ [\,r^O_i\,]_2[\,\beta^O_{k,k}\,]_0 \to [\,r_i\,]_2[\;\;]_0\end{array}\right\}\text{for}\begin{array}{c}O\in\{L,R\},\\ r\in\{t,f\},\\ 1\le i\le n,\\ 1\le j\le n,\\ 1\le k\le n\end{array}$$

$$[\,\alpha^O_j\,]_2[\,\beta^O_{k,k}\,]_0 \to [\,\alpha_j\,]_2[\;\;]_0\ ,\ \text{for}\begin{array}{c}O\in\{L,R\},\\ 1\le j\le p+1,\\ 0\le k\le n\end{array}$$

$$\left.\begin{array}{l}[\,x'''_{i,j,k}\,]_1[\,\gamma\,]_0 \to [\,x_{i,j,k}\,]_1[\;\;]_0\\ [\,\overline{x}'''_{i,j,k}\,]_1[\,\gamma\,]_0 \to [\,\overline{x}_{i,j,k}\,]_1[\;\;]_0\\ [\,x^{*'''}_{i,j,k}\,]_1[\,\gamma\,]_0 \to [\,x^*_{i,j,k}\,]_1[\;\;]_0\end{array}\right\}\text{for}\begin{array}{c}1\le i\le n,\\ 1\le j\le p,\\ 0\le k\le n\end{array}$$

$$[\;\;]_1[\,\beta_{l,k}\,]_0 \to [\,\beta_{l,k}\,]_1[\;\;]_0\ ,\ \text{for } 0\le k\le n, k+1\le l\le n$$

2.1 Rules to check satisfied clauses.

$$\left.\begin{array}{l}[\,t_i\,]_2[\,x_{i,j,n+j-1}\,]_1 \to [\,c_{j,j}\,t_i\,]_2[\;\;]_1\\ [\,t_i\,]_2[\,\overline{x}_{i,j,n+j-1}\,]_1 \to [\,t_i\,]_2[\;\;]_1\\ [\,t_i\,]_2[\,x^*_{i,j,n+j-1}\,]_1 \to [\,t_i\,]_2[\;\;]_1\\ [\,f_i\,]_2[\,x_{i,j,n+j-1}\,]_1 \to [\,f_i\,]_2[\;\;]_1\\ [\,f_i\,]_2[\,\overline{x}_{i,j,n+j-1}\,]_1 \to [\,c_{j,j}\,f_i\,]_2[\;\;]_1\\ [\,f_i\,]_2[\,x^*_{i,j,n+j-1}\,]_1 \to [\,f_i\,]_2[\;\;]_1\end{array}\right\}\text{for } 1\le i\le n, 1\le j\le p$$

$$\left.\begin{array}{l}[\,x_{i,j,n+k}\,]_1[\,\gamma\,]_0 \to [\,x_{i,j,n+k+1}\,]_1[\;\;]_0\\ [\,\overline{x}_{i,j,n+k}\,]_1[\,\gamma\,]_0 \to [\,\overline{x}_{i,j,n+k+1}\,]_1[\;\;]_0\\ [\,x^*_{i,j,n+k}\,]_1[\,\gamma\,]_0 \to [\,x^*_{i,j,n+k+1}\,]_1[\;\;]_0\end{array}\right\}\text{for}\begin{array}{c}1\le i\le n,\\ 1\le j\le p,\\ 0\le k\le j-2\end{array}$$

$$[\,c_{j,k}\,]_2[\,\gamma\,]_0 \to [\,c_{j,k+1}\,]_2[\;\;]_0\ ,\ \text{for } 1\le j\le p, j\le k\le p-1$$

3.1 Rules to check if all clauses are satisfied by a truth assignment.

$$[\,\alpha_{p+1}\,]_2[\,\delta'_{4n+p}\,]_1 \to [\,\alpha'_{p+1}\,]_2[\;\;]_0$$

$$[\,\alpha_j\,c_{j,p}\,]_2[\;\;]_0 \to [\;\;]_2[\,\#\,]_0\ ,\ \text{for } 1\le j\le p$$

4.1 General counters.

$$[\,\delta_i\,]_1[\,\gamma\,]_0 \to [\,\delta_{i+1}\,]_1[\;\;]_0\ ,\ \text{for } 0\le i\le 4n+p+1$$

$$[\,\delta'_{4i+1}\,]_1[\,\gamma\,]_0 \to [\,{\delta'}^2_{4i+2}\,]_1[\;\;]_0\ ,\ \text{for } 0\le i\le n-1$$

$$[\,\delta'_{4i+k}\,]_1[\,\gamma\,]_0 \to [\,\delta'_{4i+k+1}\,]_1[\;\;]_0\ ,\ \text{for } 0\le i\le n-1, k\in\{0,2,3\}$$

$$[\,\delta'_{4n+i}\,]_1[\,\gamma\,]_0 \to [\,\delta'_{4n+i+1}\,]_1[\;\;]_0\ ,\ \text{for } 0\le i\le p-1$$

4.2 Rules to give a negative answer.

$$[\,\alpha_j\,\alpha'_{p+1}\,]_2[\;\;]_0 \to [\;\;]_2[\,n_1\,]_0\ ,\ \text{for } 1\le j\le p$$

$$[\;\;]_2[\,n_1\,]_0 \to [\,n_1\,]_2[\;\;]_0$$

$$[\,n_1\,]_2[\,\delta_{4n+p+2}\,]_1 \to [\,n_2\,]_2[\;\;]_1$$

$$[\,n_2\,]_2[\;\;]_0 \to [\;\;]_2[\,\texttt{no}\,]_0$$

4.3 Rules to give an affirmative answer.

$$[\,\alpha'_{p+1}\,]_2[\,\delta_{4n+p+2}\,]_1 \to [\,y_1\,]_2[\;\;]_1$$

$$[\,y_1\,]_2[\,\gamma\,]_0 \to [\,y_2\,]_2[\;\;]_0$$

$$[\,y_2\,]_2[\;\;]_0 \to [\;\;]_2[\,\texttt{yes}\,]_0$$

7. The input cell is the cell labelled by 1 ($i_{in} = 1$) and the output region is the environment ($i_{out} = env$).

Let $\varphi = C_1 \wedge \cdots \wedge C_p$ an instance of `SAT` problem consisting of p clauses $C_j = l_{j,1} \vee \cdots \vee l_{j,r_j}$, $1 \leq j \leq p$, where $Var(\varphi) = \{x_1, \ldots, x_n\}$, and $l_{j,k} \in \{x_i, \neg x_i \mid 1 \leq i \leq n\}$, $1 \leq j \leq p, 1 \leq k \leq r_j$. Let us assume that the number of variables, n, and the number of clauses, p, of φ, are greater than or equal to 2.

We consider the polynomial encoding (cod, s) from `SAT` in $\mathbf{\Pi}$ defined as follows: for each $\varphi \in I_{\texttt{SAT}}$ with n variables and p clauses, $s(\varphi) = \langle n, p \rangle$ and

$$cod(\varphi) = \{x_{i,j,0} \mid x_i \in C_j\} \cup \{\overline{x}_{i,j,0} \mid \neg x_i \in C_j\} \cup \{x^*_{i,j,0} \mid x_i \notin C_j, \neg x_i \notin C_j\}$$

For instance, the formula $\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee \neg x_4)$ is encoded as follows:

$$cod(\varphi) = \begin{pmatrix} x_{1,1,0} & x_{2,1,0} & \overline{x}_{3,1,0} & x^*_{4,1,0} \\ x^*_{1,2,0} & \overline{x}_{2,2,0} & x^*_{3,2,0} & x_{4,2,0} \\ x^*_{1,3,0} & \overline{x}_{2,3,0} & x_{3,3,0} & \overline{x}_{4,3,0} \end{pmatrix}$$

We define $cod_k(\varphi)$ as the set of elements of $cod(\varphi)$ when the third subscript equals k. In the same way, we define $cod'_k(\varphi)$, $cod''_k(\varphi)$ and $cod'''_k(\varphi)$ as the sets of elements of $cod(\varphi)$ when the third subscript equals k and elements are primed, double primed and triple primed, respectively. For notation convenience, we define $cod^j_k(\varphi)$ the subset of elements of $cod_k(\varphi)$ with elements of $C_j, \ldots, C_p$. For instance, $cod^2_4(\varphi)$ would be the following set:

$$cod^2_4(\varphi) = \begin{pmatrix} x^*_{1,2,4} & \overline{x}_{2,2,4} & x^*_{3,2,4} & x_{4,2,4} \\ x^*_{1,3,4} & \overline{x}_{2,3,4} & x_{3,3,4} & \overline{x}_{4,3,4} \end{pmatrix}$$

The Boolean formula φ will be processed by the system $\Pi(s(\varphi)) + cod(\varphi)$. Next, we informally describe how that system works.

The solution proposed follows a brute force algorithm in the framework of recognizer tissue P systems with separation and evolutional communication rules, and it consists of the following stages:

- *Generation stage*: Using separation rules each 4 steps, we produce 2^n membranes labelled by 2 containing each possible truths assignment. At the same time, we generate 2^n copies of $cod_n(\varphi)$. This stage spends n computation steps exactly, being n the numer of variables of φ.
- *First checking stage*: With rules from **2.1**, we can check which clauses from the input formula φ have been satisfied by a specific truth assignment. This stage takes exactly p steps.
- *Second checking stage*: With rules from **3.1**, we remove objects α_j such that they are removed from a membrane if and only if the truth assignment associated to that membrane makes true clause C_j. This stage takes exactly one step.
- *Output stage*: With rules from **4.2** and **4.3**, we can give an affirmative or a negative answer depending on if the input formula is satisfiable or not. This stage spends exactly 4 steps, regardless of whether the formula is satisfiable or not.

5 A formal verification

In this section, an exhaustive verification of the system is given.

Generation stage

At this stage, all truth assignments for the variables associated with the Boolean formula $\varphi(x_1, \ldots, x_n)$ are going to be generated, by applying separation rules from **1.2** in membranes labelled by 2. In such manner that in the $4i+2$-th step $(1 \leq i \leq n-1)$ of this stage, separation rule associated with an object $a_{i,i}$ is triggered, two new cells distributing t_i and f_i between them. In the last step of this stage, each membrane labelled by 2 will contain a truth assignment of the formula.

Proposition 1. *Let $\mathcal{C} = (\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_q)$ be a computation of the system $\Pi(s(\varphi))$ with input multset $cod(\varphi)$.*

(a_0) *For each $4k$ $(0 \leq k \leq n-1)$ at configuration $\mathcal{C}_{4k}$ we have the following:*
- $\mathcal{C}_{4k}(1) = \{\delta_{4k}, {\delta'}_{4k}^{2^k}, cod_k(\varphi)^{2^k}\} \cup \{\beta_{l,k}^{2^k} \mid k+1 \leq l \leq n\}$
- *There are 2^k membranes labelled by 2 such that each of them contains*
  - *objects $a_{k+1,k}, \ldots, a_{n,k}$;*
  - *objects $r_1, \ldots, r_k$, being $r \in \{t, f\}$; and*
  - *objects $\alpha_1, \ldots, \alpha_{p+1}$.*

(a_1) *For each $4k+1$ $(0 \leq k \leq n-1)$ at configuration $\mathcal{C}_{4k+1}$ we have the following:*
- $\mathcal{C}_{4k+1}(1) = \{\delta_{4k+1}, {\delta'}_{4k+1}^{2^k}, cod'_k(\varphi)^{2^k}\} \cup \{{\beta'}_{l,k}^{2^k} \mid k+1 \leq l \leq n\}$
- *There are 2^k membranes labelled by 2 such that each of them contains*
  - *objects $a'_{k+1,k}, \ldots, a'_{n,k}$;*
  - *objects $r''_1, \ldots, r''_k$, being $r \in \{t, f\}$*
  - *an object t'_{k+1}; and*
  - *objects $\alpha'_1, \ldots, \alpha'_{p+1}$.*

(a_2) *For each $4k+2$ $(0 \leq k \leq n-1)$ at configuration $\mathcal{C}_{4k+2}$ we have the following:*
- $\mathcal{C}_{4k+2}(1) = \{\delta_{4k+2}, {\delta'}_{4k+3}^{2^{k+1}}, cod''_{k+1}(\varphi)^{2^{k+1}}\} \cup$ $\{{\beta^O}_{l,k}^{2^k} \mid O \in \{L, R\}, k+1 \leq l \leq n\}$
- *There are 2^k membranes labelled by 2 such that each of them contains*
  - *objects $a_{k+1,k}, \ldots, a_{n,k}$;*
  - *objects $r_1, \ldots, r_k$, being $r \in \{t, f\}$; and*
  - *objects $\alpha_1, \ldots, \alpha_{p+1}$.*

(a_3) *For each $4k+3$ $(0 \leq k \leq n-1)$ at configuration $\mathcal{C}_{4k+3}$ we have the following:*
- $\mathcal{C}_{4k+3}(0) = \{{\beta^O}_{k+1,k+1}^{2^k}\} \cup \{\beta_{l,k+1}^{2^{k+1}} \mid k+2 \leq l \leq n\}$
- $\mathcal{C}_{4k+3}(1) = \{\delta_{4k+3}, {\delta'}_{4k+3}^{2^{k+1}}, cod'''_{k+1}(\varphi)^{2^{k+1}}\} \cup \{\beta_{l,k}^{2^k} \mid k+1 \leq l \leq n\}$
- *There are 2^{k+1} membranes labelled by 2 such that each of them contains*
  - *objects $a_{k+1,k}, \ldots, a_{n,k}$;*

– *objects* $r_1, \ldots, r_k$, *being* $r \in \{t, f\}$; *and*
– *objects* $\alpha_1, \ldots, \alpha_{p+1}$.

(*b*) $\mathcal{C}_{4n}(1) = \{\delta_{4n}, {\delta'}_{4n}^{2^n}, cod_{4n}(\varphi)^{2^n}\}$, *and there are* 2^n *membranes labelled by 2 such that each of them contains objects* $\alpha_1, \ldots, \alpha_{p+1}$, *as well as a different subset* $\{r_1, \ldots, r_n\}$, *being* $r \in \{t, f\}$.

Proof. (a) is going to be demonstrated by induction on k.

(a_0) The base case $k = 0$ is trivial because at the initial configuration we have: $\mathcal{C}_0(1) = \{\delta_0, \delta'_0, cod_0(\varphi)\} \cup \{\beta_{l,0} \mid 1 \le l \le n\}$ and there exists a single membrane labelled by 2 containing objects $\alpha_1, \ldots, \alpha_{p+1}$ and objects $a_{1,0}, \ldots, a_{n,0}$. Then, configuration $\mathcal{C}_0$ yields configuration $\mathcal{C}_1$ by applying the rules:

$$[\,a_{1,0}\,]_2[\,\gamma\,]_0 \to [\,a'_{1,0}\,t'_1\,]_2[\quad]_0$$
$$[\,a_{i,0}\,]_2[\,\gamma\,]_0 \to [\,a'_{i,0}\ \text{, for } 2 \le i \le n$$
$$[\,\alpha_j\,]_2[\,\gamma\,]_0 \to [\,\alpha'_j\,]_2[\quad]_0\ \text{, for } 1 \le j \le p+1$$
$$[\,\beta_{l,0}\,]_1[\,\gamma\,]_0 \to [\,\beta'_{l,0}\,]_1[\quad]_0\ \text{, for } 1 \le l \le n$$
$$\left.\begin{array}{l} [\,x_{i,j,0}\,]_1[\,\gamma\,]_0 \to [\,x'_{i,j,1}\,]_1[\quad]_0 \\ [\,\overline{x}_{i,j,0}\,]_1[\,\gamma\,]_0 \to [\,\overline{x}'_{i,j,1}\,]_1[\quad]_0 \\ [\,x^*_{i,j,0}\,]_1[\,\gamma\,]_0 \to [\,{x^*}'_{i,j,1}\,]_1[\quad]_0 \end{array}\right\}\text{for } 1 \le i \le n, 1 \le j \le p$$
$$[\,\delta_0\,]_1[\,\gamma\,]_0 \to [\,\delta_1\,]_1[\quad]_0$$
$$[\,\delta'_0\,]_1[\,\gamma\,]_0 \to [\,\delta'_1\,]_1[\quad]_0$$

(a_1) Thus, $\mathcal{C}_1(1) = \{\delta_1, \delta'_1, cod'_1(\varphi)\} \cup \{\beta'_{l,0} \mid 1 \le l \le n\}$ and in $\mathcal{C}_1$ there exists one membrane labelled by 2 such that its contents is the set of objects $\{a'_{1,0}, \ldots, a'_{n,0}\}$, the object t'_1 and objects $\alpha'_1, \ldots, \alpha'_{p+1}$. Then, configuration $\mathcal{C}_1$ yields configuration $\mathcal{C}_2$ by applying the rules:

$$[\,a'_{1,0}\,]_2[\,\gamma\,]_0 \to [\,a_{1,1}\,f_1^R\,]_2[\quad]_0$$
$$[\,t'_1\,]_2[\,\gamma\,]_0 \to [\,t_1^L\,]_2[\quad]_0$$
$$[\,a'_{i,0}\,]_2[\,\gamma\,]_0 \to [\,a^L_{i,1}\,a^R_{i,1}\,]_2[\quad]_0\ \text{,for} 2 \le i \le n$$
$$[\,\alpha'_j\,]_2[\,\gamma\,]_0 \to [\,\alpha_j^L\,\alpha_j^R\,]_2[\quad]_0\ \text{, for } 1 \le j \le p+1$$
$$[\,\beta'_{l,k}\,]_1[\,\gamma\,]_0 \to [\,\beta^L_{l,k+1}\,\beta^R_{l,k+1}\,]_1[\quad]_0\ \text{, for } k+1 \le l \le n$$
$$\left.\begin{array}{l} [\,x'_{i,j,0}\,]_1[\,\gamma\,]_0 \to [\,{x''}^2_{i,j,1}\,]_1[\quad]_0 \\ [\,\overline{x}'_{i,j,0}\,]_1[\,\gamma\,]_0 \to [\,\overline{x}''^2_{i,j,0+1}\,]_1[\quad]_0 \\ [\,{x^*}'_{i,j,0}\,]_1[\,\gamma\,]_0 \to [\,{x^*}''^2_{i,j,0+1}\,]_1[\quad]_0 \end{array}\right\}\text{for } \begin{array}{l}1 \le i \le n, \\ 1 \le j \le p\end{array}$$
$$[\,\delta_1\,]_1[\,\gamma\,]_0 \to [\,\delta_2\,]_1[\quad]_0$$
$$[\,\delta'_1\,]_1[\,\gamma\,]_0 \to [\,{\delta'}_2^2\,]_1[\quad]_0$$

(a_2) Thus, $\mathcal{C}_2(1) = \{\delta_2, {\delta'}_2^2, cod''_1(\varphi)\} \cup \{\beta^O_{l,1} \mid O \in \{L, R\}, 1 \le l \le n\}$ and in $\mathcal{C}_2$ there exists one membrane labelled by 2 such that its contents is the set of objects $\{a_{1,1}, \ldots, a_{n,1}\}$, objects t_1^L and f_1^R and objects $\alpha_1^O, \ldots, \alpha_{p+1}^O$, for $O \in \{L, R\}$. Then, configuration $\mathcal{C}_2$ yields configuration $\mathcal{C}_3$ by applying the rules:

$$[\,a_{1,1}\,]_2 \to [\,\Gamma_0\,]_2[\,\Gamma_1\,]_2\ \text{, for } 1 \le i \le n$$
$$\left.\begin{array}{l} [\,\beta^O_{1,1}\,]_1[\quad]_0 \to [\quad]_1[\,\beta^O_{1,1}\,]_0 \\ [\,\beta^O_{l,1}\,]_1[\quad]_0 \to [\quad]_1[\,\beta_{l,1}\,]_0 \end{array}\right\}\text{for } \begin{array}{l}O \in \{L, R\}, \\ k+1 \le l \le n\end{array}$$

$$\left.\begin{array}{l} [\,x''_{i,j,0}\,]_1[\,\gamma\,]_0 \rightarrow [\,x'''_{i,j,0}\,]_1[\quad]_0 \\ [\,\overline{x}''_{i,j,0}\,]_1[\,\gamma\,]_0 \rightarrow [\,\overline{x}'''_{i,j,0}\,]_1[\quad]_0 \\ [\,x^{*''}_{i,j,0}\,]_1[\,\gamma\,]_0 \rightarrow [\,x^{*'''}_{i,j,0}\,]_1[\quad]_0 \end{array}\right\}\text{for}\begin{array}{l}1 \le i \le n,\\ 1 \le j \le p,\end{array}$$

$$[\,\delta_2\,]_1[\,\gamma\,]_0 \rightarrow [\,\delta_3\,]_1[\quad]_0$$
$$[\,\delta'_2\,]_1[\,\gamma\,]_0 \rightarrow [\,\delta'_3\,]_1[\quad]_0$$

(a_3) Thus, $\mathcal{C}_3(1) = \{\delta_3, {\delta'}^2_3, cod'''_1(\varphi)\}$, at the environment there is the multiset $\{\beta^O_{1,1} \mid O \in \{L,R\}\} \cup \{\beta^2_{l,1} \mid 2 \le l \le n\}$ and in $\mathcal{C}_2$ there exists two membranes labelled by 2 such that its contents is the set of objects $\{a^O_{2,1}, \ldots, a^O_{n,1}\}$ with $O = L$ (resp., $O = R$), object t^L_1 (resp., f^R_1) and objects $\alpha^O_1, \ldots, \alpha^O_{p+1}$, for $O = L$ (resp., $O = R$). Hence, the result holds for $k = 0$

- Supposing that, by induction, result is true for k $(1 \le k \le n-1)$; that is,
  - (a_0) For each $4k$ $(0 \le k \le n-1)$ at configuration $\mathcal{C}_{4k}$ we have the following:
    - $\mathcal{C}_{4k}(1) = \{\delta_{4k}, {\delta'}^{2^k}_{4k}, cod_k(\varphi)^{2^k}\} \cup \{\beta^{2^k}_{l,k} \mid k+1 \le l \le n\}$
    - There are 2^k membranes labelled by 2 such that each of them contains
      - · objects $a_{k+1,k}, \ldots, a_{n,k}$;
      - · objects $r_1, \ldots, r_k$, being $r \in \{t, f\}$; and
      - · objects $\alpha_1, \ldots, \alpha_{p+1}$.
  - (a_1) For each $4k+1$ $(0 \le k \le n-1)$ at configuration $\mathcal{C}_{4k+1}$ we have the following:
    - $\mathcal{C}_{4k+1}(1) = \{\delta_{4k+1}, {\delta'}^{2^k}_{4k+1}, cod'_k(\varphi)^{2^k}\} \cup \{{\beta'}^{2^k}_{l,k} \mid k+1 \le l \le n\}$
    - There are 2^k membranes labelled by 2 such that each of them contains
      - · objects $a'_{k+1,k}, \ldots, a'_{n,k}$;
      - · objects $r''_1, \ldots, r''_k$, being $r \in \{t, f\}$
      - · an object t'_{k+1}; and
      - · objects $\alpha'_1, \ldots, \alpha'_{p+1}$.
  - (a_2) For each $4k+2$ $(0 \le k \le n-1)$ at configuration $\mathcal{C}_{4k+2}$ we have the following:
    - $\mathcal{C}_{4k+2}(1) = \{\delta_{4k+2},, {\delta'}^{2^{k+1}}_{4k+2} cod''_{k+1}(\varphi)^{2^{k+1}}\} \cup \{{\beta^O}^{2^k}_{l,k} \mid O \in \{L,R\}, k+1 \le l \le n\}$
    - There are 2^k membranes labelled by 2 such that each of them contains
      - · objects $a_{k+1,k}, \ldots, a_{n,k}$;
      - · objects $r_1, \ldots, r_k$, being $r \in \{t, f\}$; and
      - · objects $\alpha_1, \ldots, \alpha_{p+1}$.
  - (a_3) For each $4k+3$ $(0 \le k \le n-1)$ at configuration $\mathcal{C}_{4k+3}$ we have the following:
    - $\mathcal{C}_{4k+3}(0) = \{{\beta^O}^{2^k}_{k+1,k+1}\} \cup \{\beta^{2^{k+1}}_{l,k+1} \mid k+2 \le l \le n\}$
    - $\mathcal{C}_{4k+3}(1) = \{\delta_{4k+3}, {\delta'}^{2^{k+1}}_{4k+3}, cod'''_{k+1}(\varphi)^{2^{k+1}}\} \cup \{\beta^{2^k}_{l,k} \mid k+1 \le l \le n\}$
    - There are 2^{k+1} membranes labelled by 2 such that each of them contains
      - · objects $a_{k+1,k}, \ldots, a_{n,k}$;
      - · objects $r_1, \ldots, r_k$, being $r \in \{t, f\}$; and
      - · objects $\alpha_1, \ldots, \alpha_{p+1}$.
- Then, by the induction hypothesis, we want to prove the result for $k+1$.

(a_0) Then, configuration $\mathcal{C}_{4k+3}$ yields configuration $\mathcal{C}_{4(k+1)}$ by applying the rules:

$$\left.\begin{array}{l}[\,a^O_{i,j}\,]_2[\,\beta^O_{k+1,k+1}\,]_0 \to [\,a_{i,j}\,]_2[\quad]_0 \\ [\,r^O_i\,]_2[\,\beta^O_{k+1,k+1}\,]_0 \to [\,r_i\,]_2[\quad]_0\end{array}\right\}\text{for}\begin{array}{l}O \in \{L,R\},\\ r \in \{t,f\},\\ 1 \le i \le n,\\ 1 \le j \le n\end{array}$$

$$[\,\alpha^O_j\,]_2[\,\beta^O_{k+1,k+1}\,]_0 \to [\,\alpha_j\,]_2[\quad]_0\ \text{, for } O \in \{L,R\}, 1 \le j \le p+1$$

$$\left.\begin{array}{l}[\,x'''_{i,j,k+1}\,]_1[\,\gamma\,]_0 \to [\,x_{i,j,k+1}\,]_1[\quad]_0 \\ [\,\overline{x}'''_{i,j,k+1}\,]_1[\,\gamma\,]_0 \to [\,\overline{x}_{i,j,k+1}\,]_1[\quad]_0 \\ [\,x^{*'''}_{i,j,k+1}\,]_1[\,\gamma\,]_0 \to [\,x^*_{i,j,k+1}\,]_1[\quad]_0\end{array}\right\}\text{for } 1 \le i \le n, 1 \le j \le p$$

$$[\quad]_1[\,\beta_{l,k+1}\,]_0 \to [\,\beta_{l,k+1}\,]_1[\quad]_0\ \text{, for } k+2 \le l \le n$$

$$[\,\delta_{4k+3}\,]_1[\,\gamma\,]_0 \to [\,\delta_{4(k+1)}\,]_1[\quad]_0$$

$$[\,\delta'_{4k+3}\,]_1[\,\gamma\,]_0 \to [\,\delta'_{4(k+1)}\,]_1[\quad]_0$$

Therefore, the following holds:

– $\mathcal{C}_{4(k+1)}(1) = \{\delta_{4(k+1)}, {\delta'}^{2^{k+1}}_{4(k+1)}, cod_{k+1}(\varphi)^{2^{k+1}}\} \cup \{\beta^{2^{k+1}}_{l,k+1} \mid k+2 \le l \le n\}$
– There are 2^{k+1} membranes labelled by 2 such that each of them contains
  · objects $a_{k+2,k+1}, \ldots, a_{n,k+1}$;
  · objects $r_1, \ldots, r_{k+1}$, being $r \in \{t,f\}$; and
  · objects $\alpha_1, \ldots, \alpha_{p+1}$.

(a_1) Then, configuration $\mathcal{C}_{4(k+1)}$ yields configuration $\mathcal{C}_{4(k+1)+1}$ by applying the rules:

$$[\,a_{k+1,k}\,]_2[\,\gamma\,]_0 \to [\,a'_{k+1,k}\,t'_{k+1}\,]_2[\quad]_0$$

$$\left.\begin{array}{l}[\,t_i\,]_2[\,\gamma\,]_0 \to [\,t''_i\,]_2[\quad]_0 \\ [\,f_i\,]_2[\,\gamma\,]_0 \to [\,f''_i\,]_2[\quad]_0\end{array}\right\}\text{for } 1 \le i \le k$$

$$[\,a_{i,k+1}\,]_2[\,\gamma\,]_0 \to [\,a'_{i,k+1}\,]_2[\quad]_0\ \text{,for } 2 \le i \le n$$

$$[\,\alpha_j\,]_2[\,\gamma\,]_0 \to [\,\alpha'_j\,]_2[\quad]_0\ \text{, for } 1 \le j \le p+1$$

$$[\,\beta_{l,k+1}\,]_1[\,\gamma\,]_0 \to [\,\beta'_{l,k+1}\,]_1[\quad]_0 \}\ \text{for } k+2 \le l \le n$$

$$\left.\begin{array}{l}[\,x_{i,j,k+1}\,]_1[\,\gamma\,]_0 \to [\,x'_{i,j,k+1}\,]_1[\quad]_0 \\ [\,\overline{x}_{i,j,k+1}\,]_1[\,\gamma\,]_0 \to [\,\overline{x}'_{i,j,k+1}\,]_1[\quad]_0 \\ [\,x^*_{i,j,k+1}\,]_1[\,\gamma\,]_0 \to [\,x^{*'}_{i,j,k+1}\,]_1[\quad]_0\end{array}\right\}\text{for } 1 \le i \le n, 1 \le j \le p$$

$$[\,\delta_{4(k+1)}\,]_1[\,\gamma\,]_0 \to [\,\delta_{4(k+1)+1}\,]_1[\quad]_0$$

$$[\,\delta'_{4(k+1)}\,]_1[\,\gamma\,]_0 \to [\,\delta'_{4(k+1)+1}\,]_1[\quad]_0$$

Therefore, the folowing holds:

– $\mathcal{C}_{4(k+1)+1}(1) = \{\delta_{4(k+1)+1}, {\delta'}^{2^k}_{4(k+1)+1}, cod'_{k+1}(\varphi)^{2^{k+1}}\} \cup \{{\beta'}^{2^k}_{l,k} \mid k+1 \le l \le n\}$
– There are 2^{k+1} membranes labelled by 2 such that each of them contains
  · objects $a'_{k+2,k+1}, \ldots, a'_{n,k+1}$;
  · objects $r''_1, \ldots, r''_{k+1}$, being $r \in \{t,f\}$
  · an object t'_{k+2}; and
  · objects $\alpha'_1, \ldots, \alpha'_{p+1}$.

(a_2) Then, configuration $\mathcal{C}_{4(k+1)+1}$ yields configuration $\mathcal{C}_{4(k+1)+2}$ by applying the rules:

$[\,a'_{k+1,k}\,]_2[\,\gamma\,]_0 \to [\,a_{k+1,k+1}\, f^R_{k+1}\,]_2[\quad]_0$
$[\,t'_{k+1}\,]_2[\,\gamma\,]_0 \to [\,t^L_{k+1}\,]_2[\quad]_0$
$\left.\begin{array}{l}[\,t''_i\,]_2[\,\gamma\,]_0 \to [\,t^L_i\, t^R_i\,]_2[\quad]_0\\ [\,f''_i\,]_2[\,\gamma\,]_0 \to [\,f^L_i\, f^R_i\,]_2[\quad]_0\end{array}\right\}\text{for } 1 \le i \le k$
$[\,a'_{i,k+1}\,]_2[\,\gamma\,]_0 \to [\,a^L_{i,k+2}\, a^R_{i,k+2}\,]_2[\quad]_0$, for $2 \le i \le n$
$[\,\alpha'_j\,]_2[\,\gamma\,]_0 \to [\,\alpha^L_j\, \alpha^R_j\,]_2[\quad]_0$, for $1 \le j \le p+1$
$[\,\beta'_{l,k+1}\,]_1[\,\gamma\,]_0 \to [\,\beta^L_{l,k+2}\, \beta^R_{l,k+2}\,]_1[\quad]_0$, for $k+2 \le l \le n$
$\left.\begin{array}{l}[\,x'_{i,j,k+1}\,]_1[\,\gamma\,]_0 \to [\,x''^2_{i,j,k+2}\,]_1[\quad]_0\\ [\,\overline{x}'_{i,j,k+1}\,]_1[\,\gamma\,]_0 \to [\,\overline{x}''^2_{i,j,k+2}\,]_1[\quad]_0\\ [\,x^{*\prime}_{i,j,k+1}\,]_1[\,\gamma\,]_0 \to [\,x^{*\prime\prime 2}_{i,j,k+2}\,]_1[\quad]_0\end{array}\right\}\begin{array}{l}1 \le i \le n,\\ 1 \le j \le p\end{array}$
$[\,\delta_{4(k+1)+1}\,]_1[\,\gamma\,]_0 \to [\,\delta_{4(k+1)+2}\,]_1[\quad]_0$
$[\,\delta'_{4(k+1)+1}\,]_1[\,\gamma\,]_0 \to [\,\delta'^2_{4(k+1)+2}\,]_1[\quad]_0$

Therefore, the following holds:

– $\mathcal{C}_{4(k+1)+2}(1) = \{\delta_{4(k+1)+2}, \delta'^{2^{k+2}}_{4(k+1)+2}, cod''_{k+2}(\varphi)^{2^{k+2}}\} \cup \{\beta^{2^{k+1}}_{l,k} \mid k+1 \le l \le n\}$
– There are 2^{k+1} membranes labelled by 2 such that each of them contains
 · objects $a_{k+2,k+1}, \ldots, a_{n,k+1}$;
 · objects $r_1, \ldots, r_{k+1}$, being $r \in \{t, f\}$; and
 · objects $\alpha_1, \ldots, \alpha_{p+1}$.

(a_3) Then, configuration $\mathcal{C}_{4(k+1)+2}$ yields configuration $\mathcal{C}_{4(k+1)+3}$ by applying the rules:

$[\,a_{k+1,k+1}\,]_2 \to [\,\Gamma_0\,]_2[\,\Gamma_1\,]_2$, for $1 \le i \le n$
$\left.\begin{array}{l}[\,\beta^O_{k+1,k+1}\,]_1[\quad]_0 \to [\quad]_1[\,\beta^O_{k+1,k+1}\,]_0\\ [\,\beta^O_{l,k+1}\,]_1[\quad]_0 \to [\quad]_1[\,\beta_{l,k+1}\,]_0\end{array}\right\}\text{for } O \in \{L, R\}, k+2 \le l \le n$
$\left.\begin{array}{l}[\,x''_{i,j,k+2}\,]_1[\,\gamma\,]_0 \to [\,x'''_{i,j,k+2}\,]_1[\quad]_0\\ [\,\overline{x}''_{i,j,k+2}\,]_1[\,\gamma\,]_0 \to [\,\overline{x}'''_{i,j,k+2}\,]_1[\quad]_0\\ [\,x^{*\prime\prime}_{i,j,k+2}\,]_1[\,\gamma\,]_0 \to [\,x^{*\prime\prime\prime}_{i,j,k+2}\,]_1[\quad]_0\end{array}\right\}\text{for}\begin{array}{l}1 \le i \le n,\\ 1 \le j \le p\end{array}$
$[\,\delta_{4(k+1)+2}\,]_1[\,\gamma\,]_0 \to [\,\delta_{4(k+1)+3}\,]_1[\quad]_0$
$[\,\delta'_{4(k+1)+2}\,]_1[\,\gamma\,]_0 \to [\,\delta'_{4(k+1)+3}\,]_1[\quad]_0$

Therefore, the following holds:

– $\mathcal{C}_{4(k+1)+3}(0) = \{\beta^{O^{2^{k+1}}}_{k+2,k+2}\} \cup \{\beta^{2^{k+2}}_{l,k+2} \mid k+3 \le l \le n\}$
– $\mathcal{C}_{4(k+1)+3}(1) = \{\delta_{4(k+1)+3}, \delta'^{2^{k+2}}_{4(k+1)+3}, cod'''_{k+2}(\varphi)^{2^{k+2}}\} \cup \{\beta^{2^{k+1}}_{l,k+1} \mid k+2 \le l \le n\}$
– There are 2^{k+2} membranes labelled by 2 such that each of them contains
 · objects $a_{k+2,k+1}, \ldots, a_{n,k+1}$;
 · objects $r_1, \ldots, r_{k+1}$, being $r \in \{t, f\}$; and
 · objects $\alpha_1, \ldots, \alpha_{p+1}$.

• In order to prove (b) it is enough to notice that, on the one hand, from (a_3) configuration $\mathcal{C}_{4n-1}$[1] holds:

[1] Here, $4n - 1 = 4k + 3$ for $k = n - 1$.

- $\mathcal{C}_{4n-1}(1) = \{\delta_{4n-1}, {\delta'}^{2^n}_{4n-1}, cod'''_n(\varphi)\}$.
- There are 2^n membranes labelled by 2 such that each of them contains
  - a different subset $\{r_1^O, \dots, r_n^O\}$, being $r \in \{t, f\}$ and $O \in \{L, R\}$; and
  - objects $\alpha^O, \dots, \alpha^O_{p+1}$, for $O \in \{L, R\}$.

- On the other hand, configuration $\mathcal{C}_{4n-1}$ yields configuration $\mathcal{C}_{4n}$ by applying the rules:

$$[\,r_i^O\,]_2[\,\beta^O_{n,n}\,]_0 \to [\,r_i\,]_2[\quad]_0 \;,\ \text{for} \begin{array}{l} O \in \{L,R\}, \\ r \in \{t,f\}, \\ 1 \le i \le n \end{array}$$

$$[\,\alpha_j^O\,]_2[\,\beta^O_{n,n}\,]_0 \to [\,\alpha_j\,]_2[\quad]_0 \;,\ \text{for } O \in \{L,R\}, 1 \le j \le p+1$$

$$\left.\begin{array}{l} [\,x'''_{i,j,n}\,]_1[\,\gamma\,]_0 \to [\,x_{i,j,n}\,]_1[\quad]_0 \\ [\,\overline{x}'''_{i,j,n}\,]_1[\,\gamma\,]_0 \to [\,\overline{x}_{i,j,n}\,]_1[\quad]_0 \\ [\,{x^*}'''_{i,j,n}\,]_1[\,\gamma\,]_0 \to [\,x^*_{i,j,n}\,]_1[\quad]_0 \end{array}\right\} \text{for } 1 \le i \le n, 1 \le j \le p$$

$$[\,\delta_{4n-1}\,]_1[\,\gamma\,]_0 \to [\,\delta_{4n}\,]_1[\quad]_0$$

$$[\,\delta'_{4n-1}\,]_1[\,\gamma\,]_0 \to [\,\delta'_{4n}\,]_1[\quad]_0$$

- Then, we have $\mathcal{C}_{4n}(1) = \{\delta_{4n}, {\delta'}^{2^n}_{4n}, cod_{4n}(\varphi)^{2^n}\}$, and there are 2^n membranes labelled by 2 such that each of them contains objects $\alpha_1, \dots, \alpha_{p+1}$, as well as a different subset $\{r_1, \dots, r_n\}$, being $r \in \{t, f\}$. □

First checking stage

Following the generation stage comes the first checking stage, where objects $c_{j,k}$ are created in order to know if clause C_j has been satisfied by the truth assignment encoded in membranes labelled by 2. In each step, we fire rules for a single clause, therefore in p steps we can obtain objects $c_{j,k}$ if this clause is satisfied. This can be because of two reasons:

- Literal x_i appears in clause C_j, and the the valoration of variable x_i in a truth assignment is `True`. Then, we can say that such truth assignment satisfies this clause; or
- Literal $\neg x_i$ appears in clause C_j, and the the valoration of variable x_i in a truth assignment is `False`. Then, we can say that such truth assignment satisfies this clause.

In any other way, variable x_i has nothing to do with clause C_j. At the final step of this stage, membranes labelled by 2 will have objects $c_{j,p}$ where C_j are clauses satisfied by such truth assignment. We obtain an object α'_{p+1} to use it in the next stage.

Proposition 2. *Let $\mathcal{C} = (\mathcal{C}_0, \mathcal{C}_1, \dots, \mathcal{C}_q)$ be a compuation of th system $\Pi(s(\varphi))$ with input multiset $cod(\varphi)$.*

(a) For each k $(0 \le k \le p-1)$ at configuration $\mathcal{C}_{4n+k}$ we have the following:

- $\mathcal{C}_{4n+k}(1) = \{\delta_{4n+k}, {\delta'}^{2^n}_{4n+k}, cod^k_n(\varphi)^{2^n}\}$
- *There are 2^n membranes labelled by 2 such that each of them contains*

- *objects $r_1, \ldots, r_n$, being $r \in \{t, f\}$;*
- *objects $\alpha_1, \ldots, \alpha_{p+1}$; and*
- *objects $c_{1,k}, \ldots, c_{k,k}$, where $c_{j,k}$ represents that clause C_j has been satisfied by the truth formula encoded in such membrane.*

(*b*) $\mathcal{C}_{4n+p}(1) = \{\delta_{4n+p}, {\delta'}^{2^n}_{4n+p}\}$, *and there are 2^n membranes labelled by 2 such that each of them contains objects $\alpha_1, \ldots, \alpha_{p+1}$, a different subset $\{r_1, \ldots, r_n\}$ and objects c_j when clause C_j is satisfied in that membrane.*

Proof. (a) is going to be demonstrated by induction on k.

(a) The base case $k = 0$ is trivial because at the initial configuration we have: $\mathcal{C}_{4n}(1) = \{\delta_{4n}, {\delta'}^{2^n}_{4n}, cod_{4n}(\varphi)\}$ and there exist 2^n membranes labelled by 2 containing objects $\alpha_1, \ldots, \alpha_{p+1}$ and a different subset $\{r_1, \ldots, r_n\}$, being $r \in \{t, f\}$. Then, configuration $\mathcal{C}_{4n}$ yields configuration $\mathcal{C}_{4n+1}$ by applying the rules:

$$\left.\begin{array}{l}
[\,t_i\,]_2[\,x_{i,1,n}\,]_1 \to [\,c_{1,1}\,t_i\,]_2[\quad]_1\\
[\,t_i\,]_2[\,\overline{x}_{i,1,n}\,]_1 \to [\,t_i\,]_2[\quad]_1\\
[\,t_i\,]_2[\,x^*_{i,1,n}\,]_1 \to [\,t_i\,]_2[\quad]_1\\
[\,f_i\,]_2[\,x_{i,1,n}\,]_1 \to [\,f_i\,]_2[\quad]_1\\
[\,f_i\,]_2[\,\overline{x}_{i,1,n}\,]_1 \to [\,c_{1,1}\,f_i\,]_2[\quad]_1\\
[\,f_i\,]_2[\,x^*_{i,1,n}\,]_1 \to [\,f_i\,]_2[\quad]_1
\end{array}\right\}\text{for } 1 \le i \le n, 1 \le j \le p$$

$$\left.\begin{array}{l}
[\,x_{i,j,n+k}\,]_1[\,\gamma\,]_0 \to [\,x_{i,j,n+k+1}\,]_1[\quad]_0\\
[\,\overline{x}_{i,j,n+k}\,]_1[\,\gamma\,]_0 \to [\,\overline{x}_{i,j,n+k+1}\,]_1[\quad]_0\\
[\,x^*_{i,j,n+k}\,]_1[\,\gamma\,]_0 \to [\,x^*_{i,j,n+k+1}\,]_1[\quad]_0
\end{array}\right\}\text{for } 1 \le i \le n, 2 \le j \le p$$

$$\begin{array}{l}
[\,\delta_{4n}\,]_1[\,\gamma\,]_0 \to [\,\delta_{4n+1}\,]_1[\quad]_0\\
[\,\delta'_{4n}\,]_1[\,\gamma\,]_0 \to [\,\delta'_{4n+1}\,]_1[\quad]_0
\end{array}$$

Thus, $\mathcal{C}_{4n+1}(1) = \{\delta_{4n+1}, {\delta'}^{2^n}_{4n+1}, cod^2_{4n+1}(\varphi)^{2^n}\}$ and in $\mathcal{C}_{4n+1}$ there exist 2^n membranes labelled by 2 such that their contents are objects $\alpha_1, \ldots, \alpha_{p+1}$, a different subset $\{r_1, \ldots, r_n\}$, being $r \in \{t, f\}$ and objects $c_{1,1}$ if some literal present in C_j satisfies it[2]. Hence, the result holds for $k = 1$.

Supposing that, by induction, result is true for k $(0 \le k \le p - 1)$; that is,

- $\mathcal{C}_{4n+k}(1) = \{\delta_{4n+k}, {\delta'}^{2^n}_{4n+k}, cod^{k+1}_n(\varphi)^{2^k}\}$
- There are 2^n membranes labelled by 2 such that each of them contains
  - objects $r_1, \ldots, r_n$, being $r \in \{t, f\}$;
  - objects $\alpha_1, \ldots, \alpha_{p+1}$; and
  - objects $c_{1,k}, \ldots, c_{k,k}$, where $c_{j,k}$ represents that clause C_j has been satisfied by the truth formula encoded in such membrane.

Then, configuration $\mathcal{C}_{4n+k}$ yields configuration $\mathcal{C}_{4n+k+1}$ by applying the rules:

[2] Here, objects # are created, but they are not used anymore, so they are not going to be noted here.

$$\left.\begin{array}{l}[\,t_i\,]_2[\,x_{i,k+1,n+k}\,]_1 \to [\,c_{k+1,k+1}\,t_i\,]_2[\quad]_1\\ [\,t_i\,]_2[\,\overline{x}_{i,k+1,n+k}\,]_1 \to [\,t_i\,]_2[\quad]_1\\ [\,t_i\,]_2[\,x^*_{i,k+1,n+k}\,]_1 \to [\,t_i\,]_2[\quad]_1\\ [\,f_i\,]_2[\,x_{i,k+1,n+k}\,]_1 \to [\,f_i\,]_2[\quad]_1\\ [\,f_i\,]_2[\,\overline{x}_{i,k+1,n+k}\,]_1 \to [\,c_{k+1,k+1}\,f_i\,]_2[\quad]_1\\ [\,f_i\,]_2[\,x^*_{i,k+1,n+k}\,]_1 \to [\,f_i\,]_2[\quad]_1\end{array}\right\}\text{for } 1 \le i \le n, 1 \le j \le p$$

$$\left.\begin{array}{l}[\,x_{i,j,n+k}\,]_1[\,\gamma\,]_0 \to [\,x_{i,j,n+k+1}\,]_1[\quad]_0\\ [\,\overline{x}_{i,j,n+k}\,]_1[\,\gamma\,]_0 \to [\,\overline{x}_{i,j,n+k+1}\,]_1[\quad]_0\\ [\,x^*_{i,j,n+k}\,]_1[\,\gamma\,]_0 \to [\,x^*_{i,j,n+k+1}\,]_1[\quad]_0\end{array}\right\}\text{for } 1 \le i \le n, k+2 \le j \le p$$

$$[\,c_{j,k}\,]_2[\,\gamma\,]_0 \to [\,c_{j,k+1}\,]_2[\quad]_0\ ,\text{ for } 1 \le j \le p, k \le k \le p-1$$
$$[\,\delta_{4n+k}\,]_1[\,\gamma\,]_0 \to [\,\delta_{4n+k+1}\,]_1[\quad]_0$$
$$[\,\delta'_{4n+k}\,]_1[\,\gamma\,]_0 \to [\,\delta'_{4n+k+1}\,]_1[\quad]_0$$

Thus, $\mathcal{C}_{4n+k+1}(1) = \{\delta_{4n+k+1}, {\delta'}^{2^n}_{4n+k+1}, cod^{k+2}_{4n+k+1}(\varphi)^{2^n}\}$ and in $\mathcal{C}_{4n+k+1}$ there exist 2^n membranes labelled by 2 such that their contents are objects $\alpha_1, \dots, \alpha_{p+1}$, a different subset $\{r_1, \dots, r_n\}$, being $r \in \{t, f\}$ and objects $c_{1,k}, \dots, c_{k,k}$ if some literal present in C_j satisfies them.

In order to demonstrate (b) it is enough to notice that, on the one hand, from (a) configuration $\mathcal{C}_{4n+p-1}$ holds:

- $\mathcal{C}_{4n+p-1}(1) = \{\delta_{4n+p-1}, {\delta'}^{2^n}_{4n+p-1}, cod^p_n(\varphi)^{2^n}\}$
- There are 2^n membranes labelled by 2 such that each of them contains
  - objects $r_1, \dots, r_n$, being $r \in \{t, f\}$;
  - objects $\alpha_1, \dots, \alpha_{p+1}$; and
  - objects $c_{1,p-1}, \dots, c_{p-1,p-1}$, where $c_{j,p-1}$ represents that clause C_j has been satisfied by the truth formula encoded in such membrane.

On the other hand, configuration $\mathcal{C}_{4n+p-1}$ yields configuration $\mathcal{C}_{4n+p}$ by applying the rules:

$$\left.\begin{array}{l}[\,t_i\,]_2[\,x_{i,p,n+p-1}\,]_1 \to [\,c_{p,p}\,t_i\,]_2[\quad]_1\\ [\,t_i\,]_2[\,\overline{x}_{i,p,n+p-1}\,]_1 \to [\,t_i\,]_2[\quad]_1\\ [\,t_i\,]_2[\,x^*_{i,p,n+p-1}\,]_1 \to [\,t_i\,]_2[\quad]_1\\ [\,f_i\,]_2[\,x_{i,p,n+p-1}\,]_1 \to [\,f_i\,]_2[\quad]_1\\ [\,f_i\,]_2[\,\overline{x}_{i,p,n+p-1}\,]_1 \to [\,c_{p,p}\,f_i\,]_2[\quad]_1\\ [\,f_i\,]_2[\,x^*_{i,p,n+p-1}\,]_1 \to [\,f_i\,]_2[\quad]_1\end{array}\right\}\text{for } 1 \le i \le n, 1 \le j \le p$$

$$[\,c_{j,p-1}\,]_2[\,\gamma\,]_0 \to [\,c_{j,p}\,]_2[\quad]_0\ ,\text{ for } 1 \le j \le p-1$$
$$[\,\delta_{4n+p-1}\,]_1[\,\gamma\,]_0 \to [\,\delta_{4n+p}\,]_1[\quad]_0$$
$$[\,\delta'_{4n+p-1}\,]_1[\,\gamma\,]_0 \to [\,\delta'_{4n+p}\,]_1[\quad]_0$$

Then, we have $\mathcal{C}_{4n+p}(1) = \{\delta_{4n+p}, {\delta'}^{2^n}_{4n+p}\}$, and in $\mathcal{C}_{4n+p}$ there are 2^n membranes labelled by 2 such that each of them contains a different subset $\{r_1, \dots, r_n\}$, being $r \in \{t, f\}$[3], objects $\alpha_1, \dots, \alpha_{p+1}$ and objects $c_{j,p}$ when clause C_j has been satisfied by the truth assignment encoded in such membrane. □

[3] This subset is not used anymore, so it will not be noted from now on.

Second checking stage

Here, when rules from **3.1** are fired at the $(4n+p+1)$-th step, objects α_j within a membrane labelled by 2 are removed if and only if the truth assignment associated to that membrane makes true clause C_j, that is, if there is at least one object c_j in such membrane. At configuration $\mathcal{C}_{4n+p}$ we have $\mathcal{C}_{4n+p}(1) = \{\delta_{4n+p}, {\delta'}_{4n+p}^{2^n}\}$ and each membrane labelled by 2 contains objects $\alpha_1, \ldots, \alpha_p$ and objects c_j such that the corresponding truth assignment satisfies the clause C_j. By applying rules from **3.1** and rule $[\,\delta_{4n+p}\,]_1[\,\gamma\,]_0 \to [\,\delta_{4n+p+1}\,]_1[\quad]_0$, object δ_{4n+p} evolves into δ_{4n+p+1} within the membrane labelled by 1, and in each membrane labelled by 2, objects α_j such that their corresponding object $c_{j,p}$ are "removed" from the system, and let the next stage to check whether or not they are present, besides the object α_{p+1}, that is prepared, evolving to α'_{p+1}, to react with the remaining objects α_j. This stage takes exactly one step.

Output stage

The output phase starts at the $(4n+p+2)$-th step, and takes exactly four steps, regardless of whether the input formula φ is satisfied or not by some truth assignment.

- *Affirmative answer*: If the input formula φ of `SAT` problem is satisfiable then at least one of the truth assignments from a membrane with label 2 has satisfied all clauses. Then, there will be a membrane labelled by 2 such that all objects α_j, with $1 \leq j \leq p$ have dissapeared in the previous step. At configuration $\mathcal{C}_{4n+p+1}$, we have $\mathcal{C}_{4n+p+1}(1) = \{\delta_{4n+p+1}\}$ and in each membrane labelled by 2 there remain objects α_j if the corresponding truth assignment does **not** make true clause C_j and one object α'_{p+1}. In this step, only rule $[\,\delta_{4n+p+1}\,]_1[\,\gamma\,]_0 \to [\,\delta_{4n+p+2}\,]_1[\quad]_0$ will be fired and rules $[\,\alpha_j\,\alpha'_{p+1}\,]_2[\quad]_0 \to [\quad]_2[\,n_1\,]_0$ will be fired in membranes labelled by 2 such that at least one clause is not satisfied by the corresponding truth assignment. Then, at configuration $\mathcal{C}_{4n+p+2}$, we have $\mathcal{C}_{4n+p+2}(1) = \{\delta_{4n+p+2}, n_1^t\}$, being t the number of truth assignments that have at least one clause **not** satisfied by the corresponding truth assignment, and membranes labelled by 2 contains an object α'_{p+1} if and only if the corresponding truth assignment makes true all clauses from φ, and can contain objects α_j, $1 \leq j \leq p$, if clause C_j is not satisfied by the corresponding truth assignment.
  In the next step, applying rules $[\quad]_2[\,n_1\,]_0 \to [\,n_1\,]_2[\quad]_0$ and $[\,\alpha'_{p+1}\,]_2[\,\delta_{4n+p+2}\,]_1 \to [\,y_1\,]_2[\quad]_1$, we obtain an object y_1 in a membrane labelled by 2 if and only if the corresponding truth assignment makes true the input formula. Let us remark that more than one membrane labelled by 2 can contain a truth assignment that makes true φ, but in this case, we as we want to know if *at least* one truth assignment makes true the input formula φ, we only want one object y_1. Then, at configuration $\mathcal{C}_{4n+p+3}$ we have that $\mathcal{C}_{4n+p+3}(1) = \emptyset$ and in membranes la-

belled by 2, we can have objects n_1[4], adding up to t in all membranes labelled by 2, being t the number of truth assignments that do not make true the input formula, an object α'_{p+1} if the corresponding truth assignment makes true all clauses, excepting one membrane labelled by 2 which corresponding truth assignment makes true the input formula that will contain an object y_1, and can contain objects α_j, $1 \le j \le p$, if clause C_j is not satisfied by the corresponding truth assignment. In the next step the only rule that can be fired is $[\,y_1\,]_2[\,\gamma\,]_0 \to [\,y_2\,]_2[\quad]_0$, that will be useful to synchronize the affirmative and the negative answer. Let us note that rule $[\,n_1\,]_2[\,\delta_{4n+p+2}\,]_1 \to [\,n_2\,]_2[\quad]_1$ cannot be fired because object δ_{4n+3} has been consumed in the previous step by an object α'_{p+1}. Then, at configuration $\mathcal{C}_{4n+p+4}$, we have that $\mathcal{C}_{4n+p+4}(1) = \emptyset$ and in membranes labelled by 2, we can have objects n_1, adding up to t in all membranes labelled by 2, being t the number of truth assignments that do not make true the input formula, an object α'_{p+1} if the corresponding truth assignment makes true all clauses, excepting one membrane labelled by 2 which corresponding truth assignment makes true the input formula that will contain an object y_2, and can contain objects α_j, $1 \le j \le p$, if clause C_j is not satisfied by the corresponding truth assignment. At the last step of the computation, rule $[\,y_2\,]_2[\quad]_0 \to [\quad]_2[\,\texttt{yes}\,]_0$ is fired, sending an object `yes` to the environment. Then, at configuration $\mathcal{C}_{4n+p+5}$, we have that $\mathcal{C}_{4n+p+5}(1) = \emptyset$ and in membranes labelled by 2, we can have objects n_1, adding up to t in all membranes labelled by 2, being t the number of truth assignments that do not make true the input formula, an object α'_{p+1} if the corresponding truth assignment makes true all clauses, excepting one membrane labelled by 2 which corresponding truth assignment makes true the input formula, and can contain objects α_j, $1 \le j \le p$, if clause C_j is not satisfied by the corresponding truth assignment, and there will be an object `yes` in the environment. Here, the computation halts and returns an affirmative answer.

- *Negative answer*: If the input formula φ of `SAT` problem is not satisfiable then none of the truth assignments encoded by a membrane labelled by 2 makes the formula φ true. Thus, some object α_j $(1 \le j \le p)$ will be within all membranes labelled by 2 will not remain in such membranes. At configuration $\mathcal{C}_{4n+p+1}$, we have $\mathcal{C}_{4n+p+1}(1) = \{\delta_{4n+p+1}\}$ and in each membrane labelled by 2 there remain objects α_j if the corresponding truth assignment does **not** make true clause C_j. In this step, only rules $[\,\alpha_j\,\alpha'_{p+1}\,]_2[\quad]_0 \to [\quad]_2[\,n_1\,]_0$, for $1 \le j \le p$ and rule $[\,\delta_{4n+p+1}\,]_1[\,\gamma\,]_0 \to [\,\delta_{4n+p+2}\,]_1[\quad]_0$ will be fired. Then, at configuration $\mathcal{C}_{4n+p+2}$ we have in the environmet 2^n copies of object n_1, $\mathcal{C}_{4n+p+2}(1) = \{\delta_{4n+p+2}\}$ and membranes labelled by 2 will contain objects α_j $(1 \le j \le p)$ when clauses C_j are not satisfied by the corresponding truth assignment. In the $(4n + p + 3)$-th step, rule $[\quad]_2[\,n_1\,]_0 \to [\,n_1\,]_2[\quad]_0$ will be fired. Here, objects n_1 will be sent to a membrane labelled by 2. Then,

[4] Let us note that a membrane containing an object n_1 does not say that the corresponding truth assignment does not makes true the input formula. In fact, we can have more than one object n_1 within a single membrane labelled by 2.

at configuration $\mathcal{C}_{4n+p+3}$ we have $\mathcal{C}_{4n+p+3}(1) = \{\delta_{4n+p+2}\}$ and membranes labelled by 2 contain objects α_j $(1 \leq j \leq p)$ if clause C_j is not satisfied by the corresponding truth assignment, and can contain t objects n_1 $(0 \leq t \leq 2^n)$. At the $(4n+p+4)$-th step rule $[\, n_1\,]_2[\, \delta_{4n+p+2}\,]_1 \rightarrow [\, n_2\,]_2[\quad]_1$ is fired, since object δ_{4n+3} has not been consumed by any rule from **4.3**, creating an object n_2 in a membrane labelled by 2. Then, at configuration $\mathcal{C}_{4n+p+4}$ we have $\mathcal{C}_{4n+p+4}(1) = \emptyset$ and membranes labelled by 2 contain objects α_j $(1 \leq j \leq p)$ if clause C_j is not satisfied by the corresponding truth assignment, and can contain t objects n_1 $(0 \leq t \leq 2^n)$, and one of them contains an object n_2. At the last step of the computation, rule $[\, n_2\,]_2[\quad]_0 \rightarrow [\quad]_2[\, \mathtt{no}\,]_0$ is fired, sending an object no to the environment. Then, at configuration $\mathcal{C}_{4n+p+5}$ we have that $\mathcal{C}_{4n+p+5}(1) = \emptyset$ and membranes labelled by 2 contain objects α_j $(1 \leq j \leq p)$ if clause C_j is not satisfied by the corresponding truth assignment, and can contain t objects n_1 $(0 \leq t \leq 2^n)$, and there will be an object no in the environment. Here, the computation halts and returns a negative answer.

Result

Proof. The family of P systems previously constructed verifies the following:

- Every system of the family $\mathbf{\Pi}$ is a recognizer P systems from $\mathbf{TSEC}(2, 2)$.
- The family $\mathbf{\Pi}$ is polynomially uniform by Turing machines because for each $n, p \in \mathbb{N}$, the rules of $\Pi(\langle n, p\rangle)$ of the family are recursively defined from $n, p \in \mathbb{N}$, and the amount of resources needed to build an element of the family is of a polynomial order in n and p, as shown below:
  - Size of the alphabet: $9n^2p + 6n^2 + \frac{3np^2}{2} - 3np + 22n + \frac{p^2}{2} + \frac{13p}{2} + 14 \in \Theta(max\{n^2p, np^2\})$.
  - Initial number of cells: $2 \in \Theta(1)$.
  - Initial number of objects in cells: $n^2 + n(p+1) + p + 3 \in \Theta(n^2)$.
  - Number of rules: $8n^3 + \frac{27n^2p}{2} + 4n^1 + \frac{19np}{2} + 23n + \frac{p^2}{2} + \frac{17p}{2} + 11 \in \Theta(n^3)$.
  - Maximal number of objects involved in any rule: $4 \in \Theta(1)$.
- The pair (cod, s) of polynomial-time computable functions defined fulfill the following: for each input formula φ of SAT problem, $s(\varphi)$ is a natural number, $cod(\varphi)$ is an input multiset of the system $\Pi(s(\varphi))$, and for each $n \in \mathbb{N}$, $s^{-1}(n)$ is a finite set.
- The family $\mathbf{\Pi}$ is polynomially bounded: indeed for each input formula φ of SAT problem, the deterministic P system $\Pi(s(\varphi)) + cod(\varphi)$ takes exactly $4n+p+5$ steps, being n the number of variables of φ and p the number of clauses.
- The family $\mathbf{\Pi}$ is sound with regard to (X, cod, s): indeed, for each formula φ, if the computation of $\Pi(s(\varphi)) + cod(\varphi)$ is an accepting computation, then φ is satisfiable.
- The family $\mathbf{\Pi}$ is complete with regard to (X, cod, s): indeed, for each input formula φ such that it is satisfiable, the computation of $\Pi(s(\varphi)) + cod(\varphi)$ is an accepting computation. □

Corollary 1. $\mathbf{NP} \cup \mathbf{co} - \mathbf{NP} \subseteq \mathbf{PMC}_{\mathbf{TSEC}(2,2)}$.

Proof. It suffices to notice that `SAT` problem is a **NP**-complete problem, `SAT` $\in \mathbf{PMC}_{\mathbf{TSEC}(2,2)}$, and the complexity class $\mathbf{PMC}_{\mathbf{TSEC}(2,2)}$ is closed under polynomial-time reduction and under complement. □

6 Conclusions and future work

In [6] a tight frontier of efficiency in the framework of tissue P systems with evolutional symport/antiport rules and cell separation is defined by the length of the RHS, that is, passing from 1 to 2 is enough to pass from non-efficiency to presumably efficiency while the length of the LHS is at least 3. This result is demonstrated giving a solution of the `SAT` problem by means of a family of P system from $\mathbf{TSEC}(3,2)$. But an open problem remains open here: what happens with P systems from $\mathbf{TSEC}(k,2)$ ($k \geq 2$)? Can we solve computationally hard problems restricting the length of the LHS to 2?

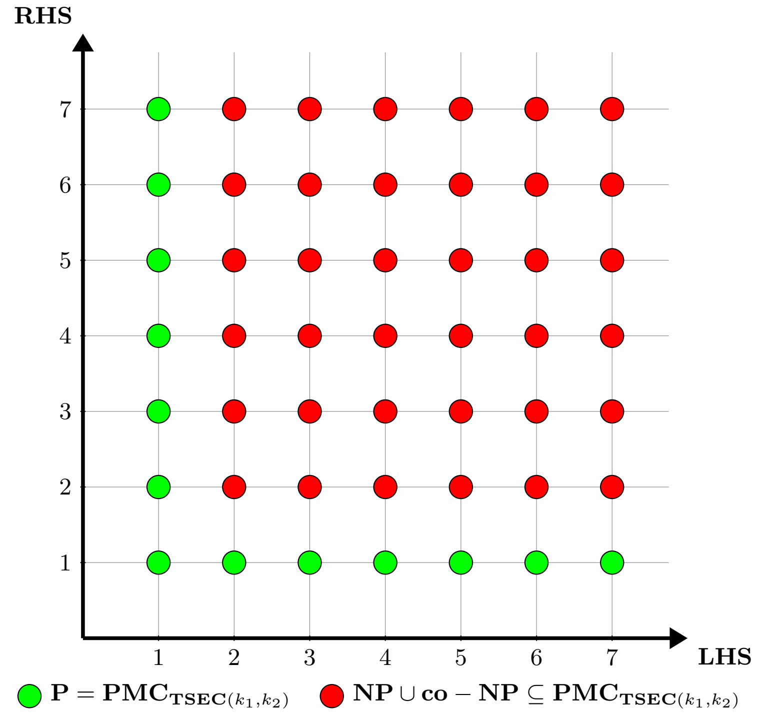
In this paper, an efficient solution to the **SAT** problem is given by means of a family of P systems from $\mathbf{TSEC}(2,2)$, so the previous problem is solved. Then, we can conclude here with a similar figure to the presented in [6] but with the new results included.

Of course, after this work we can define several clear research lines to continue investigating these kinds of P systems.

– What happens when the environment "dissapear"?
– Do the structure matter? By this we mean using cell-like structure with this kind of rules.
– In [12] another definition of length is given. Let k be the length of the rule defined as follows: if $r \equiv [\,u\,]_i[\,v\,]_j \to [\,v'\,]_i[\,u'\,]_j$, $k = |u| + |v| + |u'| + |v'|$. Then the complexity class of tissue P systems with evolutional communication rules with at most length k and cell separation is denoted by $\mathbf{PMC}_{\mathbf{TSEC}(k)}$. What are the borderline here?
– What is the upper bound of these systems? In [3] a characterization of tissue P systems with symport/antiport rules and both cell division and separation is given matching their efficiency to the class $\mathbf{P}^{\#\mathbf{P}}$, and it seems that this class of P system can reach the same complexity class.

Acknoledgements

This work was supported by Project TIN2017-89842-P of the Ministerio de Economía y Competitividad of Spain and by Grants No 61320106005 of the National Natural Science Foundation of China.



References

1. R. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, M. Rius–Font. Characterizing tractability by tissue–like P systems. In R. Gutiérrez–Escudero, M.A. Gutiérrez–Naranjo, Gh. Păun, I. Pérez–Hurtado and A. Riscos Núñez (eds.) *Proceedings of the Seventh Brainstorming Week on Membrane Computing*, Fénix Editora, Seville, 2009, pp. 169–180.
2. M. Ionescu, Gh. Păun, T. Yokomori. Spiking Neural P Systems. *Fundamenta Informaticae*, **71**, 2,3 (2006), 279–308.
3. A. Leporati, L. Manzoni, A.E. Porreca, C. Zandron. Characterising the complexity of tissue P systems with fission rules. *Journal of Computer and System Sciences*, **90** (2017), 115–128.
4. C. Martín-Vide, Gh. Păun, J. Pazos, A. Rodríguez-Patón. Tissue P systems. *Theoretical Computer Science*, **296**, Issue 2, 2003, 295–326.
5. L. Pan, M.J. Pérez-Jiménez, A. Riscos-Núñez, M. Rius-Font. New frontiers of the efficiency in tissue P systems. L. Pan, Gh. Păun, T. Song (eds.) *Pre-proceedings of Asian Conference on Membrane Computing (ACMC 2012)*, Huazhong University of Science and Technology, Wuhan, China, October 15-18, 2012, pp. 61-73.
6. L. Pan, B. Song, L. Valencia-Cabrera, M.J. Pérez-Jiménez. The computational complexity of tissue P systems with evolutional symport/antiport rules. *Complexity*, 21 pages, 2018.
7. 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

8. M.J. Pérez-Jiménez. An approach to computational complexity in Membrane Computing. In G. Mauri, Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg, A. Salomaa (eds.). *Membrane Computing, International Workshop, WMC5, Milano, Italy, 2004, Selected Papers, Lecture Notes in Computer Science*, **3365** (2005), 85-109.
9. M.J. Pérez-Jiménez, Á. Romero-Jiménez, F. Sancho-Caparrini. Complexity classes in models of cellular computing with membranes. *Natural Computing*, **2**, 3 (2003), 265–285.
10. M.J. Pérez-Jiménez, P. Sosík. Improving the efficiency of tissue P systems with cell separation. In M. García-Quismondo, L.F. Macías-Ramos, Gh. Păun, I. Pérez- Hurtado, L. Valencia-Cabrera (eds.) *Proceedings of the Tenth Brainstorming Week on Membrane Computing*, Volume II, Seville, Spain, January 30 - February 3, 2012, Report RGNC 01/2012, Fénix Editora, 2012, pp. 105-140.
11. A.E. Porreca, N. Murphy, M.J. Pérez-Jiménez. An Optimal Frontier of the Efficiency of Tissue P Systems with Cell Division. In M.Á. Martínez-del-Amor, Gh. Păun, I. Pérez–Hurtado and F.J. Romero-Campero (eds.) *Proceedings of the Tenth Brainstorming Week on Membrane Computing*, Volume II, Seville, Spain, January 30 - February 3, 2012, Report RGNC 01/2012, Fénix Editora, 2012, 141–166.
12. B. Song, C. Zhang, L. Pan. Tissue-like P systems with evolutional symport/antiport rules. *Information Sciences*, **378** (2017), 177-.193.

Author Index