Proceedings of the 24th International Conference on Membrane Computing (CMC2023)

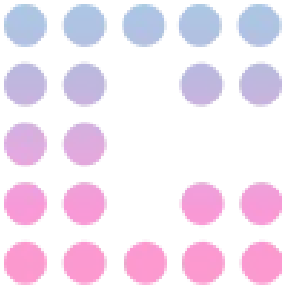
August 28–30, 2023 Opava Czech Republic

Edited by Lucie Ciencialová

SLEZSKÁ UNIVERZITA
FILOZOFICKO-PŘÍRODOVĚDECKÁ FAKULTA V OPAVĚ

http://cmc2023.slu.cz

Proceedings of the Twenty-fourth International Conference on Membrane Computing (CMC2023)



28-31 August 2023
Opava, Czech Republic

Lucie Ciencialová
editor

Proceedings of the 24th International Conference on Membrane Computing (CMC2023)
Edited by: Lucie Ciencialová

Published in August 2023

Preface

The present volume contains the invited contributions and a selection of papers presented at the 24th Conference on Membrane Computing(CMC2023), held in Opava, Czech Republic from August 28 to August 31, 2023. Further additional information on this conference can be found at the following website: http://cmc2023.slu.cz

The CMC series started with three workshops organized in Curtea de Arges, Romania, in 2000, 2001 and 2002. The workshops were then held in Tarragona, Spain (2003), Milan, Italy (2004), Vienna, Austria (2005), Leiden, The Netherlands(2006), Thessaloniki, Greece (2007), and Edinburgh, UK (2008). The 10th edition was organized again in Curtea de Arges, in August 2009, where it was decided to continue the series as the Conference on Membrane Computing (CMC). The following editions were held in Jena, Germany (2010),Fontainebleau, France (2011), Budapest, Hungary (2012), Chisinau, Moldova (2013), Praha, Czech Republic (2014), Valencia, Spain (2015) and Milan, Italy(2016), Bradford, UK(2017), Dresden, Germany (2018), Curtea de Arges, Romania (2019). Due to the pandemic caused by the Corona virus, the two main conferences organized annually by the Membrane Computing community through IMCS, Asian Conference on Membrane Computing (ACMC) and Conference on Membrane Computing (CMC) were united in one joint conference, online-only conference (2020) and Chengdu, China and Debrecen, Hungary (2021). CMC23 was helt in Trieste, Italy in 2022.

CMC2023 has been organized, under the auspices of the International Membrane Computing Society, and the European Molecular Computing Consortium by Institute of Computer Science, Faculty of Philosophy and science, the Silesian Univerzity in Opava, Czech Republic.

The invited lectures were given by Artiom Alhazov (Chisinau, Moldova) and José M. Sempere Luna (Valencia, Spain). The editors express their gratitude to the Program Committee, the invited speakers, the authors of the papers, the reviewers, and all the participants for their contributions to the success of CMC2023. The support of the Faculty of Philosophy and Science, the Silesian University in Opava, sponzors Compacer, ICZ and Jettimodel are gratefully acknowledged.

August 2023

Lucie Ciencialová
Program Chair
CMC 2023

Organization

Steering Committee of CMC series

Henry Adorna	Quezon City, Philippines
Artiom Alhazov	Chișinău, Moldova
Bogdan Aman	Iași, Romania
Matteo Cavaliere	Manchester, UK
Erzsébet Csuhaj-Varjú	Budapest, Hungary
Giuditta Franco	Verona, Italy
Rudolf Freund	Wien, Austria
Marian Gheorghe	Bradford, UK - Honorary member
Thomas Hinze	Cottbus, Germany
Florentin Ipate	Bucharest, Romania
Shankara N. Krishna	Bombay, India
Alberto Leporati	Milan, Italy
Ferrante Neri	Nottingham, UK
Taishin Y. Nishida	Toyama, Japan
Linqiang Pan	Wuhan, China – co-chair
Gheorghe Păun	Bucharest, Romania - Honorary member
Mario J. Pérez-Jiménez	Sevilla, Spain
Agustín Riscos-Núñez	Sevilla, Spain
Jose M. Sempere	Valencia, Spain
Petr Sosík	Opava, Czech Republic
K. G. Subramanian	Chennai, India
György Vaszil	Debrecen, Hungary
Sergey Verlan	Paris, France
Claudio Zandron	Milan, Italy – co-chair
Gexiang Zhang	Chengdu, China

Organizing Committee of CMC 2023

Luděk Cienciala - chair
Kateřina Bissell
Lucie Ciencialová
Kamil Matula
Anna Novotná
Šárka Vavrečková
Petr Sosík

Program Committee of CMC 2023

Henry Adorna	Quezon City, Philippines
Artiom Alhazov	Chișinău, Moldova
Bogdan Aman	Iași, Romania
Lucie Ciencialová	Opava, Czech Republic - chair
Erzsébet Csuhaj-Varjú	Budapest, Hungary
Giuditta Franco	Verona, Italy
Thomas Hinze	Cottbus, Germany
Florentin Ipate	Bucharest, Romania
Sergiu Ivanov	Paris, France
Alberto Leporati	Milan, Italy
Luca Manzoni	Trieste, Italy
David Orellana-Martín	Seville, Spain
Antonio Enrico Porreca	Marseille, France
Mario Pérez-Jiménez	Sevilla, Spain
Agustín Riscos-Núñez	Sevilla, Spain
Jose M. Sempere	Valencia, Spain
Petr Sosík	Opava, Czech Republic
K.G. Subramanian	Chennai, India
György Vaszil	Debrecen, Hungary
Sergey Verlan	Paris, France
Claudio Zandron	Milan, Italy
Gexiang Zhang	Chengdu, China

Table of Contents

III Informal talks

Part I

Invited talks

Towards an Online Simulator Exploring Non-Deterministic Networks of Cells

Artiom Alhazov

State University of Moldova,
Vladimir Andrunachievici Institute of Mathematics and Computer Science
Academiei 5, Chișinău, MD-2028, Moldova
artiom@math.md

Abstract. Vast majority of existing simulators, given a non-deterministic P system, either assume it is confluent, and compute the first non-deterministic choice, or compute a random non-deterministic choice. If the goal is to examine a proof construction, this is unacceptable. In this talk we discuss a possible approach of an online simulator solving such a problem, and satisfying some additional requirements. Topics mentioned include different proposals to keep such software efficient, run it directly in the browser, cover most existing rule types and more (e.g., capable of combining antiport with promoters/inhibitors and with spiking in the same rule if desired, in a uniform representation) for static P systems, work in various derivation modes, while in the same time being simple enough to be feasible to be implemented by a single developer in some reasonable time. Attention is paid to usefulness/conveneince to a user examining constructions from theorems, visualization, interactivity and other features.

Membrane computing: A wonderful framework for systems and computational biology

José M. Sempere

Departamento de Sistemas Informáticos y Computacíon (Universidad Politécnicade Valencia)
46071 Valencia, Spain jsempere@dsic.upv.es

Abstract. From the beginning, membrane computing has been a very suitable computational model to address modeling problems in biology. In this talk, we will show the main components that must be defined when defining digital twins of biological elements by using membrane computing. We will show a methodology adapted for membrane computing to the modeling of biological systems. We will see how to define objects and rules adapted to the biological application domain, and we will also talk about the different stochastic simulation engines. Throughout the talk, we will show some successful cases where membrane computing has been used for the modeling of epidemiological systems and control of physiological markers.

Part II

Regular papers

Simple P systems with Prescribed Teams of Sets of Rules

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

[1] State University of Moldova,
Vladimir Andrunachievici Institute of Mathematics and Computer Science
Academiei 5, Chișinău, MD-2028, Moldova
artiom@math.md

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

[3] IBISC, Univ. Évry, Paris-Saclay University
23, boulevard de France 91034 Évry, France
sergiu.ivanov@ibisc.univ-evry.fr

Abstract. In this paper we consider simple P systems with prescribed teams of sets of rules, with the application of the rule sets in the teams possibly depending on some given condition, as well as, in the general case, the different sets of rules in a prescribed team working in different derivation modes, whereas in homogeneous systems for all sets of rules the same derivation mode comes into action.
We prove some general results, for example, (i) how with such simple P systems with prescribed teams of sets of rules we can simulate label controlled P systems, where only rules with the same label can be applied, (ii) how simple purely catalytic P systems can be mimicked by simple P systems with prescribed teams of sets of non-cooperative rules with all sets of rules working in the sequential derivation mode, and (iii) how simple catalytic P systems can be mimicked by simple P systems with prescribed teams of sets of non-cooperative rules with some sets of rules working in the sequential derivation mode and only one working in the maximally parallel derivation mode.
Computational completeness of these simple P systems with prescribed teams of sets of non-cooperative rules therefore immediately follows from the well-known results for simple catalytic and purely catalytic P systems, respectively. On the other hand, homogeneous simple P systems with prescribed teams of sets of non-cooperative rules with all teams working in the maximally parallel derivation mode have the same computational power as $ET0L$ systems used for multisets.

Keywords: applicability condition, computational completeness, $ET0L$ systems, P systems, prescribed teams

1 Introduction

A quarter of a century ago, membrane (P) systems were introduced in [12] as a multiset-rewriting model of computing inspired by the hierarchical membrane structure and the functioning of the living cell, with the molecules/objects evolving in parallel. Since then, this area of biologically motivated computing has emerged in a fascinating way. A lot of interesting theoretical models have been developed by scientists all over the world, many of them already documented in two textbooks, see [13] and [14]. For actual information, we refer to the *P systems webpage* [16] as well as to the issues of the *Bulletin of the International Membrane Computing Society* and of the *Journal of Membrane Computing*.

P systems traditionally operate on multisets of objects, hence, the power of non-cooperative rules (even) when working in the maximally parallel derivation mode is rather restricted; for example, the multiset language $\{b^{2^n} \mid n \in \mathbb{N}\}$ cannot be obtained with non-cooperative rules by halting computations. Therefore, one of the fundamental questions which has attracted a lot of attention in the area of P systems is, how variants of different ways of cooperation of the rules and various control mechanisms affect the computational power. For example, allowing for cooperative rules rather easily boosts the power of specific variants of P systems to computational completeness.

One of the well-known control mechanisms forcing some rules to only be applied together (in the sequential derivation mode) are matrix grammars, in which the rules are grouped into sequences, which in the given order must be applied one after another. A less strict variant where the rules in a set of rules called *prescribed teams* can be applied in any order was introduced in [6]. In [4], such prescribed teams are working on different objects.

In contrast to the original model, in which the rules of a team can be applied together only sequentially, we here consider a *team* as a set of sets of rules, where each set of rules has assigned (i) its own applicability condition and (ii) its own derivation mode in which the rules in this set have to be applied, and based on one of these teams a suitable multiset of rules to be applied to the underlying configuration is constructed.

In the model of (*simple*, i.e., only one membrane region is considered) P systems with prescribed teams of sets of rules, the application of a team means applying each set of rules in the chosen team to be used in the derivation mode assigned to the set in this team, provided the applicability condition based on the features of the underlying configuration is fulfilled. In *internally homogenous* systems, all sets of rules in a team have assigned the same derivation mode, whereas in *globally homogenous* systems all teams have assigned the same derivation mode for all sets of rules in the teams. In this paper, we mainly focus on the sequential and the maximally parallel derivation mode; investigations with other derivation modes, as, for example considered in the formal framework for static P systems, see [9], or others then defined in [5,2,3,1], we leave for future research.

One obvious result we are going to prove is that globally homogenous simple P systems working in the maximally parallel derivation mode for the teams of sets of non-cooperative rules have the same computational power as $ET0L$ systems, i.e., extended tabled Lindenmayer systems. Simple P systems with prescribed teams of sets of rules can simulate label controlled P systems, where only rules with the same label can be applied. Moreover, simple purely catalytic P systems can be mimicked

by simple P systems with prescribed teams of sets of non-cooperative rules with the sets of rules working in the sequential derivation mode. For the simulation of catalytic P systems, one additional set working in the maximally parallel derivation mode is needed. Computational completeness of these simple P systems with prescribed teams of sets of non-cooperative rules therefore can immediately be inferred from the well-known results for simple catalytic and purely catalytic P systems, respectively. Furthermore, using sets of symbols as permitting and forbidden context conditions for the sets of rules in the teams allows for an easy direct simulation of register machines, either with using non-cooperative rules or else insertion and deletion rules.

2 Definitions

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

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

The family of regular, context-free, and recursively enumerable string languages is denoted by $\mathcal{L}(REG)$, $\mathcal{L}(CF)$, and $\mathcal{L}(RE)$, respectively. As $Ps\mathcal{L}(REG) = Ps\mathcal{L}(CF)$, in the area of multiset rewriting $\mathcal{L}(CF)$ plays no role at all, and in the area of membrane computing we often only get characterizations of $Ps\mathcal{L}(REG)$ and $Ps\mathcal{L}(RE)$ or else $Ps\mathcal{L}(ET0L)$, where $\mathcal{L}(ET0L)$ denotes the family of languages generated by extended tabled Lindenmayer systems ($ET0L$ systems).

For further notions and results in formal language theory we refer to textbooks like [7] and [15].

2.1 Register Machines

Register machines are well-known universal devices for computing on (or generating or accepting) sets of vectors of natural numbers. The following definitions and propositions are given as in [1].

Definition 1. *A* register machine *is a construct*

$$M = (m, B, l_0, l_h, P)$$

where

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

The instructions of M *can be of the following forms:*

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

A configuration *of a register machine is described by the contents of each register and by the value of the current label, which indicates the next instruction to be executed.* M *is called* deterministic *if the ADD-instructions all are of the form* $p : (ADD(r), q)$*.*

Throughout the paper, B_{ADD} *denotes the set of labels of ADD-instructions* $p : (ADD(r), q, s)$ *of arbitrary registers* r*, and* $B_{SUB(r)}$ *denotes the set of labels of all SUB-instructions* $p : (SUB(r), q, s)$ *of a decrementable register* r*. Moreover, for any* $p \in B \setminus \{l_h\}$*,* $Reg(p)$ *denotes the register affected by the ADD- or SUB-instruction labeled by* p*; for the sake of completeness, in addition* $Reg(l_h) = 1$ *is taken.*

In the *accepting* case, a computation starts with the input of an l-vector of natural numbers in its first l registers and by executing the first instruction of P (labeled by l_0); 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.

In the *generating* case, a computation starts with all registers being empty and by executing the first instruction of P (labeled by l_0); it terminates with reaching the $HALT$-instruction and the output of a k-vector of natural numbers in its last k registers. Without loss of generality, we may assume all registers except the last k output registers to be empty at the end of the computation.

In the *computing* case, a computation starts with the input of an l-vector of natural numbers in its first l registers and by executing the first instruction of P (labeled by l_0); it terminates with reaching the $HALT$-instruction and the output of a k-vector of natural numbers in its last k registers. Without loss of generality, we may assume all registers except the last k output registers to be empty at the end of the computation.

For useful results on the computational power of register machines, we refer to [11]; for example, to prove our main theorem, we need the following formulation of results

for register machines generating or accepting recursively enumerable sets of vectors of natural numbers with k components or computing partial recursive relations on vectors of natural numbers:

Proposition 1. *Deterministic register machines can accept any recursively enumerable set of vectors of natural numbers with l components using precisely $l+2$ registers. Without loss of generality, we may assume that at the end of an accepting computation all registers are empty.*

Proposition 2. *Register machines can generate any recursively enumerable set of vectors of natural numbers with k components using precisely $k+2$ registers. Without loss of generality, we may assume that at the end of a generating computation the first two registers are empty, and, moreover, on the output registers, i.e., the last k registers, no SUB-instruction is ever used.*

Proposition 3. *Register machines can compute any partial recursive relation on vectors of natural numbers with l components as input and vectors of natural numbers with k components as output using precisely $l+2+k$ registers, where without loss of generality, we may assume that at the end of a successful computation the first $l+2$ registers are empty, and, moreover, on the output registers, i.e., the last k registers, no SUB-instruction is ever used.*

In all cases it is essential that the output registers never need to be decremented.

2.2 Extended Tabled Lindenmayer Systems

An *extended tabled Lindenmayer system* (an $ET0L$ *system* for short) is a construct

$$G = (V, \Sigma, T_1, \ldots, T_n, A) \text{ where}$$

- V is a set of objects;
- $\Sigma \subseteq V$ is a set of *terminal objects*;
- T_j, $1 \leq i \leq n$, called *tables* are finite sets of non-cooperative rules over V, i.e., of the form $a \to u$ with $a \in V$ and $u \in V^*$;
- $A \in V^+$ is the axiom.

A computation in the $ET0L$ system G starts with the axiom A; then, in each computation step, a table T_j is chosen and the rules in T_j are applied to the current configuration in a parallel way. The language generated by G is the set of all terminal strings in Σ^* obtained in that way from the axiom A, i.e.,

$$L(G) = \{w \in \Sigma^* \mid A \Longrightarrow^* w\}.$$

$ET0L$ systems can also be considered as computing models working on multisets instead of strings, i.e., the axiom A is the initial multiset and the configurations are multisets on which the non-cooperative rules in the tables work in parallel. In the following, such $ET0L$ systems working on multisets will be denoted as $mET0L$ systems. Obviously, we have $\mathcal{L}(mET0L) = Ps\mathcal{L}(ET0L)$.

Remark 1. As a technical detail we mention that many authors require every table to contain at least one rule for every object in V. We observe that incomplete tables missing a rule for some $x \in V$ can easily be made complete by adding the unit rules $x \to x$ for all $x \in V$ for which so far no rule is already present in the table.

3 Simple P systems

Taking into account the well-known flattening process, which means that computations in a P system with an arbitrary (static) membrane structure can be simulated in a P system with only one membrane, e.g., see [8], in this paper we only consider simple P systems, i.e., with the simplest membrane structure of only one membrane region:

Definition 2. *A* simple P system *is a construct*

$$\Pi = (V, \mathcal{C}, \Sigma, w, \mathcal{R}, \delta)$$

where

- V *is the alphabet of* objects*;*
- $\mathcal{C} \subseteq V$ *is the alphabet of* catalysts*;*
- $\Sigma \subseteq (V \setminus \mathcal{C})$ *is the alphabet of* terminal objects*;*
- $w \in V^\circ$ *is the multiset of objects initially present in the membrane region;*
- $\mathcal{R}$ *is a finite set of* evolution rules *over* V*; these evolution rules are multiset rewriting rules* $u \to v$ *with* $u, v \in V^\circ$*;*
- δ *is the* derivation mode.

A *catalytic rule* is of the form $ca \to cv$, a *non-cooperative* rule is of the form $a \to v$, where c is a catalyst, a is an object from $V \setminus \mathcal{C}$, and v is a string from $(V \setminus \mathcal{C})^*$. A simple P system only using catalytic and non-cooperative rules is called *catalytic*, and it is called *purely catalytic* if only catalytic rules are used. The *type* of a (simple) P system only using non-cooperative rules is abbreviated by $ncoo$, the types of catalytic and purely catalytic P systems are abbreviated by cat and $pcat$, respectively.

The multiset in the single membrane region of Π constitutes a *configuration* of the P system. The *initial configuration* is given by the initial multiset w; in case of accepting or computing P systems the input multiset w_0 is assumed to be added to w, i.e., the initial configuration then is ww_0.

A transition between configurations is governed by the application of the evolution rules, which is done in the given derivation mode δ. The application of a rule $u \to v$ to a multiset M results in subtracting from M the multiset identified by u, and then in adding the multiset identified by v. Observe that each catalyst can be used (at most) once in every derivation step.

If no catalysts are used, we omit $\mathcal{C}$ and simply write $\Pi = (V, \Sigma, w, \mathcal{R}, \delta)$.

3.1 Variants of Derivation Modes

Given a P system $\Pi = (V, \mathcal{C}, \Sigma, w, \mathcal{R}, \delta)$, the set of multisets of rules applicable to a configuration C is denoted by $Appl(\Pi, C)$.

The set of all multisets of rules applicable to a given configuration can be restricted by imposing specific conditions, thus yielding the following basic derivation modes (for example, see [9] for formal definitions):

- asynchronous mode (abbreviated $asyn$): at least one rule is applied;

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

If $Appl(\Pi, C)$ is not empty, this set equals the set $Appl(\Pi, C, asyn)$ of multisets of rules applicable in the *asynchronous derivation mode* (abbreviated $asyn$).

In [5], these derivation modes are restricted in such a way that each rule can be applied at most once, thus yielding the set modes $sasyn$, $smax$, $smax_{rules}$, and $smax_{objects}$ (the sequential mode is already a set mode by definition).

In this paper we shall restrict ourselves to the derivation modes $sequ$ and max:

The set $Appl(\Pi, C, sequ)$ denotes the set of multisets of rules applicable in the *sequential derivation mode* (abbreviated $sequ$), where in each derivation step exactly one rule is applied.

The standard parallel derivation mode used in P systems is the *maximally parallel derivation mode* (max for short), in which only non-extendable multisets of rules can be applied:

$$\begin{aligned} Appl(\Pi, C, max) = \{&R \in Appl(\Pi, C) \mid \\ &\text{there is no } R' \in Appl(\Pi, C) \\ &\text{such that } R' \supset R\}. \end{aligned}$$

For some new variants of derivation modes we refer to [2,3].

3.2 Computations in a P system

The P system continues with applying multisets of rules according to the given derivation mode until there remain no applicable rules in the single region of Π, i.e., as usual, with all these variants of derivation modes as defined above, we consider *halting computations*.

We may generate or accept or even compute functions or relations. The inputs/outputs may be multisets or strings, defined in the well-known way. When the system *halts*, in case of computing with multisets we consider the number of objects from Σ contained in the membrane region at the moment when the system halts as the *result* of the underlying computation of Π.

We would like to emphasize that as results we only take the objects from the terminal alphabet Σ, especially the catalysts are not counted to the result of a computation. On the other hand, with all the proofs given in this paper, except for the catalysts – if any – no other "garbage" remains in the membrane region at the end of a halting computation, i.e., we could even omit Σ.

3.3 (Simple) P systems With Label Control

We may extend the model of a simple P system to the model of a *simple P system with label control*

$$\Pi = (V, \mathcal{C}, \Sigma, w, B, \mathcal{R}, \delta)$$

by labelling each rule in $\mathcal{R}$ by an element from a set of labels B. Then in any derivation step only rules labeled by the same label $r \in B$ are allowed to be used together. Such controlled P systems were investigated in [10].

Example 1. Consider the simple P system of type $ncoo$ without catalysts

$$\Pi = (V = \{a, b\}, \Sigma = \{a\}, w = b, B = \{1, 2\}, \mathcal{R} = \{1 : b \to bb, 2 : b \to a\}, max)$$

with the two labels 1 and 2 in B as well as the labeled rules $1 : b \to bb$ and $2 : b \to a$ in $\mathcal{R}$.

Applying rule $1 : b \to bb$ $n \geq 0$ times we obtain b^{2^n}; by applying the second rule $2 : b \to a$ we finally obtain the terminal multiset a^{2^n}. Hence, $L(\Pi) = \{a^{2^n} \mid n \geq 0\}$, a multiset language which cannot be obtained by a simple P system of type $ncoo$ without additional control mechanism.

4 Simple P systems with Prescribed Teams of Sets of Rules

We now consider a new model of simple P systems, where in one derivation step specific sets of rules – called *teams* – are applied in their assigned derivation mode. As usual, we start with a finite multiset of objects until no such team can be applied any more.

Definition 3. *A* simple P system with prescribed teams of sets of rules – *a* PPT system *for short – is a construct*

$$\Pi = (V, \Sigma, P, T_1, \ldots, T_n, A) \text{ where}$$

- V *is a set of objects;*
- $\Sigma \subseteq V$ *is a set of* terminal objects*;*
- P *is a finite set of* multiset rules*, i.e., each rule is the form* $u \to v$ *with* $u \in V^*$ *and* $v \in V^+$*;*
- *each* prescribed team T_j, $1 \leq i \leq n$*, is a finite set of sets of rules from* P *together with the associated derivation mode* δ_j *and possibly some applicability condition* K_j*, i.e.,* $T_j = (\{(K_{j,i}, R_{j,i}, \delta_{j,i}) \mid 1 \leq i \leq n_j\})$*, where the* $R_{j,i} \subseteq P$ *are finite sets of rules from* P*;*
- $A \in V^\circ$ *is a finite multiset of initial objects from* V*.*

As usual, a rule $p \in P,\ p = u \to v$, is called *applicable* to a *configuration*, i.e., an object $x \in V^\circ$, if and only if u is a subset of x. The set of all rules applicable to x is denoted by $Appl(\Pi, x)$

The number n of teams is called the *degree* of Π. $|T_j|$ is called the *size* of the prescribed team T_j. If all prescribed teams have at most size s, then Π is called a *PPT*

system of size s. If the number of rules in the sets of rules is at most m, then Π is called a *PPT system of rule size* m. Π is called a *PPT system of type* (n, s, m), if it is of degree n, size s, and rule size m. Moreover, if all rules in the sets of rules in the teams are of a specific type α (for example $ncoo$), we call Π a *PPT system of type* $(\alpha; n, s, m)$.

The family of sets of multisets generated/accepted by PPT system of type $(\alpha; n, s, m)$ is denoted by $\mathcal{L}(PPT_{gen}(\alpha; n, s, m))$/$\mathcal{L}(PPT_{acc}(\alpha; n, s, m))$. Any of the parameters n, s, m can be replaced by $*$, if the number cannot be bounded; α can also be omitted.

As derivation modes, we will restrict ourselves to the sequential derivation mode $sequ$ and the maximally parallel derivation mode max.

The conditions $K_{j,i}$ in the most general case can be any computable/recursive features of the underlying configuration. Here we essentially will consider random context conditions, i.e., $K_{j,i} = (P_{j,i}, Q_{,j,i})$, where $P_{j,i}$ and $Q_{j,i}$ are finite sets of multisets over V; $P_{j,i}$ is the set of *permitting contexts* and $Q_{j,i}$ is the set of *forbidden contexts*. The random context condition $K_{j,i} = (P_{j,i}, Q_{j,i})$ is fulfilled by a multiset x if and only if x contains each multiset in $P_{j,i}$, but none of the multisets in $Q_{j,i}$. If no conditions $K_{j,i}$ are specified, we simply write $T_j = \{(R_{j,i}, \delta_{j,i}) \mid 1 \leq i \leq n_j\}$.

If different derivation modes appear in a team, the whole PPT system is called *non-homogenous*. The system is called *locally homogenous*, if for all teams, the sets of rules in the team all are applied in the same derivation mode δ_j, and we write $T_j = (\{R_{j,i} \mid 1 \leq i \leq n_j\}, \delta_j)$, $1 \leq j \leq n$. Finally, the system is called *globally homogenous* if the derivation mode is the same δ for all T_j, and we only write $T_j = \{R_{j,i} \mid 1 \leq i \leq n_j\}$ and specify δ by writing $\Pi = (V, \Sigma, P, T_1, \ldots, T_n, \delta, A)$.

Computations in a PPT system Given a prescribed team of sets of rules

$$T_j = (\{(K_{j,i}, R_{j,i}, \delta_{j,i}) \mid 1 \leq i \leq n_j\})$$

with the derivation modes $\{\delta_{j,i} \mid 1 \leq i \leq n_j\} \subseteq \{sequ, max\}$, a derivation step with T_j on the configuration x can be carried out in the following way:

1. we choose a multiset of rules $R \in Appl(\Pi, x)$; the multiset of objects which the rules in R bind is denoted by $Bind(R, x)$, the multiset of objects in $x \setminus Bind(R, x)$ is denoted by $Idle(R, x)$;
2. we now for all $1 \leq i \leq n_j$ check whether x fulfills the applicability conditions $K_{j,i}$;
3. each rule in R must be assigned to one of the sets $R_{j,i}$ for which the applicability condition $K_{j,i}$ is fulfilled, yielding the multiset of rules $R'_{j,i}$; the multiset of objects which the rules in $R'_{j,i}$ bind is denoted by $Bind(R, R'_{j,i}, K)$;
4. for $\delta_{j,i} = max$ we now check that $R'_{j,i}$ cannot be extended by using an additional rule from $R_{j,i}$ on objects from $Idle(R, x)$;
5. for $\delta_{j,i} = sequ$ we first check whether $|R'_{j,i}| = 1$; if $|R'_{j,i}| = 1$, then we have to check that no rule from $R_{j,i}$ can be applied to objects from $Idle(R, x)$; if $|R'_{j,i}| > 1$ the check fails, i.e., R cannot be applied;
6. if all checks from above have been passed correctly, the multiset of rules R can be applied to the current configuration x.

We emphasize that the rule sets in a team compete for the objects available in the underlying configuration, but at the end each set of rules for itself makes its part of the transition from the underlying configuration to the next configuration in a correct way according to its assigned derivation mode, as no additional rule could bind an idle object. Moreover, we observe that a set of rules $R_{j,i}$ from T_j can only be chosen if the applicability condition $K_{j,i}$ is fulfilled by x. Finally, a team T_j can only be applied if the multiset of rules obtained by the procedure described above is not empty.

As some variant of the general model we may also consider prescribed teams of sets of rules for which the applicability conditions $K_{j,i}$ are the same for all $1 \leq i \leq n_j$, i.e., just one condition K_j, and then we write

$$T_j = (K_j, \{(R_{j,i}, \delta_{j,i}) \mid 1 \leq i \leq n_j\})$$

and can simplify the procedure for applying T_j by first checking that the current configuration fulfills K_j.

As a first example we show how label control can easily be simulated by teams of sets of rules:

Example 2. Consider the globally homogenous PPT system of type $ncoo$

$$\Pi = (V = \{a, b\}, \Sigma = \{a\}, P, T_1, T_2, max, b) \text{ where}$$

$P = \{b \to bb, b \to a\}$, $T_1 = \{\{b \to bb\}\}$, and $T_2 = \{\{b \to a\}\}$. Π is a globally homogenous PPT system of type $(ncoo; 2, 1, 1)$.

Applying team T_1, i.e., the rule $b \to bb$, in the maximally parallel way $n \geq 0$ times we obtain b^{2^n}; by applying the second team T_2, i.e., the rule $b \to a$, in the maximally parallel way once, we finally obtain the terminal multiset a^{2^n} as in Example 1. Hence, we conclude $L(\Pi) = \{a^{2^n} \mid n \geq 0\}$ as well as

$$\{a^{2^n} \mid n \geq 0\} \in \mathcal{L}(gh(max)PPT_{gen}(ncoo; 2, 1, 1)),$$

with the prefix $gh(max)$ indicating that we consider globally homogenous PPT systems working in the derivation mode max.

5 PPT Systems Simulating P systems With Label Control

As can already be guessed by looking at Example 2, PPT systems can easily simulate P systems with label control – *without catalysts* – which are working in the derivation mode max by putting the rules with the same labels into one team:

Theorem 1. *Every P systems with label control Π, without catalysts, and working in the derivation mode max, can be simulated by a PPT system of type $(n, 1, *)$, where n is the number of different labels for the rules in Π.*

Proof. Given a simple P system with label control, without catalysts,

$$\Pi = (V, \Sigma, w, B, \mathcal{R}, max)$$

where each rule in $\mathcal{R}$ is labelled by an element from a set of labels B, $B = \{l_j \mid 1 \leq j \leq n\}$, we construct a globally homogenous PPT system Ψ of degree n and size 1, simulating (the computations of) Π:

$$\Psi = (V, \Sigma, P, T_1, \ldots, T_n, max, w)$$

where we define

$$P = \{p \mid l_j : p \in \mathcal{R},\ 1 \leq j \leq n\}$$

as well as the teams T_j, $1 \leq j \leq n$, as follows:

$$T_j = \{\{p \mid l_j : p \in \mathcal{R}\}\}$$

By definition, the size of T_j is 1, whereas the number of rules in the single set of rules in a team can be arbitrarily large.

We observe that applying the set of rules in in the team T_j in Ψ in the maximally parallel way has the same effect as applying exactly the rules with label l_j in Π in the maximally parallel way. □

6 PPT Systems Simulating $mET0L$ Systems

We now show that the computational power of $mET0L$ systems equals the computational power of globally homogenous PPT systems of type $ncoo$ without applicability conditions working in the derivation mode max.

Theorem 2. *Every $mET0L$ system with n tables can be simulated by a globally homogenous PPT system of type $ncoo$, degree n, and size 1 without applicability conditions working in the derivation mode max.*

Proof. The $mET0L$ system $G = (V, \Sigma, T'_1, \ldots, T'_n, A)$ can be simulated by the globally homogenous PPT system of type $ncoo$, degree n, and size 1 without applicability conditions working in the derivation mode max

$$\Pi = (V, \Sigma, P, T_1, \ldots, T_n, max, A)$$

where we simply take

$$T_j = \{T'_j \setminus \{a \to a \mid a \in \Sigma\}\},\ \ 1 \leq j \leq n,$$

i.e., the work of the table T'_j is simulated by the single set of rules in the team T_j of the PPT system Π.

We observe that we have to exclude the unit rules $a \to a$ for the terminal symbols $a \in \Sigma$, from the sets of rules T'_j, $1 \leq j \leq n$, in order to ensure that Π halts as soon as a terminal configuration (i.e., a configuration only containing terminal symbols) has been reached. Finally, we mention that every (useless) team T_j of the form $\{\emptyset\}$ is to be omitted. □

In order to show the inverse inclusion, we need the following lemma:

Lemma 1. *Every globally homogenous PPT system of degree n and size k without applicability conditions working in the derivation mode max can be simulated by a globally homogenous PPT system of degree n and size 1 without applicability conditions working in the derivation mode max.*

Proof. The globally homogenous PPT system of degree n and size k without applicability conditions working in the derivation mode max

$$\Pi = (V, \Sigma, P, T_1, \ldots, T_n, max, A)$$

with $T_j = \{R_{j,i} \mid 1 \leq j \leq n_j\}$, $1 \leq j \leq n$, can be simulated by the corresponding globally homogenous PPT system of degree n and only size 1 without applicability conditions working in the derivation mode max

$$\Pi' = (V, \Sigma, P, T'_1, \ldots, T'_n, max, A)$$

where we take $T'_j = \{\{y \mid y \in R_{j,i},\ 1 \leq i \leq n_j\}\}$. We observe that by definition the rules in the $R_{j,i}$ work in parallel on the underlying configurations in the same way if they are grouped in the $R_{j,i}$ or just in one set of rules $\{y \mid y \in R_{j,i},\ 1 \leq i \leq n_j\}$. We observe that Π' again is of degree n, but only of size 1. □

Based on this lemma, we now can show how a globally homogenous PPT system of type $ncoo$, degree n without applicability conditions working in the derivation mode max can be simulated by an $mET0L$ system with n tables:

Theorem 3. *Every globally homogenous PPT system of type $ncoo$, degree n, and size k without applicability conditions working in the derivation mode max can be simulated by an $mET0L$ system with n tables.*

Proof. According to Lemma 1, without loss of generality we may assume that the size k is only one. Hence, we may start with a globally homogenous PPT system of type $ncoo$, degree n, and size 1 without applicability conditions working in the derivation mode max

$$\Pi = (V, \Sigma, P, T_1, \ldots, T_n, max, A)$$

where for the teams T_j we have $T_j = \{T'_j\}$, $1 \leq j \leq n$, with T'_j being a set of non-cooperative rules.

Then the $mET0L$ system

$$G = (V, \Sigma, T'_1, \ldots, T'_n, A)$$

simulates the (computations of the) PPT system Π, as the work of the table T'_j simulates the application of the team T_j of the PPT system Π with the single set of rules T'_j.

As a technical detail we mention that the tables T'_j have to be extended by unit rules $x \to x$ for every $x \in V$ for which no rule is already present in it, in order to fulfill the requirement for $ET0L$ systems as already discussed in Remark 1. □

In sum, we have shown the following result (where $gh(max)PPT(ncoo)$ denotes the globally homogenous PPT systems of type $ncoo$ working in the derivation mode max):

Theorem 4. $\mathcal{L}(mET0L) = \mathcal{L}(gh(max)PPT(ncoo))$.

7 PPT Systems Simulating [Purely] Catalytic P systems

We first consider purely catalytic P systems, which correspond to PPT systems where all sets of non-cooperative rules in the unique team work in the sequential derivation mode.

Theorem 5. *Every purely catalytic P system with n catalysts can be simulated by a corresponding globally homogenous PPT system of type $ncoo$, degree 1, and size n without applicability conditions working in the derivation mode $sequ$.*

Proof. The purely catalytic P system with n catalysts

$$\Pi = (V, \mathcal{C}, \Sigma, w, \mathcal{R}, max),$$

i.e., $\mathcal{C} = \{c_k \mid 1 \leq k \leq n\}$, can be simulated by the globally homogenous PPT system of type $ncoo$, degree 1, and size n without applicability conditions working in the sequential derivation mode

$$\Pi = (V, \Sigma, P, T_1, sequ, w)$$

with $T_1 = \{R_{1,k} \mid 1 \leq k \leq n\}$ and

$$P = \{a \to u \mid c_k a \to c_k u \in \mathcal{R} \text{ for some } 1 \leq \mathrm{k} \leq \mathrm{n}\}$$

as well as

$$R_{1,k} = \{a \to u \mid c_k a \to c_k u \in \mathcal{R}\},\ 1 \leq k \leq n.$$

The applicability of the unique team works by applying (at most) one rule $a \to u$ from each $R_{1,k}$, $1 \leq k \leq n$, which corresponds to applying the corresponding rule $c_k a \to c_k u$ in Π. □

Simple catalytic P systems with n catalysts can be mimicked by simple P systems with prescribed teams of sets of rules, where as in the case of purely catalytic P systems the work of the n catalysts is simulated by n sets of non-cooperative rules in the team working in the sequential mode and one additional set of non-cooperative rules simulates the set of non-catalytic rules working in the maximally parallel derivation mode.

Theorem 6. *Every catalytic P system with n catalysts can be simulated by a corresponding PPT system of type $ncoo$, degree 1, and size $n+1$ without applicability conditions with n components of the unique team working in the derivation mode $sequ$ and one working in the derivation mode max.*

Proof. The catalytic P system with n catalysts

$$\Pi = (V, \mathcal{C}, \Sigma, w, \mathcal{R}, max),$$

i.e., $\mathcal{C} = \{c_k \mid 1 \leq k \leq n\}$, can be simulated by the PPT system of type $ncoo$, degree 1, and size $n+1$ without applicability conditions

$$\Pi = (V, \Sigma, P, T_1, w)$$

with $T_1 = \{R_{1,k} \mid 1 \leq k \leq n+1\}$ and

$$P = \{a \to u \mid c_k a \to c_k u \in \mathcal{R} \text{ for some } 1 \leq \mathrm{k} \leq \mathrm{n}\} \cup \{\mathrm{a} \to \mathrm{u} \mid \mathrm{a} \to \mathrm{u} \in \mathcal{R}\}$$

as well as

$$R_{1,k} = (\{a \to u \mid c_k a \to c_k u \in \mathcal{R}\}, sequ),\ 1 \leq k \leq n,$$

and

$$R_{1,n+1} = (\{a \to u \mid a \to u \in \mathcal{R}\}, max),$$

The application of the unique team works by applying (at most) one rule $a \to u$ from each $R_{1,k}$, $1 \leq k \leq n$, which corresponds to applying the corresponding rule $c_k a \to c_k u$ in Π, as well as the rules in $R_{1,n+1}$ in the maximally parallel way.

Observe that in contrast to the globally homogenous PPT systems for simulating purely catalytic P systems, we now have non-homogenous PPT systems, as we have to use both derivation modes $sequ$ and max in the unique team. □

According to Propositions 2 and 1, from Theorems 5 and 6 we immediately infer the following results:

Corollary 1. *For any $d \geq 1$, we have*

1. $N^d\mathcal{L}(RE) = \mathcal{L}(PPT_{gen}(ncoo; 1, 3, *))$ *and*
2. $N^d\mathcal{L}(RE) = \mathcal{L}(PPT_{acc}(ncoo; 1, d+3, *))$*;*

moreover,
$Ps\mathcal{L}(RE) = \mathcal{L}(PPT_{gen}(ncoo; 1, *, *)) = \mathcal{L}(PPT_{acc}(ncoo; 1, *, *))$.
In all cases, the degree of the PPT systems is only 1.

8 PPT Systems Directly Simulating Register Machines

In this section we show how register machines can directly be simulated by PPT systems in an easy way when using applicability conditions represented by sets of permitting and forbidden contexts, see the defintion on page 15.

Theorem 7. *The computations of a register machine can be simulated by a globally homogenous PPT system of type $ncoo$ and size* 2 *using permitting and forbidden contexts as applicability conditions in the sequential derivation mode.*

Proof. Consider the register machine

$$M = (m, B, l_0, l_h, R)$$

with B_{ADD} denoting the set of labels of *ADD*-instructions $p : (ADD(r), q, s)$ of arbitrary registers r, and $B_{SUB(r)}$ denoting the set of labels of all *SUB*-instructions $p : (SUB(r), q, s)$ of a decrementable register r. Moreover, for any $p \in B \setminus \{l_h\}$, $Reg(p)$ denotes the register affected by the *ADD*- or *SUB*-instruction labeled by p.

We now construct the globally homogenous PPT system of size 2 working in the sequential derivation mode using permitting and forbidden contexts as applicability conditions

$$\Pi = (V, \Sigma, P, T_1, \ldots, T_n, sequ, w).$$

Throughout the computation of Π, one symbol $p \in B$ represents the instruction from the register machine to be simulated next, and the number of symbols a_r represents the contents of register r, $1 \leq r \leq m$. Hence, we start with the axiom $w = l_0 w_0$, where w_0 represents the initial contents of the registers. If the final label l_h appears, we know that the computation in M has been successful and finally can erase l_h, so that a multiset over Σ remains as the result of the computation.

Moreover, the set of symbols V only consists of the labels in B and the symbols a_r representing the registers:

$$V = B \cup \{a_r \mid 1 \leq r \leq m\}.$$

The set of terminal symbols Σ consists of only those symbols from the set $\{a_r \mid 1 \leq r \leq m\}$ which represent output registers.

We need the following simple non-cooperative rules in P for the simulation of the instructions of M:

$$\begin{aligned} P = &\{p \to qa_r,\ p \to sa_r \mid p : (ADD(r), q, s) \in R\} \\ &\cup \{p \to q,\ p \to s \mid p : (SUB(r), q, s) \in R\} \\ &\cup \{a_r \to \lambda \mid 1 \leq r \leq m \text{ and r is a decrementable register}\} \\ &\cup \{l_h \to \lambda\} \end{aligned}$$

The teams of sets of rules with applicability conditions for simulating the instructions of the register machine defined below form the teams $T_1, \ldots, T_n$.

– $p : (ADD(r), q, s), p \in B_{ADD}$, is simulated by the team

  $R_p = \{((\{p\}, \emptyset), \{p \to qa_r,\ p \to sa_r\})\}$
  which can also be written in a simpler way as
  $R_p = \{\{p \to qa_r, p \to sa_r\}\}$, because the rules in this set of rules in this team can anyway only be applied if p is present.

– $p : (SUB(r), q, s); p \in B_{SUB(r)}$, is simulated by the team of sets of rules

  $R_{p,1} = ((\{p, a_r\}, \emptyset), \{\{p \to q\}, \{a_r \to \lambda\}\})$

  (both the presence of p and a_r have to be checked in order to guarantee that both rules $p \to q$ and $a_r \to \lambda$ are applied) as well as by the team of sets of rules

  $R_{p,2} = \{((\{p\}, \{a_r\}), \{p \to s\})\}$

  (both the presence of p and the absence of a_r have to be checked to guarantee that we only proceed to label s if no symbol a_r is present).

– $l_h : HALT$ is simulated by the team
$R_h = \{((\{l_h\}, \emptyset), \{l_h \to \lambda\})\}$,
which can also be written in a simpler way as
$R_h = \{\{l_h \to \lambda\}\}$.

Only in the case of applying a team $R_{p,1}$ two rules are applied in one step, otherwise only one rule is applied.

The application of a team is only possible if the current label symbol p appears in the underlying configuration, and in the case of a *SUB*-instruction also the presence/absence of $a_{Reg(p)}$ is correctly given. Throughout the computation in Π exactly one of the teams of sets of rules is applicable before finally a configuration only containing terminal symbols is reached. □

9 Computational Completeness

According to Subsection 2.1, register machines are a model being computationally complete for multisets, i.e., every partial recursive relation on multisets can be computed by a register machine. Hence, from Theorem 7 we immediately infer the following result:

Theorem 8. *PPT systems of type* $ncoo$ *and size* 2 *using applicability conditions represented by sets of permitting and forbidden contexts in the sequential derivation mode are computationally complete for multisets.*

Instead of non-cooperative rules we can also use the simple rules of insertion and deletion:

– $I(a)$ inserts an object a in the underlying multiset (and can be interpreted as the rule $\lambda \to a$).
– $D(a)$ deletes an object a from the underlying multiset, if at least one a is present (and can be interpreted as the rule $a \to \lambda$).

Based on the proof of Theorem 7, we easily get the following result for PPT systems using insertion and deletion rules (called PPT systems of type $InsDel$ for short):

Corollary 2. *PPT systems of type* $InsDel$ *and size* 3 *using applicability conditions represented by sets of permitting and forbidden contexts in the sequential derivation mode are computationally complete for multisets.*

Proof. Consider the register machine

$$M = (m, B, l_0, l_h, R)$$

with B_{ADD} denoting the set of labels of *ADD*-instructions $p : (ADD(r), q, s)$ of arbitrary registers r, and $B_{SUB(r)}$ denoting the set of labels of all *SUB*-instructions $p : (SUB(r), q, s)$ of a decrementable register r. Moreover, for any $p \in B \setminus \{l_h\}$, $Reg(p)$ denotes the register affected by the *ADD*- or *SUB*-instruction labeled by p.

We now construct the globally homogenous PPT system of type $InsDel$ and size 3 working in the sequential derivation mode using permitting and forbidden contexts as applicability conditions

$$\Pi = (V, \Sigma, P, T_1, \ldots, T_n, sequ, w).$$

Throughout the computation of Π, one symbol $p \in B$ represents the instruction from the register machine to be simulated next, and the number of symbols a_r represents the contents of register r, $1 \leq r \leq m$. Hence, we start with the axiom $w = l_0 w_0$, where w_0 represents the initial contents of the registers. If the final label l_h appears, we know that the computation in M has been successful and finally can erase l_h, so that a multiset over Σ remains as the result of the computation.

Moreover, the set of symbols V only consists of the labels in B and the symbols a_r representing the registers:

$$V = B \cup \{a_r \mid 1 \leq r \leq m\}.$$

The set of terminal symbols Σ consists of only those symbols from the set $\{a_r \mid 1 \leq r \leq m\}$ which represent output registers.

We need the following simple insertion and deletion rules in P for the simulation of the instructions of M:

$$\begin{aligned} P = &\{I(p), D(p) \mid p \in B\} \\ &\cup \{I(a_r) \mid 1 \leq r \leq m\} \\ &\cup \{D(a_r) \mid 1 \leq r \leq m \text{ and r is a decrementable register}\} \end{aligned}$$

The teams of sets of rules with applicability conditions for simulating the instructions of the register machine defined below form the teams $T_1, \ldots, T_n$.

- $p : (ADD(r), q, s)$, $p \in B_{ADD}$, is simulated by the team

$$\begin{aligned} R_p = \{&((\{p\}, \emptyset), \{D(p)\}), \\ &((\{p\}, \emptyset), \{I(q),\ I(s)\}), \\ &((\{p\}, \emptyset), \{I(a_r)\})\}, \end{aligned}$$

  which in a shorter way could be written as

$$R_p = ((\{p\}, \emptyset), \{\{D(p)\}, \{I(q),\ I(s)\}, \{I(a_r)\}\})$$

  as the applicability condition $(\{p\}, \emptyset)$ is required for all sets of rules in the team. Observe that the size of these teams now is 3!

- $p : (SUB(r), q, s)$; $p \in B_{SUB(r)}$, is simulated by the team

$$R_{p,1} = ((\{p, a_r\}, \emptyset), \{\{D(p)\}, \{I(q)\}, \{D(a_r)\}\})$$

  (again the size of these teams now is 3)

as well as by the team

$$R_{p,2} = ((\{p\}, \{a_r\}), \{\{D(p)\}, \{I(s)\}\})$$

– $l_h : HALT$ is simulated by the team

$$R_h = \{\{D(l_h)\}\}.$$

Throughout the computation in Π exactly one of the teams of sets of rules is applicable before finally a configuration only containing terminal symbols is reached. □

As is well-known, catalytic and purely catalytic P systems are computationally complete (for multisets), too. Therefore, based on the results shown in Section 7 we get the following results (compare with Corollary 1):

Corollary 3. *Globally homogenous PPT systems of type* $ncoo$ *and degree* 1 *without applicability conditions working in the sequential derivation mode are computationally complete for multisets.*

Corollary 4. *PPT systems of type* $ncoo$ *and degree* 1 *without applicability conditions with one set of rules in the unique team working in the maximally parallel derivation mode and all the other sets of rules working in the sequential derivation mode are computationally complete for multisets.*

10 Conclusion

In this paper we have considered the concept of using prescribed teams of sets of rules being applied in different derivation modes, with the applicability of a team possibly depending on a given condition. Among other general results, we have shown that simple purely catalytic P systems with n catalysts can be simulated by simple P systems with one prescribed team of sets of rules with all n sets of non-cooperative rules in this team working in the sequential derivation mode thus simulating the work of the n catalysts, as well as that simple catalytic P systems with n catalysts can be simulated by simple P systems with one prescribed team of sets of non-cooperative rules, where one set of this team works in the maximally parallel derivation mode and the other n sets of rules in this team work in the sequential mode thus again simulating the work of the n catalysts. From the results known for simple (purely) catalytic P systems, we immediately infer the corresponding computational completeness results for the new variants of simple P systems, with on one hand only using non-cooperative rules and no applicability conditions. On the other hand, we can show computational completeness for different variants of simple P systems with prescribed teams of sets of non-cooperative rules by directly simulating register machines, thereby using applicability conditions given as sets of (atomic) promoters and inhibitors.

Throughout this paper, we have restricted ourselves to the two basic derivation modes, i.e., the sequential one and the maximally parallel derivation mode. A thorough investigation of simple P systems with prescribed teams of sets of rules using other derivation modes remains for future research. Moreover, other kinds of rules might be used, too; for example, insertion and deletion rules instead of non-cooperative rules as already used for simulating register machines in Corollary 2.

Acknowledgements

The authors gratefully acknowledge the useful comments of the anonymous referees.

Artiom Alhazov acknowledges project 20.80009.5007.22 "Intelligent information systems for solving ill-structured problems, processing knowledge and big data" by the National Agency for Research and Development.

References

1. Artiom Alhazov, Rudolf Freund, and Sergiu Ivanov. When catalytic P systems with one catalyst can be computationally complete. *Journal of Membrane Computing*, 3(3):170–181, 2021.
2. Artiom Alhazov, Rudolf Freund, Sergiu Ivanov, and Marion Oswald. Variants of simple purely catalytic P systems with two catalysts. In György Vaszil, Claudio Zandron, and Gexiang Zhang, editors, *International Conference on Membrane Computing ICMC 2021, Proceedings*, pages 39–53, 2021.
3. Artiom Alhazov, Rudolf Freund, Sergiu Ivanov, and Sergey Verlan. Variants of simple P systems with one catalyst being computationally complete. In György Vaszil, Claudio Zandron, and Gexiang Zhang, editors, *International Conference on Membrane Computing ICMC 2021, Proceedings*, pages 21–38, 2021.
4. Artiom Alhazov, Rudolf Freund, Sergiu Ivanov, and Sergey Verlan. Prescribed teams of rules working on several objects. In Jérôme Durand-Lose and György Vaszil, editors, *Machines, Computations, and Universality – 9th International Conference, MCU 2022, Debrecen, Hungary, August 31 – September 2, 2022, Proceedings*, volume 13419 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2022.
5. Artiom Alhazov, Rudolf Freund, and Sergey Verlan. P systems working in maximal variants of the set derivation mode. In Alberto Leporati, Grzegorz Rozenberg, Arto Salomaa, and Claudio Zandron, editors, *Membrane Computing – 17th International Conference, CMC 2016, Milan, Italy, July 25-29, 2016, Revised Selected Papers*, volume 10105 of *Lecture Notes in Computer Science*, pages 83–102. Springer, 2017.
6. E. Csuhaj-Varjú, J. Dassow, and J. Kelemen. *Grammar Systems: A Grammatical Approach to Distribution and Cooperation*. Topics in computer mathematics. Gordon and Breach, 1994.
7. Jürgen Dassow and Gheorghe Păun. *Regulated Rewriting in Formal Language Theory*. Springer, 1989.
8. Rudolf Freund, Alberto Leporati, Giancarlo Mauri, Antonio E. Porreca, Sergey Verlan, and Claudio Zandron. Flattening in (tissue) P systems. In Artiom Alhazov, Svetlana Cojocaru, Marian Gheorghe, Yurii Rogozhin, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 8340 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2014.

9. Rudolf Freund and Sergey Verlan. A formal framework for static (tissue) P systems. In George Eleftherakis, Petros Kefalas, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 4860 of *Lecture Notes in Computer Science*, pages 271–284. Springer, 2007.
10. Kamala Krithivasan, Gh. Păun, and Ajeesh Ramanujan. On controlled P systems. *Fundam. Inform.*, 131(3–4):451–464, 2014.
11. Marvin L. Minsky. *Computation. Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, NJ, 1967.
12. Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.
13. Gheorghe Păun. *Membrane Computing: An Introduction*. Springer, 2002.
14. Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors. *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.
15. Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages*. Springer, 1997.
16. The P Systems Website. http://ppage.psystems.eu/.

P Systems with Reactive Membranes

Artiom Alhazov[1], Rudolf Freund[2], Sergiu Ivanov[3], David Orellana-Martín[4,5], Antonio Ramírez-de-Arellano[4,5], José Antonio Rodríguez Gallego[6]

[1] State University of Moldova,
Vladimir Andrunachievici Institute of Mathematics and Computer Science
Academiei 5, Chișinău, MD-2028, Moldova
artiom@math.md

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

[3] IBISC, Univ. Évry, Paris-Saclay University
23, boulevard de France 91034 Évry, France
sergiu.ivanov@ibisc.univ-evry.fr

[4] Research Group on Natural Computing,
Department of Computer Science and Artificial Intelligence,
Universidad de Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
{dorellana,aramirezdearellano}@us.es

[5] SCORE Laboratory, I3US, Universidad de Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain

[6] Departmento de Construcciones Arquitectónicas I,
Universidad de Sevilla
Avda. Reina Mercedes 2, 41012 Sevilla, Spain
jrodriguez14@us.es

Abstract. Membranes are one of the key concepts in P systems and membrane computing, and a lot of research focuses on their properties and possible extensions: membrane division, membrane dissolution, mobile membranes, etc. In this work, we explore the possibility of using membranes for thinking about the emergence of milieu separations at the origins of life. We propose a new variant of P systems with reactive membranes, in which every symbol is initially surrounded by an elementary membrane, and in which membranes can non-deterministically merge and split, leading to the formation of bigger and more complicated membranes. We show that such non-deterministic splitting and merging does not seem to radically affect the computational power: P systems with reactive membranes and non-cooperative rules generate at least all semilinear languages, and cooperative rules allow for simulating partially blind register machines. We briefly discuss using P systems with reactive membranes for illustrating the emergence of autocatalytic cycles, but actual constructions are left for future work.

Keywords: origins of life, P systems, self-assembly, space and topology.

1 Introduction

Membrane computing is a multiset rewriting-based theoretical construct for natural computing, originally introduced by Gheorghe Păun in [23], and extensively studied ever since. The structure of a membrane system—or a P system—mimics that of a living cell: it is a hierarchical family of nested membranes, each carrying a multiset of abstract objects and multiset rewriting rules. The objects can be seen as formal representations of chemical species, and the rewriting rules capture the biochemical interactions these species may have.

Beyond the obvious abstraction arrow between biochemical species and formal objects, membrane computing parallels biological systems in another interesting way. In biology, centralization of functions is quite frequent (e.g., central nervous systems, specialized organs, etc.), but not fundamental. Only as a first example, simple organisms carry out many activities in a decentralized way, weakly orchestrated by interference between related processes. Take unicellular organisms: a computer scientist may be tempted to consider the genetic material as the program for the whole cell, but it is now known (e.g. [10]) that the relationship between the genotype and the phenotype—its manifestation—is very far from the clear program–execution duality imbuing computer science. As an abstraction of hierarchically structured biochemistry, P systems inherit this weakly centralized way of functioning, which makes them a good candidate for supporting the thought process about some grand laws of biology.

In this paper, we lay the groundwork for using P systems as a tool for thinking about some aspects of the emergence of life. The particular question we focus on is the emergence of milieu separations, which played an essential role as they allowed to isolate and protect relevant processes from the environment [11]. Since P systems already include membranes as first-class citizens, we will use them as a framework for thinking about the emergence of complex regions from simpler ones.

The approach we take here is to posit that every copy of a symbol a is endowed with some *elementary space*—a membrane which initially only contains the multiset a. Two such symbols can bond by merging their membranes, thereby yielding a more complex membrane containing 2 symbols. Such membranes can further merge, yielding bigger and bigger regions. Dually, membranes containing multiple symbols can split into a pair of simpler membranes, with the contents of the original larger membrane distributed across its children. This is in fact membrane separation (e.g. [7,21,22]).

Measuring the complexity of a membrane by the number of symbols it contains is simultaneously simple and appropriate: cooperative evolution rules are allowed, so more symbols means more applicable rules and therefore more interactions. In the setup we establish in this paper, all membranes share the same common set of evolution rules. The rules can naturally be seen as defining a chemistry, while membrane merging and splitting can on the other hand be seen as some lower-level ground laws governing who may interact with whom, i.e. the topology of the interactions. The resulting abstract structures featuring merging and splitting membranes are therefore systems in which objects interact based on the non-deterministic variations in their neighborhoods. We call such structures P systems with *reactive membranes*.

Before using P systems with reactive membranes as a formal tool, a number of important details have to be sorted out. In particular, we show that the definition of

membrane splitting and merging turns out to be rather nontrivial. Choosing when to recover and how to interpret the result impacts the form of the computations of a P system with reactive membranes, as well as what kind of results one can expect. Finally, this P system variant as informally introduced above and defined in Section 3 is very basic and may be extended in many ways, as we briefly show in Section 5.

Note that we do not pretend to faithfully model in any way the processes which happened at the origins of life. Rather, we acknowledge the exceptional complexity of these processes, as well as the impossibility to experimentally verify any of the related hypotheses (e.g., [17]). The intended role of P systems with reactive membranes is to serve as a formal vehicle for an otherwise abstract thought process, to help to verify the latter in a basic way, and to help the researcher to deal with complex questions. This approach is similar in spirit to the works [26,27], in which sign Boolean networks are used with a similar purpose.

P systems with reactive membranes are naturally part of the lineage of P systems with active membranes, and feature similarities with other variants in this family. Among closely related variants are P systems with mobile membranes, in which membranes are allowed to move across the membrane structure, and thereby change their immediate neighbors [8,9,20]. Another variant are P systems with vesicles of multisets, in which multisets are contained in vesicles, which are contained in membranes, implying that entire multisets of symbols can travel between different membranes, thereby activating different sets of rules [5,15]. A key specificity of P systems with reactive membranes setting them apart from the other variants is that membrane splitting and merging is global, compulsory, and independent of the contents of the membranes or of the rules. This feature introduces a basic form of space, through which the entities travel and in which they interact in their immediate neighborhood. On the other hand, compulsory splitting and merging to be compulsary modulates the computational power in interesting ways.

This paper is structured as follows. In Section 2 we recall some basic concepts from formal languages and P systems. In Section 3 we introduce P systems with reactive membranes, and define the precise semantics of splitting and merging of membranes. In Section 4 we present some first results concerning the computational power of P systems with reactive membranes, with non-cooperative and cooperative rules. In Section 5 we give some examples of possible extensions to the new variant. Finally, in Section 6, we discuss the potential of reactive membranes for illustrating some processes which happened at the origins of life, as well as some aspects of their computational power.

2 Preliminaries

For an alphabet V, a finite non-empty set of abstract symbols, the free monoid generated by V under the operation of concatenation, i.e., the set containing all possible strings over V, is denoted by V^*. The empty string is denoted by λ, and $V^* \backslash \{\lambda\}$ is denoted by V^+.

For two natural numbers $a, b \in \mathbb{N}$, $a \leq b$, we use the notation $[a..b]$ to refer to the interval of natural numbers between a and b, both included: $[a..b] = \{a, a+1, \ldots, b\}$.

Given a finite set A, a multiset over A is a function $w : A \to \mathbb{N}$, assigning the number of times an element of A appears in w. The infinite set of all multisets over A is denoted by A°. The family of finite sets of finite multisets over A is denoted by $\mathcal{P}_{fin}(A^\circ)$.

To spell out a multiset w, we will generally write any string containing exactly the same symbols with the same multiplicities. For example, the strings aab, aba, ba^2 will be used to refer to the same multiset w with the property $w(a) = 2$, $w(b) = 1$, and $w(c) = 0$ for all $c \in A \setminus \{a, b\}$. We denote the empty multiset by Λ, i.e., $\forall a \in A : \Lambda(a) = 0$, and its string representation is λ, the empty string.

Given two multisets w_1 and w_2 over A, their multiset union $w_1 \cup w_2$ is defined as $(w_1 \cup w_2)(a) = w_1(a) + w_2(a)$, for all $a \in A$. As their multiset intersection $w_1 \cap w_2$ we define $(w_1 \cap w_2)(a) = \min\{w_1(a), w_2(a)\}$. A restriction of the multiset $w : A \to \mathbb{N}$ to the subset $B \subseteq A$ is the multiset $w|_B : A \to \mathbb{N}$ with the property that $w|_B(a) = w(a)$ if $a \in B$ and $w|_B(a) = 0$ otherwise.

The family of regular, context-free, and recursively enumerable string languages is denoted by $\mathcal{L}(REG)$, $\mathcal{L}(CF)$, and $\mathcal{L}(RE)$, respectively. For a family of languages FL, the family of Parikh images of languages in FL is denoted by $PsFL$. As $Ps\mathcal{L}(REG) = Ps\mathcal{L}(CF)$, in the area of multiset rewriting $\mathcal{L}(CF)$ plays no role at all, and in the area of membrane computing we often only get characterizations of $Ps\mathcal{L}(REG)$ and $Ps\mathcal{L}(RE)$.

For further notions and results in formal language theory we refer to textbooks like [12] and [25].

In the rest of this section, we briefly recall P systems and the related concepts. For more extensive overviews, we refer the reader to [18,24].

A *(transition) P system* is a construct

$$\Pi = (O, T, \mu, w_1, \ldots, w_n, R_1, \ldots R_n, \delta, h_i, h_o) \text{ where}$$

- O is the alphabet of *objects*,
- $T \subseteq O$ is the alphabet of *terminal objects*,
- μ is the *membrane structure* injectively labelled by the numbers from $[1..n]$ and usually given by a sequence of correctly nested brackets,
- w_i are the multisets giving the *initial contents* of each membrane i, $1 \leq i \leq n$,
- R_i is the finite *set of rules* associated with membrane i, $1 \leq i \leq n$,
- δ is the *derivation mode*, and
- h_i and h_o are the labels of the *input membrane* and the *output membrane*, respectively; $1 \leq h_i \leq n$, $1 \leq h_o \leq n$.

Taking into account the well-known flattening process, which means that computations in a P system with an arbitrary (static) membrane structure can be simulated in a P system with only one membrane, e.g., see [14], often only *simple P systems* are considered, i.e., with the simplest membrane structure of only one membrane region, and then we write:

$$\Pi = (O, T, w_1, R_1, \delta)$$

Quite often, the rules associated with membranes are multiset rewriting rules (or special cases of such rules). Multiset rewriting rules have the form $u \to v$, with $u \in O^\circ \setminus \{\lambda\}$ and $v \in O^\circ$, where O° is the set of multisets over O, and $\lambda(a) = 0$,

for all $a \in O$. If $|u| = 1$, the rule $u \to v$ is called *non-cooperative*, otherwise it is called *cooperative*. In *communication P systems*, rules are additionally allowed to send symbols to the neighbouring membranes. In this case, for rules in R_i, $v \in (O \times Tar_i)^\circ$, where Tar_i contains the symbols out (corresponding to sending the symbol to the parent membrane), $here$ (indicating that the symbol should be kept in membrane i), and in_j (indicating that the symbol should be sent into the child membrane j of membrane i). When writing out the multisets over $O \times Tar_i$, the indication $here$ is often omitted.

In P systems, rules are often applied in the maximally parallel way: in one derivation step, only a non-extendable multiset of rules can be applied. The rules are not allowed to consume the same instance of a symbol twice, which creates competition for objects and may lead to non-deterministic choices between the maximal collections of rules applicable in one step. The *maximally parallel derivation mode* is generally denoted by the symbol max. Other derivation modes include the *sequential derivation mode* $sequ$ in which exactly one rule is applied in every step, the *set maximally parallel derivation mode* $smax$ only allowing multisets of rules in which every rule has multiplicity 1, as well as the *asynchronous derivation mode* $asyn$ under which no restriction is imposed on the applied multiset of rules. We refer to the works [3,4,6,16] for an in-depth discussion of the matter.

A *computation* of a P system is traditionally considered to be a sequence of configurations it can successively visit, stopping at the halting configuration. A *halting* configuration is a configuration in which no rule can be applied any more, in any membrane. The *result of a computation* in a P system Π as defined above is the contents of the output membrane h_o projected over the terminal alphabet T.

We will use the notations $N(\Pi)$ and $Ps(\Pi)$ to respectively refer to the number language and the language of multisets generated by Π. The notation $OP_n(\delta, \tau)$ will refer to the family of P systems with at most n membranes, operating under the derivation mode δ and relying on the rules of type τ, where $\tau = coo$ if cooperative rules are allowed and $\tau = ncoo$ if all rules are non-cooperative. Finally, we use the notations $NOP_n(\delta, \tau)$ and $PsOP_n(\delta, \tau)$ to refer to the family of number languages and multiset languages, respectively, generated by the P systems in the family $OP_n(\delta, \tau)$.

Example 1. Figure 1 shows the graphical representation of the P system formally given by

$$\begin{aligned}\Pi &= (\{a, b, c, d\}, \{a, d\}, [_1[_2]_2]_1, d, ab, R_1, R_2, max, 1, 1),\\ R_2 &= \{a \to aa, b \to b\,(c, out)\},\\ R_1 &= \emptyset.\end{aligned}$$

In the maximally parallel mode, the inner membrane 2 of Π will apply as many instances of the rules as possible, thereby doubling the number of a, and ejecting a copy of c into the surrounding (skin) membrane in each step. The symbol d in the skin membrane is not used. Therefore, after k steps of evolution, membrane 2 will contain the multiset $a^{2^k} b$ and membrane 1 the multiset $c^k d$. Since all rules are always applicable in Π, this P system never halts. □

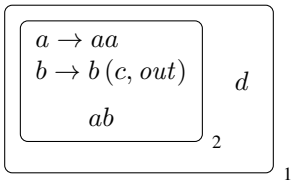


Fig. 1: An example of a simple P system.

3 Reactive Membranes

A *P system with reactive membranes* is the following construct:

$$\Pi = (O, T, W_0, R, \delta) \text{ where}$$

- O is the alphabet of *objects*,
- $T \subseteq O$ is the alphabet of *terminal objects*,
- $W_0 \in \mathcal{P}_{fin}(O^\circ)$ is the (finite) *initial set of multisets* over O,
- $R \subseteq O^\circ \times O^\circ$ is the set of *evolution rules*, and
- δ is the *derivation mode*.

For all rules in R, we will require at least one of the sides to be non-empty, i.e.,

$$\forall u \to v \in R : u \neq \lambda \vee v \neq \lambda.$$

We immediately stress two major features of this definition. On the one hand, we do not include any membrane structure. Indeed, as W_0 hints, we simply use individual multisets to represent the contents of the individual membranes, without explicitly representing the membranes themselves. Incidentally, this means that membranes do not nest in this model. On the other hand, the evolution of all symbols in (the multisets in) all the membranes is governed by the same common set of rules R.

A configuration of Π is any set of multisets over O. Similarly to networked models of computing like networks of evolutionary processors (e.g. [19]) or tissue P systems with vesicles of multisets [5], a computation step in P systems with reactive membranes consists of two stages:

1. splitting and merging,
2. evolution.

Informally, the splitting and merging stage implements the non-deterministic evolution of the membranes—individual multisets under this definition—as described in the introduction: any two multisets may merge, and any multiset may split in two. The evolution stage consists in applying the evolution rules in R to every multiset of the configuration, according to the derivation mode δ. In the following paragraphs we give a formal description of both stages, applied to a configuration $W_i \in \mathcal{P}_{fin}(O^\circ)$.

Splitting and merging stage

1. Non-deterministically partition W_i into 3 subsets:

   $$W_i = M_i \cup S_i \cup I_i$$

   such that $|M_i|$ is even, and the sets S_i, M_i, and I_i are mutually disjoint, i.e., $S_i \cap M_i = S_i \cap I_i = M_i \cap I_i = \emptyset$. The multisets in M_i will be merged pairwise, the multisets in S_i will be split, and the multisets in I_i will remain intact.
2. Partition M_i into a set of disjoint pairs. Non-deterministically pick a bijection $\varphi : [1..|M_i|] \to M_i$ and construct the following set:

   $$\hat{M}_i = \{(\varphi(2k-1), \varphi(2k)) \mid 1 \leq k \leq |M_i|/2\}.$$

   Then define $M'_i = \{w_1 \cup w_2 \mid (w_1, w_2) \in \hat{M}_i\}$.
3. Define $\mathrm{split}(w)$ to be the set of all possible ways to split the multiset w into two multisets:

   $$\mathrm{split}(w) = \{(w_1, w_2) \mid w_1 \cup w_2 = w, w_1, w_2 \in O^\circ\}.$$

   Define the set of all possible ways of splitting the multisets in S_i:

   $$\hat{S}_i = \prod_{w \in S_i} \mathrm{split}(w).$$

   Non-deterministically pick $S'_i \in \hat{S}_i$.
4. Compute the new intermediate configuration as

   $$W'_i = M'_i \cup \mathrm{flatten}(S'_i) \cup I_i,$$

   where $\mathrm{flatten}(S'_i) = \{w_1, w_2 \mid (w_1, w_2) \in S'_i\}$.

In the above presentation we describe merging before splitting, but the order of the two substeps does not matter, since they occur on disjoint sets M_i and S_i. Furthermore, we stress that multiple intermediate configurations W'_i may be obtained from the same configuration W_i.

Evolution stage The evolution stage is defined in the conventional way by applying the rules in R to every multiset in W'_i individually, according to the derivation mode δ:

$$W_{i+1} = \{w \mid w' \overset{\delta,R}{\Longrightarrow} w, w' \in W'_i\},$$

where w is a multiset derived from w' by applying the rules in R under the mode δ.

A configuration W is *halting* if no rules are applicable in the evolution stage, for any intermediate configuration W' which can be obtained from W in the splitting and merging stage. An *n-step halting computation* of a P system with reactive membranes Π is a finite sequence of configurations $(W_i)_{0 \leq i \leq n}$ such that W_{i+1} is obtained from W_i by the computation step described above, and W_n is a halting configuration.

As the *result of a computation* in a P system with reactive membranes Π as defined above we take all the terminal objects appearing in the membranes present in a halting configuration W_n:

$$\left(\bigcup_{w \in W_n} w \right) \Big|_T = \bigcup_{w \in W_n} w|_T.$$

To conclude the introduction of P systems with reactive membranes, we again stress that the splitting and merging of multisets (or membranes) is non-deterministic, imposed in every computation step, and independent of the features of the configuration or of the rules in R. More concretely, the rules in R cannot directly influence which symbols will appear next to which after the splitting and merging stage.

Example 2. Consider the following P system with reactive membranes:

$$\begin{aligned} \Pi &= (O, T, W_0, R, max), \text{ where} \\ O &= \{a, b, c, d, e, f\}, \\ T &= \{d, f\}, \\ W_0 &= \{a, b, c\}, \\ R &= \{ab \to d, abc \to f, a \to e\}. \end{aligned}$$

For the first step of the computation, Π may decide to not split or merge any multisets ($M_0 = S_0 = \emptyset$, $I_0 = W_0$), meaning that the evolution rules will be applied directly to singleton multisets a, b, and c. While no rules are applicable to b or c individually, the rule $a \to e$ will have to be applied to a, yielding the next configuration $W_1 = \{b, c, e\}$. We can immediately conclude that the rules $ab \to d$ and $abc \to f$ will never be applicable any more later in this computation, as there is no way to reintroduce a. In sum, this halting computation yields the result Λ.

Now suppose that Π decides to merge the multisets a and b in the first step, yielding the intermediate configuration $W_0' = \{ab, c\}$. In this case a non-deterministic choice will appear in the evolution stage between applying the rule $ab \to d$ or $a \to e$ (both singleton sets of rules are non-extendable). As a consequence, the following two possibilities exist for the second configuration: $\{d, c\}$ and $\{eb, c\}$. In sum, these halting computations yield the results d and Λ, respectively.

Finally, note that the rule $abc \to f$ will never be applicable with $W_0 = \{a, b, c\}$, since putting a, b, and c together in one membrane requires at least two mergers, and a will necessarily be consumed by $a \to e$ or $ab \to d$ along the way. On the other hand, if we put together a, b and c in one multiset from the start, or even if we put ab together and c apart, the rule $abc \to f$ will have a chance to be applied. In particular, in the case in which the initial configuration is $\{ab, c\}$ it suffices to consider the branch of the computation along which Π decides to merge the two multisets in the first splitting and merging stage. In sum, with the initial sets $\{ab, c\}$ and $\{abc\}$ we can get the results Λ, d, and f. □

As indicated by the example discussed above, it makes a difference in how many multisets the initial multiset of objects is divided. Thus, we will use the notation $Re_nOP(\delta, \tau)$ to refer to the family of P systems with reactive membranes starting with n initial multisets, running under the mode δ and using rules of type $\tau \in \{coo, ncoo\}$,

as well as the notations $NRe_nOP(\delta,\tau)$ and $PsRe_nOP(\delta,\tau)$ to refer to the family of number languages and multiset languages, respectively, generated by the P systems with reactive membranes from $Re_nOP(\delta,\tau)$.

Whereas on the one hand the previous example shows the effect of having more than one initial membrane, prohibiting the application of some evolution rules, the next example shows that the halting condition can be fulfilled due to the fact that symbols are distributed over several membranes, although some rule could be applied if all symbols on its left-hand side could be put into the same membrane by a merge operation. As merging can only combine the contents of two membranes, we can already get the situation that a rule with three symbols in its left-hand side cannot be applied any more.

Example 3. Consider the following P system with reactive membranes:

$$\begin{aligned}\Pi &= (O,T,W_0,R,max), \text{ where}\\ O &= \{a_i,a_i',a_i'' \mid 1\le i\le 3\}\cup\{f\},\\ T &= \{a_1'',a_2'',a_3'',f\},\\ W_0 &= \{a_1a_2a_3\},\\ R &= \{a_i\to a_i', a_i'\to a_i'', a_1''a_2''a_3''\to f\}.\end{aligned}$$

If in the first two steps of the computation, Π decides to not split or merge any multisets, from $W_0=\{a_1a_2a_3\}$ with applying the rules $\{a_i\to a_i' \mid 1\le i\le 3\}$, after the first evolution step we obtain $W_1=\{a_1'a_2'a_3'\}$, and by then applying the rules $\{a_i'\to a_i'' \mid 1\le i\le 3\}$, after the second evolution step we obtain $W_2=\{a_1''a_2''a_3''\}$. Keeping $\{a_1''a_2''a_3''\}$ in the same membrane then allows for applying the rule $a_1''a_2''a_3''\to f$, thus obtaining the terminal result $W_3=\{f\}$, as W_3 is a halting configuration.

Yet with two splits, but still applying the rules $\{a_i\to a_i' \mid 1\le i\le 3\}$ in the first evolution step and the rules $\{a_i'\to a_i'' \mid 1\le i\le 3\}$ in the second evolution step, we get a two-step halting computation

$$\{a_1a_2a_3\}\Longrightarrow\{a_1',a_2'a_3'\}\Longrightarrow\{a_1'',a_2'',a_3''\}$$

yielding the terminal result $a_1''a_2''a_3''$.

We also mention that with having $T=\{f\}$ only, this halting computation yields the result Λ. □

4 Computational Power: First Results

In this section, we list some first results regarding the computational power of P systems with reactive membranes. We start by remarking that the halting condition can be checked in an easier way when the system only includes non-cooperative rules.

Remark 1. When using only non-cooperative rules, the halting condition for a configuration W can be checked without considering all possible splits and mergers and then the non-applicability of the rules in all membranes; instead it suffices to check the non-applicability of the rules to the objects in $\mathrm{flatten}(W)$, i.e., the union of the multisets in all the membranes of W.

The following result even holds for non-cooperative rules and cooperative rules.

Lemma 1. *For every $Re_1OP(\delta, \tau)$ system there exists an equivalent $Re_nOP(\delta, \tau)$ system, for every $n > 1$.*

Proof. Given a P system with reactive membranes using rules of type τ

$$\Pi' = (O, T, \{w\}, R, \delta),$$

an equivalent P system with reactive membranes using rules of type τ with n initial membranes is

$$\Pi' = (O, T, \{w, w_2 = \Lambda, \ldots, w_n = \Lambda\}, R, \delta).$$

In an empty membrane Λ, no non-cooperative rules or cooperative rules are applicable. Moreover, merging a membrane X with Λ yields X again, so no additional applications of rules can happen. □

Now we show that splitting and merging do not affect the (results of the) computations in a P system with reactive membranes at all, no matter which derivation mode is used, when only non-cooperative rules are used. Hence, we get a characterization of $Ps\mathcal{L}(REG)$:

Theorem 1. *For any $\delta_1, \delta_2 \in \{asyn, seq, max, smax\}$ and $Y \in \{N, Ps\}$ as well as any $n \geq 1$,*

$$YRe_nOP(\delta_1, ncoo) = YOP_1(\delta_2, ncoo) = Y\mathcal{L}(REG).$$

Proof. The equality $YOP_1(\delta_2, ncoo) = Y\mathcal{L}(REG)$ is folklore, e.g., see [24]. The main idea for proving this result is that the evolution of symbols by applying non-cooperative rules can be described by a derivation tree, but for the resulting terminal objects it is completely irrelevant when the symbols evolve.

A similar argument now can be used here to argue that the following holds for any $\delta_1 \in \{asyn, seq, max, smax\}$:

$$YRe_nOP(\delta_1, ncoo) = YOP_1(asyn, ncoo) = Y\mathcal{L}(REG).$$

(⇒) Given a P system with reactive membranes

$$\Pi' = (O, T, \{w\}, R, \delta_1),$$

we can easily define the equivalent simple P system

$$\Pi = (O, T, w, R, asyn).$$

Even distributing the contents of a single membrane over several membranes, even at the beginning with having several initial multisets, does not affect the applicability of the non-cooperative rules. Yet we have to mention that using the sequential derivation mode in several membranes yields a kind of parallelism like $smax$, but also this has no effect on the results of computations, especially as, according to Remark 1, halting only depends on the non-applicability of all rules to the symbols in all the multisets of the underlying configuration.

($\Leftarrow$) Given a simple P system

$$\Pi = (O, T, w, R, asyn),$$

we can easily define the equivalent P system with reactive membranes

$$\Pi' = (O, T, \{w\}, R, asyn).$$

Any derivation of the 1-membrane transition P system Π operating under the asynchronous derivation mode can be directly simulated by the P system with reactive membranes Π' which uses the same rules for the evolution stage, but then always chooses to not split or merge any membranes, i.e. M_i and S_i from the splitting and merging stage are always empty. As we are only using non-cooperative rules, the applicability of all the (multisets of) rules applied in Π is also guaranteed in Π'. Finally we can apply Lemma 1 to get an equivalent P system with reactive membranes with n initial membranes.

In sum we see that P systems with reactive membranes behave as the corresponding transition P system when only non-cooperative rules are used. □

Finally, we show that P systems with reactive membranes working under the maximally parallel mode and using cooperative rules can simulate partially blind register machines. As a reminder, we mention that partially blind register machines (PBRM) have programs consisting of the following two types of instructions for incrementing and decrementing a register:

- $(p, \mathrm{ADD}(r), q, s)$: in state p increment register r and jump to state q or state s;
- $(p, \mathrm{SUB}(r), q)$: in state p try to decrement register r; if successful, jump to state q, otherwise abort the computation without producing a result.

Partially blind register machines feature a final zero check: the register machine only halts with producing a result if all non-output registers are empty when the machine reaches the halting instruction uniquely labeled by h.

We will refer to the set of multiset languages generated by partially blind register machines by $PsPBRM$.

Theorem 2. *For any $\delta \in \{asyn, sequ, max, smax\}$,*

$$PsPBRM \subseteq PsRe_1OP(\delta, coo).$$

Sketch. The main idea of the proof is that throughout the simulation of the partially blind register machine, the configurations of the P system with reactive membranes Π always contain exactly one instance of the symbol representing the label of the instruction to be carried out next. The contents of a register r is represented by the total number of symbols a_r in the configurations of Π.

The increment instruction $(p, \mathrm{ADD}(r), q, s)$ can be simulated directly by the rules $p \to qa_r$ and $p \to sa_r$.

The decrement instruction $(p, \mathrm{SUB}(r), q)$ can be simulated by the following two rules: $pa_r \to q$, $p \to p$. Moreover, for every register symbol a_r with r not being an output register, we add the unit rules $a_r \to a_r$.

Indeed, if p and a copy of a_r find themselves in the same membrane, then a successful decrement is simulated: the total number of copies of a_r in the system is reduced by one.

If there are no copies of a_r left in the system, then p only has the chance to be used with the unit rule $p \to p$; observe that in any derivation mode at least one rule has to be applied if the system is not halting, i.e., as long as there still is a rule which can be applied to some symbol. In this case, either $p \to p$ and/or some unit rule $a_r \to a_r$ can be applied in every future derivation step, hence, the computation will never halt.

If copies of a_r do appear in the system, but not in the membrane containing p, then p can use the unit rule $p \to p$, and in any derivation mode either only this rule and/or other unit rules $a_r \to a_r$ can be applied. If in some future step, p and a_r appear in the same membrane, possibly $pa_r \to q$ can be applied. Otherwise, again we obtain just non-halting computation branches.

However, there must exist another branch in which no splits and mergers have happened at all, i.e., p and a_r are together, and in which the simulation therefore will be able to proceed correctly. The same alternative holds if p and a_r share the same membrane, but the system non-deterministically would choose to only apply $p \to p$ rather than $pa_r \to p$.

As soon as the halting label h appears, we have to use the final rule $h \to \lambda$. The final zero check is simulated by the unit rules $a_r \to a_r$ for all non-output registers r, which keep the computation to go on forever if at least one such symbol a_r is still present. Observe that this argument does not depend on the distribution of the symbols in the membranes of a configuration.

In sum, we conclude that the P system with reactive membranes Π can simulate the computations of the given partially blind register machine correctly, but on the other hand cannot yield more results. □

Finally, we remark that the construction we show here is non-deterministic, even if the simulated partially blind register machine is deterministic, i.e., all increment instructions are of the form $(p, \mathrm{ADD}(r), q, q)$, which in a simpler way can be written as $(p, \mathrm{ADD}(r), q)$.

5 Extensions

Given the motivation to use P systems with reactive membranes for thinking about the emergence of space and space separations in abiotic environments, and also the richness of the ecosystem of P systems variants, multiple extensions can be proposed.

A natural one to be considered would be *limiting the size* of individual membranes, as real membranes do not generally grow very big. Limitations on the number of symbols have already been considered in P systems [2], but combined with constant splitting and merging this ingredient may have a drastically different impact. It would be necessary to decide what happens when a membrane attains its maximal capacity. The approach in [2] is to prevent it to accept new symbols, but in the context of reactive membranes it may

be appropriate to bias the splitting and merging stage of the computational step to force such full membranes to split. The contribution of such limitations to the computational power is yet unclear, but probably in some strong relation to the size of the left-hand sides of the evolution rules.

An extension in the spirit of generalized P systems [13] would be to *subject the rules to splitting and merging*. With such an extension, membranes would contain objects and rules, and splitting and merging would affect not only which symbols can interact, but also which rules will ensure their interaction.

Finally, splitting and merging could also be applied to rules: for example, a rule $u \to v$ could split into two rules $u \to \alpha$ and $\alpha \to v$, which could later merge back into $u \to v$. Similarly to splitting and merging of membranes, splitting and merging of rules delays some interactions. Relevance to thinking about the origins of life and the computational power of this variant remain to be explored.

6 Conclusion and Perspectives

This paper is a first attempt at using P systems for thinking about the origins of life, and in particular about the emergence of individual compartments separated by membranes. We introduced P systems with reactive membranes, in which every symbol is conceptually surrounded by elementary membranes, which then can merge to form bigger membranes, or split. Mimicking biochemistry, the set of rules is common to all membranes—the differences in the processes in different membranes should come from the symbols. Cooperative rules are allowed, and probably even necessary to meaningfully implement distinctions between membranes.

It is still an open research direction to actually illustrate some processes believed to have happened during abiogenesis in P systems with reactive membranes. Perhaps the most promising would be to implement autocatalytic cycles (e.g. [11]). The next step would be to implement *self-replication*, as suggested by José M. Sempere in a discussion. Indeed, in P systems with reactive membranes the membrane structure emerges spontaneously, which makes them a promising candidate for implementing self-replication of something other than symbol objects.

A parallel research direction which we started to explore in this paper is the computational power of P systems with reactive membranes. We have shown here that splitting and merging does not affect the computational power of P systems with reactive membranes using non-cooperative rules—P systems with reactive membranes using non-cooperative rules have the same computational power as simple P systems provided we only start with one singleton multiset, no matter which derivation mode we use. Based on this result, we have shown that P systems with reactive membranes can characterize the family of Parikh sets of semilinear languages when using only non-cooperative rules in any derivation mode.

Finally, when cooperative rules are allowed, P systems with reactive membranes can generate all multiset languages generated by partially blind register machines.

Several questions still remain to be addressed, in particular: can splitting and merging *augment* the computational power? It would indeed be surprising, but it has already been shown that non-deterministic shuffling of rule right-hand sides allows for generating

non-semilinear languages [1], meaning that random shuffling of symbol neighborhoods as described in this paper may boost the power of the variant in some specific cases.

A subtle aspect which we do not discuss in depth in this paper is the halting condition and the procedure for retrieving the result. There is an asymmetry between these two: halting occurs when no more evolution rules are applicable after all possible splits and mergers. On the other hand, getting out the result essentially happens by merging *all* membranes into a single one.

Since the computational results we give in this paper seem to depend directly on the halting condition and on the procedure for obtaining the result, it would be relevant to explore how slight variations in these two affect the computational power of P systems with reactive membranes.

Finally, in Section 5 we have suggested several possible extensions of the new variant. A formal exploration of the computational power of such extensions would be quite relevant. Even more importantly, it would be very relevant to identify which extensions are more useful for using P systems with reactive membranes in thinking about the origins of life.

Acknowledgements

The authors would like to thank the Organizing Committee of the 19th Brainstorming Week on Membrane Computing[7] (BWMC 2023) for organizing this fruitful event, which allowed the authors to jointly develop the ideas presented in this paper. The authors would also like to thank José M. Sempere for multiple helpful suggestions about self-replication and emergence of space as well as the anonymous referees for their useful comments.

Artiom Alhazov acknowledges project 20.80009.5007.22 “Intelligent information systems for solving ill-structured problems, processing knowledge and big data” by the National Agency for Research and Development.

References

1. Artiom Alhazov, Rudolf Freund, and Sergiu Ivanov. P systems with randomized right-hand sides of rules. *Theor. Comput. Sci.*, 805:144–160, 2020.
2. Artiom Alhazov, Rudolf Freund, and Sergiu Ivanov. P systems with limited number of objects. *J. Membr. Comput.*, 3(1):1–9, 2021.
3. Artiom Alhazov, Rudolf Freund, Sergiu Ivanov, and Marion Oswald. Variants of derivation modes for which purely catalytic P systems are computationally complete. *Theor. Comput. Sci.*, 920:95–112, 2022.
4. Artiom Alhazov, Rudolf Freund, Sergiu Ivanov, and Sergey Verlan. Variants of derivation modes for which catalytic P systems with one catalyst are computationally complete. *J. Membr. Comput.*, 3(4):233–245, 2021.
5. Artiom Alhazov, Rudolf Freund, Sergiu Ivanov, and Sergey Verlan. Tissue P systems with vesicles of multisets. *Int. J. Found. Comput. Sci.*, 33(3&4):179–202, 2022.

[7] http://www.gcn.us.es/19bwmc

6. Artiom Alhazov, Rudolf Freund, and Sergey Verlan. P systems working in maximal variants of the set derivation mode. In Alberto Leporati, Grzegorz Rozenberg, Arto Salomaa, and Claudio Zandron, editors, *Membrane Computing – 17th International Conference, CMC 2016, Milan, Italy, July 25-29, 2016, Revised Selected Papers*, volume 10105 of *Lecture Notes in Computer Science*, pages 83–102. Springer, 2016.
7. Artiom Alhazov and Tseren-Onolt Ishdorj. Membrane operations in P systems with active membranes. *Second Brainstorming Week on Membrane Computing*, pages 37–44, 2004.
8. Bogdan Aman and Gabriel Ciobanu. Simple, enhanced and mutual mobile membranes. *Trans. Comp. Sys. Biology*, 11:26–44, 2009.
9. Luca Cardelli and Gheorghe Păun. An universality result for a (mem)brane calculus based on mate/drip operations. *Int. J. Found. Comput. Sci.*, 17(1):49–68, 2006.
10. Matthew Cobb. 60 years ago, Francis Crick changed the logic of biology. *PLOS Biology*, 15(9):1–8, 09 2017.
11. Bruce Damer and David Deamer. The hot spring hypothesis for an origin of life. *Astrobiology*, 20(4):429–452, 2020.
12. Jürgen Dassow and Gheorghe Păun. *Regulated Rewriting in Formal Language Theory*. Springer, 1989.
13. Rudolf Freund. Generalized P-systems. In Gabriel Ciobanu and Gheorghe Păun, editors, *Fundamentals of Computation Theory, 12th International Symposium, FCT '99, Iasi, Romania, August 30 – September 3, 1999, Proceedings*, volume 1684 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 1999.
14. Rudolf Freund, Alberto Leporati, Giancarlo Mauri, Antonio E. Porreca, Sergey Verlan, and Claudio Zandron. Flattening in (tissue) P systems. In Artiom Alhazov, Svetlana Cojocaru, Marian Gheorghe, Yurii Rogozhin, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 8340 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2014.
15. Rudolf Freund and Marion Oswald. Tissue P systems and (mem)brane systems with mate and drip operations working on strings. In Nadia Busi and Claudio Zandron, editors, *Proceedings of the First Workshop on Membrane Computing and Biologically Inspired Process Calculi, MeCBIC@ICALP 2006, Venice, Italy, July 9, 2006*, volume 171 (2) of *Electronic Notes in Theoretical Computer Science*, pages 105–115. Elsevier, 2006.
16. Rudolf Freund and Sergey Verlan. A formal framework for static (tissue) P systems. In George Eleftherakis, Petros Kefalas, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing, 8th International Workshop, WMC 2007, Thessaloniki, Greece, June 25-28, 2007. Revised Selected and Invited Papers*, volume 4860 of *Lecture Notes in Computer Science*, pages 271–284. Springer, 2007.
17. Nicolas Glade. *Le Vivant Rare, Faible et Amorphe - Évolution depuis les Origines jusqu'à la Vie telle qu'elle nous Apparaît. (Rare, Weak and Amorphous Life – Evolution of Life from the Origins until Life as it Appears Nowadays)*. HAL Open Archive, 2022.
18. Bulletin of the International Membrane Computing Society (IMCS). http://membranecomputing.net/IMCSBulletin/index.php.
19. Sergiu Ivanov, Yurii Rogozhin, and Sergey Verlan. Small universal networks of evolutionary processors. *J. Autom. Lang. Comb.*, 19(1–4):133–144, 2014.
20. Shankara Narayanan Krishna and Gheorghe Păun. P systems with mobile membranes. *Nat. Comput.*, 4(3):255–274, 2005.
21. David Orellana-Martín, Luis Valencia-Cabrera, and Mario J. Pérez-Jiménez. P systems with evolutional communication and separation rules. In Jérôme Durand-Lose and György Vaszil, editors, *Machines, Computations, and Universality – 9th International Conference, MCU 2022, Debrecen, Hungary, August 31 – September 2, 2022, Proceedings*, volume 13419 of *Lecture Notes in Computer Science*, pages 143–157. Springer, 2022.

22. Linqiang Pan and Tseren-Onolt Ishdorj. P systems with active membranes and separation rules. *J. Univers. Comput. Sci.*, 10(5):630–649, 2004.
23. Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.
24. Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors. *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.
25. Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages*. Springer, 1997.
26. Rémi Segretain, Sergiu Ivanov, Laurent Trilling, and Nicolas Glade. A methodology for evaluating the extensibility of Boolean networks' structure and function. In Rosa M. Benito, Chantal Cherifi, Hocine Cherifi, Esteban Moro, Luis Mateus Rocha, and Marta Sales-Pardo, editors, *Complex Networks & Their Applications IX - Volume 2, Proceedings of the Ninth International Conference on Complex Networks and Their Applications, COMPLEX NETWORKS 2020, 1–3 December 2020, Madrid, Spain*, volume 944 of *Studies in Computational Intelligence*, pages 372–385. Springer, 2020.
27. Rémi Segretain, Laurent Trilling, Nicolas Glade, and Sergiu Ivanov. Who plays complex music? On the correlations between structural and behavioral complexity measures in sign Boolean networks. In *21st IEEE International Conference on Bioinformatics and Bioengineering, BIBE 2021, Kragujevac, Serbia, October 25–27, 2021*, pages 1–6. IEEE, 2021.

Queens of the Hill

Artiom Alhazov[1], Sergiu Ivanov[2], David Orellana-Martín[3,4]

[1]Vladimir Andrunachievici Institute of Mathematics and Computer Science,
The State University of Moldova, Academiei 5, Chișinău, MD-2028, Moldova
artiom@math.md

[2]IBISC Laboratory, Université Paris-Saclay, Univ Évry
91020, Évry-Courcouronnes, France
E-mail : sergiu.ivanov@univ-evry.fr

[3]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@us.es

[4]SCORE Laboratory, I3US, Universidad de Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain

Abstract. Inspired by the programming game Core Wars, we propose in this work a framework and the organisation of king of the hill-style tournaments between P systems. We call these tournaments Queens of the Hill and the individual contestants valkyries. The goal of each valkyrie is to dissolve as many membranes of as many other valkyries as possible, while at the same time resisting the attacks. Valkyries are transition P systems with cooperative rules, target indication, and rudimentary matter–anti-matter annihilation rules. These ingredients are sufficient for computational completeness, but the context of Queens of the Hill reduces the relevance of this statement. We give some tentative examples of strategies and discuss their advantages and drawbacks. Finally, we describe how Queens of the Hill can be used as a teaching exercise, and also a tool to federate the students' creativity to push the frontiers of membrane computing.

Keywords: Core Wars, membrane dissolution, anti-matter, interaction.

1 Core Wars

To cite [12], *"Core War (or Core Wars) is a programming game where assembly programs try to destroy each other in the memory of a simulated computer."* In Core Wars, programmers design programs—called warriors—with two goals in mind:

1. kill as many other programs as possible,
2. survive for as long as possible against the attacks of the other programs.

In the most basic setup, all programs are loaded in the same shared memory space, and only feature instruction segments, i.e. their memory only contains code, and data

is stored as part of some of the instructions. No memory protection is available for the instructions, so all programs can write anywhere, including to the instruction segments of competitors, which is the primary way of attacking. The simplest warrior is called the Imp and only consists of a single instruction in the special assembly language called Redcode:

```
MOV 0, 1
```

The numbers correspond to addresses in the memory space relative to the current instruction, so `0` refers to the current instruction slot, and `1` refers to the next one. This program copies its only instruction to the next memory slot, which then copies itself to the next one, etc. The Imp therefore ends up populating the whole memory with copies of itself.

As small and impressive as it is, the Imp will never actually win, because it just reproduces itself, possibly over the code of the competitors, but it never kills any competitor. To kill a process, Redcode features the special instruction `DAT`. When it is executed, the current process is killed. A simple winning code would throw `DAT` over the whole memory, while simultaneously avoiding to run this instruction in its own execution. This is what the warrior called the Dwarf does, whose detailed presentation is given in [12].

Multiple servers exist continuously running Code Wars tournaments in the king of the hill mode (see section "Climbing the hill" in [12]): 10 to 30 warriors are loaded in the same shared memory space and are run sequentially, on a single virtual processor, which interleaves the execution of the instructions of every warrior. The score of a warrior in a match roughly corresponds to the number of other warriors it has killed. The warrior with the highest score is the current king of the hill, and the warrior with the lowest score falls off the hill: it is replaced by a new warrior.

2 Queens of the Hill

In this submission we propose a framework for running king of the hill style tournaments between P systems. We refer to such tournaments as (P) *Queens of the Hill*, and we call individual contestants *valkyries*. In this section, we propose the formal framework to be used for the valkyries as well as the rules for Queens of the Hill tournaments.

2.1 Valkyries

Our choice of the P system variant for the valkyries is guided by the following two principles: ability to interact with the other contestants and ease of programming. We choose here a variation on what is sometimes called transition P systems, which is partially inspired by P automata with matter–anti-matter annihilation rules as shown in [6] and by P colonies [3]. As a reminder, the original P automata rely on antiport rules exclusively [4].

We define a (valkyrie) P system as the following tuple:

$$\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n), \text{ where}$$

- $O = \Sigma \cup \Delta_k$ is a finite alphabet of objects,
- $\Delta_k = \{\delta_t, \bar{\delta}_t \mid 1 \leq t \leq k\} \cup \{\delta\}$ for some fixed $k \in \mathbb{N}$,
- μ is the hierarchical membrane structure bijectively labeled by the numbers from 1 to n and usually presented as a sequence of correctly nested brackets,
- w_i is the initial multiset in membrane i, $1 \leq i \leq n$,
- R_i is the finite set of rules in membrane i, $1 \leq i \leq n$.

The rules in R_i feature full cooperation and may use target indications. More precisely, a rule in R_i has the form $u \to v$, where $u \in \Sigma^\circ$, $u \neq \lambda$, is a non-empty multiset over Σ, and $v \in (O \times Tar)^\circ$ is a multiset of symbols over O, each equipped with target indications $Tar = \{in, here, out\}$. A symbol appearing with the indication in in v will be sent into a non-deterministically chosen inner membrane of membrane i, a symbol with the indication $here$ will remain in membrane i, and a symbol with the indication out will be sent to the parent membrane. If membrane i does not have any inner membranes, the symbols with target indication in will be kept in membrane i, i.e. the target indications in and $here$ are equivalent in the case of elementary membranes. For readability, we will always omit the indication $here$, i.e. instead of writing $(a, here)(a, here)(b, out)$ we will write $aa(b, out)$.

The symbol $\delta \in \Delta_k$ has the special semantics of dissolving the membrane in which it appears. More formally, once δ is introduced into membrane i, all of its objects and inner membranes are moved to its parent membrane, and membrane i is removed from the system—non-elementary membrane dissolution is allowed. Membrane dissolution happens at the end of a computation step, and all introduced copies of δ are removed from the system after all dissolutions are performed. It follows incidentally that introducing any number of copies of δ in a membrane produces exactly the same as effect as introducing one copy of δ. Dissolution of the outermost (skin) membrane is forbidden, i.e. introducing a copy of δ into the skin membrane will have no effect and the symbol δ will be immediately removed.

All sets R_i, $1 \leq i \leq n$, also include the following rules:

$$R_k^\delta = \{\delta_t \to \delta_{t-1} \mid 2 \leq t \leq k\} \cup \{\delta_1 \to \delta\} \\ \cup \{\delta_t \bar{\delta}_t \to \lambda \mid 1 \leq t \leq k\}.$$

Informally, the symbol δ_t is equipped with a timer which triggers the dissolution of the containing membrane after t steps. The rules $\delta_t \bar{\delta}_t \to \lambda$ have weak priority, meaning that if both a copy of δ_t and $\bar{\delta}_t$ are present, then they must be erased (annihilate), preempting the evolution rule for δ_t[1].

Since the left-hand sides of the rules in $R_i \setminus R_k^\delta$ are multisets over Σ, these rules cannot directly detect or rewrite the symbols in Δ_k. However, they can produce the

[1]We recall that in P systems weak priority of annihilation rules means that if both δ_t and $\bar{\delta}_t$ are present in a configuration, then they *must* interact according to the annihilation rule $\delta_t \bar{\delta}_t \to \lambda$, even if other different rules involving δ_t or $\bar{\delta}_t$ are present. The opposite of weak priority is strong priority—if rule r_1 has strong priority over rule r_2, then in the situations in which r_1 may be applied, r_2 must *not* be applied, even if its left-hand side is a subset of the current configuration and has no intersections with the left-hand side of r_1. See [17] for further details on priorities and [1] for an introduction on matter–anti-matter annihilation rules in P systems.

anti-symbol $\bar{\delta}_t$ to force the annihilation of a symbol δ_t if it is present in the current membrane.

The rules are applied in the maximally parallel way, with weak priority of the annihilation rules $\delta_t\bar{\delta}_t \to \lambda$. A computation steps proceeds in the classical fashion, by first non-deterministically choosing a non-extendable multiset of rules to apply, applying it, and performing all the necessary dissolutions. A halting configuration is a configuration in which no more rules are applicable. We can consider halting computations of P systems, but due to the continual nature of the tournament, we will generally consider infinite or time-limited computations instead.

Example 1. Consider the following valkyrie P system:

$$\begin{aligned}
\Pi &= (O, [_1[_2[_3]_3]_2]_1, d, b\bar{\delta}_1, a, R_1, R_2, R_3),\\
O &= \{a, b, c, d\} \cup \Delta_2,\\
R_1 &= \{d \to d, d \to d(\delta_2, in)\} \cup R_2^\delta,\\
R_2 &= \{bc \to b\} \cup R_2^\delta,\\
R_3 &= \{a \to aa(c, out), a \to \delta\} \cup R_2^\delta.
\end{aligned}$$

As a reminder, $\Delta_2 = \{\delta_2, \delta_1, \delta\} \cup \{\bar{\delta}_1, \bar{\delta}_2\}$ and $R_2^\delta = \{\delta_2 \to \delta_1, \delta_1 \to \delta\} \cup \{\delta_2\bar{\delta}_2 \to \lambda, \delta_1\bar{\delta}_1 \to \lambda\}$. Figure 1 gives a graphical illustration of the P system above. For conciseness, we omit the rules in R_2^δ from such graphical illustrations.

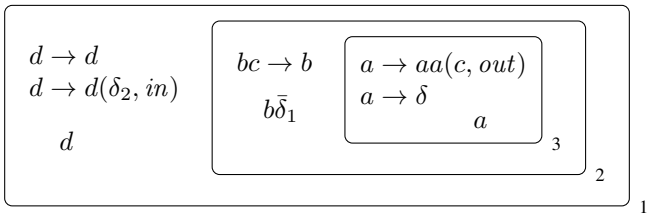


Fig. 1: A simple valkyrie P system. The rules from R_2^δ are not represented.

The rule $a \to aa(c, out)$ in membrane 3 doubles some of the a, and also ejects the corresponding number of c in membrane 2. The remaining copies of a are used to produce δ, which will dissolve membrane 3, copying all instances of a it contains into the parent membrane 2. The symbol b in membrane 2 will progressively erase all the copies of c ejected by membrane 1.

The symbol d in the skin may choose between simply maintaining itself, or also injecting δ_2 into membrane 2. The first copy of δ_2 injected into 2 will undergo the evolution rule $\delta_2 \to \delta_1$, and will afterwards annihilate with $\bar{\delta}_1$ already present there from the start. However, the second copy of δ_2 the rule $d \to d(\delta_2, in)$ will inject into membrane 2 will be free to produce δ in two steps thereby dissolving membrane 2. If by this time the rule $a \to \delta$ has not yet been applied in membrane 3, membrane 3 will become the direct inner membrane of membrane 1, so the next application of the rule $d \to d(\delta_2, in)$ will send δ_2 in membrane 3, leading to its dissolution in two steps.

Therefore, this valkyrie P system always converges to a cycle of configurations in which there is only the skin membrane containing a copy of d, a copy of b, possibly some copies of c, some copies of a, as well as a symbol from $\{\delta_2, \delta_1\}$, which always ticks down to δ without any effect, since the dissolution of the skin membrane is disallowed. □

2.2 Tournament Setup

The setup of Queens of the Hill tournaments is partially inspired by P colonies [3]: a set of valkyrie P systems is grouped together in a big skin membrane, which always sends back in whatever is sent out. More formally, we define an m-Queens of the Hill tournament as the following tuple:

$$\mathcal{Q} = (O, \Pi_1, \ldots, \Pi_m),$$

where $O = \Sigma \cup \Delta_k$ and Π_j is a valkyrie P system as defined in Section 2.1. All P systems Π_j share the same sets of symbols Σ and O. The tournament $\mathcal{Q}$ is a P system obtained by placing all Π_j into a common outer membrane 0 with the empty initial multiset and with the following set of rules:

$$R_0 = \{a \to (a, in) \mid a \in O\} \cup R_k^\delta.$$

In other words, R_0 always sends in whatever symbols are sent out from the individual valkyries, but due to non-determinism these symbols do not necessarily end up in the valkyrie which produced them. Note that R_0 contains 2 rules for symbols δ_t: such a symbol may be sent in, or it may evolve into δ_{t-1}. As before, if the corresponding anti-symbol $\bar{\delta}_t$ is also present, the annihilation rule $\delta_t \bar{\delta}_t \to \lambda$ will have to be applied. Finally note that R_0 is the only set of rules in which the left-hand sides are allowed to include δ_t.

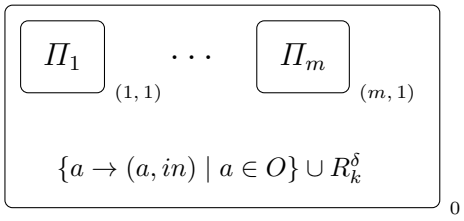


Fig. 2: An informal picture of an m-Queens of the Hill tournament $\mathcal{Q}$. The skin membranes of the valkyries Π_j are relabelled as $(j, 1)$.

A Queens of the Hill tournament obeys the same semantics as valkyrie P systems defined in Section 2.1. In particular, this means that the dissolution of the skin membranes of a valkyrie Π_j *is allowed*, because at this time it is surrounded by the bigger skin membrane of the whole tournament $\mathcal{Q}$. To preserve consistent membrane labelling, a membrane i in the valkyrie P system Π_j is renamed into membrane (j, i) in the tournament $\mathcal{Q}$.

2.3 Tournament Organization

An m-Queens of the Hill tournament runs all the valkyries in the maximally parallel mode multiple times and for a limited number of steps. At the end, the score of each valkyrie is computed from the number of its membranes that was dissolved. Non-determinism in the computations is resolved probabilistically, as it is done in the P-Lingua framework [5,7]: at every non-deterministic branching point, one of the branches is chosen under the uniform probability distribution.

More concretely, the tournament runs in the following way:

1. Run the computation for N steps, resolving non-determinism according to the uniform probability distribution.
2. Repeat Step 1 M times.

The score of a valkyrie is computed according to the following formula:

$$\mathrm{score}(\Pi_j) = \frac{1}{|\Pi_j|}\left(|\Pi_j| - \frac{1}{M}\sum_{i=1}^{M}\mathrm{diss}_i(\Pi_j)\right),$$

where $\mathrm{diss}_i(\Pi_j)$ is the number of membranes of Π_j that were dissolved during the i-th computation (i-th run of Step 1 above), and $|\Pi_j|$ is the total number of membranes in Π_j.

Example 2. Suppose that Π_j has 5 membranes, $|\Pi_j| = 5$, and take $M = 3$. Further suppose that 2, 3, and 4 membranes of Π_j were dissolved respectively in the first, second, and third computations, i.e. $\mathrm{diss}_1(\Pi_j) = 2$, $\mathrm{diss}_2(\Pi_j) = 3$, and $\mathrm{diss}_3(\Pi_j) = 4$. Then the score of Π_j in this tournament will be:

$$\frac{1}{5}\left(5 - \frac{2+3+4}{3}\right) = \frac{2}{5}.$$

Informally, the score of a valkyrie is how many membranes on average it retains by the end of a computation of the tournament, normalized by its total number of membranes. □

A valkyrie has the highest score of 1 if none of its membranes is ever dissolved in the tournament. It has the lowest score of 0 if all its membranes are always dissolved.

2.4 Tournament Parameters

Table 1 summarizes the parameters governing a Queens of the Hill tournament that were introduced in the previous sections. The values of these parameters may have a significant impact on the strategies adopted by the individual valkyries. Smaller values of $|\Sigma|$ reduce the richness of the behaviors of a valkyrie and make it less robust to perturbations coming from the skin membrane 0, i.e. from the other valkyries. Larger values of k mean more opportunities for the symbols δ_t to be captured. Larger values of m mean lower probability of receiving a symbol δ_t after emitting it into the skin membrane 0. Shorter computation lengths N mean that lightning attacks may be more feasible, while smaller values for M mean fewer computations in a tournament, which increases the contribution of randomness to the outcome.

$\lvert\Sigma\rvert$	10	The number of working symbols.
k	5	The maximal value of the index t in δ_t.
m	10–20	The number of entrants in the tournament.
N	1000	The length of a computation in the tournament.
M	50	The total number of computations in the tournament.

Table 1: A summary of the parameters governing a Queens of the Hill tournament, together with the possible values for these parameters. We suggest these values based on previous experience from running similar tournaments with multi-agent systems. These values should be taken as a starting point for further exploration.

3 A Note on Computational Complexity

Valkyrie P systems as defined in Section 2 are quite obviously computationally complete, even with a subset of the ingredients. In particular, full cooperation together with the maximally parallel mode suffice to simulate arbitrary register machines. We refer the reader to the first publication in membrane computing [16] for the very first proofs, as well as to the more recent [11,17] for a sample of the wide variety of techniques for proving computational completeness of P system variants. For the record and for the sake of the discussion of the possible strategies in Queens of the Hill tournament, we briefly recall a proof of computational completeness of P systems as defined above.

A (deterministic) register machine is an abstract computational device consisting essentially of a finite set of registers and a program. The registers can contain natural numbers or zero. The program consists of the following two types of instructions:

- $(p, \mathrm{ADD}(r), q)$: in state p, increment register r and go to state q;
- $(p, \mathrm{SUB}(r), q, s)$: in state p, check the value of register r; if its value is strictly positive, decrement it and go to state q; otherwise go to state s.

Register machines are famously computationally complete. We refer the reader to [14] for a much more in-depth discussion.

One-membrane valkyrie P systems can simulate both types of register machine instructions, even without dissolution or anti-matter rules. Classically, the alphabet Σ will include one symbol per state p of the register machine, and the value of register r will be represented by the multiplicity of symbol a_r. The instruction $(p, \mathrm{ADD}(r), q)$ can be directly simulated by the rule $p \to q a_r$. The simulation of $(p, \mathrm{SUB}(r), q, s)$ is more intricate, as usual, and relies on non-determinism and maximal parallelism: the symbol p non-deterministically guesses whether the register is empty, and a trap symbol is produced if the guess is wrong. The following table lists the rules for both branches, arranged by steps:

	Decrement	*Zero test*
1.	$p \to \bar{p}_1 \hat{p}_1$	$p \to \tilde{p}_1 \dot{p}_1$
2.	$\bar{p}_1 a_r \to \bar{p}_2,\ \hat{p}_1 \to \hat{p}_2$	$\dot{p}_1 a_r \to \#,\ \tilde{p}_1 \to \tilde{p}_2$
3.	$\hat{p}_2 \bar{p}_2 \to q,\ \hat{p}_2 \bar{p}_1 \to \#$	$\tilde{p}_2 \dot{p}_1 \to s$

The decrement branch begins by splitting the state symbol p into $\bar{p}_1$ and $\hat{p}_1$. The symbol $\bar{p}_1$ erases a copy of a_r if it is present in the system and evolves into $\bar{p}_2$. It does not evolve if no copies of a_r are present. At the same time, $\hat{p}_1$ evolves into $\hat{p}_2$. In the third step, $\hat{p}_2$ evolves into q in the presence of $\bar{p}_2$, i.e. in the case in which the decrement was successful. If the decrement could not happen, $\hat{p}_2$ finds $\bar{p}_1$, which produces the trap symbol.

The zero test branch begins by splitting the state symbol p into $\tilde{p}_1$ and $\dot{p}_1$. The symbol $\dot{p}_1$ must evolve into the trap symbol $\#$ if it finds a copy of a_r, as $\dot{p}_1 a_r \to \#$ is the only rule which may transform $\dot{p}_1$. In the meantime, $\tilde{p}_1$ evolves into $\tilde{p}_2$. If in the third step $\dot{p}_1$ is still present in the system, this means that it did not find any copies of a_r, the register is empty, and the symbol s is produced. Otherwise $\tilde{p}_2$ cannot evolve, but this also means that a trap symbol was produced at step 2, meaning that the computation will never halt.

The argument above shows that the language of valkyries in Queens of the Hill tournaments is rich enough. However, note how this argument relies on two essential details which are partially relevant or even irrelevant in Queens of the Hill: non-determinism and halting. On the one hand, non-determinism is resolved probabilistically, meaning that not all possibilities will be explored, and that some of them may be explored multiple times. Furthermore, proofs of computational completeness in P systems classically consider the results produced at the end of halting computations, while in Queens of the Hill halting does not have a central role. What is important in Queens of the Hill is communicating with the other valkyries, i.e. attempting to dissolve as many of their membranes as possible, as soon as possible. From this standpoint, efficiency is important, while actual computational complexity is much less relevant, as long as the valkyrie manages to attain a relatively high score. Finally, note how $|\Sigma|$ is a powerful tool for modulating the complexity and the efficiency of individual valkyries.

4 Tentative Strategies

The main goal of Queens of the Hill is turning P system design into a game involving teams of students on the front line, backed by researchers collecting and systematizing the explicit and implicit knowledge produced by the teams designing the valkyries. In this section, we present several tentative strategies, whose efficiency or relevance will be the subject of immediate future work.

One of the first strategies one may think of when seeing the rules of Queens of the Hill is the Bomber: eject δ_t for some value of t into the skin membrane 0 and hope that none of those symbols is sent back into the same membrane. The efficiency of the bomber

$$\boxed{\begin{array}{c}\{a \to a(\delta_t, out) \mid 1 \leq t \leq k\} \\ a\end{array}}_1$$

Fig. 3: The Bomber.

decreases as the number of valkyries decreases. For example, when there is only one

other valkyrie, the probability is quite high that the ejected δ_t lands back in the Bomber. Note that this probability is not exactly $\frac{1}{2}$, since the rule $\delta_t \to \delta_{t-1}$ can also be applied in the skin, potentially until the production of δ.

The Bomber can be made more robust by making it accumulate copies of $\bar{\delta}_t$ for some values of t, so that the symbols δ_t coming from the skin annihilate with the corresponding copies of $\bar{\delta}_t$. While this strategy can deal with an occasional $\bar{\delta}_t$, it will be quickly overwhelmed when sharing the tournament with a considerable population of bombers.

$$\begin{array}{|c|}\hline \{a \to a\bar{\delta}_t(\delta_t, out) \mid 1 \le t \le k\} \\ a \\ \hline \end{array}_{\,1}$$

Fig. 4: The Bar Bomber.

Another variation of the Bomber is the strategy of ejecting $\bar{\delta}_t$ in the hope of neutralizing δ_t before it even gets into the valkyrie. This has the obvious disadvantage that it will also protect the other valkyries from δ_t.

$$\begin{array}{|c|}\hline \{a \to a(\bar{\delta}_t, out)(\delta_t, out) \mid 1 \le t \le k\} \\ a \\ \hline \end{array}_{\,1}$$

Fig. 5: The Anti-Bomber.

In case the number of competing valkyries m—or an upper bound on m—is known, robustly dealing with such bomber strategies is in fact not very difficult: it suffices to ensure the presence of $r(m-1)$ copies of $\bar{\delta}_1$ at all times, where $r \in \mathbb{N} \setminus \{0\}$ is a natural factor which we discuss in the following paragraph. Indeed, it is not necessary to provide for $\bar{\delta}_t$ for $t > 1$, as these symbols will inextricably tick down to δ_1 and will have to annihilate with $\bar{\delta}_1$.

$$\begin{array}{|c|}\hline a \to a\delta_1^{r(m-1)} \\ a\delta_1^{r(m-1)} \\ \hline \end{array}_{\,1}$$

Fig. 6: The Delta Wall.

The idea behind the factor r is that other strategies may try to beat the Delta Wall by having rules emitting a large number of δ_t. However, the more such symbols are emitted,

the lower the probability that they end up in the same valkyrie, meaning that the Delta Wall will have a high degree of resilience, even for smaller values of r, like 3 or even 2.

Another protective strategy consists in wrapping the valkyrie in a couple of additional membranes. In this way, the valkyrie can tolerate several membrane dissolutions without being thrown out of the game.

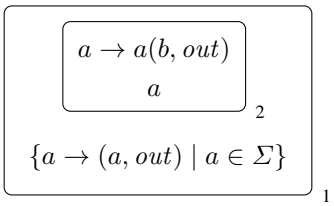


Fig. 7: The 2-layer Onion.

Remark that the Onion will have trouble emitting δ_t symbols. Firstly, the rules in the valkyrie are not allowed to contain δ_t in their left-hand sides, so the rules of the shape $\delta_t \to (\delta_t, out)$ are not allowed. If instead of emitting δ_t a different symbol d is used, then it is necessary to convert d into δ_t at some moment. If such conversion rules only appear in the outermost membrane of the Onion, then dissolving that membrane will remove those rules. On the other hand, including such rules in every layer of the Onion will create the possibility that an inner level inadvertently causes the dissolution of an outer level. Therefore, a strategy which may work best with the Onion would consist in relying on the relative scarcity of the symbols in Σ and in trying to destabilize the other valkyries by forcing some unexpected symbols into their membranes. For this to work, $|\Sigma|$ has to be sufficiently small.

Finally, the last tentative strategy we present in this section is the Bombshell. The idea is to have multiple inner membranes which are all released into the skin membrane of the tournament, therefore creating a family of cooperating agents belonging to the same team. This allows for exceeding the total number of valkyies m, but comes at the price of dissolving a membrane, which will be reflected in the final score.

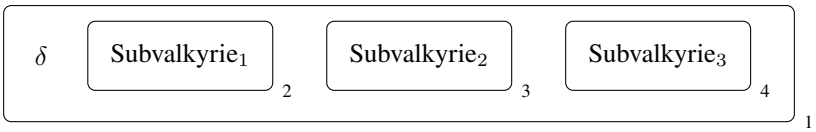


Fig. 8: The scheme of a 3-charge Bombshell.

5 Future Work and Perspectives

The immediate future work is setting up Queens of the Hill tournaments between valkyries designed by teams of students taking a course in formal languages or in natural computing. Queens of the Hill can be seen as a programming exercise in the language of an unconventional model of computing with a concrete goal: attacking all other contestants and surviving against their attacks for as long as possible. This context can also be used to introduce questions from theoretical biology about evolution and robustness, somewhat in the spirit of [18,19]. We remark that such exercises are quite widespread in teaching of multi-agent systems and autonomic systems, as NetLogo-related resources illustrate [20].

To us as teachers and researches (enseignants-chercheurs as they say in French), Queens of the Hill is a great opportunity to employ our students' creativity to push the frontiers of what can be done with P systems. In the particular setup we describe in this paper we focus on transition P systems with non-elementary membrane dissolution and some *r*udimentary matter-antimatter annihilation rules, a model directly supported by P-Lingua. Obviously, other variants of P systems and the corresponding simulator engines can be used as the underlying formalism, thereby stimulating the students' interest in these other variants. Among the salient examples we cite kernel P systems [8,13] and cP systems [9,10,15].

While valkyrie P systems are in principle computationally complete (Section 3), individual computational steps are less expressive than register machine instructions, meaning that designing valkyries *de facto* explores the capabilities of a less powerful language. Furthermore, good valkyrie design will require estimating the probabilities of different branches of computation, which will encourage the students to delve deeper into probability theory.

The setup we propose in this paper is at an early stage. We will most likely need to further tune the values of the parameters in Table 1, and probably also adjust some aspects of the definitions of valkyrie P systems as well as of the tournament in order to avoid trivial edge cases and incite the design of complex strategies. An important question is the relevance of the scoring function $\mathrm{score}(\Pi_j)$ introduced in Section 2.3—other scoring functions may better capture the results of the competition. It is also possible to define scoring functions measuring the production of a certain set of symbols, thereby shifting the focus away from membrane dissolution entirely. One could also think about tracking the origins of the symbols, which could in principle allow saying which valkyrie dissolved which other valkyrie. This would require a rather fine analysis of the computations.

On a final note, we remark that while Queens of the Hill tournaments are directly inspired by Core Wars, the P system context shuffles things up quite a bit. In particular, data is secondary in Core Wars, and warriors interact by writing over each other's code. If we take the rules to be the program in P systems, then the programs of the valkyries are immutable in the sense that individual rules cannot be modified[2]. However, it is possible to instantly and entirely erase parts of their programs by dissolving the corresponding membranes. Furthermore, P systems are inherently non-deterministic, which we translate into a probabilistic framework, while warriors in Core Wars are deterministic. These

[2]This may be an opportunity for plugging in polymorphic P systems and other P system variants with dynamic rules [2].

remarks make us believe that Queens of the Hill tournaments have great potential waiting to be explored.

Acknowledgements

The authors would like to thank the Organizing Committee of the 19th Brainstorming Week on Membrane Computing[3] (BWMC 2023) for organizing this fruitful event.

References

1. Artiom Alhazov, Bogdan Aman, Rudolf Freund, and Gheorghe Paun. Matter and anti-matter in membrane systems. In Helmut Jürgensen, Juhani Karhumäki, and Alexander Okhotin, editors, *Descriptional Complexity of Formal Systems - 16th International Workshop, DCFS 2014, Turku, Finland, August 5-8, 2014. Proceedings*, volume 8614 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2014.
2. Artiom Alhazov, Rudolf Freund, and Sergiu Ivanov. Polymorphic P systems: A survey. Technical report, Bulletin of the International Membrane Computing Society, December 2016.
3. Lucie Ciencialová, Erzsébet Csuhaj-Varjú, Ludek Cienciala, and Petr Sosík. P colonies. *J. Membr. Comput.*, 1(3):178–197, 2019.
4. Erzsébet Csuhaj-Varjú and György Vaszil. P automata or purely communicating accepting P systems. In Gheorghe Păun, Grzegorz Rozenberg, Arto Salomaa, and Claudio Zandron, editors, *Membrane Computing, International Workshop, WMC-CdeA 2002, Curtea de Arges, Romania, August 19-23, 2002, Revised Papers*, volume 2597 of *Lecture Notes in Computer Science*, pages 219–233. Springer, 2002.
5. Ignacio Pérez-Hurtado et al. The P-Lingua Website. http://www.p-lingua.org/wiki/index.php/Main_Page, Retrieved in May 2023.
6. Rudolf Freund, Sergiu Ivanov, and Ludwig Staiger. Going beyond turing with P automata: Regular observer ω-languages and partial adult halting. *Int. J. Unconv. Comput.*, 12(1):51–69, 2016.
7. Manuel García-Quismondo, Rosa Gutiérrez-Escudero, Miguel A. Martínez-del-Amor, Enrique Orejuela-Pinedo, and Ignacio Pérez-Hurtado. P-lingua 2.0: A software framework for cell-like P systems. *Int. J. Comput. Commun. Control*, 4(3):234–243, 2009.
8. Marian Gheorghe, Rodica Ceterchi, Florentin Ipate, Savas Konur, and Raluca Lefticaru. Kernel P systems: From modelling to verification and testing. *Theor. Comput. Sci.*, 724:45–60, 2018.
9. Alec Henderson and Radu Nicolescu. Actor-like cP systems. In Thomas Hinze, Grzegorz Rozenberg, Arto Salomaa, and Claudio Zandron, editors, *Membrane Computing - 19th International Conference, CMC 2018, Dresden, Germany, September 4-7, 2018, Revised Selected Papers*, volume 11399 of *Lecture Notes in Computer Science*, pages 160–187. Springer, 2018.
10. Alec Henderson, Radu Nicolescu, and Michael J. Dinneen. Solving a PSPACE-complete problem with cP systems. *J. Membr. Comput.*, 2(4):311–322, 2020.
11. Bulletin of the International Membrane Computing Society (IMCS). http://membranecomputing.net/IMCSBulletin/index.php.
12. Ilmari Karonen. The beginners' guide to Redcode. https://vyznev.net/corewar/guide.html, version 1.23, August 11, 2020.

[3] http://www.gcn.us.es/19bwmc

13. Savas Konur, Laurentiu Mierla, Florentin Ipate, and Marian Gheorghe. kPWorkbench: A software suit for membrane systems. *SoftwareX*, 11:100407, 2020.
14. Ivan Korec. Small universal register machines. *Theor. Comput. Sci.*, 168(2):267–301, 1996.
15. Radu Nicolescu and Alec Henderson. An introduction to cP systems. In Carmen Graciani Díaz, Agustín Riscos-Núñez, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Enjoying Natural Computing - Essays Dedicated to Mario de Jesús Pérez-Jiménez on the Occasion of His 70th Birthday*, volume 11270 of *Lecture Notes in Computer Science*, pages 204–227. Springer, 2018.
16. Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.
17. Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors. *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.
18. Rémi Segretain, Sergiu Ivanov, Laurent Trilling, and Nicolas Glade. A methodology for evaluating the extensibility of boolean networks' structure and function. In Rosa M. Benito, Chantal Cherifi, Hocine Cherifi, Esteban Moro, Luis Mateus Rocha, and Marta Sales-Pardo, editors, *Complex Networks & Their Applications IX - Volume 2, Proceedings of the Ninth International Conference on Complex Networks and Their Applications, COMPLEX NETWORKS 2020, 1-3 December 2020, Madrid, Spain*, volume 944 of *Studies in Computational Intelligence*, pages 372–385. Springer, 2020.
19. Rémi Segretain, Laurent Trilling, Nicolas Glade, and Sergiu Ivanov. Who plays complex music? On the correlations between structural and behavioral complexity measures in sign Boolean networks. In *21st IEEE International Conference on Bioinformatics and Bioengineering, BIBE 2021, Kragujevac, Serbia, October 25-27, 2021*, pages 1–6. IEEE, 2021.
20. Uri Wilensky. NetLogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

Pure 2D Eilenberg P Systems

Somnath Bera[1], Atulya K. Nagar[2], K.G. Subramanian[2,*], and Gexiang Zhang[3]

[1] School of Advanced Sciences-Mathematics, Vellore Institute of Technology, Chennai, Tamil Nadu 600 127 India
[2] School of Mathematics, Computer Science and Engineering, Liverpool Hope University, Hope Park, Liverpool L16 9JD, UK
([*] Honorary Visiting Professor; Email: kgsmani1948@gmail.com)
[3] School of Automation, Chengdu University of Information Technology, Chengdu, Sichuan, China

Abstract. The computational power of Eilenberg P systems with string objects and rewriting rules in generating languages, has been studied. Extending this system to two dimensions, we introduce here an Eilenberg P system $P2DEPS$ with rectangular picture array objects and pure 2D context-free rules and examine the array language generating power of these systems. We show that with two membranes, any pure 2D context-free language can be generated. We also show that $P2DEPS$ has more generative power than pure 2D context-free grammar ($P2DCFG$). We also compare $P2DEPS$ with certain other picture array generating grammars.

Keywords: Eilenberg P system · pure context-free rules · pure 2D context-free grammar.

1 Introduction

In the area of membrane computing [7,8], based on the biologically inspired computing model of P system introduced by Gh. Păun with string objects and evolution rules involving rewriting operation [6], a P system, called Eilenberg P system (EPS) was introduced in [1]. An EPS involves an Eilenberg machine which is similar to a finite state machine having associated with each transition, a specific set of evolution rules that are context-free rewriting rules belonging to a region of the EPS. The EPS starts in an initial state with an initial set of string objects. In the current state, with a current set of string objects, the EPS evolves by applying the rules associated with one of the transitions emerging out from the current state. The EPS continues its activity from the next state of the transition. Here we extend the concept of an EPS to two dimensions by introducing a P system, called pure 2D Eilenberg P system ($P2DEPS$) having picture array objects and tables of pure 2D context-free rules [12] as evolution rules in its regions. Here again as in the case of EPS, we have an Eilenberg machine with the evolution rules from the regions associated with the transitions. We examine the picture array generative power of $P2DEPS$ and show that any pure 2D context-free language [10] can be generated by a $P2DEPS$ with two membranes and that $P2DEPS$ has more generative power than pure 2D context-free grammar ($P2DCFG$). We also compare the generative power of $P2DEPS$ with certain other picture array grammars.

2 Preliminaries

For formal language theory related notions, the reader can refer to [9] and to [5,12] for two-dimensional array grammars and languages. For P systems and array P systems we refer to [6,11]. For the definitions of Eilenberg P system and Eilenberg machine we refer to [1,3]. We now recall certain basic notions on words and picture arrays as well as the definition of pure 2D context-free grammar [12].

A finite set T of symbols is called an alphabet. A word or a string $w = a_1a_2 \dots a_m$ $a_i \in T, 1 \leq i \leq m, (m \geq 1)$ of length m over an alphabet T is a finite sequence of symbols belonging to T. We denote by $|w|$, the length of the word w. The set of all words over T, including the empty word λ with no symbols, is denoted by T^*. For any word $w = a_1a_2 \dots a_n$, $t(w)$ is the vertical word with the word w written vertically. For example, if $w = aab$ over the alphabet $\{a, b\}$, then $t(w)$ is $\begin{matrix} a \\ a \\ b \end{matrix}$. A $p \times q$ array with p rows and q columns (also called a $p \times q$ picture array) M over an alphabet T is of the form

$$M = \begin{matrix} a_{11} & \cdots & a_{1q} \\ \vdots & \ddots & \vdots \\ a_{p1} & \cdots & a_{pq} \end{matrix}$$

where each $a_{ij} \in T, 1 \leq i \leq p$, $1 \leq j \leq q$. We denote by $|M|_r$, the number of rows of M and by $|M|_c$, the number of columns of M. The set of all picture arrays over T is denoted by T^{**}, which includes the empty array λ. $T^{++} = T^{**} - \{\lambda\}$. A picture array language is a subset of T^{**}.

Definition 1. *A pure 2D context-free grammar ($P2DCFG$) is given by* $G = (T, R_1, R_2, X)$ *where*

- T *is a finite set of symbols ;*
- $R_1 = \{c_i | 1 \leq i \leq m\}, R_2 = \{r_j | 1 \leq j \leq n\}$;

Each $c_i, (1 \leq i \leq m)$, called a column table, is a set of pure context-free rules of the form $a \to \alpha, a \in T, \alpha \in T^*$ such that for any two rules $a \to \alpha, b \to \beta$ in c_i, we have $|\alpha| = |\beta|$; Each $r_j, (1 \leq j \leq n)$, called a row table, is a set of pure context-free rules of the form $d \to t(\gamma), d \in T$ and $\gamma \in T^*$ such that for any two rules $d \to t(\gamma), e \to t(\delta)$ in r_j, we have $|\gamma| = |\delta|$;

- $X \subseteq T^{**} - \{\lambda\}$ *is a finite set of axiom arrays.*

A derivation in a $P2DCFG$ is defined as follows: For any two arrays M_1, M_2, we write $M_1 \Rightarrow M_2$ if M_2 is obtained from M_1 by either rewriting a column of M_1 by rules of some column table c_i in R_1 or a row of M_1 by rules of some row table r_j in R_2. The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$.
The picture array language $L(G)$ generated by G is the set of picture arrays $\{M | M_0 \Rightarrow^* M \in T^{**}$, for some $M_0 \in X\}$.

The family of picture array languages generated by pure 2D context-free grammars is denoted by $P2DCFL$.

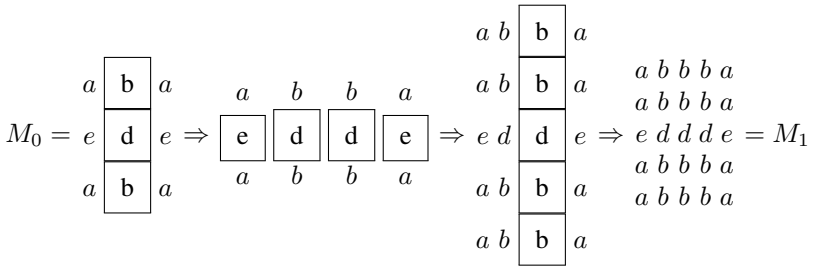


Fig. 1: Derivation $M_0 \Rightarrow^* M_1$

We illustrate picture array generation in a $P2DCFG$ with an example.

Example 1. Consider the pure 2D context-free grammar $G_1 = (T, R_1, R_2, \{M_0\})$ where $T = \{a, b, d, e\}$, $R_1 = \{c\}, R_2 = \{r\}$. The column table c and the row table r are given by $c = \{b \to bb, d \to dd\}$, $r = \left\{ \begin{matrix} a & b \\ e \to e\,, d \to d \\ a & b \end{matrix} \right\}$, $M_0 = \begin{matrix} a\ b\ a \\ e\ d\ e \\ a\ b\ a \end{matrix}$
On using c, r, c in this order, we have the derivation $M_0 \Rightarrow^* M_1$, which is shown in Fig. 1. The symbols rewritten in a column or a row are enclosed in boxes in the Figure.

3 Pure 2D Eilenberg P System

We now introduce a new array P system, called pure 2D Eilenberg P system ($P2DEPS$) extending the definition of EPS to two dimensions with the regions of the P system having rectangular picture array objects and pure 2D context-free grammar type of tables of rules but the picture arrays evolve on applying the rules of the tables in the regions that are associated with a transition emerging out of the current state. Thus the definition of the two-dimensional extension of EPS is very similar to the string case [1,3] except that a deterministic version of the Eilenberg machine is used in the sense that the system from a given state moves to at most one state on reading the label of a transition. Some or all the states can be final states. The working of the two-dimensional extension of EPS is also similar to the string case except that the result, namely, the picture array generated, is collected in a designated output region of the system.

Definition 2. *A pure 2D Eilenberg P system ($P2DEPS$) is given by* $\Pi = (\mu, T, Q, X_1, \cdots, X_n, \mathcal{C}, \delta, q_0, F, i_o)$ *where*

(i) μ *is a membrane structure having* n *membranes;*
(ii) T *is the alphabet of the system;*
(iii) Q *is a finite set of states;*
(iv) $X_i, (1 \leq i \leq n)$ *is a finite set (which can be empty) of picture arrays representing the initial value in the region labelled* i*;*

(iv) $\mathcal{C} = (C_1, \cdots, C_p)$, *where for* $1 \leq i \leq p$, *the component* $C_i = (P_{i,1}, \cdots, P_{i,n})$ *with* $P_{i,j}$, *a finite set (which can be empty) of column and/or row tables of pure 2D context-free grammar rules (as in Definition 1) with a target* ($here$, in *or* out) *for each table of rules and the elements of* $P_{i,j}$ *belong to region* i;
(v) $\delta : Q \times \mathcal{C}; \to Q$ *is the next-state function;*
(vi) q_0 *is the initial state;*
(vii) F *is a set of final states;*
$(viii)$ i_0 *is the label of the output region where the result of the system is collected.*

A computation in a $P2DEPS$ *takes place as follows: A computation starts with the initial state* q_0 *and the initial configuration of the system* $(M_1, \cdots, M_n)$. *A column or a row table of pure context-free rules in the component associated with the transition emerging out of* q_0, *is used to rewrite (with the rewriting done as in a pure 2D context-free grammar) the picture arrays in the corresponding region. At a time, only one table of rules can be applied to a picture array rewriting a column or a row of symbols and all the picture arrays which can be rewritten are to be rewritten The picture arrays evolved either remain in the same region or are sent to an immediate inner or outer region depending on the target* $here$, in *or* out *indicated with the table of rules. The process is continued using a table of rules in the component associated with a transition from the current state to which the system reaches from an immediate earlier step. At a given step of computation, a picture array in a region to which a non-deterministically chosen table of rules in the component used is applicable, is rewritten (as done in a pure 2D context-free grammar), thus evolving the picture array. The process repeats and the resulting configuration of the system in a computation step consists of picture arrays evolved at that moment. A computation halts when no table of rules associated with the current transition is applicable to the picture arrays in the regions. When the system reaches a final state and a halting computation, the picture arrays collected in the output region* i_0 *constitute the picture array language* $L(\Pi)$ *generated by the* $P2DEPS$.

The family of picture array languages generated by $P2DEPS$ *is denoted by* $\mathcal{L}(P2DEPS)$. *If we indicate the three parameters, namely, the number of membranes at most* m, *the number of states at most* s *and the number of components at most* p, *then we denote the system by* $P2DEPS(m, s, p)$ *and the corresponding family by* $\mathcal{L}(P2DEPS(m, s, p))$.

We illustrate with an example.

Example 2. Consider the $P2DEPS(1, 3, 3)$
$\Pi_1 = ([_1]_1, \{a, b, d, e\}, \{q_1, q_2, q_3\}, X_1, \mathcal{C}, \delta, q_1, 1, \{q_3\})$ where
$\mathcal{C} = (C_1, C_2, C_3)$, $\delta(q_1, C_1) = q_2, \delta(q_2, C_2) = q_1, \delta(q_1, C_3) = q_3$, with $C_1 = (\{c_1\}), C_2 = (\{r\}), C_3 = (\{c_2\})$. The column tables c_1, c_2 are given by $c_1 = \{e \to aeb, d \to ddd\}$, $c_2 = \{e \to ab, d \to dd\}$. The row table r is given by $r = \{a \to \begin{matrix} a \\ d \end{matrix}, e \to \begin{matrix} e \\ d \end{matrix}, b \to \begin{matrix} b \\ d \end{matrix}\}$. All the tables of rules have the target $here$. $X_1 = \{M_1\}$, with the initial value in the region being $M_1 = \begin{matrix} a\ e\ b \\ d\ d\ d \end{matrix}$.
The picture array language $L(\Pi_1)$ generated by Π_1 consists of $(n+1) \times (2n+2), (n \geq 1)$,

picture arrays M where $M_{1j} = a$ for $1 \leq j \leq n+1$, $M_{1j} = b$, for $n+2 \leq j \leq 2n+2$ and all other entries are d. A picture array of $L(\Pi_1)$ is shown in Fig. 2. The $P2DEPS$

$$\begin{array}{c} a \cdots a\ e\ b \cdots b \\ d \cdots d\ d\ d \cdots d \\ \vdots\ \ddots\ \vdots\ \vdots\ \vdots\ \ddots\ \vdots \\ d \cdots d\ d\ d \cdots d \end{array}$$

Fig. 2: A picture array of $L(\Pi_1)$

Π_1 has only one membrane with an initial value, namely, the picture array M_1 in the membrane. The system Π_1 starts in the initial state q_1. If the emerging transition at q_1 given by $\delta(q_1, C_1) = q_2$ is used, then the rules of the column table c_1 are used to rewrite the column $t(ed)$ of the only initial picture array M_1 to yield a picture array

$$M_2 = \begin{array}{c} a\ a\ e\ b\ b \\ d\ d\ d\ d\ d \end{array}.$$

The next state reached by the system is q_2. The system now uses the only emerging transition given by $\delta(q_2, C_2) = q_1$ and so the rules of the row table r are applied rewriting the row $aaebb$ of M_2 yielding the picture array

$$M_3 = \begin{array}{c} a\ a\ e\ b\ b \\ d\ d\ d\ d\ d \\ d\ d\ d\ d\ d \end{array}$$

while the system reaches the next state q_1. The process can repeat as many times as needed until the emerging transition at q_1 given by $\delta(q_1, C_3) = q_3$ is used which makes the system reach the final state q_3 while the rules of the column table c_2 are applied to the picture array rewriting the symbols in the column $t(ed \cdots d)$ and yielding the picture array of the form shown in Fig. 2. The computation halts as there is no outgoing transition in the state q_3. Since the system reaches a halting computation and is also in the final state, the picture array generated is collected in the picture array language $L(\Pi_1)$. In fact $L(\Pi_1)$ belongs to the family $\mathcal{L}(P2DEPS(1, 3, 3))$.

4 Comparison results

We now compare the picture array generative power of $P2DEPS$ with $P2DCFG$. We show that a $P2DCFL$ can be generated by a $P2DEPS$ with two membranes, one state and two components. We also construct such a $P2DEPS$ generating a picture array language that cannot be generated by any $P2DCFG$.

Theorem 1. $P2DCFL \subseteq \mathcal{L}(P2DEPS(2, 1, 2))$

Proof. Consider a $P2DCFG\ G = (T, R_1, R_2, X)$ generating a picture array language L. We construct a $P2DEPS\ \Pi = ([_1[_2]_2]_1, T, \{q_1\}, X, \mathcal{C}, \delta, q_1, \{q_1\}, 2)$ where $\mathcal{C} = (C_1, C_2)$ with $C_1 = (R_1 \cup R_2, \emptyset)$ $C_2 = (\{c\}, \emptyset)$ where the column table $c = \{a \to a \mid a \in T\}$. $\delta(q_1, C_1) = \delta(q_1, C_2) = q_1$. The tables of rules that belong to $R_1 \cup R_2$ have the target $here$ and the column table c has the target in. Corresponding to a derivation in the $P2DCFG\ G$, a computation in the $P2DEPS\ \Pi$ is done as follows: The system Π starts in the initial state q_1 and an initial value which is an axiom picture array from X in the membrane with label 1. Since $\delta(q_1, C_1) = q_1$, rules of a column or row table in $R_1 \cup R_2$ can be applied simulating a corresponding step of derivation in G with the system remaining in the state q_1 itself. Thus a sequence of derivation steps in a derivation in G will be captured in the computation in Π. At any intermediate step of computation, the system Π can use the rules of the column table c rewriting any column in the picture array of that intermediate step. The application of the rules of the column table c does not change the picture array but the picture array is sent to the inner membrane with label 2. The computation halts as there are no rules (of the region 2) in the components that can be applied and the system is in the final state q_1 itself. Thus every picture array of L is collected in the output region 2. This proves the inclusion in the statement of the theorem.

Theorem 2. $\mathcal{L}(P2DEPS(2,1,2)) - P2DCFL \neq \phi$.

Proof. We now consider a $P2DEPS$ $\Pi_1 = ([_1[_2]_2]_1, \{a, b, e\}, \{q_1\}, \{M_0\}, \mathcal{C}, \delta, q_1, \{q_1\}, 2)$ where $M_0 = \begin{matrix} a\ e\ b \\ a\ e\ b \end{matrix}$, $\mathcal{C} = (C_1, C_2)$ with $C_1 = (\{c_1, r\}, \emptyset)$ $C_2 = (\{c_2\}, \emptyset)$ where the column tables are $c_1 = \{e \to aeb\}$, $c_2 = \{e \to ab\}$, the row table is $r = \{a \to \begin{matrix} a \\ a \end{matrix}, b \to \begin{matrix} b \\ b \end{matrix}, e \to \begin{matrix} e \\ e \end{matrix}\}$, $\delta(q_1, C_1) = \delta(q_1, C_2) = q_1$. The tables c_1 and r have the target $here$ and the table c_2 has the target in. The system Π_1 generates a picture array language L_1 consisting of $m \times 2n$ $(m \geq 2, n \geq 2)$ picture arrays M such that $M_{ij} = a$ for $1 \leq i \leq m, 1 \leq j \leq n$, and $M_{ij} = b$ for $1 \leq i \leq m, n+1 \leq j \leq 2n$. A picture array of L_1 is shown below:

$$\begin{matrix} a \cdots a\ b \cdots b \\ \vdots\ \ddots\ \vdots\ \vdots\ \ddots\ \vdots \\ a \cdots a\ b \cdots b \end{matrix}$$

In fact the system Π_1 starts in the initial state q_1 with the initial picture array M_0 in the region with label 1. If the transition $\delta(q_1, C_2) = q_1$ is used, then the rules of the column table c_2 are applied to M_0 which evolves into the picture array $M_1 = \begin{matrix} a\ a\ b\ b \\ a\ a\ b\ b \end{matrix}$. M_1 is sent to the region with label 2. The computation halts with the system in the only state q_1 which is a final state. The picture array M_1 is collected in the language. If the transition $\delta(q_1, C_1) = q_1$ is used and the rules of the column table c_1 are applied to M_0, then it evolves into the picture array $\begin{matrix} a\ a\ e\ b\ b \\ a\ a\ e\ b\ b \end{matrix}$ that remains in the same membrane. The process can continue and at any step of computation, the rules of the row table r can be used if

the transition selected is $\delta(q_1, C_1) = q_1$ and a row of the form $a^k e b^k$ (for some $k \geq 1$) is added to the picture array. Again if the transition $\delta(q_1, C_2) = q_1$ is selected with the system continuing in the state q_1, then the picture array generated is sent to region with label 2 and the computation halts. As the state q_1 is a final state, the picture array generated is collected in the language L_1. But this language cannot be generated by any $P2DCFG$. If such a $P2DCFG$ G_1 generates L_1, then starting from an axiom picture array, a column of $a's$ or a column of $b's$ are to be rewritten by rules of a column table of rules. But then any pure context-free rule for a or b will only yield pictures not in the language since every picture array in L_1 has a certain number of continuous columns (starting from the first column) of $a's$ followed by an equal number of columns of $b's$.

A two-dimensional picture array generating grammar, called regular matrix grammar (RMG), was introduced in [10] and extensively investigated in many studies. This two-dimensional grammar was later renamed as two-dimensional right-linear grammar ($2RLG$) [5]. Extensions of RMG to the context-free and context-sensitive cases have been considered in [10] and their properties have been studied. We refer to these two-dimensional grammars as two-dimensional context-free grammar ($2CFG$) and two-dimensional context-sensitive grammar ($2CSG$) in line with the naming of $2RLG$. We compare the generative power of $P2DEPS$ with those of $2RLG$, $2CFG$, and $2CSG$. In all these three grammars, namely, $2RLG, 2CFG, 2CSG$, there are two phases of derivations with the first phase involving respectively, regular, context-free and context-sensitive string grammars generating a string language over a set of symbols, called intermediates. Every string of intermediates obtained in the first phase is rewritten in the second phase in all these three grammars in the vertical direction using nonterminal regular grammar rules of the form $A \to aB$ applied together or terminal rules of the form $A \to a$ that are applied together, to generate the columns of the picture arrays over terminal symbols. The families of picture languages generated by $2RLG, 2CFG, 2CSG$ are respectively denoted by $2RLL, 2CFL, 2CSL$.

Theorem 3. $\mathcal{L}(P2DEPS(2, 1, 2)) - 2RLL \neq \emptyset$.

Proof. Consider the picture array language L_2 consisting of $m \times (2n + 1)$ $(n \geq 1)$ picture arrays M such that $M_{1j} = a$, for $1 \leq j \leq n$, $M_{1(n+1)} = d$, $M_{1j} = b$, for $n + 2 \leq j \leq 2n + 1$, $M_{ij} = d$, otherwise. A $P2DEPS$ with two membranes, 1 state and 2 components, generating L_2 is given by

$\Pi'_2 = ([_1[_2]_2]_1, \{a, b, e, d\}, \{q_1\}, \{M_0\}, \mathcal{C}, \delta, q_1, \{q_1\}, 2)$ where $M_0 = \begin{matrix} a\ e\ b \\ d\ d\ d \end{matrix}$, $\mathcal{C} = (C_1, C_2)$ with $C_1 = (\{c_1, r\}, \emptyset)$, $C_2 = (\{c_2\}, \emptyset)$, where the column tables are $c_1 = \{e \to aeb, d \to ddd\}$, $c_2 = \{e \to d, d \to d\}$, the row table $r = \{a \to \begin{matrix} a \\ d \end{matrix}, b \to \begin{matrix} b \\ d \end{matrix}, e \to \begin{matrix} e \\ d \end{matrix}\}$. $\delta(q_1, C_1) = q_1$, $\delta(q_1, C_2) = q_1$. The tables of rules c_1, r have the target $here$ and c_2 has target in. The system Π'_2 starts in the initial state q_1 with the initial picture array M_0 in region with label 1. If the transition $\delta(q_1, C_2) = q_1$ is used, then the rules of the column table c_1 or the rules of the row table r are applied to M_0 yielding a picture array which remains in the same membrane and the next state is the same state q_1. The process can repeat as many times as needed generating a picture array of the form

$$\begin{array}{c} a \cdots a\ e\ b \cdots b \\ d \cdots d\ d\ d \cdots d \\ \vdots\ \ddots\ \vdots\ \vdots\ \vdots\ \ddots\ \vdots \\ d \cdots d\ d\ d \cdots d \end{array}$$

With the system at the state q_1, if the transition $\delta(q_1, C_2) = q_1$ is used, then the rules of the column table c_2 are applied to the picture array evolved at that moment yielding a picture array belonging to L_2. This picture array is collected in the language as the computation halts with the system remaining in the final state q_1. But the picture array language L_2 cannot be generated by any $2RLG$ since the first rows of the picture arrays constitute a context-free language, namely $\{a^n db^n \mid n \geq 1\}$. This would mean that a $2CFG$ will be required to generate L_2.

Theorem 4. $\mathcal{L}(P2DEPS(1,2,2)) - 2RLL \neq \emptyset$.

Proof. The picture array language L_2 considered in the proof of Theorem 3 cannot be generated by any $2RLG$ as seen in the proof of Theorem 3. A $P2DEPS$ with one membrane, 2 states and 2 components, generating L_2 is given by $\Pi_2 = ([_1]_1, \{a, b, d_1, d_2, e, d\}, \{q_1, q_2\}, \{M_0\}, \mathcal{C}, \delta, q_1, \{q_2\}, 1)$ where $M_0 = \begin{matrix} d_1\ e\ d_2 \\ d\ \ d\ \ d \end{matrix}$, $\mathcal{C} = (C_1, C_2)$ with $C_1 = (\{c, r_1\})$, $C_2 = (\{r_2\})$, where the column table is $c = \{e \to d_1 e d_2, d \to ddd\}$, the row tables are $r_1 = \{d_1 \to \begin{matrix} d_1 \\ d \end{matrix}, d_2 \to \begin{matrix} d_2 \\ d \end{matrix}, e \to \begin{matrix} e \\ d \end{matrix}\}$, $r_2 = \{d_1 \to a, d_2 \to b, e \to d\}$. $\delta(q_1, C_1) = q_1$, $\delta(q_1, C_2) = q_2$. The tables of rules have the target $here$. The system Π_2 starts in the initial state q_1 with the initial picture array M_0 in region with label 1. If the transition $\delta(q_1, C_1) = q_1$ is used, then the rules of the column table c or the rules of the row table r_1 are applied to M_0 yielding a picture array which remains in the same membrane and the next state is the same state q_1. The process can repeat as many times as needed generating a picture array of the form

$$\begin{array}{c} d_1 \cdots d_1\ e\ d_2 \cdots d_2 \\ d\ \cdots\ d\ \ d\ \ d\ \cdots\ d \\ \vdots\ \ddots\ \vdots\ \vdots\ \vdots\ \ddots\ \vdots \\ d\ \cdots\ d\ \ d\ \ d\ \cdots\ d \end{array}$$

With the system at the state q_1, if the transition $\delta(q_1, C_2) = q_2$ is used, then the rules of the row table r_2 are applied to the picture array evolved at that moment yielding a picture array of the form

$$\begin{array}{c} a \cdots a\ d\ b \cdots b \\ d \cdots d\ d\ d \cdots d \\ \vdots\ \ddots\ \vdots\ \vdots\ \vdots\ \ddots\ \vdots \\ d \cdots d\ d\ d \cdots d \end{array}$$

This picture array is collected in the language as the computation halts with the system reaching a final state q_2.

Theorem 5. $\mathcal{L}(P2DEPS(2,2,4)) - 2CFL \neq \emptyset$.

Proof. Consider the picture array language L_3 consisting of $m \times (3n+2)$ $(m \geq 2, n \geq 1)$ picture arrays M such that $M_{1j} = a$, for $1 \leq j \leq n$, $M_{1j} = b$, for $n+2 \leq j \leq 2n+1$, $M_{1j} = e$, for $2n+3 \leq j \leq 3n+2$, $M_{ij} = d$, otherwise. A $P2DEPS$ with two membranes, 2 states and 4 components, generating L_3 is given by
$\Pi_3 = ([_1[_2]_2]_1, \{a, b, e, e_1, e_2\}, \{q_1, q_2\}, \{M_0\}, \mathcal{C}, \delta, q_1, q_1, 2)$ where
$M_0 = \begin{matrix} a & e_1 & b & e_2 & e \\ d & d & d & d & d \end{matrix}$, $\mathcal{C} = (C_1, C_2, C_3, C_4)$ with $C_1 = (\{r_1\}, \emptyset)$, $C_2 = (\{c_1\}, \emptyset)$, $C_3 = (\{c_2\}, \emptyset)$, $C_4 = (\{r_2\}, \emptyset)$ where the column tables are $c_1 = \{e_1 \rightarrow ae_1b, d \rightarrow ddd\}$, $c_2 = \{e_2 \rightarrow e_2e, d \rightarrow dd\}$, the row tables are $r_1 = \{a \rightarrow \begin{matrix} a \\ d \end{matrix}, e_1 \rightarrow \begin{matrix} e_1 \\ d \end{matrix}, b \rightarrow \begin{matrix} b \\ d \end{matrix}, e_2 \rightarrow \begin{matrix} e_2 \\ d \end{matrix}, e \rightarrow \begin{matrix} e \\ d \end{matrix}, \}$, $r_2 = \{a \rightarrow a, b \rightarrow b, e \rightarrow e, e_1 \rightarrow d, e_2 \rightarrow d\}$, $\delta(q_1, C_1) = q_1$, $\delta(q_1, C_2) = q_2$, $\delta(q_2, C_3) = q_1$, $\delta(q_1, C_4) = q_1$. The column tables of rules c_1, c_2 and the row table of rules r_1 have the target $here$ and the table of rules r_2 has target in. The system Π_3 starts in the initial state q_1 with the initial picture array M_0 in the region with label 1. If the transition $\delta(q_1, C_4) = q_1$ is used, then the rules of the row table r_2 are applied to M_0 yielding a picture array of the form $\begin{matrix} a & d & b & d & e \\ d & d & d & d & d \end{matrix}$ which is sent to the output region 2 while the system If the transition $\delta(q_1, C_1) = q_1$ is used, then the rules of the row table r_1 are applied to M_0 yielding a picture array with a row of $d's$ added below the first row. This picture array remains in the same membrane and the next state is the same state q_1. The process can repeat as many times as needed generating a picture array of the form

$$\begin{matrix} a & e_1 & b & e_2 & e \\ d & d & d & d & d \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ d & d & d & d & d \end{matrix}$$

If the transition $\delta(q_1, C_2) = q_2$ followed by $\delta(q_2, C_3) = q_1$ are used, then the rules of the column table c_1 are applied followed by the application of the rules of the column table c_2 to the picture array remaining in region 1, generating a picture array of the form

$$\begin{matrix} a \cdots a & e_1 & b \cdots b & e_2 & e \cdots e \\ d \cdots d & d & d \cdots d & d & d \cdots d \\ \vdots \ddots \vdots & \vdots & \vdots \ddots \vdots & \vdots & \vdots \ddots \vdots \\ d \cdots d & d & d \cdots d & d & d \cdots d \end{matrix}$$

Note that this process can be repeated with the state moving from q_1 to q_2 and returning back to q_1. When the system is at the state q_1, if the transition $\delta(q_1, C_4) = q_3$ is used, then the rules of the row table r_2 are applied to the picture array evolved at that moment yielding a picture array of the form

$$\begin{matrix} a \cdots a & d & b \cdots b & d & c \cdots c \\ d \cdots d & d & d \cdots d & d & d \cdots d \\ \vdots \ddots \vdots & \vdots & \vdots \ddots \vdots & \vdots & \vdots \ddots \vdots \\ d \cdots d & d & d \cdots d & d & d \cdots d \end{matrix}$$

This picture array is collected in the language as the computation halts with the system reaching the final state q_1. But the picture array language L_3 cannot be generated by any $2CFG$ since the first row of the picture arrays constitute a context-sensitive language, namely $\{a^n db^n dc^n \mid n \geq 1\}$. This would mean that a $2CSG$ will be required to generate L_2.

Theorem 6. $\mathcal{L}(P2DEPS(1,3,4)) - 2CFL \neq \emptyset$.

Proof. The picture array language L_3 considered in the proof of Theorem 5 cannot be generated by any $2CFG$ as seen in the proof of Theorem 5. A $P2DEPS$ with one membrane, 3 states and 4 components, generating L_3 is given by
$\Pi_3 = ([_1]_1, \{a, b, d_1, d_2, d_3, e, d\}, \{q_1, q_2, q_3\}, \{M_0\}, \mathcal{C}, \delta, q_1, \{q_3\}, 1)$ where $M_0 = \begin{matrix} d_1 & e_1 & d_2 & e_2 & d_3 \\ d & d & d & d & d \end{matrix}$, $\mathcal{C} = (C_1, C_2, C_3, C_4)$ with $C_1 = (\{r_1\})$, $C_2 = (\{c_1, \})$ $C_3 = (\{c_2, \})$, $C_4 = (\{r_2\})$ where the column tables are $c_1 = \{e_1 \rightarrow d_1e_1d_2, d \rightarrow ddd\}$, $c_2 = \{e_2 \rightarrow e_2d_3, d \rightarrow dd\}$, the row tables are $r_1 = \{d_1 \rightarrow \begin{matrix} d_1 \\ d \end{matrix}, e_1 \rightarrow \begin{matrix} e_1 \\ d \end{matrix}, d_2 \rightarrow \begin{matrix} d_2 \\ d \end{matrix}, e_2 \rightarrow \begin{matrix} e_2 \\ d \end{matrix}, d_3 \rightarrow \begin{matrix} d_3 \\ d \end{matrix}, \}$, $r_2 = \{d_1 \rightarrow a, d_2 \rightarrow b, d_3 \rightarrow c, e_1 \rightarrow d, e_2 \rightarrow d\}$. $\delta(q_1, C_1) = q_1$, $\delta(q_1, C_2) = q_2$, $\delta(q_2, C_3) = q_1$, $\delta(q_1, C_4) = q_3$. All the tables of rules have the target $here$. The system Π_3 starts in the initial state q_1 with the initial picture array M_0 in the region with label 1. If the transition $\delta(q_1, C_1) = q_1$ is used, then the rules of the row table r_1 are applied to M_0 yielding a picture array which remains in the same membrane and the next state is the same state q_1. The process can repeat as many times as needed generating a picture array of the form

$$\begin{matrix} d_1 & e_1 & d_2 & e_2 & d_3 \\ d & d & d & d & d \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ d & d & d & d & d \end{matrix}.$$

If the transition $\delta(q_1, C_2) = q_2$ followed by $\delta(q_2, C_3) = q_1$ are used, then the rules of the column table c_1 are applied followed by the application of the rules of the table c_2 to the picture array remaining in region 1, generating a picture array of the form

$$\begin{matrix} d_1 \cdots d_1 & e_1 & d_2 \cdots d_2 & e_2 & d_3 \cdots d_3 \\ d \cdots d & d & d \cdots d & d & d \cdots d \\ \vdots \ddots \vdots & \vdots & \vdots \ddots \vdots & \vdots & \vdots \ddots \vdots \\ d \cdots d & d & d \cdots d & d & d \cdots d \end{matrix}.$$

When the system is at the state q_1, if the transition $\delta(q_1, C_4) = q_3$ is used, then the rules of the row table r_2 are applied to the picture array evolved at that moment yielding a picture array of the form

$$\begin{matrix} a \cdots a & d & b \cdots b & d & c \cdots c \\ d \cdots d & d & d \cdots d & d & d \cdots d \\ \vdots \ddots \vdots & \vdots & \vdots \ddots \vdots & \vdots & \vdots \ddots \vdots \\ d \cdots d & d & d \cdots d & d & d \cdots d \end{matrix}.$$

This picture array is collected in the language as the computation halts with the system reaching the final state q_3.

5 Conclusion

We have introduced here an extension of the string language generating Eilenberg P system (EPS) to two dimensions with sets of pure 2D context-free rules in the regions of the system. The resulting P system, namely, $P2DEPS$ generates picture array languages. It will be of interest to compare the generative power of $P2DEPS$ with other picture array grammar models such as [2,4].

Acknowledgement

The authors thank the reviewers for their very useful and relevant comments which helped them to prepare this revised version. This work was supported by the National Natural Science Foundation of China (61972324), the Sichuan Science and Technology Program (23NSFTD0049, 23ZDYF0247, 2022YFG0181).

References

1. Balanescu, T., Gheorghe, M., Holcombe, M., Ipate, F.: Eilenberg P systems, In: Gh. Păun et al (eds): WMC – CdeA 2002, LNCS 2597, pp. 43-57, 2003.
2. Bera, S., Ceterchi, R., Sriram, S., Subramanian,K.G.: Array P systems and pure 2D context-free grammars with independent mode of rewriting. J. Membr. Comput. **4(1)**: 11-20 (2022)
3. Bernardini, Gheorghe, M., Holcombe, M. :PX Systems = P Systems + X Machines, Natural Computing, **2**, 201-213 (2003).
4. Fernau, H., Freund, R., Ivanov, S., Schmid, M.L., Subramanian, K.G. : Array Insertion and Deletion P Systems. In: Mauri, G., Dennunzio, A., Manzoni, L., Porreca, A.E. (eds) Unconventional Computation and Natural Computation. UCNC 2013. Lecture Notes in Computer Science, vol **7956**. Springer, Berlin, 67-78.
5. Giammarresi, D., Restivo, A.: Two-dimensional languages. In: Rozenberg, G., Salomaa, G. (eds.) Handbook of Formal Languages, pp. 215—267. Springer, Berlin (1997).
6. Păun, Gh. : Computing with membranes, J. Comp. System Sci., **61**, 108– 143 (2000).
7. Păun, Gh.: Membrane Computing: An Introduction. Springer-Verlag Berlin, Heidelbrg, 2000.
8. Păun, Gh., Rozenberg, G., Salomaa, A. : The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)
9. Rozenberg, G., Salomaa, A. (Eds.): Handbook of Formal Languages. Vols. 1 - 3, Springer-Verlag, Berlin (1997).
10. Siromoney, G., Siromoney, R., Krithivasan, K.: Abstract families of matrices and picture languages. Computer Graphics and Image Proc. **1**, 284–307 (1972).
11. Subramanian, K.G. : P systems and picture languages. Lecture Notes in Computer Science, Springer Verlag, Vol. 4664, 99–109 (2007).
12. Subramanian, K.G., Ali, R.M., Geethalakshmi, M., Nagar, A.K.: Pure 2D picture grammars and languages. Discrete Appl. Math. **157**, 3401-–3411 (2009).

Solving QUBO problems with cP systems

Lucie Ciencialová[1 [0000−0002−6026−5284]], Michael J. Dinneen[2 [0000−0001−9977−525X]], Radu Nicolescu[2 [0000−0003−2498−1002]], and Luděk Cienciala[1 [0000−0001−9473−5945]]

[1] Institute of Computer Science, Silesian University in Opava, Czech Republic
Research Institute of the IT4Innovations Centre of Excellence,
Silesian University in Opava, Czech Republic
lucie.ciencialova@fpf.slu.cz
[2] School of Computer Science, University of Auckland,
Private Bag 92019, Auckland 1142, New Zealand
mjd@cs.auckland.ac.nz, r.nicolescu@auckland.ac.nz

Abstract. P systems with compound terms (cP systems) have been proposed by Radu Nicolescu in 2018. These expressive cP systems have been used to solve well known NP-complete problems efficiently such as the Hamiltonian path, traveling salesman, 3-coloring, and problems in software verification. In this paper, we use cP systems to give an efficient parallel solution to the integer-valued Quadratic Unconstrained Boolean Optimization (QUBO) problem.

Keywords: Membrane computing, cP Systems, adiabatic quantum computing, QUBO

1 Introduction

One of the first models for membrane computing was the P systems; this model was first proposed by Gheorghe Păun in [15]. Many variants and extensions of P systems have since been developed; see for example [16]. We note that, for many of the later P system models, the focus has been more on computation (parallel and distributed) rather than being primarily motivated by biological computation. A recent version of P systems with compound terms (cP systems) [14] has been proposed by Radu Nicolescu. These expressive cP systems have been used to solve well known NP-complete problems efficiently such as the Hamiltonian path, traveling salesman [4], 3-coloring [3], and problems in software verification [11]. This powerful framework can also solve PSPACE-hard problems such as TQBF in linear time [7].

In this paper, we use cP systems to give an efficient parallel solution to the integer-valued Quadratic Unconstrained Boolean Optimization (QUBO) problem. These QUBO instances are generated from many combinatorial NP-hard optimization problems, which is a fundamental framework for adiabatic quantum computations [13,17]. Here these quantum computers, such as those produced by D-Wave Systems, are designed to solve explicitly one type of hard optimization problem (e.g. the QUBO problem). To solve other NP-hard problems, one does the traditional polynomial-time reduction to QUBO on a classical computer, then attempt to solve the QUBO on these adiabatic quantum machines, then post-process the quantum output on a classical machine to get a solution to the original NP-hard problem.

The first development of simple polynomial-time reductions from well-known NP-hard problems to QUBO (equivalently Ising) was given by Lucas in [12]. Later many other problems such as dominating set [5], sub/graph isomorphism [2,8], bounded Steiner tree [10], broadcast-time [1] were developed. The focus recently has been on reducing the input sizes when mapped to QUBO so as to be able to run on the currently sized quantum annealers, such as for MAX-3SAT [6] and k-densest common subgraph [9].

For the remaining part of this paper, we first describe cP systems in Section 2, then address how to represent negative numbers with complex P system objects in Section 3, formally define the QUBO problem in Section 4, and then show how to solve them efficiently in Section 5. Finally, we end with some final comments in Section 6.

2 Introduction to cP systems

A cP system is a kind of P systems that is formed from separated cells called top-level cells and their sub-cell systems. The rules are associated only with top-level cells, cells inside top-level cells are used as structured storage. In this paper, we use a simplified version of cP system with one top-level cell only.

Let us focus on sub-cells and the objects inside them. The elementary object is called term. It can be contained directly inside the top-level cell or inside a sub-cell. These sub-cells are called compound terms and they can contain both terms and compound terms. The name of a compound term is a label of the sub-cell. Terms and compound terms are written using lower-case letters, typically from the Latin alphabet, numbers or special symbols, e.g. $a, b, 1$ for terms and $f(), g(), +()$ for compound terms. As it is common in membrane systems literature, we use upper indices for the multiplicity appearance of an object, e.g. a^3b^2 and λ for the empty term – empty multiset. These terms and compound terms are called ground terms. For better understanding relation between the cell system and compound terms see Figure 1.

The rules associated with top-level cell can manipulate with terms and compound terms too. They are designed as rewriting rules of the form

$$\textit{current-state lhs} \rightarrow \textit{ target-state rhs}$$

where *lhs* and *rhs* are left-hand side and right-hand side of the rule. States of the top-level cell are objects - terms that are involved in the system only for the purpose of determining the state of the cell. If a rule application do not depend on a state of the cell the state can be omitted from both sides of rule.

The speciality of the rules used in cP systems is use of variables. They are usually labelled by uppercase letters, e.g. A, B, C. The variables can be replaced by any multiset of objects - terms. Anonymous (discarded) variables are denoted by an underscore(_) in cP systems.

The rules used in *lhs* and *rhs* can contain terms, compound terms with variables. For example $+(aXY^2)$ can be part of a rule. Before application of the rule with variables, all symbols which appear in rules can be matched against ground terms, using a one-way first-order syntactic unification (pattern matching). A term can only match another copy of itself, but a variable can match any multiset of ground terms (including λ). This may create combinatorial non-determinism, when a combination of two or more

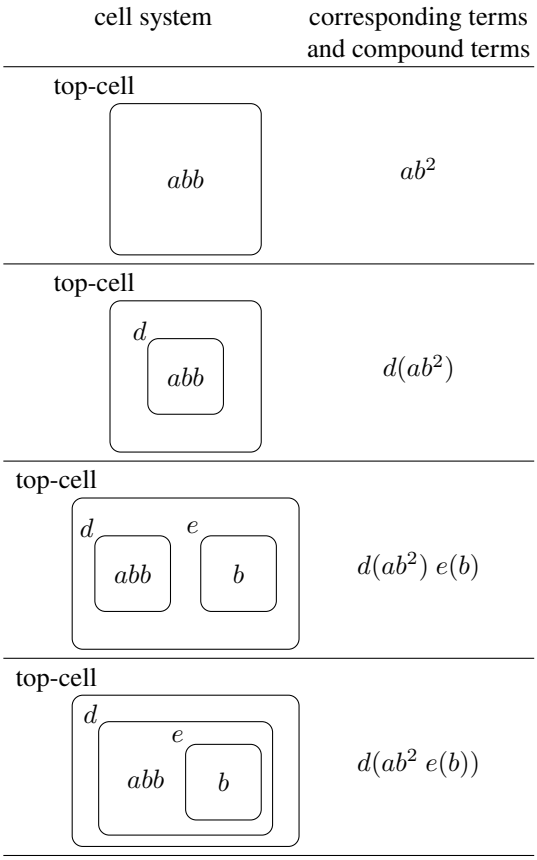


Fig. 1: Relation between cells and terms.

variables are matched against the same multiset, in which case an arbitrary matching is chosen. For example if *lhs* $= +(aX)Y^2$ and inside top-level cell there is compound term $+(a^2c)b^2$ there is only one matching $X = ac,\ Y = b$. If *lhs* $= +(XY)$ and inside top-level cell there is compound term $+(ab)$ there are four sets of matching $X = a,\ Y = b;\ X = b,\ Y = a;\ X = \lambda,\ Y = ab;\ X = ab,\ Y = \lambda$.

The *rhs* can contain promoters or (and) inhibitors. Promoters are objects that must be present within the top-level cell for the rule to be applicable but are not removed by the rule. Inhibitors are objects that must not be present for the rule to be applicable. If promoters are present, they are denoted following a | per promoter, and inhibitors by $\neg$. To define additional useful matchings expressively, promoters and inhibitors may also use virtual "equality" terms, written in infix format, with the = operator. For example, including the term $(ab = X)$ indicates that in valid matching X equals to ab.

So, the rule with variables can be unified in more than one way. Such rule can be seen as template for applicable rules that are made by matching.

There are two modes of application of rules: *min* and *max*. If rule is applied in *min* mode, system non-deterministically selects and applies one of applicable rules arise from matching of this rule. If rule applies in *max* mode all these applicable rules are executed. The mode of application is associated with every rule and it is stated as sub-index of $\rightarrow$ ($\rightarrow_{\min}$ and $\rightarrow_{\max}$).

Formally, a single-cell cP system is a construct

$$\Pi = (T, A, O, R, S, \overline{s}), \text{ where}$$

T is the set of top-level cells at the start of the evolution of the system; A is the alphabet of the system; O is the set of multisets of initial objects in the top-level cells; R is the set of rule-sets for each top-level cell; S is the set of possible states of the top-level cells; and $\overline{s} \in S$ is the starting state of every top-level cell in the system.

3 Representing and operating on integers

When solving discrete numerical problems, we encounter the problem of representing negative integers in terms of cP systems.

In cP systems, as in other membrane systems, to represent natural numbers, a multiset of identical objects whose number is just that number is used. For example, the number 4 can be represented as four occurrences of the object 1, and the absence of such an object can represent the number 0.

Consider a representation where each integer consists of two components, a positive and a negative, just as a complex number consists of a real and an imaginary component. For every integer a there is

$$i(x, y), \text{ where } x, y \in \mathbb{N}_0 \Longleftrightarrow a = x - y$$

For example:

$$i(1,4) \Longleftrightarrow -3 = 1 - 4$$

Note that there are multiple such representations for each integer. We consider a number representation in canonical form if at least one of its components is equal to zero.

$$i(5,8) \Longleftrightarrow -3 = 5 - 8$$
$$i(8,11) \Longleftrightarrow -3 = 8 - 11$$
$$i(0,3) \Longleftrightarrow -3 = 0 - 3$$

Every representation $i(x,y)$ can be converted into canonical form:

$$i(x,y) \sim \begin{cases} i(x',0) \text{ for } x \geq y \text{ where } x' = x - y \\ i(0,y') \text{ for } x < y \text{ where } y' = y - x \end{cases}$$

We can also define operations that are similar to those with integers such as addition, subtraction and multiplication. Division is not given because of the the set of integers is not closed under division.

$$i(x,y) + i(x',y') = i(x + x', y + y')$$

$i(x,y) + i(x',y') \Longleftrightarrow (x-y) + (x'-y') = x + x' - y - y' = (x+x') - (y+y') \Longleftrightarrow i(x + x', y + y')$

$$i(x,y) - i(x',y') = i(x + y', y + x')$$

$i(x,y) - i(x',y') \Longleftrightarrow (x-y) - (x'-y') = x - x' - y + y' = (x+y') - (y+x') \Longleftrightarrow i(x + y', y + x')$

$$i(x,y) \cdot i(x',y') = i(x \cdot x' + x' \cdot y', y \cdot x' + x \cdot y')$$

$i(x,y) \cdot i(x',y') \Longleftrightarrow (x-y) \cdot (x'-y') = x \cdot x' - x \cdot y' - y \cdot x' + y \cdot y' = (x \cdot x' + y \cdot y') - (x \cdot y' + y \cdot x') \Longleftrightarrow i(x \cdot x' + x' \cdot y', y \cdot x' + x \cdot y')$

For simulation of QUBO we do not need to use multiplication, only addition and subtraction is needed.

3.1 Counting with integers in cP systems

In cP systems, the number $i(x,y)$ can be seen as a cell containing two sub-cells labeled by $+$ and $-$, $i\,(+(x)\,-(y))$. For example

$$i(3,2) \rightsquigarrow i\,(+(111)\,-(11)) = i\,(+(3)\,-(2))$$
$$i(1,4) \rightsquigarrow i\,(+(1)\,-(1111)) = i\,(+(1)\,-(4))$$
$$i(2,0) \rightsquigarrow i\,(+(11)\,-()) = i\,(+(2)\,-())$$
$$i(0,0) \rightsquigarrow i\,(+()\,-())$$

In cP systems, one rule is sufficient to implement the addition and subtraction of two integers:

$$\longrightarrow_{\min} k\,(+(AC)\,-(BD))\;\mid i\,(+(A)\,-(B))\;\; j\,(+(C)\,-(D))$$
$$\longrightarrow_{\min} k\,(+(AD)\,-(BC))\;\mid i\,(+(A)\,-(B))\;\; j\,(+(C)\,-(D))$$

4 QUBO

Motivated recently by the emerging popularity of adiabatic quantum computing, QUBO (Quadratic Unconstrained Binary Optimisation) is an NP-hard mathematical optimization problem. For this paper, we are interested in the integer version of the problem of minimizing a quadratic objective function

$$x^* = \min_{\vec{x}} \vec{x}^T Q \vec{x}$$

where:

- $n \in \mathbb{N}_0$ - the number of variables in $\vec{x}$
- $i, j \in \mathbb{N}_0$
- $\vec{x}$ is a n-vector of binary (Boolean) variables $x_i \in \{0; 1\}, 0 \leq i \leq n-1$
- Q is an upper-triangular $n \times n$ matrix where $q_{i,j} \in \mathbb{Z}, 0 \leq i \leq j \leq n-1$ are possibly non-zero coefficients

Formally, QUBO problems are of the form:

$$x^* = \min_{\vec{x}} \sum_{i \leq j} x_i q_{i,j} x_j, \quad \text{where } x_i, x_j \in \{0, 1\}$$

This restriction on the matrix Q to integer values does not limit the power of the QUBO problem, as most reductions from other NP-hard problems to it maps to this form. (See, for example, [12,5,1].)

5 cP system QUBO solver

We now develop a cP system that finds minimal value of a QUBO in three phases of computation.

1. In the first phase, all possible values assignment is generated.
2. The second phase is devoted to generating of all polynomials.
3. In the third phase, related coefficients are added together to evaluate potential solutions for the assignments produced from phases 1 and 2.

We write number $z \in \mathbb{N}$ instead of 1^z inside elemental cells.
Input:

– For every variable x_i storing value $y_i \in \{0, 1\}$ there is complex object

$$a\,(in(i)val(y_i'))\ \text{ where } y_i' \in \{\lambda, 1\}.$$

– For every coefficient $q_{i,j}$ there is complex object

$$q\,(in1(i)in2(j)val(+(x)\,-(y)))\ \text{ where } q_{i,j} = x - y.$$

– Two counters (counter-like objects): $C_1(\lambda), C_2(n)$.
– Empty list (complex object) of values of variables: $l(C_1(\lambda))$ with counter $C_1(\lambda)$ inside.

5.1 1st phase

The first phase aims to generate all possible combinations of values that variables $x_0, \ldots x_{n-1}$ can be set to.

$$S_1 \quad C_2(1X) \longrightarrow_{\min} S_2 \quad v(\lambda)v(1)C_2(X) \tag{1}$$

By the execution of the rule (1), two complex objects - $v(\lambda)$ and $v(1)$ are generated and the number of 1s inside $C_2()$ is decreased by one.

$$\begin{aligned} S_2 \ \longrightarrow_{\max} S_1\ l\,(a\,(in(X)val(Y))\,C_1(X1)Z) \mid l\,(C_1(X)Z) \\ \mid v(Y) \end{aligned} \tag{2}$$

For every combination of X, Z and Y there can be one unified rule. Because X is the same for all objects in cP system at current step of computation, Z is unique for every object $l()$, and there are two possible evaluations for Y there is $1 \times |l()| \times 2$ rules that can be executed in parallel.

$$S_2\ l(\_) \longrightarrow_{\max} S_1 \tag{3}$$
$$S_2\ v(\_) \longrightarrow_{\max} S_1 \tag{4}$$

The rules (3) and (4) can erase all objects $l()$ and $v()$ inside the cell.

S_2 : $\quad l(C_1(\lambda))C_2(n-1) \quad v(\lambda)v(1)$ example configuration

$$
\begin{aligned}
&(2)_1 \qquad X = \lambda, Y = \lambda, Z = \lambda \\
&S_2 \qquad\quad \longrightarrow_{\max} S_1\ l\,(a\,(in(\lambda)val(\lambda))\,C_1(1)) \mid l\,(C_1(\lambda)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid v(\lambda)
\end{aligned}
$$

$$
\begin{aligned}
&(2)_2 \qquad X = \lambda, Y = 1, Z = \lambda \\
&S_2 \qquad\quad \longrightarrow_{\max} S_1\ l\,(a\,(in(\lambda)val(1))\,C_1(1)) \mid l\,(C_1(\lambda)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid v(1)
\end{aligned}
$$

$$
\begin{aligned}
&(3) \\
&S_2\ l\,(C_1(\lambda)) \longrightarrow_{\max} S_1
\end{aligned}
$$

$$
\begin{aligned}
&(4)_1 \\
&S_2\ v(\lambda) \longrightarrow_{\max} S_1
\end{aligned}
$$

$$
\begin{aligned}
&(4)_2 \\
&S_2\ v(1) \longrightarrow_{\max} S_1
\end{aligned}
$$

$$
S_1: \quad l\,(a\,(in(\lambda)val(\lambda))\,C_1(1))\ l\,(a\,(in(\lambda)val(1))\,C_1(1))\,C_2(n-1)
$$

When the number stored in the counter C_2 is reduced to zero, rule (1) is no longer applicable. The top-level cell contains 2^n complex objects $l()$ having different combinations of variable values. To move to the next stage, we need to restore the value of counter C_2 and insert a component $m()$ into the objects $l()$ for use in the next stage. The object $l'()$ stores the values of variables.

$$
\begin{aligned}
&S_1\ C_2() \qquad\quad \longrightarrow_{\min} S_2'\ C_2(X1) \qquad\quad \mid l\,(ZC_1(X)) \qquad (5) \\
&S_2'\ l\,(ZC_1(X)) \longrightarrow_{\max} S_3\ l\,(ZC_1()m()l'(Z)) \qquad (6)
\end{aligned}
$$

5.2 2nd phase

The idea of the second phase is to generate objects $p()$, which will contain representatives of quadratic elements that will be multiplied by the coefficients of one (say the i-th) row of the matrix Q. However, if the value of the variable x_i is zero, the generation of the row is omitted since its value will be zero.

Since Q is upper-triangular, the i-th row has at most $n-i$ non-zero coefficients that will be multiplied by quadratic elements consisting of x_i and x_j, where $n-1 \geq j \geq i \geq 0$.

For example: Let $\vec{x} = (1, 1, 1, 1)$. After the second phase, the complex object $m()$ will contain the following objects:

$$
\begin{array}{llllll}
p(w(\lambda) & a(in(\lambda)val(1)) & a(in(1)val(1)) & a(in(11)val(1)) & a(in(111)val(1))) \\
p(w(1) & & a(in(1)val(1)) & a(in(11)val(1)) & a(in(111)val(1))) \\
p(w(11) & & & a(in(11)val(1)) & a(in(111)val(1))) \\
p(w(111) & & & & a(in(111)val(1)))
\end{array}
$$

When there is some value zero in the vector $\vec{x}$ the corresponding $p()$ is present in $m()$ in “an empty" form. For example, if $\vec{x} = (1, 0, 1, 0)$:

$$
\begin{array}{llllll}
p(w(\lambda) & a(in(\lambda)val(1)) & a(in(1)val(0)) & a(in(11)val(1)) & a(in(111)val(0))) \\
p(w(1)) & & & & \\
p(w(11) & & & a(in(11)val(1)) & a(in(111)val(0))) \\
p(w(111) & & & &
\end{array}
$$

$$
\begin{aligned}
&S_3\ l\,(m\,(Z')\,a\,(in(X)val(1))\,Zl'(Y)C_1(X)) \longrightarrow_{\max} S_3' \\
&\qquad\qquad l\,(m\,(p\,(w(X)a\,(in(X)val(1))\,Z)\,Z')\,Zl'(Y)C_1(X)) \qquad (7) \\
&S_3\ l\,(m\,(p\,(w(X))\,Z')\,a\,(in(X)val(\lambda))\,Zl'(Y)C_1(X)) \longrightarrow_{\max} S_3' \\
&\qquad\qquad l\,(m\,(Z')\,Zl'(Y)C_1(X)) \qquad (8)
\end{aligned}
$$

If the top-level cell is in state S_3, exactly one rule is applied to each object $l()$, either rule (7) or rule (8). In the state S_3', rule (9) is executed in max mode, which causes the counter C_1 to increase inside all objects $l()$. After that, rule (10) is applied to decrease the value of counter C_2 by one.

$$
\begin{aligned}
&S_3'\ l\,(C_1(X)Z) \longrightarrow_{\max} S_3''\ l\,(C_1(1X)Z) \qquad (9) \\
&S_3''\ C_2(1X) \longrightarrow_{\min} S_3\ C_2(X) \qquad (10)
\end{aligned}
$$

If the counter C_2 is empty, all objects $p()$ have been generated, and it is time to proceed to the next stage. First, using rule (11), we delete the counter, and the top-left cell goes to state S_4. Then we delete the unnecessary objects from all objects $l()$ as well using rule (12).

$$
\begin{aligned}
&S_3\ C_2() \longrightarrow_{\min} S_4 \qquad (11) \\
&S_4\ l\,(m(X)l'(Z)C_1(Y)) \longrightarrow_{\max} S_4\ l\,(X\ r(+(\lambda)\ -(\lambda))\ l'(Z)C_1(Y)) \qquad (12)
\end{aligned}
$$

5.3 3rd phase

In the third stage, for each combination of non-zero values of $x_i x_j$, we will add the value of the coefficient q_{ij} to the result in the object $l()$.

$$S_4\ l\,(r\,(+(U')\ -(V'))\,p\,(w(X)a\,(in(Y)val(1))\,Z)\,Z'C_1(X)) \\ \longrightarrow_{\max} S_4\ l\,(r\,(+(UU')\ -(VV'))\,p\,(w(X)Z)\,Z'C_1(X)) \\ \mid q\,(in1(X)in2(Y)val(+(U)\ -(V))) \quad (13)$$

$$S_4\ l\,(p\,(w(X)a\,(in(Y)val())\,Z)\,Z'C_1(X)) \\ \longrightarrow_{\max} S_4\ l\,(p\,(w(X)Z)\,Z'C_1(X)) \\ \mid q\,(in1(X)in2(Y)val(Z'')) \quad (14)$$

When an object $p()$ with a leading variable x_i no longer contains any object $a()$, i.e. a complex object $l()$ contains an object $p(w(i))$ and an inner counter $C_1(i)$, the internal counter needs to be decreased. If counter $C_1()$ is already zero, the top-level cell changes its state to S_5.

$$S_4\ l(p(w(1X))ZC_1(1X)) \longrightarrow_{\max} S_4\ l(ZC_1(X)) \quad (15)$$

$$S_4\ l(p(w())Zl'(Y)C_1()) \longrightarrow_{\max} S_5\ l(Zl'(Y)) \quad \neg l(C_1(1X)\_) \quad (16)$$

Normalisation:

$$S_5\ l(r(+(XY)\ -(Y))\_) \longrightarrow_{\max} S_6\ l(r(+(X)\ -(\lambda))\_) \quad (17)$$

$$S_5\ l(r(+(X)\ -(XY))\_) \longrightarrow_{\max} S_6\ l(r(+(\lambda)\ -(Y))\_) \quad (18)$$

If there is at least one negative integer in $r()$s we will search for maximum of negative part of $r()$. If there are only positive integers (including zero) we will search for the minimum of the positive part of $r()$.

$$S_6\ l(r(+(\_)\ -(1Y))\_) \longrightarrow_{\min} S_7\ l(r(+(\_)\ -(1Y))\_) \quad (19)$$

$$S_6\ l(r(+(X)\ -(\lambda))\_) \longrightarrow_{\min} S_8\ l(r(+(X)\ -(\lambda))\_) \\ \neg l(r(+(\_)\ -(1Y))\_) \quad (20)$$

When the top-level cell is in the state S_7, there is at least one negative result stored in $r()$. Then we need to find maximum of negative part of $r()$.

$$S_7 \longrightarrow_{\min} S_F\ res(val(+(\lambda)\ -(X))Z) \mid l(r(+(\lambda)\ -(X))Z) \\ \neg l(r(+(\lambda)\ -(X1Y))\_) \qquad (21)$$

To find a minimum of positive and zero values we need to add one to the content of $+()$ so the value of each $r()$ is at least one. Then we find a minimum of positive parts of $r()$s.

$$S_8\ l(r(+(X)\ -(\lambda))Z) \longrightarrow_{\max} S_9\ \ l(r(+(x1)\ -(\lambda))Z) \qquad (22)$$

$$S_9 \longrightarrow_{\min} S_F\ res(val(+(X)\ -(\lambda))Z) \\ \mid l(r(+(1X)\ -(\lambda))Z) \\ \neg(X = YW)\ l(r(+(Y)\ -(\lambda))\_) \qquad (23)$$

The result of computation is complex object that contains variable values in sub-cell $res()$.

6 Conclusions

In this paper we showed how integers can be represented and used in cP systems. The second part of the paper was devoted to cP systems that can give an efficient parallel solution to the integer-valued Quadratic Unconstrained Boolean Optimization (QUBO) problem.

Acknowledgments

We thank James Cooper and Yezhou Liu for their helpful discussions.

Research is partially supported by the Silesian University in Opava under the Student Funding Plan, project SGS/11/2023

References

1. Calude, C.S., Dinneen, M.J.: Solving the broadcast time problem using a D-Wave quantum computer. In: Advances in Unconventional Computing, pp. 439–453. Springer (2017)
2. Calude, C.S., Dinneen, M.J., Hua, R.: Quantum solutions for densest k-subgraph problems. Journal of Membrane Computing **2**(1), 26–41 (2020). https://doi.org/10.1007/s41965-019-00030-1
3. Cooper, J., Nicolescu, R.: Alternative representations of P systems solutions to the graph colouring problem. Journal of Membrane Computing **1**(2), 112–126 (2019)
4. Cooper, J., Nicolescu, R.: The Hamiltonian cycle and travelling salesman problems in cP systems. Fundamenta Informaticae **164**(2-3), 157–180 (2019)

5. Dinneen, M.J., Hua, R.: Formulating graph covering problems for adiabatic quantum computers. In: Proceedings of the Australasian Computer Science Week Multiconference. pp. 18:1–18:10. ACSW '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3014812.3014830
6. Fowler, A.: Improved QUBO Formulations for D-Wave Quantum Computing. Master's thesis, University of Auckland (2017)
7. Henderson, A., Nicolescu, R., Dinneen, M.J.: Solving a PSPACE-complete problem with cP systems. Journal of Membrane Computing **2**(4), 311–322 (Dec 2020). https://doi.org/10.1007/s41965-020-00064-w
8. Hua, R., Dinneen, M.J.: Improved QUBO formulation of the graph isomorphism problem. SN Computer Science **1**(19), 1–18 (2020). https://doi.org/10.1007/s42979-019-0020-1
9. Huang, N.: A QUBO formulation for the k-densest common subgraph isomorphism problem via quantum annealing. In: 2020 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE). pp. 1–7 (2020). https://doi.org/10.1109/CSDE50874.2020.9411586
10. Liu, K., Dinneen, M.J.: Solving the bounded-depth Steiner tree problem using an adiabatic quantum computer. In: Proceedings of IEEE CSDE 2019, Melbourne, Australia (Dec 2019), https://researchspace.auckland.ac.nz/handle/2292/49490, http://ilab-australia.org/CSDE2019/
11. Liu, Y., Nicolescu, R., Sun, J.: Formal verification of cP systems using PAT3 and ProB. Journal of Membrane Computing **2**(2), 80–94 (2020)
12. Lucas, A.: Ising formulations of many NP problems. Frontiers in Physics **2**, 5 (2014)
13. McGeoch, C.C.: Adiabatic quantum computation and quantum annealing: Theory and practice. Synthesis Lectures on Quantum Computing **5**(2), 1–93 (2014)
14. Nicolescu, R., Henderson, A.: An introduction to cP Systems. In: Graciani, C., Riscos-Núñez, A., Păun, G., Rozenberg, G., Salomaa, A. (eds.) Enjoying Natural Computing: Essays Dedicated to Mario de Jesús Pérez-Jiménez on the Occasion of His 70th Birthday. pp. 204–227. LNCS 11270, Springer (2018)
15. Păun, G.: Computing with membranes. Journal of Computer and System Sciences **61**(1), 108–143 (2000)
16. Sosík, P.: P systems attacking hard problems beyond NP: a survey. Journal of Membrane Computing **1**(3), 198–208 (2019). https://doi.org/10.1007/s41965-019-00017-y
17. Wikipedia contributors: D-Wave Systems — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=D-Wave_Systems (2022), [Online; accessed 15-August-2022]

Implementing Perceptrons by Means of Water Based Computing

Nicoló Civiero[1], Alec Henderson[2], Thomas Hinze[3], Radu Nicolescu[4], and Claudio Zandron[1]

[1] Dipartimento di Informatica, Sistemistica e Comunicazione (DISCo)
Università degli Studi di Milano-Bicocca
Viale Sarca 336, 20126 Milan, Italy {claudio.zandron}@unimib.it
[2] Australian Institute of Tropical Health and Medicine, James Cook University, Townsville, Australia alec.henderson@jcu.edu.au
[3] Friedrich Schiller University Jena, Department of Bioinformatics
Ernst-Abbe-Platz 2, D-07743 Jena, Germany thomas.hinze@uni-jena.de
[4] University of Auckland, School of Computer Science, Auckland, New Zealand
r.nicolescu@auckland.ac.nz

Abstract. Water based computing emerged as a branch of membrane computing in which water tanks act as permeable membranes connected via pipes. Valves residing at the pipes control the flow of water in terms of processing rules. Resulting water tank systems provide a promising platform for exploration and for case studies of information processing by flow of liquid media like water. We first discuss the possibility of realizing a single layer neural network using tanks and pipes systems. Moreover, we discuss the possibility to create a multi-layer neural network, which could be used to solve more complex problems. Two different solutions are considered: in a first solution, the weight values of the connections between the network nodes are represented by tanks. This means that the network diagram includes multiplication structures between the weight tanks and the input tanks. The second solution aims at simplifying the network proposed in the previous solution, by considering the possibility to modify the weight values associated to neuron by varying the diameter of the connecting pipes between the tanks. The multiplication structures are replaced with a timer that regulates the opening of the outlet valves of all the tanks. These two solutions can be compared to evaluate their efficiency, and considerations will be made regarding the simplicity of implementation.

Keywords: Membrane Systems, Water Based Computing, Neural Networks

1 Introduction

P systems, initially introduced by Gh. Păun in [13], are a computational model inspired by biological membranes that operate in a parallel and distributed manner. These systems are characterized by their decentralized nature and their evolution is based on the content of interconnected membranes. Numerous variants of P systems have been proposed and extensively studied, including P systems with active membranes [14,16], spiking neural P systems [6,8], and tissue P systems [7,11].

Recent research efforts have focused on simulating P systems on mainstream hardware [1,17], formal verification techniques [9,10], or employing more visual approaches like [2].

Another recently introduced idea concerns membrane water computing, introduced in [3] and driven by the goal of obtaining a parallel computing system without any central control. In such a system, the flow of water is solely regulated by local measurements of tank filling levels in a finite number of water tanks, each capable of holding an initial volume of water and storing or collecting water up to a maximum capacity. The water tank system can be viewed as a membrane system: the water tanks can be seen as membranes permeable to inflow and/or outflow of water molecules, whose presence is dynamically regulated by local measurements (interaction rules).

The volume of water contained in a tank serves as both data carrier and a medium for data processing, achieved by manipulating the volume over time. Tanks are interconnected using pipes, which allow the directed transfer of water from one tank to another when opened. The pipes can be equipped with one or more valves, which can be configured in different ways. A valve has two states: "fully open" or "fully closed", and it is determined by monitoring the filling level in a specific tank. When the level of water exceeds a predetermined threshold or indicates a nearly empty tank, the valve fully opens or remains closed at the hosting pipe during the ongoing time step, respectively. A pipe will transport water only if all its valves are fully opened and the supplying tank has water available. The entrance of a pipe can be positioned at any desired filling level in its supply tank, requiring a minimum amount of water in the tank before the pipe can be filled.

Water tank systems provide a promising platform for exploration and for case studies of information processing based on the controlled flow of liquid media like water. This concept gives a strong motivation and substantiates the significance of further work in detail for application scenarios.

A water tank system can operate in either analog or binary mode. In analog mode, the volume of water within a tank represents a non-zero natural number. To facilitate this, we introduce water tank systems for arithmetic operations such as addition, non-negative subtraction, division, and multiplication. These systems can be assembled to perform sequenced or nested computations. Furthermore, a ring oscillator, consisting of a cyclic structure with at least three water tanks, emulates a clock signal. In binary mode, an empty or nearly empty water tank corresponds to the logical value "0," while a full or nearly full tank corresponds to "1," with latencies during the filling or emptying process.

The obtained systems operate autonomously in a decentralized manner, simply relying on local measurements of filling levels. We stress the fact that, since a water tank can be viewed as a membrane that allows the inflow and/or outflow of water molecules, dynamically regulated by local measurements, such an approach is closely related to tissue membrane systems.

In the original paper [3], authors define basic logic gates such as OR, AND, and a bit duplicator for water-based logic operations. These logic gates can be connected to form Boolean circuits with the ability of inherent self-synchronization, eliminating the need for external control.

In this paper, we first discuss the possibility of realizing a single layer neural network using tanks and pipes systems, through which water flows. Moreover, we discuss the

possibility to create a multi-layer neural network, which could be used to solve more complex problems. We stress the fact that one advantage of such an implementation lies in the possibility to adopt it for explaining the functioning of Neural Network at different levels, to students or even more general audience, clearly illustrating the basic principles behind a Neural Network. In fact, the flow and the containment of water are easily visible by human senses, and the process of learning can be directly observed in details.

Two different solutions are considered: in a first solution, the weight values of the connections between the network nodes are obtained by using specific tanks. The second solution aims at simplifying the network proposed in the previous solution, by considering the possibility to modify the weight values associated to each neuron by varying the diameter of the connecting pipes between the tanks.

The paper is organized as follows. In Section 2 we recall some definitions related to water based computing, and we recall some basic multiplication schemes realized by means of water tanks, which will be used in the rest of the paper. In Section 3 two different implementations of the basic perceptron are presented. In Section 4 an implementation of the multilayer-perceptron and the description of three different activation functions are discussed. In Section 5 we draw some conclusions and give some directions for future investigations.

2 Basic Definitions

In this section, we shortly recall some definitions that will be useful while reading the rest of the paper. For a complete introduction to P systems, we refer the reader to *The Oxford Handbook of Membrane Computing* [15].

A *water tank system* represents a special type of membrane systems in which a single membrane is described by a *water tank* able to store an amount of water up to its predefined finite capacity. The communication between membranes has been managed by *pipes* that enable a controllable flow of water from one tank to another one. Communication rules appear by definition of *valves*. Here, each pipe can be equipped with an arbitrary number of valves. By default, a valve fully closes its hosting pipe. A valve either fully opens or remains closed its hosting pipe based on measurements iterated in discrete time steps. For instance, a valve opens if and only if the filling level in a specific water tank exceeds a certain threshold, otherwise it closes. If the condition for an open valve is not fulfilled any more, it closes at the end of the ongoing time step. Water gets transported via a pipe if and only if all residing valves are opened and the supply tank contains water.

The first formal definition of a water tank system was given in [3]. Later, a more simplified version was published in [4,5]. In order to cope with the needs for emulation of perceptrons, the modelling framework for water tank systems undergoes a further stage of extension by additional types of valves and by additional parameters for specification of pipes.

Formally, a water tank system is a construct

$$\Pi = (W, A, \tau, E, r, P, v_0, s_0, \Delta t) \tag{1}$$

with its components:

- W is a finite and non-empty set of tank identifiers (water tanks).
- A is a finite and non-empty set of valve identifiers (actuators).
- $\tau : W \longrightarrow \mathbb{R}_+ \cup \{\infty\}$ is a function assigning a capacity to each tank (tank capacity). $\mathbb{R}_+$ stands for the set of positive rational numbers. The capacity defines the maximum volume of water a tank can store. Excessive water is removed from a tank by overflow drain. Please note that tanks with an infinite capacity are allowed to act as a reservoir.
- $E \subset W \times \{<, =, \leq, >, \geq, \neq\} \times \mathbb{R}$ specifies a finite set of decision rules resulting from measurements (evaluation). A measurement reveals the current volume of water in a tank from W. We assume that each measurement returns a non-negative rational number in $\mathbb{R}$ including zero. An element $e \in E$ stands for a comparison by means of a relational operator. This comparison is carried out what finally implies an underlying decision by answering "true" or "false", respectively.
- $r : A \longrightarrow E$ defines a mapping that assigns a decision rule to each valve. Hence, each valve comes with a dedicated behaviour (reaction).
- $P \subset W \times W \times \mathcal{P}(A)$ symbolises a finite set of pipes in which each pipe starts at a tank from W, ends at a tank from W and hosts none, one, or several valves. These valves have been given by an element from the power set $\mathcal{P}(A)$.
- $v_0 : W \longrightarrow \mathbb{R} \cup \{\infty\}$ specifies the initial volume of water for each water tank in W. For all water tanks $w \in W$, it holds $v_0(w) \leq \tau(w)$.
- $s_0 : P \longrightarrow \mathbb{R}_+$. This function assigns an initial diameter (size) to each pipe. Diameters are expressed by rational numbers greater than zero.
- $\Delta t \in \mathbb{R}_+$ defines the duration of a discrete time step given by a constant non-negative rational number.

A water tank system evolves in discrete constant time steps beginning with its initial configuration. The execution of a time step consists of a sequence of actions. Valves are closed by default. First, all measurements are done simultaneously in all involved tanks. Then, all decisions based on these measurements have been made. Next, the valves update their state according to the corresponding decision rules. In case a decision ends up with "true", the valve fully opens. Otherwise, the valve remains closed. Now, water can flow or not through the pipes. Each pipe whose valves are all fully opened transports a portion of water during the ongoing time step when supplied. In addition, the portion of water depends on the size of the pipe. As a consequence, the water volume of either related tanks needs to be updated (increased or decreased). Finally, the size of each pipe can be adapted (made smaller or larger). After all the aforementioned actions have been carried out, the processing within the current time step is finalised, all valves become closed again, and the subsequent time step might begin. The water tank system stops if the water volumes in all water tanks keep constant over two successive time steps indicating a final system's configuration.

We recall now the integer multiplication scheme, a copy of the one presented in [3]. In this scheme, the operation works as follows: a unit is subtracted from tank x at each iteration of the loop until the tank x becomes empty, while at each iteration, the value of y is added to the result tank. For implementation details and a detailed description of how it works, we refer the reader to [3].

For the sake of completeness, the schemes related to multiplication with rational values are also provided. However, it is important to keep in mind that multiplication

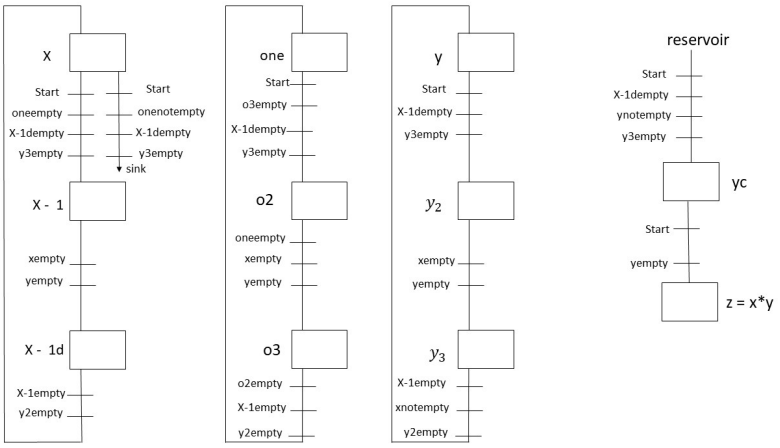


Fig. 1: Integer multiplication

involving rational numbers introduces approximations and lacks precision. For optimal outcomes, we have categorized the multiplication process into three cases, taking into account the input values (x, y).

1. x, y < 0.81
2. x, y < 1 and x or y > 0.8
3. x or y > 1

We depict below the schemes for case 1 (see fig. 2) and case 2 (see fig. 3); the subtraction schema can be found in [3].

The subtraction works as follows: the valves placed on the pipes connecting the tanks (x and y) to the sink are opened simultaneously. The contents of both tanks are drained at the same time until one of the tanks becomes empty. The result is then taken from tank x. From this description, it is evident that if x is less than y, then the result of the operation will be zero.

For the third case, the schema is the same as the multiplication with integers presented earlier. However, it is important to note that, for this operation, it is advisable to position the larger number in the tank labeled as x, to ensure a more accurate result. Conversely, if the numbers are swapped, the outcome may not be as precise; as an example, the operation 0.3 * 5.5 would give a result of 5.5.

3 Implementing Perceptrons through Water Based Computing

Artificial Neural Networks (or simply Neural Networks) are mathematical models composed of nodes (neurons) that are inspired by the functioning of the human brain, where interconnected neurons exchange information. A neural network is an "adaptive" system capable of modifying its structure (nodes, interconnections, and weights) based

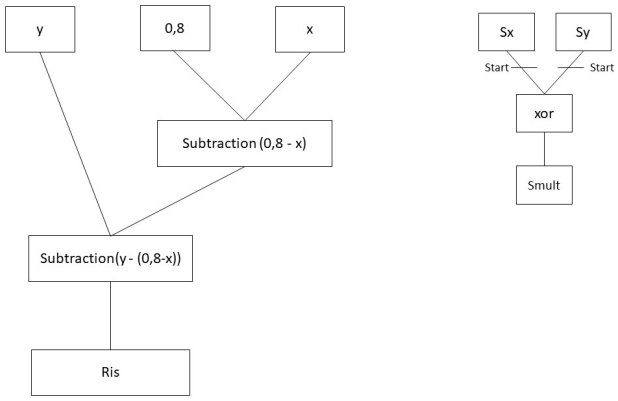


Fig. 2: rational values multiplication case 1

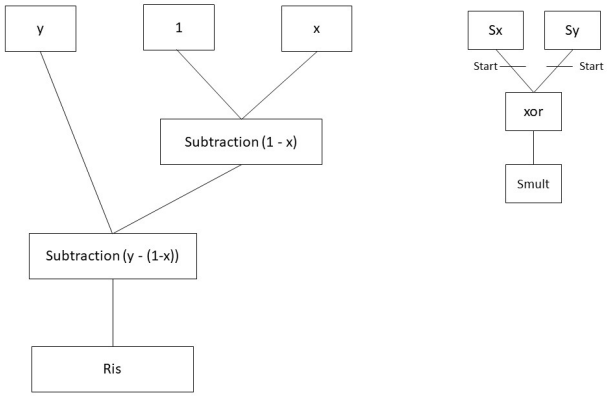


Fig. 3: rational values multiplication case 2

on both external data and internal information that connect and pass through the neural network during the learning phase.

The perceptron, introduced by McCulloch-Pitts in [12], is a machine learning algorithm used for supervised learning of binary classifiers. These classifiers are functions that determine whether an input, represented by a numerical vector, belongs to a particular class or not.

In this section, we propose an implementation of a perceptron and a simple multilayer perceptron by means of Water Based Computing. In particular, we will consider water based systems operating in analog mode: in this mode, the volume of water within a tank corresponds to a non-zero natural number. A XOR gate is used to track negative or

positive results: we assume that positive values encode to logical value ‘0’ and negative value to ‘1’. The XOR gate can be obtained as a modified version of the OR gate presented in [3], with an added valve in the result tank.

The water volume from the two input tanks is combined in a result tank. The input tanks have a maximum capacity corresponding to the logic level 1, while the result tank has two times this capacity. When both inputs tanks are 1, the result of the XOR gate must be 0. This issue can be resolved by using a simple valve added to the original OR gate, that opens when the content of the result tank reaches its full level. All the water is then drained, resulting in an empty result tank, corresponding to a 0. This logic gate will be used to control the sign of the operations of multiplication used to design perceptrons.

As a starting point, a simple version of a network with a single node of binary activation (0,1) was considered. This node is called a "simple perceptron" because it uses a simple step function as its activation function. The activation function of a node is a mathematical function that defines the output of the node after receiving the sum of weighted inputs.

We discuss two possible implementations of the network, referred to as solution 1 and solution 2. We stress the fact that the subtractions used in the schemes of the two presented solutions allow for negative results. The operation is similar to the non-negative subtraction schema presented in [3]. Water is simultaneously discharged from both tanks until one becomes empty. At this point, the remaining water in the other tank flows towards the result tank. Additionally, on the right side of the diagram, there is a control tank which indicates if the subtraction result is negative.

3.1 Solution 1

In the first proposed solution, the chosen approach is to multiply the input by the corresponding weight value. Formally, the system to implement the perceptron is defined as follows:

$$W = \{(w1), (x1), (w2), (x2), (b), (S+), (S-), (Res), (Sw1), (Sx1), (Sw2), (Sx2), (Sb), (Smultiplication1), (Smultiplication2), (reservoir)\}$$

$$A = \{(w1new), (w2new), (bnew), (sw1new), (sw2new), (sbnew), (e\,'), (Start), (SMult1e), (SMult1ne), (SMult2e), (SMult2ne), (Sbe), (Sbne), (b)\}$$

$$\tau = \{(w1, 2), (x1, 2), (w2, 2), (x2, 2), (b, 2), (S+, 10), (S-, 10), (Res, 2), (Sw1, 1), (Sx1, 1), (Sw2, 1), (Sx2, 1), (Sb, 1), (Smultiplication1, 1), (Smultiplication2, 1), (reservoir, \infty)\}$$

$$E = \{(w1new > 0), (w2new > 0), (bnew > 0), (sw1new > 0), (sw2new > 0), (sbnew > 0), (e\,' > 0), (Smultiplication1 > 0), (Smultiplication2 > 0), (Sb > 0), (b > 0)\}$$

r=

$$w1new = \begin{cases} 1 & \text{if } Vw1new > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$w2new = \begin{cases} 1 & \text{if } Vw2new > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$bnew = \begin{cases} 1 & \text{if } Vbnew > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$sw1new = \begin{cases} 1 & \text{if } Vsw1new > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$sw2new = \begin{cases} 1 & \text{if } Vsw2new > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$sbnew = \begin{cases} 1 & \text{if } Vsbnew > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$e\,' = \begin{cases} 1 & \text{if } Ve\,' > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$start = \begin{cases} 1 & \text{if } time > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$SMult1e = \begin{cases} 1 & \text{if } VSmultiplication1 = 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$SMult1ne = \begin{cases} 1 & \text{if } VSmultiplication1 > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$Smult2e = \begin{cases} 1 & \text{if } VSmultiplication2 = 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$Smult2ne = \begin{cases} 1 & \text{if } VSmultiplication2 > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$Sbe = \begin{cases} 1 & \text{if } VSb = 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$Sbne = \begin{cases} 1 & \text{if } VSb > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$b = \begin{cases} 1 & \text{if } Vb = 0 \\ 0 & \text{if } otherwise \end{cases}$$

Meaning that 1 marks the valve to be open and 0 closed, respectively.

$$\begin{aligned} P = \{&(reservoir, w1, \{w1new\}), (w1, Multiplication1, \{start\}), \\ &(x1, Multiplication1, \{start\}), (reservoir, w2, \{w2new\}), \\ &(w2, Multiplication2, \{start\}), (x2, Multiplication2, \{start\}), \\ &(reservoir, b, \{bnew\}), (Multiplication1, S+, \{SMult1e\}), \\ &(Multiplication1, S-, \{SMult1ne\}), (Multiplication2, S+, \{SMult2e\}), \\ &(Multiplication2, S-, \{SMult2ne\}), (b, S+, \{Sbe\}), \\ &(b, S-, \{Sbne\}), (S+, [(S+) - (S-)], \{\}), (S-, [(S+) - (S-)], \{\}), \\ &([(S+) - (S-)], Res, \{\}), (Res, sink, \{\}), (reservoir, Sw1, \{sw1new, e'\}), \\ &(reservoir, Sw2, \{sw2new, e'\}), (reservoir, Sb, \{sbnew, e'\}), \\ &(Sw1, xor, \{start\}), (Sx1, xor, \{start\}), (Sw2, xor, \{start\}), \\ &(Sx2, xor, \{start\}), (Sb, sink, \{start, b\}), (xor, Smultiplication1, \{\}), \\ &(xor, Smultiplication2, \{\})\} \end{aligned}$$

$$\begin{aligned} V0 = \{&(w1; 0), (x1; 0), (w2; 0), (x2; 0), (b; 0), (S+; 0), (S-; 0), (Res; 0), (Sw1; 0), \\ &(Sx1; 0), (Sw2; 0), (Sx2; 0), (Sb; 0), (Smultiplication1; 0), \\ &(Smultiplication2; 0)\} \end{aligned}$$

$$
\begin{aligned}
S0 = \{&(reservoir, w1, \{w1new\}, 0.1), (w1, Multiplication1, \{start\}, 0.1),\\
&(x1, Multiplication1, \{start\}, 0.1), (reservoir, w2, \{w2new\}, 0.1),\\
&(w2, Multiplication2, \{start\}, 0.1), (x2, Multiplication2, \{start\}, 0.1),\\
&(reservoir, b, \{bnew\}, 0.1), (Multiplication1, S+, \{SMult1e\}, 0.1),\\
&(Multiplication1, S-, \{SMult1ne\}, 0.1),\\
&(Multiplication2, S+, \{SMult2e\}, 0.1),\\
&(Multiplication2, S-, \{SMult2ne\}, 0.1), (b, S+, \{Sbe\}, 0.1),\\
&(b, S-, \{Sbne\}, 0.1), (S+, [(S+) - (S-)], \{\}, 0.1),\\
&(S-, [(S+) - (S-)], \{\}, 0.1), ([(S+) - (S-)], Res, \{\}, 0.1),\\
&(Res, sink, \{\}, 0.1)(reservoir, Sw1, \{sw1new, e'\}, 0.1),\\
&(reservoir, Sw2, \{sw2new, e'\}, 0.1), (reservoir, Sb, \{sbnew, e'\}, 0.1),\\
&(Sw1, xor, \{start\}, 0.1), (Sx1, xor, \{start\}, 0.1),\\
&(Sw2, xor, \{start\}, 0.1), (Sx2, xor, \{start\}, 0.1),\\
&(Sb, sink, \{start, b\}, 0.1), (xor, Smultiplication1, \{\}, 0.1),\\
&(xor, Smultiplication2, \{\}, 0.1)\}
\end{aligned}
$$

$$\Delta t = 1second$$

We have decided to set a maximum size of 2 for the input and weight tanks to keep the network dimensions limited. For the control tanks, the maximum value is set to one because the possible values of a control tank are either 1 or 0.

To represent the weight of the connection between the input and the node, a tank is used, and the content of the tank corresponds to the weight value (see fig. 4).

On the left side of the schema, the actual network is depicted, with tanks for the input (x) and tanks for the weights (W). In this hypothesis, these values are multiplied together (multiplication1, multiplication2) using the multiplication schema mentioned earlier. The result of the multiplication then flows into the tanks S (sum), while observing the value contained in the tanks Smult (multiplication sign). The tank (S+) contains positive values, while the tank (S-) contains negative values. The Bias value is added to one of these two tanks, depending on the value of the tank Sb (bias sign). The subtraction is then performed between the values contained in the tank (S+) and (S-) to obtain the result of the network, which is then passed through the activation function to obtain the network's output.

On the right side, there are control tanks used to calculate the signs of multiplication and bias. In particular, the XOR operation is used for the multiplication sign, and the values contained in the tanks Smult will be 0 for a positive sign and 1 for a negative sign. The FeedForward phase is then followed by the weight update phase (see fig. 5).

On the left side of the schema, the subtraction between the desired value (t) and the value obtained from the network (y) is performed first. Then, the multiplications between the result of the subtraction, the input, and the learning rate are carried out. On the right side of the schema, control tanks are present for the sign value of the multiplication

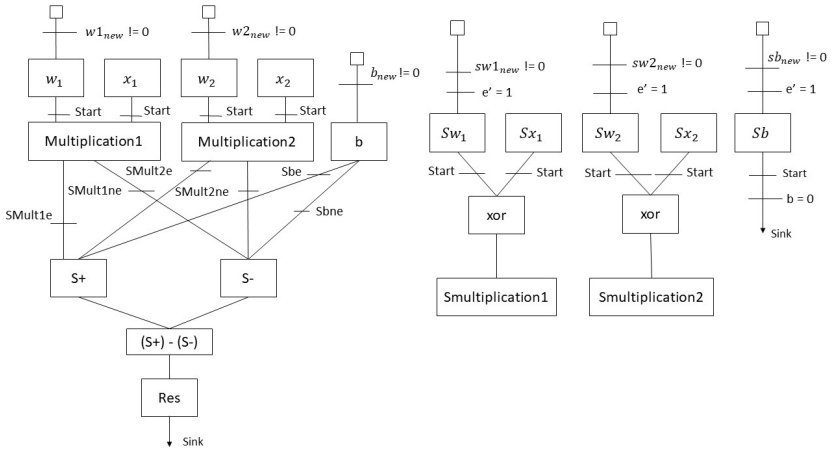


Fig. 4: Solution 1: Feed Forward phase

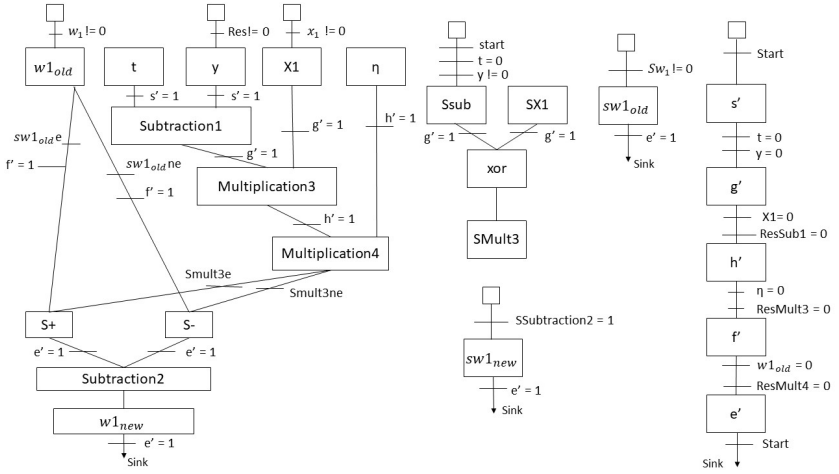


Fig. 5: Weight Update phase

(Smult3), sign value of old weight(Sw1old), and sign value of new weight(Sw1new). Finally, there is a column of control tanks used to adjust the valves of the input tanks to carry out the operations in the correct order.

3.2 Solution 2

In the second proposed solution, it has been chosen not to use tanks for the weights but instead variable-sized pipes that connect the tanks. Formally, the system to implement the perceptron is defined as follows:

$$W = \{(x1), (x2), (b), (S+), (S-), (Res), (Sw1), (Sx1), (Sw2), (Sx2), (Sb), (Smultiplication1), (Smultiplication2), (reservoir)\}$$

$$A = \{(bnew), (sw1new), (sw2new), (sbnew), (e\,'), (Start), (Smult1e), (Smult1ne), (Smult2e), (Smult2ne), (Sbe), (Sbne), (b), (v)\}$$

$$\tau = \{(x1, 2), (x2, 2), (b, 2), (S+, 10), (S-, 10), (Res, 2), (Sw1, 1), (Sx1, 1), (Sw2, 1), (Sx2, 1), (Sb, 1), (Smultiplication1, 1), (Smultiplication2, 1), (reservoir, \infty)\}$$

$$E = \{(bnew > 0), (sw1new > 0), (sw2new > 0), (sbnew > 0), (e\,' > 0), (Smultiplication1 > 0), (Smultiplication2 > 0), (Sb > 0), (b > 0), (v < n)\}$$

$$bnew = \begin{cases} 1 & \text{if } Vbnew > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$sw1new = \begin{cases} 1 & \text{if } Vsw1new > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$sw2new = \begin{cases} 1 & \text{if } Vsw2new > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$sbnew = \begin{cases} 1 & \text{if } Vsbnew > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$e\,' = \begin{cases} 1 & \text{if } e\,' > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$start = \begin{cases} 1 & \text{if } time > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$Smult1e = \begin{cases} 1 & \text{if } VSmultiplication1 = 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$Smult1ne = \begin{cases} 1 & \text{if } VSmultiplication1 > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$Smult2e = \begin{cases} 1 & \text{if } VSmultiplication2 = 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$Smult2ne = \begin{cases} 1 & \text{if } VSmultiplication2 > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$Sbe = \begin{cases} 1 & \text{if } VSb = 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$Sbne = \begin{cases} 1 & \text{if } VSb > 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$b = \begin{cases} 1 & \text{if } Vb = 0 \\ 0 & \text{if } otherwise \end{cases}$$

$$v = \begin{cases} 1 & \text{if } Vv < n \\ 0 & \text{if } otherwise \end{cases}$$

Meaning that 1 marks the valve to be open and 0 closed, respectively.

$$\begin{aligned} P = \{&(reservoir, b, \{bnew\}), (x1, S+, \{Smult1e, v\}), (x1, S-, \{Smult1ne, v\}), \\ &(x2, S+, \{Smult2e, v\}), (x2, S-, \{Smult2ne, v\}), (b, S+, \{Sbe, v\}), \\ &(b, S-, \{Sbne, v\}), (S+, [(S+) - (S-)], \{\}), (S-, [(S+) - (S-)], \{\}), \\ &([(S+) - (S-)], Res, \{\}), (Res, sink, \{\}), (reservoir, Sw1, \{sw1new, e'\}), \\ &(reservoir, Sw2, \{sw2new, e'\}), (reservoir, Sb, \{sbnew, e'\}), \\ &(Sw1, xor, \{start\}), (Sx1, xor, \{start\}), (Sw2, xor, \{start\}), \\ &(Sx2, xor, \{start\}), (Sb, sink, \{start, b\}), (xor, Smultiplication1, \{\}), \\ &(xor, Smultiplication2, \{\})\} \end{aligned}$$

$$\begin{aligned} V0 = \{&(x1; 0), (x2; 0), (b; 0), (S+; 0), (S-; 0), (Res; 0), (Sw1; 0), \\ &(Sx1; 0), (Sw2; 0), (Sx2; 0), (Sb; 0), (Smultiplication1; 0), \\ &(Smultiplication2; 0)\} \end{aligned}$$

$$\begin{aligned} S0 = \{&(reservoir, b, \{bnew\}, 0.1), (x1, S+, \{SMult1e, v\}, 0.1), \\ &(x1, S-, \{SMult1ne, v\}, 0.1), (x2, S+, \{SMult2e, v\}, 0.1), \\ &(x2, S-, \{SMult2ne, v\}, 0.1), (b, S+, \{Sbe, v\}, 0.1), \\ &(b, S-, \{Sbne, v\}, 0.1), (S+, [(S+) - (S-)], \{\}, 0.1), \\ &(S-, [(S+) - (S-)], \{\}, 0.1), ([(S+) - (S-)], Res, \{\}, 0.1), \\ &(Res, sink, \{\}, 0.1), (reservoir, Sw1, \{sw1new, e'\}, 0.1), \\ &(reservoir, Sw2, \{sw2new, e'\}, 0.1), (reservoir, Sb, \{sbnew, e'\}, 0.1), \\ &(Sw1, xor, \{start\}, 0.1), (Sx1, xor, \{start\}, 0.1), (Sw2, xor, \{start\}, 0.1), \\ &(Sx2, xor, \{start\}, 0.1), (Sb, sink, \{start, b\}, 0.1), \\ &(xor, Smultiplication1, \{\}, 0.1), (xor, Smultiplication2, \{\}, 0.1)\} \end{aligned}$$

$\Delta t = 1second$

In the previous hypothesis, the connecting pipes between the tanks had the same diameter and did not affect the amount of water flowing through the network. To allow the passage of a "weighted" amount of water, a timer is used to regulate the opening of valves present in each connecting pipe between the nodes. Assuming that each second, an amount of water equal to the diameter of the pipe passes through each pipe, the water flow between the tanks can be controlled. The idea is to simplify the network by reducing the number of tanks and eliminating the multiplications involving the inputs and weights. (see fig. 6)

To update the weights, we proceed as for the previous solution. In this case, however, the timer cannot be utilized to calculate the new weight, because we need the weight value in a tank in order to add it to the updated value.

To create a specific timer, the schema of Ring Oscillator presented in [3] is modified, by adding a valve between tank T1 and T2 (we have changed the name of the tanks to avoid confusion with the tanks representing the weights). The condition applied to the valve is $v \neq n$, which means that the valve remains open until the value n is reached in tank v (we also changed the name of tank y in v, to avoid confusion with tank y in the update weight phase). Additionally, in tank v the tube and valve that allow water drainage are removed. This way, in each iteration, it is ensured that tank v will receive a quantity of water equal to 1 ,ideally assuming each iteration is 1 second (see fig. 7).

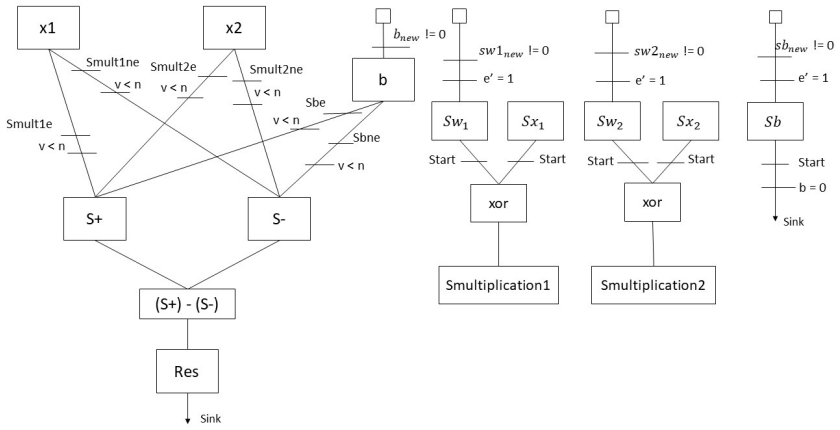


Fig. 6: Solution 2: Feed Forward phase

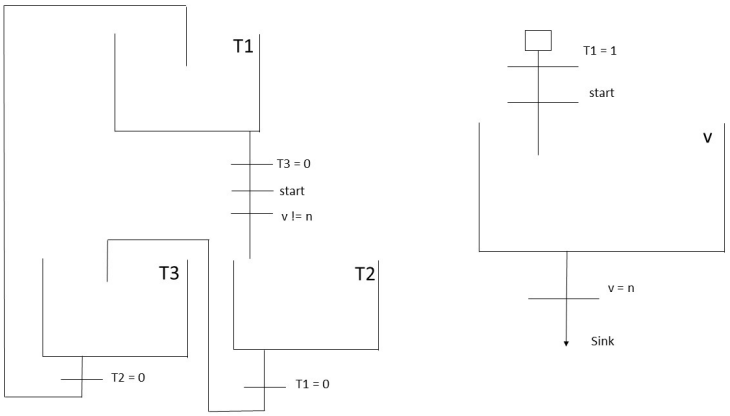


Fig. 7: Timer

4 Multilayer Perceptron

In multilayer Perceptron nodes have the same internal structure as for perceptron unit, but it is necessary to create new structure/schema for activation functions and backward propagation, and to modify the schema for updating weights.

Concerning activation functions, we have considered and implemented ReLu, Tanh, and Sigmoid.

ReLu ReLu activation function is the simplest, for positive value the result is the input, for negative value the result is zero. The implementation consists of a tank, and a control valve so if the input is negative this valve is open and the volume of water in the tank is zero. (For keeping the amount of water small, it is possible to set a max limit, to do that we use a pipe set to a height value, if volume of water is over the value the water goes into the sink).

Tanh Tanh activation function is more complex. We consider volume of water between 0 and 2. Over 2 we consider as the value is 2. We separate the input volume of water into 3 tanks: in the first and second tank the maximum value of water is 0.5, while in the third tank it is 1. To separate the input water, we consider a water cascade like the one presented in [3], fig. 2, and we operate on the volume of water of each tank in a different way. For the first tank we take all the volume of water, (for example if it is 0,4 the result of tanh is 0,4). For second and third tank we set an outflow pipe with a different diameter than 0,1 (default value), so for every second we have set a different outflow than 0,1, equal to the diameter of the pipe. This volume of water is drained into another tank that has two outflow pipes: one leads the water into the sink and the other into result tank, and these pipes have a different diameter than 0,1.

Sigmoid For sigmoid function we use the formula $(tanh(x/2) + 1)/2$.

We simply operate/compute the multiplication (x*0.5) on the input of tanh, add 1 and perform another multiplication on the result.

The activation functions just described can be compared with respect to three difference aspects:

- Accuracy with respect to the mathematical function they implement
- The time needed to provide a result
- Ease of implementation, i.e. the number of tanks and pipes required to implment them

Regarding ease of implementation, it appears that ReLU is the simplest, requiring only one tank and a valve, while tanh and sigmoid are more complex, needing numerous tanks and valves. This complexity affects the speed at which results are obtained. The more complex a system is, the more time it takes to complete operations and achieve a result. Regarding accuracy, ReLu stands out as the most accurate activation function, providing precise results without any significant approximation errors. Tanh, as defined, has a slight approximation error, while sigmoid suffers from significant errors due to the multiplication by the value 0.5. In conclusion, it becomes evident that ReLU outperforms both tanh and sigmoid functions in these three aspects.

4.1 Forward Propagation

Forward propagation phase is the same as for perceptron forward propagation, but in a feedforward neural network structure, a unit will receive information of several units belonging to the previous layer. We suppose to add all this information in one input tank for each node.

4.2 Backward Propagation

In order to implement this phase, we first need to get the error value of the net. We use the formula

$$\delta = (t - y) * f'(y)$$

where $f'(y)$ is the first derivative, y is the output of the net, and t is the label value. The idea for backward propagation is to rotate the net by 180 degrees, so that the output tank become the input of the net. In this tank we put the error value, and then we propagate that value in the net like we did in forward propagation. In each node we get a delta value (that is, the error for each node); the delta values are used for updating the value of weights.

4.3 Update Weights

In this phase we use the formula

$$w_{jk}^{new} = w_{jk}^{old} + n * \delta_k * y_j$$

where w_{jk} is the weight between the node k and node j, δ_k is the error calculated in the current node, and y_j is the output of upper layer node. The resulting schema is almost the same as perceptron updating weights schema.

5 Discussion and Conclusions

In solution 1, the size of the network will be larger compared to that of solution 2, mainly due to the higher number of multiplications and tangent operations required.

To implement the timer that regulates the valves in solution 2, a pump will be needed to allow water to cycle through the three tanks. In a future implementation, the effects of the pump on the network will also need to be taken into account.

Regarding training, the network in solution 2 requires that the connecting tubes between the various nodes be changed with every weight update. This will surely make the training process of the network much harder with respect to solution 1.

Summarizing, the network in solution 2 requires a smaller number of tanks, tubes, and multiplication structures: as a consequence, it is the simplest to implement in terms of size. On the contrary, the network in solution 1 is the simplest to train, as it does not require modifying the dimension of the tubes every time the weight values change.

Future investigations concern the tests that could be conducted: for example, using more precise learning rates (e.g. vary in hundredths rather than just decimal places), to check whether or not better results can be obtained.

Studies on more complex networks such as RNN (Recurrent Neural Network) or LSTM (Long Short-Term Memory) are another important research direction.

Acknowledgements

The work of Claudio Zandron was partially supported by Università degli Studi di Milano-Bicocca, Fondo Ateneo per la Ricerca 2023, project 2023-ATE-0333.

References

1. Ballesteros, K. J., Cailipan, D. P. P., de la Cruz, R. T. A., Cabarle, F. G. C., Adorna, H. N. (2022), Matrix representation and simulation algorithm of numerical spiking neural P systems, Journal of Membrane Computing, 4(1), 2022, 41–55.
2. Dupaya, A. G. S., Galano, A. C. A. P., Cabarle, F. G. C., De La Cruz, R. T., Ballesteros, K. J., Lazo, P. P. L., A web-based visual simulator for spiking neural P systems, Journal of Membrane Computing, 4(1), 2022, 21–40.
3. T. Hinze, H. Happe, A. Henderson, R. Nicolescu, Membrane Computing with Water, Journal of Membrane Computing, Springer, 2, 2020, 121–136.
4. A. Henderson, R. Nicolescu, M.J. Dinneen, T. Chan, H. Happe, T. Hinze, Turing Completeness of Water Computing, Journal of Membrane Computing, Springer, 3(3), 2021, 182–193.
5. A. Henderson, R. Nicolescu, M.J. Dinneen, T. Chan, H. Happe, T. Hinze, Programmable and Parallel Water Computing, Journal of Membrane Computing, Springer, 5(1), 2023, 25–54.
6. Ionescu, M., Păun, Gh., Yokomori, T., Spiking neural P systems, Fundamenta informaticae, 71(2, 3), 2006, 279–308.
7. A. Leporati, L. Manzoni, G. Mauri, 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.
8. Leporati, A., Mauri, G., Zandron, C. Spiking neural P systems: main ideas and results, Natural Computing, 21(4), 2022, 629-649.

9. Liu, Y., Nicolescu, R., Sun, J., Formal verification of cP systems using PAT3 and ProB, Journal of Membrane Computing, 2(2), 2020, 80–94.
10. Liu, Y., Nicolescu, R., Sun, J., Formal verification of cP systems using Coq, Journal of Membrane Computing, 3(3), 205–220.
11. Martín-Vide, C., Păun, Gh., Pazos, J., Rodriguez-Paton, A., Tissue P systems, Theoretical Computer Science, 296(2), 2003, 295–326.
12. McCulloch, W., Pitts, W., A Logical Calculus of Ideas Immanent in Nervous Activity, Bulletin of Mathematical Biophysics 5 (4), 1943, 115–133. doi:10.1007/BF02478259.
13. Gh. Păun, Computing with membranes, Journal of Computer and System Sciences, 61(1), 2000, 108–143.
14. Gh. Păun, P systems with active membranes: Attacking NP-Complete problems, Journal of Automata, Languages and Combinatorics, 6(1), 2001, 75–90.
15. Gh. Păun, G. Rozenberg, A. Salomaa, (Eds.), The Oxford Handbook of Membrane Computing, Oxford University Press, New York, 2010.
16. Sosik, P, P systems attacking hard problems beyond NP: a survey, Journal of Membrane Computing 1, 2019, 198–208.
17. Valencia-Cabrera, L., Perez-Hurtado, I., Martinez-del Amor, M.A., Simulation challenges in membrane computing, Journal of Membrane Computing, 2, 2020, 1–11.

Conditional Uniport P Systems with Two Cells

Erzsébet Csuhaj-Varjú[1] and Sergey Verlan[2]

[1] Department of Algorithms and Their Applications,
Faculty of Informatics,
ELTE Eötvös Loránd University, Budapest,
Pázmány Péter sétány 1/c, 1117, Hungary
csuhaj@inf.elte.hu
[2] Univ Paris Est Creteil, LACL, F-94010, Creteil, France
verlan@u-pec.fr

Abstract. Generalized communicating P systems or GCPSs are tissue-like P systems (networks of cells) where each rule moves only two objects across the cells. Depending on the source and target locations there are 8 types of such rules, the most known being the symport and the antiport rules. Conditional uniport-in and uniport-out rules are the simplest possible, involving the movement of a single object, the other one remaining in its original location. For most variants of GCPSs, three cells and their common environment have been shown to be sufficient to achieve computational completeness with a single type of rules. In this paper, we prove that GCPSs with only conditional-uniport-in rules, only two cells and their common environment are Turing complete. We also show that there exist $k > 0$ such that the family of recursively enumerable sets of numbers containing only integers greater than or equal to k is equal to the family of sets of numbers generated by GCPSs with only conditional uniport-out rules, only two cells, and their common environment. These results are major improvements on the previous results.

1 Introduction

Purely communicating P systems are membrane systems that operate only by moving objects from one region to another, including the environment. They are of particular interest because many of these P system variants are computationally complete, proving that object evolution can be replaced by communication with the environment, where an arbitrary number of instances of a type of object can be found. This implies that there are always as many instances of these types of objects in the environment as are necessary to perform the actual transition between the P system and the environment. The reader can find detailed information on purely communicating P systems in [9].

Well-known variants of purely communicating P systems are generalized communicating P systems (or GCPSs), introduced in [17], to provide a common generalization of various such models.

A generalized communicating P system is a P system having a hypergraph structure (a network of cells) where each node represents a cell and each hyperedge corresponds to a rule. Every node contains a multiset of objects that can be communicated; communication

means a move of objects between the cells across the hyperedge, according to prescribed interaction (communication) rules.

The form of an interaction rule is $(i, a)(j, b) \to (k, a)(l, b)$ where a and b are objects and i, j, k, l are labels for the input and the output cells. Such a rule means that an object a from cell i and an object b from cell j move synchronously (in one step) to cell k and cell l, respectively. These rules are very simple since they describe the movement of only two objects.

The system is embedded in an environment, represented by cell 0. The environment may have certain objects in an arbitrary number of copies (also called in an unbounded number of copies) and certain objects only in a finite number of copies. The generalized communicating P system and the environment interact by using the communication (interaction) rules given above, with the restriction that at every computation step only a finite number of objects is allowed to enter any cell from the environment.

In each computation step, the rules are applied in a maximally parallel manner, as is usual in P systems theory. A computation step may change the multisets representing the contents of the cells (also called the configuration of the GCPS). A computation in a GCPS is a sequence of configurations directly following each other, starting from the initial configuration and ending in a halting configuration. The result of the computation is the number of objects found in a distinguished cell, called the output cell.

It is worth noting that GCPS is the fruit of several mathematical generalizations of the model of symport/antiport P systems [13] that corresponds to the formalization of the biological process of co-transport. In [16] conditional uniport was introduced as a simplification of the symport/antiport model, which also corresponds to a model of biological communication through ion channels. Next, in [17] the GNPS model was born, aiming to unify the definition of both concepts. It also introduced more complex rules corresponding to hypergraph communication which in turn inspired the creation of the formal framework for P systems [11] that allowed to capture many formal aspects of the P system models [10,7,18,19].

During the years, GCPSs have been studied in detail. The simplicity of their rules and their relation to other fields like the theory of Petri nets [5,4] raised interest in their study.

One of the most interesting problems is how many cells a GPCS must have to achieve a certain computational power and how much its interaction rules can be simplified.

It has been shown that even restricted variants of generalized communicating P systems (with respect to the form of rules) are able to generate any recursively enumerable set of numbers. Furthermore, they include computationally complete models with a small number of cells and a simple underlying hypergraph architecture [8,6,5,9,16,17]. For example, GCPSs with only three cells and with only join rules, or only split rules, or only chain rules are computationally complete [8]. In these cases, any rule operates with three cells. It is also shown that the maximal computational power can also be obtained if the alphabet of objects of the GPCSs is a singleton [5].

It is an interesting question whether in the case of the uniport-out and the uniport-in rules, which involve two cells in each rule, two or more cells are necessary to obtain computational completeness.

The conditional-uniport-out rule (the $uout$ rule) sends symbol b to cell l provided that both symbol a and b are in cell i [16]; the conditional-uniport-in rule (the uin rule) brings symbol b to cell i provided that a is in that cell.

In this paper, we prove that GCPSs with only conditional-uniport-in rules, only two cells and their common environment are Turing complete. We also show that there exist $k > 0$ such that the family of recursively enumerable sets of integers greater than or equal to k is equal to the family of sets of numbers generated by GCPSs with only conditional uniport-out rules, only two cells and their common environment.

These results are improvements on previous results from [6] where 30 cells were used for the same result.

The paper is organized as follows. Section 2 gives the definitions and introduces the model. The two main results are presented in Section 3. Finally, Section 4 gives some ideas for the further research.

2 Definitions

The reader is supposed to be familiar with formal language theory and membrane computing; for further details consult [15] and [14]. NRE denotes the family of recursively enumerable sets of natural numbers. $\mathbb{N}_k$ denotes the set of natural numbers greater than or equal to k and N_kRE denotes the family of recursively enumerable sets of natural numbers greater than or equal to k.

For a finite multiset of symbols X over an alphabet V, $supp(X)$ denotes the set of symbols in X (the support of X) and $|X|$ denotes the total number of its symbols (its size). The number of occurrences of symbol x in X is denoted by $|X|_x$.

Throughout the paper, every finite multiset X is presented as a string w, where X and w have the same number of occurrences of symbol a, for each $a \in V$. The empty multiset is denoted by λ.

If no confusion arises, then the set of all finite multisets over V is denoted by V^*.

A counter automaton is a 5-tuple $M = (Q, \mathcal{R}, q_0, q_f, P)$, where Q is a finite non-empty set, called the set of states, $\mathcal{R} = \{A_1, \ldots, A_k\}$, $k \geq 1$, is a set of counters, called also registers, $q_0 \in Q$ is the initial state, and $q_f \in Q$ is the final state. P is a set of instructions of the following forms: $(p, A+, q, s)$, where $p, q, s \in Q, p \neq q_f, A \in \mathcal{R}$, called an increment instruction, or $(p, A-, q)$, where $p, q \in Q, p \neq q_f, A \in \mathcal{R}$, called a decrement instruction, or $(p, A0, q)$, where $p, q \in Q, p \neq q_f, A \in \mathcal{R}$, called a zero-check instruction. Without loosing the generality, it can be supposed that for every $p \in Q$, $(p \neq q_f)$, there is exactly one instruction of the form either $(p, A+, q, s)$ or $(p, A-, q)$, or $(p, A0, q)$.

A configuration of a counter automaton M, defined above, is a $(k + 1)$-tuple $(q, m_1, \ldots, m_k)$, where $q \in Q$ and $m_1, \ldots, m_k$ are non-negative integers; q is the current state of M and $m_1, \ldots, m_k$ are the current numbers stored in the registers (the current contents of the registers or the value of the registers) $A_1, \ldots A_k$, respectively.

A transition of the counter automaton consists in executing an instruction. An increment instruction $(p, A+, q, s) \in P$ is performed if M is in state p, the number stored in register A is increased by 1, and after that M enters either state q or state s, chosen non-deterministically. A decrement instruction $(p, A-, q) \in P$ is performed if

M is in state p, and if the number stored in register A is positive, then it is decreased by 1, and then M enters state q. If the number stored in register A is zero, then the computation "blocks" and the corresponding non-deterministic computation branch is considered to fail. A zero-check instruction $(p, A0, q) \in P$ is performed if M is in state p, and if the number stored in register A is 0, then the contents of A remains unchanged and M enters state q. If the contents of register A is not zero, then the computation "blocks" and the corresponding non-deterministic computation branch is considered to fail.

A counter automaton $M = (Q, \mathcal{R}, q_0, q_f, P)$, with k registers, given as above, generates a non-negative integer n, if starting from the initial configuration $(q_0, 0, 0, \ldots, 0)$ it enters (in a non-deterministic manner) the final configuration $(q_f, n, 0, \ldots, 0)$. The set of non-negative integers generated by M is denoted by $N(M)$.

We remark that counter automata are very closely related to register machines [12]. In fact, in a register machine the operations of minus and zero check are combined in a single instruction $(p, A-, r, s)$ that corresponds to instructions $(p, A-, r)$ and $(p, A0, s)$ of the counter automaton.

Next we recall the basic definitions concerning generalized communicating P systems [17].

A generalized communicating P system (a GCPS) of degree n, where $n \geq 1$, is an $(n+4)$-tuple $\Pi = (O, E, w_1, \ldots, w_n, R, h)$ where

1. O is an alphabet, called the set of objects of Π;
2. $E \subseteq O$; called the set of environmental objects of Π;
3. $w_i \in O^*$, $1 \leq i \leq n$, is the multiset of objects initially associated to cell i;
4. R is a finite set of interaction rules or communication rules of the form $(i, a)(j, b) \to (k, a)(l, b)$, where $a, b \in O$, $0 \leq i, j, k, l \leq n$, and if $i = 0$ and $j = 0$, then $\{a, b\} \cap (O \setminus E) \neq \emptyset$; i.e., at least one of a and b is not element of E.
5. $h \in \{1, \ldots, n\}$ is the output cell.

The system consists of n cells, labeled by natural numbers from 1 to n, which contain multisets of objects over O. Initially, cell i contains multiset w_i (the initial contents of cell i is w_i). An additional special cell, labeled by 0 and called the environment is distinguished. The environment contains objects of E in an infinite number of copies.

The cells interact by means of the rules $(i, a)(j, b) \to (k, a)(l, b)$, with $a, b \in O$ and $0 \leq i, j, k, l \leq n$. As the result of the application of the rule, object a moves from cell i to cell k and b moves from cell j to cell l. If two objects from the environment move to some other cell or cells, then at least one of them must not appear in the environment in an infinite number of copies.

The structure of the system is a hypergraph implicitly deduced from the set of rules. Indeed, a rule $(i, a)(j, b) \to (k, a)(l, b)$ induces an hyperedge $\{i, k, j, l\}$. A generalization of GNPS using more cells in a rule and also performing the rewriting lead to the notion of the formal framework for P systems [11].

A configuration of a GCPS Π, as above, is an $(n+1)$-tuple $(z_0, z_1, \ldots, z_n)$ with $z_0 \in (O \setminus E)^*$ and $z_i \in O^*$, for all $1 \leq i \leq n$; z_0 is the multiset of objects present in the environment in a finite number of copies, whereas, for all $1 \leq i \leq n$, z_i is the multiset of objects present inside cell i. The initial configuration of Π is the $(n+1)$-tuple $(\lambda, w_1, \ldots, w_n)$.

Given a multiset of rules $\mathcal{R}$ over R and a configuration $u = (z_0, z_1, \ldots, z_n)$ of Π, we say that $\mathcal{R}$ is applicable to u if all its elements can be applied simultaneously to the objects of multisets $z_0, z_1, \ldots, z_n$ such that every object is used by at most one rule. If there is no multiset $\mathcal{R}'$ where $\mathcal{R}$ is a proper submultiset of $\mathcal{R}'$ can be applied to configuration $u = (z_0, z_1, \ldots, z_n)$ of Π, then a new configuration $u' = (z'_0, z'_1, \ldots, z'_n)$ is obtained by applying $\mathcal{R}$ in a non-deterministic maximally parallel manner.

One such application of a multiset of rules satisfying the conditions listed above represents a transition in Π from configuration u to configuration u'. A transition sequence is said to be a successful generation by Π if it starts with the initial configuration of Π and ends with a halting configuration, i.e., with a configuration where no further transition step can be performed.

In this paper, we deviate from the standard notation of the configuration to make it easier to follow the movement between cells. Instead of $u = (z_0, z_1, \ldots,$
$z_n)$, we will use the notation $u = (0, z_0)(1, z_1) \cdots (n, z_n)$. The numbers $1, \ldots, n$ refer to the label of the cell, and z_i refers to the contents of cell i.

Π generates a non-negative integer n if there is a successful generation by Π such that n is the size of the multiset of objects present inside the output cell in the halting configuration. The set of non-negative integers generated by a GCPS Π in this way is denoted by $N(\Pi)$. If instead of counting all the objects present inside the output cell in the halting configuration at the end of successful generations of Π we consider only the number of objects from a nonempty subset $O' \subseteq O$, then we denote the corresponding set of numbers generated by $N_{O'}(\Pi)$.

In the following we recall the notions of the possible restrictions on the interaction rules (modulo symmetry). We distinguish the following cases, called GCPSs with minimal interaction:

1. $i = j = k \neq l$: the conditional-uniport-out rule (the $uout$ rule) sends b to cell l provided that a and b are in cell i [16];
2. $i = k = l \neq j$: the conditional-uniport-in rule (the uin rule) brings b to cell i provided that a is in that cell [16];
3. $i = j$, $k = l$, $i \neq k$: the symport2 rule (the $sym2$ rule) corresponds to the minimal symport rule [14], i.e., a and b move together from cell i to k;
4. $i = l$, $j = k$, $i \neq j$: the antiport1 rule (the $anti1$ rule) corresponds to the minimal antiport rule [14], i.e., a and b are exchanged in cells i and k;
5. $i = k$ and $i \neq j$, $i \neq l$, $j \neq l$: the presence-move rule (the $presence$ rule) moves the object b from cell j to l, provided that there is an object a in cell i and i, j, l are pairwise different cells;
6. $i = j$, $i \neq k$, $i \neq l$, $k \neq l$: the $split$ rule sends a and b from cell i to cells k and l, respectively;
7. $k = l$, $i \neq j$, $k \neq i$, $k \neq j$: the $join$ rule brings a and b together to cell k;
8. $l = i$, $i \neq j$, $i \neq k$ and $j \neq k$: the $chain$ rule moves a from cell i to cell k while b is moved from cell j to cell i, i.e., to the cell where a located previously;
9. i, j, k, l are pairwise different numbers: the parallel-shift rule (the $shift$ rule) moves a and b from two different cells to another two different cells.

$NOtP_n(x)$ denotes the set of numbers generated by generalized communicating P systems with minimal interaction of degree n, $n \geq 1$, and with rules of type x, where

$x \in \{uout, uin, sym2, anti1, presence, split, join, chain, shift\}$. $NOtP_*(x)$ is the notation for $\bigcup_{n=1}^{\infty} NOtP_n(x)$.

3 Results

In this section, we discuss whether generalized communicating P systems of the $uout$ or uin type and with only two cells are universal or not. First, we prove that there exist $k > 0$ such that the family of recursively enumerable sets of natural numbers greater than or equal to k is equal to the family of sets of numbers generated by GCPSs with only conditional uniport-out rules, only two cells and their common environment. We then show that any recursively enumerable set of non-negative integers can be generated by a GCPS with only conditional-uniport-in rules and only two cells and their common environment. The converse of the latter statement also holds, thus this variant of GCPSs is computationally complete.

We first describe the computational power of GCPSs of type $uout$ and with only two cells.

Theorem 1. *There exist $k > 0$ such that $NOtP_2(uout) \supseteq N_kRE$.*

Proof. Consider an arbitrary counter automaton $M = (Q, C, q_0, q_f, P)$. We construct the following GCPS $\Pi = (O, E, w_1, w_2, R, 1)$ (with conditional-uniport-out rules only).

$O = Q \cup \{p_1, p_2, p_2' \mid (p, A+, q, q') \in P\} \cup \{p_1, p_2, p_3 \mid (p, A_0, q, q') \in P\} \cup \{p_1, p_2, p_3, p_4 \mid (p, A-, q, q') \in P\} \cup \{L_1, L_2\} \cup E$.

$E = C \cup \{C_A \mid A \in C\} \cup \{L_0, Z\}$.

$w_1 = \{q_0, L_1, Z\} \cup \{p_1, p_2 \mid p \in Q\} \cup \{p_2' \mid (p, A+, q, q') \in P\} \cup \{p_4 \mid (p, A-, q, q') \in P\}$.

$w_2 = (Q \setminus \{q_0\}) \cup \{p_3, p' \mid p \in Q\} \cup \{L_2, Z\}$.

The set of rules R is defined as follows.

For any instruction (p, A_i+, q, q') of M the set of rules R contains the following rules:

$p.1.1 : (1, p)(1, p_1) \to (1, p)(2, p_1)$
$p.2.1 : (1, p)(1, p_2) \to (1, p)(0, p_2)$ $p.2.2 : (1, p)(1, p_2') \to (1, p)(0, p_2')$
$p.2.3 : (2, p_1)(2, p_3) \to (2, p_1)(1, p_3)$
$p.3.1 : (0, p_2)(0, A_i) \to (0, p_2)(2, A_i)$ $p.3.2 : (0, p_2')(0, A_i) \to (0, p_2')(2, A_i)$
$p.3.3 : (1, p_3)(1, p) \to (1, p_3)(0, p)$
$p.4.1 : (0, p)(0, p_2) \to (0, p)(2, p_2)$ $p.4.2 : (0, p)(0, p_2') \to (0, p)(2, p_2')$
$p.4.3 : (2, A_i)(2, p_1) \to (2, A_i)(1, p_1)$ $p.4.4 : (1, Z)(1, p_3) \to (1, Z)(0, p_3)$
$p.5.1 : (0, p_3)(0, p) \to (0, p_3)(2, p)$ $p.5.2 : (2, p_2)(2, A_i) \to (2, p_2)(1, A_i)$
$p.5.3 : (2, p_2')(2, A_i) \to (2, p_2')(1, A_i)$
$p.6.1 : (2, p_2)(2, q) \to (2, p_2)(1, q)$ $p.6.2 : (2, p_2')(2, q') \to (2, p_2')(1, q')$
$p.6.3 : (0, Z)(0, p_3) \to (0, Z)(2, p_3)$
$p.7.1 : (2, p_3)(2, p_2) \to (2, p_3)(1, p_2)$ $p.7.2 : (2, p_3)(2, p_2') \to (2, p_3)(1, p_2')$
$p.l.1 : (2, p_2)(2, L_2) \to (2, p_2)(0, L_2)$ $p.l.2 : (2, p_2')(2, L_2) \to (2, p_2')(0, L_2)$
$p.l.3 : (2, A_i)(2, L_2) \to (2, A_i)(0, L_2)$ $p.l.4 : (0, p)(0, L_0) \to (0, p)(1, L_0)$

For any instruction $(p, A_i 0, q)$ of M the set of rules R contains the following rules:

$p.1.1 : (1, p)(1, p_1) \to (1, p)(2, p_1)$
$p.2.1 : (1, p)(1, p_2) \to (1, p)(0, p_2)$ $p.2.2 : (2, p_1)(2, p_3) \to (2, p_1)(1, p_3)$
$p.3.1 : (0, p_2)(0, C_{A_i}) \to (0, p_2)(2, C_{A_i})$ $p.3.2 : (1, p_3)(1, p) \to (1, p_3)(0, p)$
$p.4.1 : (0, p)(0, p_2) \to (0, p)(2, p_2)$ $p.4.2 : (2, C_{A_i})(2, p_1) \to (2, C_{A_i})(1, p_1)$
$p.4.3 : (1, Z)(1, p_3) \to (1, Z)(0, p_3)$
$p.5.1 : (0, p_3)(0, p) \to (0, p_3)(2, p)$ $p.5.2 : (2, p_2)(2, C_{A_i}) \to (2, p_2)(1, C_{A_i})$
$p.6.1 : (2, p_2)(2, q) \to (2, p_2)(1, q)$ $p.6.2 : (0, Z)(0, p_3) \to (0, Z)(2, p_3)$
$p.6.3 : (1, C_{A_i})(1, A_i) \to (1, C_{A_i})(2, A_i)$
$p.7.1 : (2, p_3)(2, p_2) \to (2, p_3)(1, p_2)$
$p.8.1 : (1, p_2)(1, C_{A_i}) \to (1, p_2)(0, C_{A_i})$
$p.l.1 : (2, p_2)(2, L_2) \to (2, p_2)(0, L_2)$ $p.l.2 : (2, C_{A_i})(2, L_2) \to (2, C_{A_i})(0, L_2)$
$p.l.3 : (0, p)(0, L_0) \to (0, p)(1, L_0)$ $p.l.4 : (2, A_i)(2, L_2) \to (2, A_i)(0, L_2)$

For any instruction $(p, A_i -, q)$ of M the set of rules R contains the following rules:

$p.1.1 : (1,p)(1,p_1) \to (1,p)(2,p_1)$ $\quad p.1.2 : (1,p_2)(1,p_4) \to (1,p_2)(0,p_4)$
$p.2.1 : (1,p)(1,p_2) \to (1,p)(0,p_2)$ $\quad p.2.2 : (2,p_1)(2,p_3) \to (2,p_1)(1,p_3)$
$p.3.1 : (0,p_2)(0,p_4) \to (0,p_2)(2,p_4)$ $\quad p.3.2 : (1,p_3)(1,p) \to (1,p_3)(0,p)$
$p.4.1 : (0,p)(0,p_2) \to (0,p)(2,p_2)$ $\quad p.4.2 : (2,p_1)(2,p_4) \to (2,p_1)(1,p_4)$
$p.4.3 : (1,p_3)(1,A_i) \to (1,p_3)(0,A_i)$
$p.5.1 : (1,p_4)(1,p_3) \to (1,p_4)(2,p_3)$ $\quad p.5.2 : (2,p_2)(2,p_1) \to (2,p_2)(1,p_1)$
$p.6.1 : (1,p_1)(1,p_4) \to (1,p_1)(0,p_4)$ $\quad p.6.2 : (2,p_2)(2,q) \to (2,p_2)(1,q)$
$p.7.1 : (0,p_4)(0,p) \to (0,p_4)(2,p)$
$p.8.1 : (2,p)(2,p_2) \to (2,p)(1,p_2)$
$p.l.1 : (0,p_2)(0,L_0) \to (0,p_2)(1,L_0)$ $\quad p.l.2 : (1,p_4)(1,L_1) \to (1,p_4)(0,L_1)$
$p.l.3 : (1,p_3)(1,L_1) \to (1,p_3)(0,L_1)$

The system also contains the following rules:

$L.1 : (0,Z)(0,L_2) \to (0,Z)(1,L_2)$ $\quad L.2 : (1,Z)(1,L_2) \to (1,Z)(0,L_2)$
$L.3 : (1,Z)(1,L_0) \to (1,Z)(2,L_0)$ $\quad L.4 : (2,Z)(2,L_0) \to (2,Z)(1,L_0)$
$L.5 : (0,Z)(0,L_1) \to (0,Z)(2,L_1)$ $\quad L.6 : (2,Z)(2,L_1) \to (2,Z)(0,L_1)$

We claim that Π simulates M. More precisely, we claim that there exists $k \geq 0$ such that $N(M) + k = N(\Pi)$ ($x \in N(M)$ implies $x + k \in N(\Pi)$ and $y \in N(\Pi)$ implies $y - k \geq 0$ and $y - k \in M$).

In the proof we encode the configuration of M in Π and then we show that there exists a bisimulation between configurations of M and the corresponding encoding in Π. For a configuration $\mathbb{C} = (p, v_1, \ldots, v_n)$, $n = |C|$ of M we define $code(\mathbb{C}) = (0,\lambda)(1, w_1 p \bigcup_{c \in C} A_c^{v_c})(2, w_2)$. We observe that the initial configuration of Π is $code(q_0, 0, \ldots, 0)$.

We will start by showing that if there is a transition $\mathbb{C} \Rightarrow \mathbb{C}'$ in M, then in Π there is a computation $code(\mathbb{C}) \Rightarrow^* code(\mathbb{C}')$. Let us consider the case of each instruction.

Suppose that $(p, v_1, \ldots, v_i, \ldots, v_n) \Rightarrow_r (q, v_1, \ldots, v_i + 1, \ldots, v_n)$, where $r : (p, A_i+, q, q')$. In Π there exists the following derivation (in configurations below we indicate only objects that are moved):

$$(0,\lambda)(1, pp_1p_2p_2')(2, p_3qq') \Rightarrow_{p.1.1} (0,\lambda)(1, pp_2p_2')(2, p_1p_3qq') \Rightarrow_{p.2.1,p.2.3}$$
$$(0,p_2)(1, pp_2'p_3)(2, p_1qq') \Rightarrow_{p.3.1,p.3.3} (0, pp_2)(1, p_2'p_3)(2, A_ip_1qq') \Rightarrow_{p.4.1,p.4.3,p.4.4}$$
$$(0, pp_3)(1, p_1p_2')(2, A_ip_2qq') \Rightarrow_{p.5.1,p.5.2} (0,p_3)(1, A_ip_1p_2')(2, pp_2qq') \Rightarrow_{p.6.1,p.6.3}$$
$$(0,\lambda)(1, A_ip_1p_2'q)(2, pp_2p_3q') \Rightarrow_{p.7.1} (0,\lambda)(1, A_ip_1p_2p_2'q)(2, pp_3q')$$

It is obvious that if rule $p.3.2$ is used instead of $p.3.1$ then in the final configuration q is replaced by q'.

Now consider $(p, v_1, \ldots, v_{i-1}, 0, \ldots, v_n) \Rightarrow_r (q, v_1, \ldots, v_{i-1}, 0, \ldots, v_n)$, where $r : (p, A_i 0, q)$. In Π there exists the following derivation (in configurations below we indicate only objects that are moved):

$$
\begin{aligned}
&(0, C_{A_i})(1, pp_1p_2)(2, p_3q) \Rightarrow_{p.1.1} (0, C_{A_i})(1, pp_2)(2, p_1p_3q) \Rightarrow_{p.2.1,p.2.2} \\
&(0, C_{A_i}p_2)(1, pp_3)(2, p_1q) \Rightarrow_{p.3.1,p.3.2} (0, pp_2)(1, p_3)(2, C_{A_i}p_1q) \Rightarrow_{p.4.1,p.4.2,p.4.3} \\
&(0, pp_3)(1, p_1)(2, C_{A_i}p_2q) \Rightarrow_{p.5.1,p.5.2} (0, p_3)(1, C_{A_i}p_1)(2, pp_2q) \Rightarrow_{p.6.1,p.6.2} \\
&(0, \lambda)(1, C_{A_i}p_1q)(2, pp_2p_3) \Rightarrow_{p.7.1} (0, \lambda)(1, C_{A_i}p_1p_2q)(2, pp_3) \Rightarrow_{p.8.1} \\
&(0, C_{A_i})(1, p_1p_2q)(2, pp_3) = (0, \lambda)(1, p_1p_2q)(2, pp_3)
\end{aligned}
$$

This simulation is very similar to the increment case. In fact, instead of bringing a copy of register symbol A_i a checker symbol C_{A_i} is brought to cell 1 using identical rules. The checker symbol verifies that at step 7 there are no copies of A_i in cell 1 (by not taking part in rule $p.7.2$) and then goes back to cell 0.

Now consider $(p, v_1, \ldots, v_i, \ldots, v_n) \Rightarrow_r (q, v_1, \ldots, v_i - 1, \ldots, v_n)$, where $v_i > 0$ and $r : (p, A_i -, q)$. In Π there exists the following derivation (in configurations below we indicate only objects that are moved):

$$
\begin{aligned}
&(0, \lambda)(1, A_ipp_1p_2p_4)(2, p_3q) \Rightarrow_{p.1.1,p.1.2} (0, p_4)(1, A_ipp_2)(2, pp_1p_3q) \Rightarrow_{p.2.1,p.2.2} \\
&(0, p_2p_4)(1, A_ipp_3)(2, p_1q) \Rightarrow_{p.3.1,p.3.2} (0, pp_2)(1, A_ip_3)(2, p_1p_4q) \Rightarrow_{p.4.1,p.4.2,p.4.3} \\
&(0, A_ip)(1, p_3p_4)(2, p_1p_2q) \Rightarrow_{p.5.1,p.5.2} (0, A_ip)(1, p_1p_4)(2, p_2p_3q) \Rightarrow_{p.6.1,p.6.2} \\
&(0, A_ipp_4)(1, p_1q)(2, pp_2p_3) \Rightarrow_{p.7.1} (0, A_ip_4)(1, p_1q)(2, pp_2p_3) \Rightarrow_{p.8.1} \\
&(0, A_ip_4)(1, p_1p_2q)(2, pp_3)
\end{aligned}
$$

The overall simulation in this case also follows the above pattern, however here p_2 brings back to cell 1 symbol p_4 that allows to move out p_3 that performs the decrement.

We remark, that on the first step, all symbols p_4, $(p, A_i -, q) \in P$ will go to cell 0 and will remain there until the end of the computation, returning to cell 1 only if corresponding instruction is simulated.

Now let us show that any evolution of Π except the above ones leads to an infinite loop. This will prove the converse part of the bisimulation.

Consider a configuration of Π corresponding to $code(p, v_1, \ldots, v_n)$, where p is an increment instruction $(p, A_i +, q, q')$.

Step 1: The corresponding configuration in Π is $(0, \lambda)(1, pp_1p_2p_2')(2, p_3qq')$. Instead of rule $p.1.1$ one of rules $p.2.1$ or $p.2.2$ could be applied. We will consider the first case, the other one is symmetrical.

$$(0, \lambda)(1, pp_1p_2p_2')(2, p_3qq') \Rightarrow_{p.2.1} (0, p_2)(1, pp_1p_2')(2, p_3qq')$$

At this moment there are only two possible applicable multisets of rules: $p.1.1, p.3.1$ and $p.2.2, p.3.1$. When applying the first multiset, we obtain:

$$(0, p_2)(1, pp_1p_2')(2, p_3qq') \Rightarrow_{p.1.1,p.3.1} (0, p_2)(1, pp_2')(2, A_ip_1p_3qq')$$

At this moment two multisets of rules are applicable $p.2.2, p.l.3$ and $p.2.2, p.4.3$, $p.3.1$. In the first case, the loop symbol L_2 is brought to cell 0 and the system will get

stuck in an infinite loop using rules $L.1$ and $L.2$. By using the second group of rules an additional A_i is brought to cell 2 and at the next step it will push the loop symbol L_2 to cell 0 using rule $p.l.3$, bringing the computation to an infinite loop.

Not let us consider the case of the application of the multiset $p.2.2, p.3.1$:

$$(0, p_2)(1, pp_1p_2')(2, p_3qq') \Rightarrow_{p.2.1,p.3.1} (0, p_2p_2')(1, pp_1p_2')(2, A_ip_3qq')$$

At the next step rule $p.l.3$ will be necessarily applied, yielding an infinite loop.

Step 2: On the second step it is also possible to apply the multiset $p.2.2, p.2.3$, however this case is symmetrical with respect to the considered one.

Step 3: There is only one other option: to apply the multiset of rules $p.2.2, p.4.4, p.3.1$:

$$(0, p_2)(1, pp_3p_2')(2, p_1qq') \Rightarrow_{p.2.2,p.4.4,p.3.1} (0, p_2p_2'p_3)(1, p)(2, A_ip_1qq')$$

Now two multisets of rules are applicable: $p.3.1, p.3.2, p.l.3, p.4.4$ and also $p.3.1, p.3.2, p.4.4, p.4.3$. In the first case the loop symbol L_2 is introduced to cell 0, yielding an infinite loop. In the second case, cell 2 contains 3 symbols A_i, hence at the next step the rule $p.l.3$ will be applied, yielding an infinite loop for the computation.

Step 4: On the third step three more multisets of rules are applicable:
$p.3.1, p.4.4, p.l.3, p.l.4$; $p.3.1, p.4.4, p.4.3, p.l.4$ and $p.4.1, p.4.4, p.l.3$. However they either involve the application of rule $p.l.3$ or $p.l.4$ (by not keeping busy symbol A_i in cell 2 or symbol p in cell 0).

Step 5: It is not difficult to observe that at this step the only way to keep busy symbols A_i and p is to apply rules $p.5.1$ and $p.5.2$. Any other combination of rules force one of these symbols to be used in rule $p.l.3$ or $p.l.4$.

Step 6: The other applicable multiset of rules is $p.l.1$, which results in an infinite loop.

Step 7: Here again, the only other possibility is to apply rule $p.l.1$, which results in an infinite loop.

We remark that the corresponding sequences of states are shown on the figure uniport_out_plus in the appendix.

Consider a configuration of Π corresponding to $code(p, v_1, \ldots, v_n)$, where p is an increment instruction (p, A_i-, q). Suppose that the value of the register A_i is not zero (hence, there is at least one symbol A_i). By considering the derivation given above, the only options at each step involve the use of the rule $p.l.1$ or $p.l.2$, see the appendix picture uniport_out_minus_nz for more details.

If the value of the register is zero, then on step 4 symbol p_3 cannot be involved in rule $p.4.3$, hence rule $p.l.3$ will be applied, bringing L_1 to cell 0 and starting an infinite loop. See more details in the appendix, picture uniport_out_minus_z.

Consider a configuration of Π corresponding to $code(p, v_1, \ldots, v_n)$, where p is an increment instruction $(p, 0A_i, q)$. Suppose that the value of the register A_i is zero (hence, there is no symbol A_i in cell 1).

Step 1: Beside the evolution above, the rule $p.2.1$ can be applied (and then only one case is possible for the next step):

$$(0, C_{A_i})(1, pp_1p_2)(2, p_3q) \Rightarrow_{p.2.1} (0, C_{A_i}p_2)(1, pp_1)(2, p_3q) \Rightarrow_{p.1.1,p.3.1} (0, p_2)(1, p)(2, C_{A_i}p_1p_3q)$$

Now there are two multisets of rules applicable: $p.3.1, p.2.2, p.l.2$ and also $p.3.1, p.4.2$. In the first case the system gets into an infinite loop, while in the second case two copies of C_{A_i} are present in cell 2, which means that at the next step rule $p.l.2$ will be applied, yielding an infinite loop.

Step 2: There are no other applicable rules.

Step 3: There is another applicable multiset of rules $p.3.1, p.4.3$:

$$(0, C_{A_i}p_2)(1, pp_3)(2, p_1q) \Rightarrow_{p.3.1,p.4.3} (0, p_2p_3)(1, p)(2, C_{A_i}p_1q)$$

At this moment the two possible options lead to an infinite loop.

Steps 4–7: All possible variants at these steps involve the application of one of the rules $p.l.1 - p.l.4$, yielding to an infinite loop. See picture uniport_out_zero_z in the appendix for more details.

If the value of the register is not zero, then on step 8 symbol C_{A_i} will be involved in rule $p.6.3$, sending a copy of A_i to cell 1. At the next step rule $p.l.4$ will be applied, starting an infinite loop. See more details in the appendix, picture uniport_out_zero_nz.

Hence, we have shown that a successful computation in Π passes through configurations corresponding to an encoding of the configuration of M. Hence, it is possible to reconstruct a computation in M from any computation in Π.

To conclude the proof, we observe that the value of k corresponds to the number of additional symbols in cell 1 and which depend in their turn on the number of instructions of the register machine. It is equal to $k = 4N_+ + 3(N_0 + N_-) + 1$, where N_+, N_- and N_0 is the number of increment, decrement and zero instructions, respectively. Since there exists a universal counter automaton (hence having a fixed number of instructions), the statement of the Theorem follows.

By invoking the Church-Turing thesis we obtain the following result.

Corollary 1. *There exist $k > 0$ such that $NOtP_2(uout) = N_kRE$.*

Now we examine the case of GCPSs with only conditional-uniport-in rules.

Theorem 2. $NOtP_2(uin) \supseteq NRE$.

Proof. We show that any recursively set of non-negative integers can be generated by a GCPS with two cells and with only conditional-uniport-in rules. Unlike the proof of the previous theorem, we give only the main ideas and the necessary details of the proof, leaving the rest to the reader. Let $M = (Q, \mathcal{R}, q_0, q_f, P)$ be a counter automaton, with $\mathcal{R} = \{A_1, \ldots, A_k\}$, $k \geq 1$, defined as in Section 2. We give a GCPS

$\Pi = (O, E, w_1, w_2, R, 1)$ with only conditional-uniport-in rules such that any halting transition sequence of Π corresponds to a halting transition sequence of M. Furthermore, the numbers generated by these two transition sequences are the same.

Let Π have the following components. (Since it is clear from the context, we use the symbol A_j also in case of Π).

$O = \{A_j, C_{A_j}, X_{A_j} \mid 1 \leq j \leq k\} \cup$
$Q \cup \{p_i \mid (p, A_j+, q, q') \in P, 1 \leq i \leq 6, 1 \leq j\} \cup$
$\{p_i \mid (p, A_j 0, q) \in P, 1 \leq i \leq 6\} \cup \{p_i \mid (p, A-, q) \in P, 1 \leq i \leq 6\} \cup$
$\{L, L_1, Z, F, q_f, q'_f\}$.

$E = \{A_j, C_{A_j}, X_{A_j} \mid 1 \leq j \leq k\} \cup \{Z, L, L_1\}$.

We note that number n, $n \geq 0$ stored in register A_j in M, $1 \leq j \leq k$, is represented by A_j^n in Π.

We give the initial configuration. Let $w_1 = q_0$ and let $w_2 \in ((O \setminus \{q_0\}) \setminus E)^* \cup \{L, L_1, Z\}$ such that any symbol in w_2 appears in only one copy in w_2.

We give the sets of rules of Π which simulate the instructions of M and only that and provide the necessary details of the proof.

We start with simulating the instructions for increment in M.

For any instruction (p, A_i+, q, q') of M, $1 \leq i \leq k$, the set of rules in P consists of the following rules.

$p.1.1 : (1, p)(2, p_1) \to (1, p)(1, p_1)$

$p.2.1 : (1, p)(2, p_2) \to (1, p)(1, p_2)$ $\quad p.2.2 : (0, Z)(1, p_1) \to (0, Z)(0, p_1)$

$p.3.1 : (0, p_1)(1, p) \to (0, p_1)(0, p)$ $\quad p.3.2 : (1, p_2)(2, p_3) \to (1, p_2)(1, p_3)$

$p.4.1 : (0, p)(1, p_2) \to (0, p)(0, p_2)$ $\quad p.4.2 : (1, p_3)(2, p_4) \to (1, p_3)(1, p_4)$

$p.4.3 : (0, p_1)(2, p_4) \to (0, p_1)(0, p_4)$

$p.5.1 : (1, p_4)(2, p_5) \to (1, p_4)(1, p_5)$ $\quad p.5.2 : (1, p_3)(0, A) \to (1, p_3)(1, A)$

$p.5.3 : (2, Z)(0, p) \to (2, Z)(2, p)$ $\quad p.5.4 : (0, p_2)(2, p_6) \to (0, p_2)(0, p_6)$

$p.5.5 : (1, p_4)(2, p_6) \to (1, p_4)(1, p_6)$

$p.6.1 : (0, p_2)(1, p_3) \to (0, p_2)(0, p_3)$ $\quad p.6.2 : (2, p)(0, p_1) \to (2, p)(2, p_1)$

$p.6.3 : (1, p_5)(2, q) \to (1, p_5)(1, q)$ $\quad p.6.4 : (1, p_5)(2, q') \to (1, p_5)(1, q')$

$p.7.1 : (0, p_2)(1, p_5) \to (0, p_2)(0, p_5)$ $\quad p.7.2 : (2, p_1)(0, p_3) \to (2, p_1)(2, p_3)$

$p.8.1 : (2, p_3)(0, p_2) \to (2, p_3)(2, p_2)$ $\quad p.8.2 : (2, Z)(0, p_5) \to (2, Z)(2, p_5)$

$p.8.3 : (2, p_1)(1, p_4) \to (2, p_1)(2, p_4)$

$p.9.1 : (2, p_2)(0, p_6) \to (2, p_2)(2, p_6)$

$p.l.1 : (0, p_2)(2, L) \to (0, p_2)(0, L)$ $\quad p.l.2 : (0, p_4)(2, L) \to (0, p_4)(0, L)$

$p.l.3 : (0, p_2)(2, L) \to (0, p_2)(0, L)$ $\quad p.l.4 : (1, p_2)(2, L) \to (1, p_2)(1, L)$

$p.l.5 : (1, p_6)(2, L) \to (1, p_6)(1, L)$

$L.1 : (0, Z)(0, L) \to (0, Z)(1, L)$ $\quad L.2 : (1, Z)(1, L) \to (1, Z)(0, L)$

Each instruction (p, A_i+, q, q') of M is simulated by a sequence of transitions in Π associated to this instruction. The transition sequence starts with a configuration where cell 1 contains p and a multiset of elements of $\{A_1, \ldots, A_k\}$. Cell 2 contains a multiset consisting of elements of $Q \setminus \{p\}$, elements of $\{r_i \mid 1 \leq i \leq 6, r \in Q\}$ and L, L_1, Z, q'_f. Each symbol in cell 2 occurs in only one copy. Symbols L, L_1 are called loop symbols, Z is an auxiliary symbol, and symbols r_i, $1 \leq i \leq 6$, $r \in Q$ are auxiliary symbols associated to r. We briefly explain how the transitions of Π work. In the following, only those symbols in each configuration that are relevant for the simulation will be listed.

The simulation starts from configuration $(0, \lambda)(1, [A_i]p)(2, p_1p_2p_3p_4p_5p_6qq')$, where $[A_i]$ denotes an arbitrary finite multiset of symbols A_i.

After performing some transitions from the above initial configuration, Π should enter the configuration $(0, \lambda)(1, [A_i]A_iq)(2, p_1p_2p_3p_4p_5p_6pq')$; or with q' in cell 1 instead of q. In this case one instance of A_i left the environment and entered cell 1, and this is denoted by $[A_i]A_i$ in cell 1 (there is one more copy of A_i in the cell). Symbol p from cell 1 and symbol q (or q') from cell 2 swapped their place. The motion of these symbols is done as follows. In the first phase of the simulation, the auxiliary symbols p_1, p and p_2 move to the environment, in this order. After then, p moves to cell 2 and a symbol A_i from the environment moves to cell 1. Meanwhile, the auxiliary symbols p_3, p_4, p_5 and p_6 leave their original location and move to cell 1. The order in which the rules are executed is governed by both the movement of the auxiliary symbols and the rules that introduce and move loop symbols (L-rules for short), i.e. the rules which generate an infinite loop. If a rule introduces a loop symbol, then an infinite loop is started in the generation, thus the generation will not end.

The next step in the simulation of the instruction is to move the symbol q (or q') to cell 1 and return the auxiliary symbols p_i, $1 \leq i \leq 6$ to cell 2. The simulation of the instruction of M ends when q is in cell 1 and the auxiliary symbols p_i are in cell 2. The correct execution order of the rule sets is again ensured by the existence of the L-rules. If the rule set is executed in some incorrect order, the generation never finishes. Note that the rules are specified such that the simulation of a new instruction can only start if the new configuration is of the form $(0, \lambda)(1, [A_i]A_iq)(2, p_1p_2p_3p_4p_5p_6pq')$. If the simulation of an instruction starts before this configuration is obtained, an infinite loop is generated within a few steps.

The following generation corresponds to a correct derivation in Π, described above.

$$
\begin{aligned}
(0, \lambda)(1, [A_i]p)(2, p_1p_2p_3p_4p_5p_6qq') &\Rightarrow_{p.1.1} (0, \lambda(1, [A_i]pp_1)(2, p_2p_3p_4p_5p_6qq') \\
&\Rightarrow_{p.2.1,p.3.1} (0, p_1)(1, [A_i]pp_2)(2, p_3p_4p_5p_6qq') \Rightarrow_{p.4.1,p.4.2} \\
(0, pp_1)(1, [A_i]p_2p_3)(2, p_4p_5p_6qq') &\Rightarrow_{p.5.1,.5.2} (0, pp_1p_2)(1, [A_i]p_3p_4)(2, p_5p_6qq') \\
&\Rightarrow_{p.5.3,p.5.4} (0, p_1p_2p_6)(1, [A_i]Ap_3p_4p_5)(2, pqq') \\
&\Rightarrow_{p.6.1,p.6.3} (0, p_2p_3p_6)(1, [A_i]Ap_4p_5q)(2, pp_1q') \\
&\Rightarrow_{p.7.1,p.7.2} (0, p_2p_5p_6)(1, [A_i]Ap_4q)(2, pp_1p_3q') \\
&\Rightarrow_{p.8.1,p.8.2} (0, p_6)(1, [A_i]Aq)(2, pp_1p_2p_3p_4p_5q') \\
&\Rightarrow_{p.9.1} (0, \lambda)(1, [A_i]Aq)(2, pp_1p_2p_3p_4p_5p_6q')
\end{aligned}
$$

An example for a derivation tree that belongs to the simulation of the instruction is depicted in the appendix, in the figure uniport_in_plus. In the figure, the reader can find the configurations which imply an infinite loop of the generation. The reader can find notations $[A, k]$ (k is a number) in the figure which denotes the multiset consisting of k instances of A.

Next, we consider the simulation of decrement. We use the same notations as above.

For any instruction $(p, A_i 0, q)$ of M, $1 \leq i \leq k$, the set of rules R consists of the following rules:

$p.1.1 : (1,p)(2,p_1) \to (1,p)(1,p_1)$

$p.2.1 : (1,p)(2,p_2) \to (1,p)(1,p_2)$ $\quad p.2.2 : (0,Z)(1,p_1) \to (0,Z)(0,p_1)$

$p.3.1 : (0,p_1)(1,p) \to (0,p_1)(0,p)$ $\quad p.3.2 : (1,p_2)(2,p_3) \to (1,p_2)(1,p_3)$

$p.4.1 : (0,p)(1,p_2) \to (0,p)(0,p_2)$ $\quad p.4.2 : (1,p_3)(2,p_4) \to (1,p_3)(1,p_4)$

$p.4.3 : (0,p_1)(2,p_4) \to (0,p_1)(0,p_4)$

$p.5.1 : (1,p_4)(2,p_5) \to (1,p_4)(1,p_5)$ $\quad p.5.2 : (1,p_3)(0,C_A) \to (1,p_3)(1,C_A)$

$p.5.1 : (2,Z)(0,p) \to (2,Z)(2,p)$ $\quad p.5.2 : (0,p_2)(2,p_6) \to (0,p_2)(0,p_6)$

$p.5.3 : (1,p_4)(2,p_6) \to (1,p_4)(1,p_6)$

$p.6.1 : (0,p_2)(1,p_3) \to (0,p_2)(0,p_3)$ $\quad p.6.2 : (2,p)(0,p_1) \to (2,p)(2,p_1)$

$p.6.3 : (1,p_5)(2,q) \to (1,p_5)(1,q)$ $\quad p.6.4 : (2,Z)(1,C_{A_i}) \to (2,Z)(2,C_{A_i})$

$p.7.1 : (0,p_2)(1,p_5) \to (0,p_2)(0,p_5)$ $\quad p.7.2 : (2,p_1)(0,p_3) \to (2,p_1)(2,p_3)$

$p.7.3 : (2,C_{A_i})(1,A_i) \to (2,C_{A_i})(2,A_i)$

$p.8.1 : (2,p_3)(0,p_2) \to (2,p_3)(2,p_2)$ $\quad p.8.2 : (2,p_1)(1,p_4) \to (2,p_1)(2,p_4)$

$p.8.3 : (0,p_5)(2,C_{A_i}) \to (0,p_5)(0,C_{A_i})$

$p.9.1 : (2,p_2)(0,p_6) \to (2,p_2)(2,p_6)$ $\quad p.9.2 : (2,p_4)(0,p_5) \to (2,p_4)(2,p_5)$

$p.l.1 : (0,p_2)(2,L) \to (0,p_2)(0,L)$ $\quad p.l.2 : (0,p_4)(2,L) \to (0,p_4)(0,L)$

$p.l.3 : (0,p_3)(2,L) \to (0,p_3)(0,L)$ $\quad p.l.4 : (0,p_5)(2,L) \to (0,p_5)(0,L)$

$p.l.5 : (1,p_2)(2,L) \to (1,p_2)(1,L)$ $\quad p.l.6 : (1,p_6)(2,L) \to (1,p_6)(1,L)$

$p.l.7 : (2,A_i)(0,L_1) \to (2,A_i)(2,L_1)$

$p.l.8 : (0,Z)(2,L_1) \to (0,Z)(0,L_1)$

$L.1 : (0,Z)(0,L) \to (0,Z)(1,L)$ $\quad L.2 : (1,Z)(1,L) \to (1,Z)(0,L)$

$L.3 : (1,Z)(1,L_1) \to (1,Z)(2,L_1)$ $\quad L.4 : (2,Z)(2,L_1) \to (2,Z)(1,L_1)$

The proof is similar to the previous one. With a special symbol C_{A_i}, we check whether or not there exists a symbol A_i in the register, i.e. the number stored in the register is zero or not. The instruction is successfully performed if no A_i can be found in the register. To simulate the instruction, we start with the configuration $(0,\lambda)(1,p)(2,p_1p_2p_3p_4p_5p_6q)$,

where C_{A_i} is a symbol in E. As in the previous case, we indicate only those symbols which are involved in the generation. To perform a correct simulation, we should obtain the configuration $(0, \lambda)(1, q)(2, p_1p_2p_3p_4p_5p_6p)$. As previously, in the first phase of the generation, symbols p_1, p, and p_2, in this order, move to the environment. After then, p and p_6 swap their places, i.e. p enters cell 2 and p_6 enters the environment. At the same step, symbols q and and an instance of C_{A_i} enter cell 1, and later C_{A_i} moves to cell 2. If there is an A_i in cell 1, then rule $p.7.3 : (2, C_{A_i})(1, A) \to (2, C_{A_i})(2, A_i)$ is performed and by applying rule $p.l.7 : (2, A_i), (0, L_1) \to (2, A_i)(2, L_1)$ the generation enters a loop. The rule set is organized in such manner that if A_i occurs in cell 1, then rule $p.7.3$ should be performed before $p.8.3$, thus the presence of C_{A_i} and A_i in the configuration certainly will imply an infinite loop in the generation. Otherwise, C_{A_i} returns to the environment by applying rule $p.8.3 : (0, p_5)(2, C_{A_i}) \to (0, p_5)(0, C_{A_i})$. Since no A_i was found in cell 1, q is in cell 1, and p is in cell 2, in the rest of the the generation the auxiliary symbols return to cell 2.

We demonstrate the generation when the register does not contain symbol A_i.

$$(0, \lambda)(1, p)(2, p_1p_2p_3p_4p_5p_6q) \Rightarrow_{p.1.1} (0, \lambda)(1, pp_1)(2, p_2p_3p_4p_5p_6q) \Rightarrow_{p.2.1,p.2.2}$$
$$(0, p_1)(1, pp_2)(2, p_3p_4p_5p_6q) \Rightarrow_{p.3.1,p.3.2} (0, pp_1)(1, p_2p_3)(2, p_4p_5p_6q) \Rightarrow_{p.4.1,p.4.2}$$
$$(0, pp_1p_2)(1, p_3p_4)(2, p_5p_6q) \Rightarrow_{p.5.1,p.5.2} (0, p_1p_2p_6)(1, C_{A_i}p_3p_4p_5)(2, pq)$$
$$\Rightarrow_{p.6.1,p.6.2,p.6.3,p.6.4} (0, p_2p_3p_6)(1, p_4p_5q)(2, C_{A_i}pp_1) \Rightarrow_{p.7.1,p.7.2}$$
$$(0, p_2p_5p_6)(1, p_4q)(2, C_{A_i}pp_1p_3) \Rightarrow_{p.8.1,p.8.2,p.8.3} (0, p_5p_6)(1, q)(2, pp_1p_2p_3p_4)$$
$$\Rightarrow_{p.9.1,p.9.2} (0, \lambda)(1, q)(2, pp_1p_2p_3p_4p_5p_6)$$

When the register contains at least one A_i, then the generation is the same as above until rule $p.7.3 : (2, C_{A_i})(1, A_i) \to (2, C_{A_i})(2, A_i)$ can be performed. Then an infinite loop will occur, since rule $p.l.7 : (2, A_i)(2, L_1) \to (2, A_i)(2, L_1)$ and rule $p : (0, Z)(2, L_1) \to (0, Z)(0, L_1)$ are L-rules that form a loop and there is no rule for moving A_i from cell 2 to some other cell. The reader can find an example for the complete derivation tree in the appendix, in figure uniport_in_zero_z.

For any instruction (p, A_i-, q) of M, $1 \leq i \leq k$, the set of rules of Π consists of the following rules:

$p.1.1 : (1, p)(2, p_1) \to (1, p)(1, p_1)$

$p.2.1 : (1, p)(2, p_2) \to (1, p)(1, p_2)$ $\quad p.2.2 : (0, Z)(1, p_1) \to (0, Z)(0, p_1)$

$p.3.1 : (0, p_1)(1, p) \to (0, p_1)(0, p)$ $\quad p.3.2 : (1, p_2)(2, p_3) \to (1, p_2)(1, p_3)$

$p.4.1 : (0, p)(1, p_2) \to (0, p)(0, p_2)$ $\quad p.4.2 : (1, p_3)(2, p_4) \to (1, p_3)(1, p_4)$

$p.4.3 : (0, p_1)(2, p_4) \to (0, p_1)(0, p_4)$

$p.5.1 : (1, p_4)(2, p_5) \to (1, p_4)(1, p_5)$ $\quad p.5.2 : (1, p_3)(0, X_{A_i}) \to (1, p_3)(1, X_{A_i})$

$p.5.3 : (2, Z)(0, p) \to (2, Z)(2, p)$ $\quad p.5.4 : (0, p_2)(2, p_6) \to (0, p_2)(0, p_6)$

$p.5.5 : (1, p_4)(2, p_6) \to (1, p_4)(1, p_6)$

$p.6.1 : (0, p_2)(1, p_3) \to (0, p_2)(0, p_3)$ $\quad p.6.2 : (2, p)(0, p_1) \to (2, p)(2, p_1)$

$p.6.3 : (1, p_5)(2, q) \to (1, p_5)(1, q)$

$p.6.4 : (0, p_6)(1, A_i) \to (0, p_6)(0, A_i)$ $\quad p.6.5 : (2, Z)(1, X_{A_i}) \to (2, Z)(2, X_{A_i})$

$p.7.1 : (0, p_2)(1, p_5) \to (0, p_2)(0, p_5)$ $\quad p.7.2 : (2, p_1)(0, p_3) \to (2, p_1)(2, p_3)$

$p.7.3 : (2, X_{A_i})(0, p_6) \to (2, X_{A_i})(2, p_6)$

$p.8.1 : (2, p_3)(0, p_2) \to (2, p_3)(2, p_2)$ $\quad p.8.2 : (2, Z)(0, p_5) \to (2, Z)(2, p_5)$

$p.8.3 : (2, p_1)(1, p_4) \to (2, p_1)(2, p_4)$ $\quad p.8.4 : (0, Z)(2, X_{A_i}) \to (0, Z)(0, X_{A_i})$

$p.l.1 : (0, p_2)(2, L) \to (0, p_2)(0, L)$ $\quad p.l.2 : (0, p_4)(2, L) \to (0, p_4)(0, L)$

$p.l.3 : (0, p_3)(2, L) \to (0, p_3)(0, L)$ $\quad p.l.4 : (0, p_6)(2, L) \to (0, p_6)(0, L)$

$p.l.5 : (1, p_2)(2, L) \to (1, p_2)(1, L)$ $\quad p.l.6 : (1, p_6)(2, L) \to (1, p_6)(1, L)$

$L.1 : (0, Z)(0, L) \to (0, Z)(1, L)$ $\quad L.2 : (1, Z)(1, L) \to (1, Z)(0, L)$

The idea of the proof is based on similar ideas to the proof of the previous case. In the environment, there exists a symbol X_{A_i} which checks whether or not there exists an instance of A_i in cell 1, and if it is the case, then both A_i and X_{A_i} move to the environment. The simulation starts with a configuration of the form $(0, \lambda)(1, [A_i]p)(2, p_1p_2p_3p_4p_5p_6q)$. (As above, $[A_i]$ denotes a finite multiset of symbols A_i.) Similarly to the previous case, in the first phase of the generation symbols p_1, p, and p_2 move to the environment and auxiliary symbols p_3 and p_4 move to cell 1. After that X_{A_i} moves to cell 1 and p_6 moves to the environment.

In the next step, p and q swap their place, one occurrence of A_i (if it exists in cell 1) enters the environment with the assistance of p_6, and meanwhile X_{A_i} moves to cell 2. After that p_6 leaves to environment and joins X_{A_i} in cell 2. Thus, the first phase of the simulation is finished. In the second phase of the simulation, X_{A_i} returns to the environment and all the auxiliary symbols p_i, $1 \leq i \leq 6$ return to their original place, namely, cell 2. If no symbol A_i exists in the register, i.e., the register stores zero, then the rule $p.6.4 : (0, p_6)(1, A_i) \to (0, p_6)(0, A_i)$ cannot be performed. In this case rule $p.l.4 : (0, p_6)(2, L) \to (0, p_6)(0, L)$ is applied and an infinite loop will occur. As in the previous cases, if the multisets of rules are applied in an incorrect order, then the simulation does not ends with the configuration with the expected form.

The generation simulating the decrement instruction when there is at least one A_i in the register is as follows.

Symbol $[A_i, -1]$ denotes that the number of elements of multiset $[A_i]$ is decreased by 1, supposing that $[A_i]$ is a nonempty multiset.

$$\begin{aligned}
(0,\lambda)(1,[A_i]p)(2,p_1p_2p_3p_4p_5p_6q) &\Rightarrow_{p.1.1} (0,\lambda)(1,[A_i]pp_1)(2,p_2p_3p_4p_5p_6q)\\
&\Rightarrow_{p.1.2,p.2.2} (0,p_1)(1,[A_i]pp_2)(2,p_3p_4p_5p_6q) \Rightarrow_{p.3.1,p.3.2}\\
(0,pp_1)(1,[A_i]p_2p_3)(2,p_4p_5p_6q) &\Rightarrow_{p.4.1,p.4.2} (0,pp_1p_2)(1,[A_i]p_3p_4)(2,p_5p_6q)\\
\Rightarrow_{p.5.4,p.5.1,p.5.2,p.5.3} (0,p_1p_2p_6)(1,[A_i]X_{A_i}p_3p_4p_5)(2,pq) &\Rightarrow_{p.6.1,p.6.2,p.6.3,p.6.4,p.6.5}\\
(0,p_2p_3p_6)(1,[A_i,-1]p_4p_5q)(2,X_{A_i}pp_1) &\Rightarrow_{p.7.1,p.7.2,p.7.3}\\
(0,p_2p_5)(1],[A_i,-1]p_4q)(2,X_{A_i}pp_1p_3p_6) &\Rightarrow_{p.8.1,p.8.2,p.8.3,p.8.4}\\
(0,\lambda)(1,[A_i,-1]q)(2,pp_1p_2p_3p_4p_5p_6)&
\end{aligned}$$

When the register stores zero, i.e. does not contain at least one A_i, then rule $p.6.4 : (0,p_6)(1,A_i) \rightarrow (0,p_6)(0,A_i)$ cannot be performed. Then L-rule $p.l.4 : (0,p_6)(2,L) \rightarrow (0,p_6)(0,L)$ will be performed instead, and an infinite loop will occur.

The reader can find an example of a complete derivation tree in the appendix, figure uniport_in_minus_z.

The rule sets above were constructed in such a way that they result in a correct derivation if and only if they perform the multisets of rules in the described order. Furthermore, if during the simulation of a transition of M in Π the simulation of some transition is started, the generation fails, i.e. ends in an infinite loop. Next, we deal with the end of a successful computation. Recall that the computation ends in M in the final configuration $(q_f, n, 0, \ldots, 0)$, where q_f is the final state of M. The configuration $(q_f, n, 0, \ldots, 0)$ of M corresponds to the configuration $(0,\lambda)(1, q_f {A_1}^n)(2, w_3q'_f)$ of Π, where $w_2 = w_3q'_fq_f$. (Remember, w_2 is the content of cell 2 in the initial state of Π.) To have only ${A_1}^n$ in cell 1, we should move q_f to cell 2 or to the environment. Let us add the rule $(2,q'_f)(1,q_f) \rightarrow (2,q'_f)(2,q_f)$ to the rule set of Π. This rule can be used after q_f appears in cell 1. It can be seen that irrespective of whether this rule is used just after the occurrence of q_f or later (for example, just after the simulation of the instruction of M ends), we will obtain the halting configuration $(0,\lambda)(1,{A_1}^n)(2,w_2)$.

This implies that $N(M) = N(\Pi)$, and thus the statement holds.

By invoking the Church-Turing thesis we obtain the following result.

Corollary 2. $NOtP_2(uin) = NRE$.

4 Conclusion

In this paper we considered GCPSs using either the conditional-uniport-in or the conditional-uniport-out rules. Surprisingly, in the case of conditional-uniport-in we succeeded to show that only two cells are sufficient for the computational completeness,

substantially improving previous results. For the case of only conditional-uniport-out rules, we proved that there exist $k > 0$ such that the family of recursively enumerable sets of natural numbers greater than or equal to k is equal to the family of sets of numbers generated by GCPSs with only conditional uniport-out rules and only two cells and their common environment.

We conjecture that computational completeness is not achieved with GCPSs that contain only one cell and use only one of these two types of rules.

Based on the above two results and the results obtained for some other types of rules (split, join, etc.), an interesting observation can be made. If the rule applies to k cells ($k = 2$, or $k =$3, or $k = 4$), then the minimum number of cells required for the universal computational power is k (plus the environment cell). Such results were obtained for conditional-uniport-in rules, and for split, join, and chain rules [8]. A similar result holds for symport2 rules [3,1]. There are no yet such results for presence-move and shift rules, but we conjecture that the above observation still holds for these cases. We also conjecture that fewer than this number of cells is not sufficient to achieve computational completeness.

In the case of Theorem 1, the final configuration contains several additional objects in the output cell. We conjecture that using a cleaning procedure like in [2] at most 2–3 additional symbols are needed.

These interesting problems are waiting for future research.

References

1. Artiom Alhazov, Maurice Margenstern, Vladimir Rogozhin, Yurii Rogozhin, and Sergey Verlan. Communicative P systems with minimal cooperation. In Giancarlo Mauri, Gheorghe Păun, Mario J. Pérez-Jiménez, Grzegorz Rozenberg, and Arto Salomaa, editors, *International Workshop WMC5, Milano, Italy, 2004, LNCS, Springer, 2005*, volume 3365 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2005.
2. Artiom Alhazov and Yurii Rogozhin. Skin output in P systems with minimal symport/antiport and two membranes. In George Eleftherakis, Petros Kefalas, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing, 8th International Workshop, WMC 2007, Thessaloniki, Greece, June 25-28, 2007 Revised Selected and Invited Papers*, volume 4860 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2007.
3. Artiom Alhazov, Yurii Rogozhin, and Sergey Verlan. Minimal cooperation in symport/antiport tissue P systems. *International Journal of Foundations of Computer Science*, 18(1):163–180, 2007.
4. Francesco Bernardini, Marian Gheorghe, Maurice Margenstern, and Sergey Verlan. Producer/consumer in membrane systems and petri nets. In S. Barry Cooper, Benedikt Löwe, and Andrea Sorbi, editors, *Computation and Logic in the Real World, Third Conference on Computability in Europe, CiE 2007, Siena, Italy, June 18-23, 2007, Proceedings*, volume 4497 of *Lecture Notes in Computer Science*, pages 43–52. Springer, 2007.
5. Erzsébet Csuhaj-Varjú, György Vaszil, and Sergey Verlan. On generalized communicating P systems with one symbol. In Marian Gheorghe, Thomas Hinze, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing - 11th International Conference, CMC 2010, Jena, Germany, August 24-27, 2010. Revised Selected Papers*, volume 6501 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2010.
6. Erzsébet Csuhaj-Varjú and Sergey Verlan. On generalized communicating P systems with minimal interaction rules. *Theoretical Computer Science*, 412(1-2):124–135, 2011.

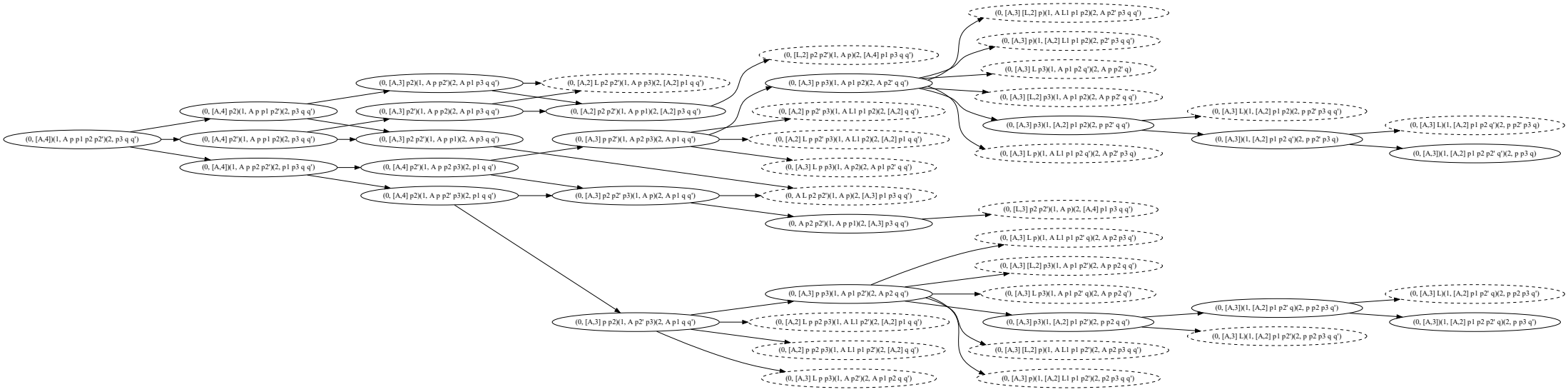
7. Erzsébet Csuhaj-Varjú and Sergey Verlan. Bi-simulation between P colonies and P systems with multi-stable catalysts. In Marian Gheorghe, Grzegorz Rozenberg, Arto Salomaa, and Claudio Zandron, editors, *Membrane Computing - 18th International Conference, CMC 2017, Bradford, UK, July 25-28, 2017, Revised Selected Papers*, volume 10725 of *Lecture Notes in Computer Science*, pages 105–117. Springer, 2017.
8. Erzsébet Csuhaj-Varjú and Sergey Verlan. Computationally complete generalized communicating P systems with three cells. In Marian Gheorghe, Grzegorz Rozenberg, Arto Salomaa, and Claudio Zandron, editors, *Membrane Computing - 18th International Conference, CMC 2017, Bradford, UK, July 25-28, 2017, Revised Selected Papers*, volume 10725 of *Lecture Notes in Computer Science*, pages 118–128. Springer, 2017.
9. Rudolf Freund, Artiom Alhazov, Yurii Rogozhin, and Sergey Verlan. Communication P systems. In Gh. Păun, G. Rozenberg, and A. Salomaa, editors, *The Oxford Handbook of Membrane Computing*, pages 118–143. Oxford University Press, 2009.
10. Rudolf Freund, Ignacio Pérez-Hurtado, Agustín Riscos-Núñez, and Sergey Verlan. A formalization of membrane systems with dynamically evolving structures. *International Journal of Computer Mathematics*, 90(4):801–815, 2013.
11. Rudolf Freund and Sergey Verlan. A formal framework for static (tissue) P systems. In George Eleftherakis, Petros Kefalas, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing, 8th International Workshop, WMC 2007, Thessaloniki, Greece, June 25-28, 2007 Revised Selected and Invited Papers*, volume 4860 of *Lecture Notes in Computer Science*, pages 271–284. Springer, 2007.
12. Marvin Minsky. *Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, 1967.
13. Andrei Păun and Gheorghe Păun. The power of communication: P systems with symport/antiport. *New Gener. Comput.*, 20(3):295–306, 2002.
14. Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors. *The Oxford Handbook of Membrane Computing*. Oxford University Press, Oxford, England, 2010.
15. Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages*. Springer-Verlag, Berlin, 1997.
16. Sergey Verlan, Francesco Bernardini, Marian Gheorghe, and Maurice Margenstern. Computational completeness of tissue P systems with conditional uniport. In Hendrik Jan Hoogeboom, Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing, 7th International Workshop, WMC 2006, Leiden, The Netherlands, July 17-21, 2006, Revised, Selected, and Invited Papers*, volume 4361 of *Lecture Notes in Computer Science*, pages 521–535. Springer, 2006.
17. Sergey Verlan, Francesco Bernardini, Marian Gheorghe, and Maurice Margenstern. Generalized communicating P systems. *Theoretical Computer Science*, 404(1-2):170–184, 2008.
18. Sergey Verlan, Rudolf Freund, Artiom Alhazov, Sergiu Ivanov, and Linqiang Pan. A formal framework for spiking neural P systems. *Journal of Membrane Computing*, 2:355–368, 2020.
19. Sergey Verlan and Gexiang Zhang. A tutorial on the formal framework for spiking neural P systems. *Natural Computing*, 22(1):181–194, 2023.

Appendix

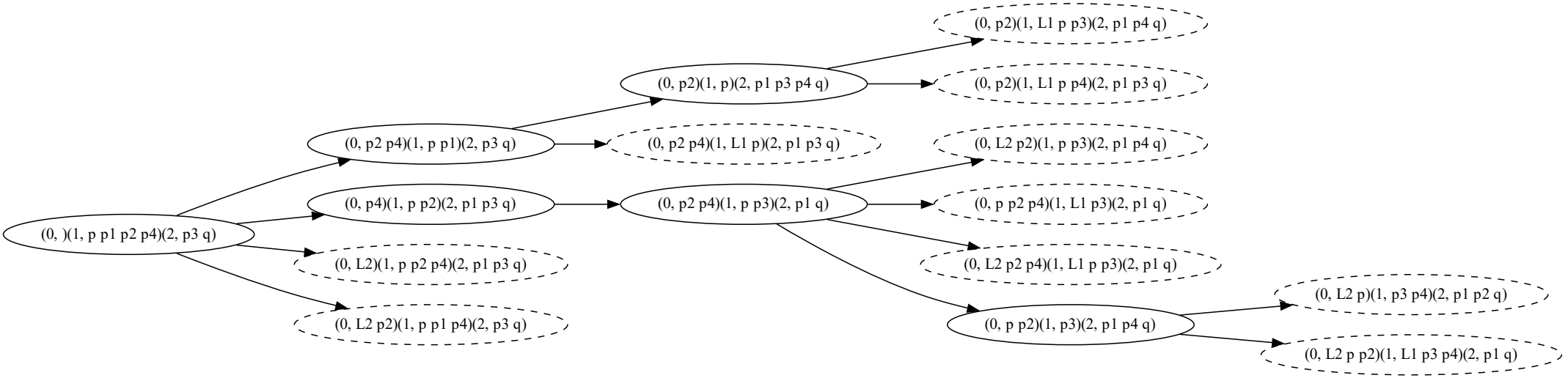
The appendix contains the configuration graph after performing all possible rule applications. These pictures were automatically computed and generated using a specially designed simulator. There are the following cases:

- uniport_out_plus — Uniport out, increment instruction
- uniport_out_minus_nz — Uniport out, decrement instruction, register is not empty
- uniport_out_minus_z — Uniport out, decrement instruction, register is empty
- uniport_out_zero_z — Uniport out, zero instruction, register is empty
- uniport_out_zero_nz — Uniport out, zero instruction, register is not empty
- uniport_in_plus — Uniport in, increment instruction
- uniport_in_minus_nz — Uniport in, decrement instruction, register is empty
- uniport_in_minus_z — Uniport in, decrement instruction, register is not empty
- uniport_in_zero_z — Uniport in, zero instruction, register is empty
- uniport_in_zero_nz — Uniport in, zero instruction, register is not empty

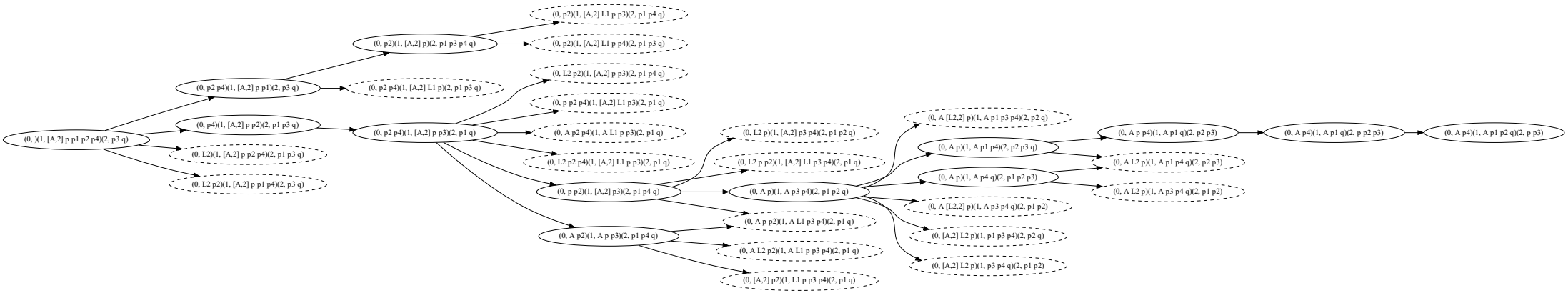
uniport_out_plus



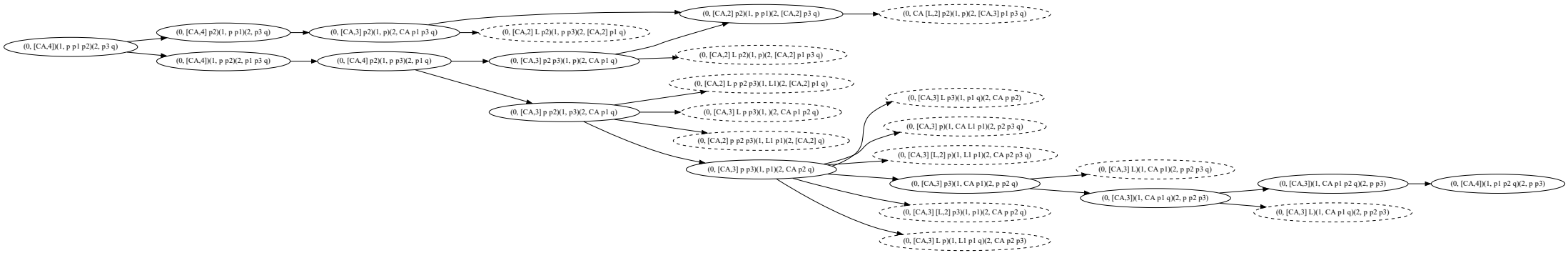
uniport_out_minus_z

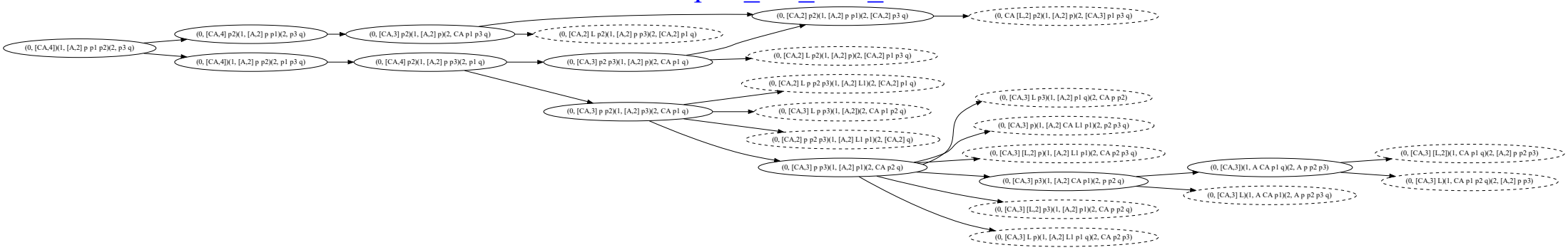


uniport_out_minus_nz

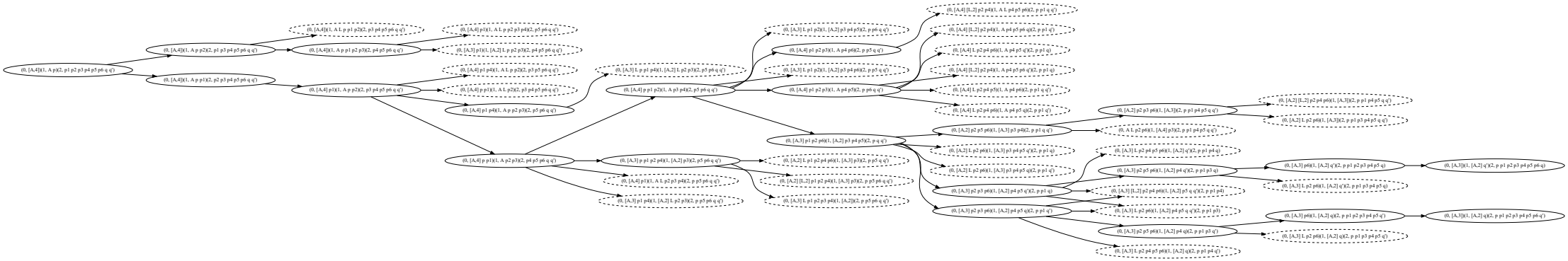


uniport_out_zero_z



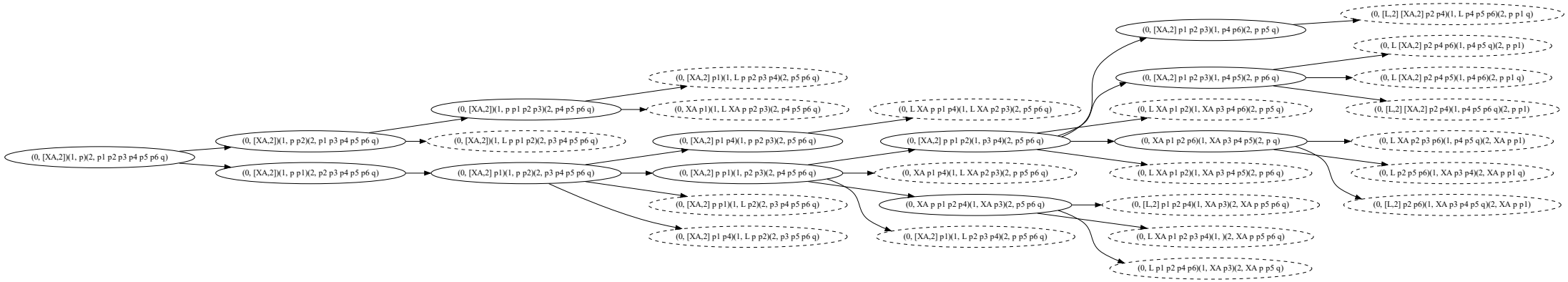

uniport_out_zero_nz
(0, [CA,4])(1, [A,2] p p1 p2)(2, p3 q)
(0, [CA,4] p2)(1, [A,2] p p1)(2, p3 q)
(0, [CA,4])(1, [A,2] p p2)(2, p1 p3 q)
(0, [CA,3] p2)(1, [A,2] p)(2, CA p1 p3 q)
(0, [CA,4] p2)(1, [A,2] p p3)(2, p1 q)
(0, [CA,2] L p2)(1, [A,2] p p3)(2, [CA,2] p1 q)
(0, [CA,2] p2)(1, [A,2] p p1)(2, [CA,2] p3 q)
(0, [CA,3] p2 p3)(1, [A,2] p)(2, CA p1 q)
(0, CA [L,2] p2)(1, [A,2] p)(2, [CA,3] p1 p3 q)
(0, [CA,2] L p2)(1, [A,2] p)(2, [CA,2] p1 p3 q)
(0, [CA,2] L p p2 p3)(1, [A,2] L1)(2, [CA,2] p1 q)
(0, [CA,3] p p2)(1, [A,2] p3)(2, CA p1 q)
(0, [CA,3] L p p3)(1, [A,2])(2, CA p1 p2 q)
(0, [CA,2] p p2 p3)(1, [A,2] L1 p1)(2, [CA,2] q)
(0, [CA,3] p p3)(1, [A,2] p1)(2, CA p2 q)
(0, [CA,3] L p3)(1, [A,2] p1 q)(2, CA p p2)
(0, [CA,3] p)(1, [A,2] CA L1 p1)(2, p2 p3 q)
(0, [CA,3] [L,2] p)(1, [A,2] L1 p1)(2, CA p2 p3 q)
(0, [CA,3] p3)(1, [A,2] CA p1)(2, p p2 q)
(0, [CA,3] [L,2] p3)(1, [A,2] p1)(2, CA p p2 q)
(0, [CA,3] L p)(1, [A,2] L1 p1 q)(2, CA p2 p3)
(0, [CA,3])(1, A CA p1 q)(2, A p p2 p3)
(0, [CA,3] L)(1, A CA p1)(2, A p p2 p3 q)
(0, [CA,3] [L,2])(1, CA p1 q)(2, [A,2] p p2 p3)
(0, [CA,3] L)(1, CA p1 p2 q)(2, [A,2] p p3)

uniport_in_plus

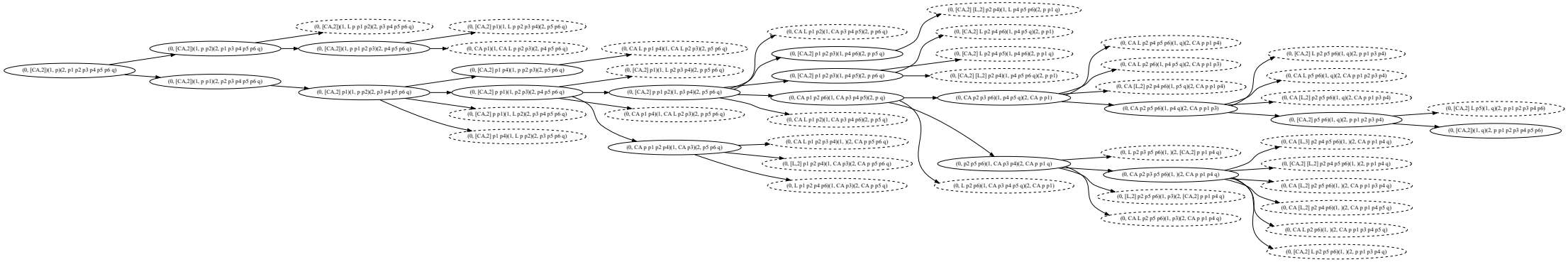


uniport_in_minus_nz

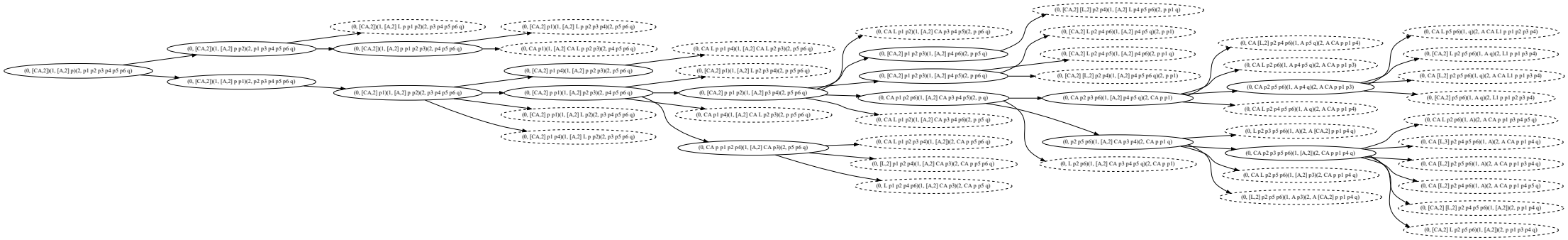
uniport_in_minus_z



uniport_in_zero_z



uniport_in_zero_nz



Simple Variants of Non-cooperative Polymorphic P Systems

Anna Kuczik and György Vaszil

Faculty of Informatics, University of Debrecen
Kassai út 26, 4028 Debrecen, Hungary
kuczik.anna@inf.unideb.hu
vaszil.gyorgy@inf.unideb.hu

Abstract. We investigate the computational power of non-cooperative polymorphic P systems with no additional ingredients. The variants we study are even more simple in the sense that the sets of possible right-hand sides of the dynamically changing rules are finite. We show that systems with this type of restriction characterize exactly the class of Parikh sets of ET0L languages.

Keywords: P systems with dynamic rules, Polymorphic P systems, P systems with non-cooperative rules, P systems with limited depth, Parikh sets of ET0L languages

1 Introduction

Polymorphic P systems were introduced in [1] motivated by the idea that the program of a computing device could be viewed as data, therefore, it could also be changed during the course of the computation. In these types of P systems, rules are not statically defined, but are dynamically inferred from the contents of pairs of membranes: The contents of one member of the pair define the multiset representing the left-hand side of the rule, the contents of the other member define the right-hand side. As the membranes can contain further membranes, the contents of the pairs, and this way the left- and right-hand sides of rules may change dynamically during the computation.

The initial results presented in [1] show the power of the model. With cooperative rules (rules with left-hand sides with more than one object) any recursively enumerable set of numbers can be generated, but non-cooperative systems (systems with rules with just one object on the left-hand side) can also generate several interesting languages, mainly based on the fact that exponential, even super-exponential growth of the number of objects inside the system can be produced.

The study of non-cooperative variants of the model was continued further in [3] by considering the case of "no ingredients", that is, when no special features (not even target indicators) are added to the system. The equivalence of so called strong and weak polymorphism was shown, left polymorphism, right polymorphism, and general polymorphism was defined. As its main contribution, [3] presented a hierarchy of computational power based on the depth of the membrane structure, but in general, the computational capabilities of the non-cooperative variant remained unclear.

In the present work, we intend to take some additional steps in this direction. We show that (1) Parikh sets of ET0L languages can be generated using non-cooperative polymorphic P systems (with no other ingredients) of depth three where all non-dynamical rules are "chain rules", and that (2) ET0L systems can generate string languages corresponding to the multiset languages of non-cooperative polymorphic P system where the set of the possible contents of regions corresponding to right-hand sides of rules is finite. This gives us an exact characterization of the class of Parikh sets of ET0L languages in terms of restricted variants of non-cooperative polymorphic P systems.

In the following we first review the necessary definitions, then present an example where a simple ET0L system is simulated, then finally generalize the idea of the simulation to a method for generating any ET0L language.

2 Preliminaries

In the following we briefly define the basic notions we will use. See [6] for more on formal language theory, and [4,5] for details about membrane computing.

An alphabet V is a finite non-empty set of symbols called letters. A string (or word) over V is a finite sequence of letters, the set of all strings over V (the free monoid generated by V) is denoted by V^*, and $V^+ = V^* \setminus \{\lambda\}$ where λ denotes the empty string. For a string $w \in V^*$, we denote by $|w|_x$ the number of occurrences of the letter $x \in V$ in w. If we fix an order $V = \{a_1, a_2, \ldots, a_n\}$ of the letters, then the vector $(|w|_{a_1}, |w|_{a_2}, \ldots, |w|_{a_n})$ is called the Parikh vector of the word $w \in V^*$.

Multisets are sets with multiplicities associated with their elements. If $\mathbb{N}$ denotes the set of nonnegative integers, then a *multiset* over a set U is a mapping $M : U \to \mathbb{N}$ where $M(a)$, for all $a \in U$, is the multiplicity of element a in the multiset M. If U is finite, $U = \{a_1, a_2, \ldots a_n\}$, then M can also be represented by a string $w = a_1^{M(a_1)} a_2^{M(a_2)} \ldots a_n^{M(a_n)}$ (and all permutations of this string) where a^j denotes the string obtained by concatenating $j \in \mathbb{N}$ occurrences of the letter $a \in V$ (with $a^0 = \lambda$).

Lindenmayer systems (or *L systems*) are parallel rewriting systems. In the following, we will use the variants which are extended, tabled, and interactionless, that is, ET0L systems in short.

An *ET0L system* is a construct $G = (V, T, U, w)$ where V is an alphabet, $T \subseteq V$ is a terminal alphabet, $w \in V^+$ is the initial word of G, and $U = (P_1, \ldots, P_m)$ where $P_i,\ 1 \leq i \leq m$, are finite sets of context-free productions over V (called *tables*), such that for each $a \in V$, there is at least one rule $a \to \alpha,\ \alpha \in V^*$ in each table. In a computational step in G, all the symbols of the current sentential form are rewritten using one of the tables of U. The language generated by G consists of all terminal strings which can be generated in a series of computational steps (a derivation) starting from the initial word, that is, $L(G) = \{u \in T^* \mid w \Rightarrow^* u\}$ where $\Rightarrow$ denotes a computational step, and $\Rightarrow^*$ is the reflexive and transitive closure of $\Rightarrow$. The family of languages generated by ET0L systems is denoted by $\mathcal{L}(ET0L)$.

It is known (see [2], for example) that for each ET0L system with an arbitrary number of tables, there exists an ET0L system with only two tables generating the very same language. This means that every task which can be solved by an arbitrary ET0L system

can also be solved by a system using two tables. Therefore, in the following we will usually assume that ET0L systems have two tables.

Moreover, since we are going to relate ET0L languages to the multiset languages of P systems, we are not interested in the string generated by the ET0L system as a sequence of letters, but only in the multiplicities of different letters, that is, in the Parikh vectors of the generated strings. We will denote by $Ps(G)$ the set of Parikh vectors corresponding the strings of $L(G)$ (also called the Parikh set of $L(G)$), and by $PsET0L$ the class of Parikh sets corresponding to the class of languages generated by ET0L systems.

Polymorphic membrane systems were introduced in [1]. Unlike in traditional membrane systems, the rules rules in polymorphic P systems are not fixed in advance, but they are defined by the contents of specific membrane regions corresponding to the left- and right-hand sides of the rule.

A *polymorphic P system* is a tuple

$$\Pi = (O, T, \mu, w_s, \langle w_{1L}, w_{1R}\rangle, \ldots, \langle w_{nL}, w_{nR}\rangle, h_o),$$

where O is the alphabet of objects, $T \subseteq O$ is the set of terminal objects, μ is the membrane structure consisting of $2n+1$ membranes labelled by a symbol from the set $H = \{s, 1L, 1R, \ldots, nL, nR\}$, the elements of the multiset w_s are the initial contents of the skin membrane, the pairs of multisets $\langle w_{iL}, w_{iR}\rangle$ correspond to the initial contents of membranes iL and iR, $1 \leq i \leq n$, and $h_o \in H$ is the label of the output membrane.

The membrane structure is usually denoted by a string of labelled and matching parentheses, but it can also be represented by a tree with its root labelled by the label of the outermost membrane, and the descendant nodes of each node labelled by the labels of membranes enclosed by the region corresponding to the given node. In the following, the number of nodes encountered during the traversal of the longest path from the root to a leaf in such a tree representation will be called the *depth* of the membrane system. (For example, the system which only has one membrane is of depth 1, while the system with two nested membranes is of depth 2.) Note that for every $1 \leq i \leq n$, the membranes iL and iR have the same parent membrane, so they are located at the same depth.

The rules of Π are not given statically in the description, but are dynamically deduced for each configuration based on the content of the membrane pairs iL and iR, $1 \leq i \leq n$. Thus, if in the configuration of the system these membranes contain the multisets u and v, then in the next step their parent membrane is transformed as if the $u \to v$ multiset rewriting rule were added to it.

If there is at least one rule in a system Π where the number of objects in u (the multiset on the left-hand side) can grow to be greater than one, then we say that Π is a *cooperative* system, otherwise, it is a *non-cooperative* system. If iL is empty for some $1 \leq i \leq n$ in a configuration, then the rule defined by the pair iL, iR is considered *disabled*, that is, no rule will be inferred from the contents of iL and iR for use in the next computational step.

A computation of the system is a series of computational steps in which the rules associated to a given region are applied in a maximally parallel way, that is, as many rules have to be applied in parallel as possible (with the restriction that each object can be rewritten by at most one rule). A P system halts (reaches a halting configuration) when no more computational steps are possible, that is, when no rule can be applied in any of the regions.

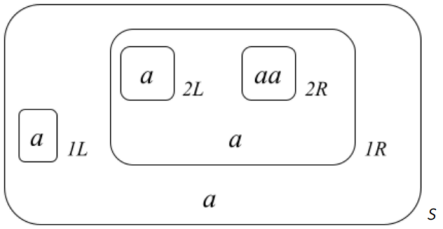


Fig. 1: The polymorphic P system Π_1 of Example 1.

The set of vectors $N(\Pi)$ generated by a P system Π with the terminal alphabet $T \subseteq O$ is the set of Parikh vectors of the strings $w \in T^*$ corresponding to the multisets of the terminal objects appearing in the output region h_o in a halting configuration of Π which is reached by a computation starting in the initial configuration of the system.

Similarly to [1], we denote non-cooperative polymorphic membrane systems and their languages as $NOP^k(polym, ncoo)$ and $\mathcal{L}(NOP^k(polym, ncoo))$ where k denotes the depth, $polym$ means polymorphism, and $ncoo$ means that the system is non-cooperative.

Now we recall an example of a simple polymorphic membrane system with superexponential growth from [1].

Example 1. Consider the polymorphic P system

$$\Pi_1 = (\{a\}, \{a\}, \mu, a, \langle a, a\rangle, \langle a, aa\rangle, s)$$

with membrane structure $\mu = [\,[\,]_{1L}\,[\,[\,]_{2L}\,[\,]_{2R}\,]_{1R}\,]_s$ as illustrated in Figure 1.

In the initial configuration, the rule corresponding to the contents of $1L, 1R$ (rule 1) is $a \to a$, and it will be applied in the skin region. The rule corresponding to $2L, 2R$ (rule 2) is $a \to aa$, and it will be applied in region $1R$. In the first step, rule 1 is applied in the skin leaving the contents of the membrane intact, and rule 2 is applied in membrane $1R$ doubling the number of a's, so rule 1 (the rule corresponding to $1L, 1R$) will be changed to $a \to aa$. In the second step, rule 1 will transform the multiset a in the skin into aa, and rule 2 will double the contents of region $1R$ again, so after this step, rule 1 becomes $a \to a^4$. In general, after k derivation steps, the contents of $1R$ will be a^{2^k}, so rule 1 will have the form $a \to a^{2^k}$. As the number of a's in the skin will be $2^{\frac{k(k-1)}{2}}$, the rate of growth of the contents of the skin membrane is superexponential.

3 Polymorphic P systems with limited depth

In this section we would like to examine the relationship of languages generated by ET0L systems and simple polymorphic P systems, where simplicity is captured by non-cooperation and limited depth. We look at an example first.

Example 2. Consider the following ET0L system $G = (V, T, U, w)$ with $V = T = \{a_1, a_2\}$, $w = a_1a_2$, and two tables $U = (P_1, P_2)$, each containing two rules

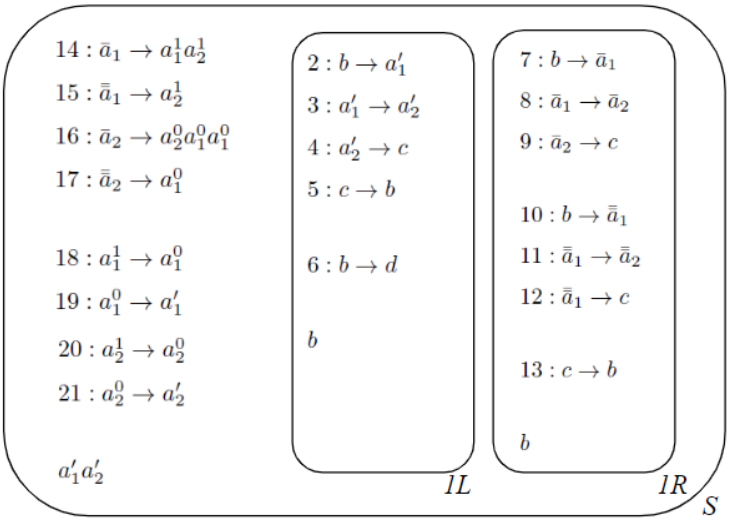


Fig. 2: The P system Π of Example 2.

$$P_1 = \{a_1 \to a_1 a_2,\ a_2 \to a_2 a_1 a_1\}, \text{ and}$$
$$P_2 = \{a_1 \to a_2,\ a_2 \to a_1\}.$$

We construct a non-cooperative polymorphic P system Π with depth 3 that can perform the choosing between rules of P_1 and P_2, and therefore simulates the operation of G.

Let $O = \{a_1, a_2, a_1', a_2', a_1^0, a_2^0, a_1^1, a_2^1, \bar{a}_1, \bar{a}_2, \bar{\bar{a}}_1, \bar{\bar{a}}_2, b, c, d, \}$, $T' = \{a_1', a_2'\}$ and

$$\Pi_2 = (O, T', \mu, w_s, \langle w_{1L}, w_{1R}\rangle, \ldots, \langle w_{21L}, w_{21R}\rangle, s)$$

where the membrane structure of Π_2 is such that the skin membrane directly contains the membranes $1L, 1R, 14L, 14R, \ldots, 21L.21R$, and the rest of the membranes are contained by $1L$ and $1R$. In more detail, μ is defined as

$$\mu = [\,[\ldots]_{1L}\,[\ldots]_{1R}\,[\,]_{14L}\,[\,]_{14R}\,\ldots[\,]_{21L}\,[\,]_{21R}\,]_s$$

where membrane $1L$ contains the inner membranes $[\,]_{2L}\ [\,]_{2R}\ldots[\,]_{6L}\ [\,]_{6R}$, and membrane $1R$ contains the inner membranes $[\,]_{7L}\,[\,]_{7R}\ldots[\,]_{13L}\,[\,]_{13R}$.

The graphical representation of μ can be seen in Figure 2 where also the initial membrane contents are depicted. Non-dynamical rules, that is, pairs of membranes $[\ w_{iL}\ ]_{iL}$, $[\ w_{iR}\ ]_{iR}$ with constant contents (contents that never change during the computation) are given in a simplified notation as $w_{iL} \to w_{iR}$. Note that in this example we only have one rule that changes dynamically, rule 1 (the rule corresponding to the regions $1L, 1R$), the other rules have the same form at each step of the computation.

The initial contents of the regions with non-constant contents are

$$w_s = a'_1 a'_2,\ w_{1L} = b,\ w_{1R} = b,$$

the initial multisets contained by the rest of the regions are given (using the simplified notation) in Figure 2.

Step	Rule 1	Contents of the Skin	Rule used in 1L	Rule used in 1R	Rules used in the Skin
1.	$b \to b$	$a'_1 a'_2$	2	7	
2.	$a'_1 \to \bar{a}_1$	$a'_1 a'_2$	3	8	
3.	$a'_2 \to \bar{a}_2$	$\bar{a}_1 a'_2$	4	9	14
4.	$c \to c$	$a_1^1 a_2^1 \bar{a}_2$	5	13	$16, 18, 20$
5.	$b \to b$	$a_1^0 a_2^0 a_2^0 a_1^0 a_1^0$	2	7 or 10	$19, 21$
6.	$a'_1 \to \bar{a}_1$ or $a'_1 \to \bar{\bar{a}}_1$	$a'_1 a'_2 a'_2 a'_1 a'_1$	...	...	...

Table 1: The polymorphic system Π_2 of example 2

The functioning of Π_2 is demonstrated in Table 1. The first column contains the step number, the second column shows the form of rule 1 (defined by the membranes $1L, 1R$) after every step, the third column contains the objects in the skin region, while the fourth, fifth, and sixth columns contain the rules we (need to) use in the corresponding computational steps.

The general idea behind the functioning of Π_2 is as follows. Rules $14 - 17$ simulate the rewriting process of the tables of G. Those with left-hand side $\bar{a}_1$ or $\bar{a}_2$ simulate the first table, those with left-hand side $\bar{\bar{a}}_1, \bar{\bar{a}}_2$ simulate the second table. The objects of the skin region correspond to the sentential form of G. Rule 1 is "dynamic", it prepares the objects of the skin membrane for the application of the rules $14 - 17$ in the appropriate order. At the beginning of a "simulating cycle", rule 1 is used to rewrite a_1 (more precisely, its variant, a'_1) to $\bar{a}_1$ or $\bar{\bar{a}}_1$ selecting this way the table to be simulated. Then, rule 1 changes to rewrite a'_2 according to the same selection while rules $14 - 17$ proceed with the actual simulation of the chosen table. The rest of the rules are needed to synchronize the whole process.

Table 1 shows how the rewriting of $a_1 a_2$ to $a_1 a_2 a_2 a_1 a_1$ by the first table of G is simulated in Π_2. In the initial state, the form of rule 1 is $b \to b$ which is not applicable because we only have objects a'_1, a'_2 in the skin region, so we have to change rule 1 in the first step.

In $1L$ we can use rule 2 ($b \to a'_1$) which rewrites b in $1L$ to a'_1 making rule 1 applicable. In parallel, we have to use rule 7 ($b \to \bar{a}_1$) or rule 10 ($b \to \bar{\bar{a}}_1$) in $1R$ depending on the table of the ET0L system we want to simulate. To simulate P_1, we must use rule 7, to simulate P_2, we must use rule 10. As we would like to simulate P_1, we use rule 7.

As can be seen in the second row of Table 1, the form of rule 1 has changed, and now we can use it in the skin region to rewrite a'_1 to $\bar{a}_1$. At the same time, the rules used in $1L$ and $1R$ ($a'_1 \to a'_2$, $\bar{a}_1 \to \bar{a}_2$, respectively) change the form of rule 1 to $a'_2 \to \bar{a}_2$ in order to be able to start rewriting a'_2-s in the next step.

After we have used rule 1 and the objects in the skin region have changed, we can use rule 14 ($\bar{a}_1 \to a_1^1 a_2^1$) which simulates the first rule from the table P_1 of G. The upper indexing of the symbols on the right-hand side starts from 1, and it will decrease in each of the following steps until the symbols are written back into the original primed form (after counting down with the indices to zero) at the appropriate step, that is, at the step when the rewriting of the other symbol, a'_2 is also finished.

Meanwhile, in step 3, rule 1 ($a'_2 \to \bar{a}_2$) is also applied to rewrite a'_2 (so the second rule of table P_1 of G can also be simulated), and rule 1 is changed to $c \to c$ (so it cannot be applied in the next step).

In the next step, with rule 16 ($\bar{a}_2 \to a_2^0 a_1^0 a_1^0$), the rule $a_2 \to a_2 a_1 a_1$ (the second rule of P_1) is simulated, while rules 18 and 20 decrement the upper indices of the objects introduced by the simulation of the previous rule, and the form of rule 1 is changed to $b \to b$.

Now, as can be seen in row 5 of Table 1, the system is ready to prepare the next simulating cycle by rewriting the objects corresponding to the sentential form of G to their original primed versions, and changing rule 1 in the appropriate way. We can return to a state that is similar to the initial state by choosing between rule 7 and rule 10 again (to simulate another step from the ET0L system), and in parallel, by rewriting a_1^0-s and a_2^0-s to a'_1-s and a'_2-s with rules 19 and 21.

The simulation of the ET0L system can be stopped at the steps which precede the table selection phase of the simulation. If both $1L$ and $1R$ contains the object b and we want to stop the simulating process, we can choose rule 6 instead of rules 7 or 10. Rule 6 shuts down the system and the simulation ends. The reason for this is that after applying rule 6, the form of rule 1 is $d \to \bar{a}_1$ or $d \to \bar{\bar{a}}_1$, none of which is applicable in the skin membrane (as no d objects can ever be present there).

The result of the computation of Π_2 is a multiset over $T' = \{a'_1, a'_2\}$ in the skin membrane (the output membrane of the system) which corresponds to the Parikh set of a string that can be generated by the ET0L system G.

Now we show how the idea presented in the example above can be generalized to arbitrary ET0L systems.

Theorem 1. $PsET0L \subseteq \mathcal{L}(NOP^3(polym, ncoo))$.

Proof. Let $G = (V, T, U, w)$ be an ET0L system, let k denote the number of letters in the alphabet, $V = \{a_1, a_2, ..., a_k\}$, and let $T = \{a_1, \ldots, a_l\} \subseteq V$ for some $l \leq k$. Without loss of generality, we assume that G has exactly two tables, $U = (P_1, P_2)$. We denote the jth rule of table 1 and table 2 (and their left- and right-hand sides) as $\alpha_{i,j} \to \beta_{i,j}$, where $\alpha_{i,j} \in V$ and $\beta_{i,j} \in V^*$, $1 \leq i \leq 2,\ 1 \leq j \leq m$. In order to simplify the notation, we assume that the cardinality of the two tables are the same, $m = |P_1| = |P_2|$. If this is not the case, then $\alpha_{1,j} \to \beta_{1,j}$ for $|P_1| < j \leq |P_2|$ (or $\alpha_{2,j} \to \beta_{2,j}$ for $|P_2| < j \leq |P_1|$) will denote the same rule as $\alpha_{1,1} \to \beta_{1,1}$ (or $\alpha_{2,1} \to \beta_{2,1}$).

Let

$$O = \{a'_i, a''_i, \bar{a}_i, \bar{\bar{a}}_i, \bar{\bar{\bar{a}}}_i \mid 1 \leq i, n \leq k\} \cup \{a_i \mid 1 \leq i \leq l\} \cup \{b, c, d\},$$
$$T = \{a_i \mid 1 \leq i \leq l\},$$

and let

$$\Pi = (O, T, \mu, w_s, \langle w_{1L}, w_{1R}\rangle, \ldots, \langle w_{pL}, w_{pR}\rangle, s)$$

where $p = 1 + (3k + 6) + 2m + k^2$ and the membrane structure of Π is such that the skin region directly contains the membranes with labels $1L, 1R$ and iL, iR for $(3k + 8) \leq i \leq p$, while the rest of the membranes are contained by $1L$ and $1R$. In more detail, μ is defined as

$$\mu = [\,[\ldots]_{1L}\,[\ldots]_{1R}\,[\,]_{(3k+8)L}\,[\,]_{(3k+8)R}\,\cdots[\,]_{pL}\,[\,]_{pR}\,]_s$$

with membrane $1L$ containing the membranes $[\,]_{2L}\,[\,]_{2R}\ldots[\,]_{(k+4)L}\,[\,]_{(k+4)R}$, and membrane $1R$ containing the membranes $[\,]_{(k+5)L}\,[\,]_{(k+5)R}\cdots[\,]_{(3k+7)L}\,[\,]_{(3k+7)R}$.

In $1L$, the number of rules depends on the number of letters in the alphabet of the ET0L system, we have to apply $k + 2$ rules for each table simulation in succession (where $k = |V|$). In general, we specify the rules for the k letters as

$$r_2 : b \to a'_1,\ r_{i+2} : a'_i \to a'_{i+1},\ \text{for } 1 \leq i \leq k-1,\ \text{and } r_{k+2} : a'_k \to c,\ r_{k+3} : c \to b.$$

These rules perform the same task as the rules of $1L$ in Example 2 do for two letters. In order to be able to finish the simulation we also need the additional rule $r_{k+4} : b \to d$ (see the end of the proof for more details).

Note that here we have used the simplified notation again for membranes with contents that remain constant for the whole computation. (Without this simplification we would have to write $\langle w_{2L}, w_{2R}\rangle$ and specify $w_{2L} = b$, $w_{2R} = a'_1$ instead of the rule $r_2 : b \to a'_1$, for example.)

Rules must be applied in $1R$ depending on the choice of the table, so we have to create rules for also P_1 and P_2. We need the rules

$$r_{k+5} : b \to \bar{a}_1,\ r_{k+5+i} : \bar{a}_i \to \bar{a}_{i+1} \text{ for } 1 \leq i \leq k-1,\ r_{2k+5} : \bar{a}_k \to c,$$

to enable the simulation of P_1, the rules

$$r_{2k+6} : b \to \bar{\bar{a}}_1,\ r_{2k+6+i} : \bar{\bar{a}}_i \to \bar{\bar{a}}_{i+1} \text{ for } 1 \leq i \leq k-1,\ r_{3k+6} : \bar{\bar{a}}_k \to c,$$

for P_2, and one more rule $r_{3k+7} : c \to b$ to finish the process.

To finish the simulation, we need to be able to stop the functioning of Π in the case when the skin region contains (primed variants of) terminal letters only, that is, objects only from the set $\{a'_1, \ldots, a'_l\} = \{a' \mid a \in T\}$. In order to do this, we introduce a mechanism that is similar to the simulation of the tables. To $1R$ we add the rules

$$r_{3k+8} : b \to \bar{\bar{\bar{a}}}_1,\ r_{3k+8+i} : \bar{\bar{\bar{a}}}_i \to \bar{\bar{\bar{a}}}_{i+1} \text{ for } 1 \leq i \leq k-1,\ r_{4k+8} : \bar{\bar{\bar{a}}}_k \to d.$$

In the skin region, we go through the objects of the alphabet applying the rules of the chosen table to each of them in a sequence, the rules for the occurrences of one specific

object at a time. For this reason, use indexed variants of the symbols which are produced in the intermediate steps, so they are able to "wait" until the rules for the rest of the objects are also applied. To achieve this effect, we add extra rules decreasing the indices in such a way that we get back to the primed form $a'_1, a'_2, \ldots, a'_k$ for all objects at the same computational step.

Recall our assumption that $|P_1| = |P_2| = m$, and that we denote the jth rule of table $i \in \{1, 2\}$ as $\alpha_{i,j} \to \beta_{i,j}$, where $\alpha_{i,j} \in V$ and $\beta_{i,j} \in V^*$, $1 \leq j \leq m$. In the following we will use the notation $\beta^l_{i,j}$ for some $1 \leq l \leq k$ to express that $\beta^l_{i,j} \in \{a^l_1, a^l_2, \ldots, a^l_k\}^*$ is the indexed version of the corresponding string $\beta_{i,j} \in \{a_1, a_2, \ldots, a_k\}^*$.

Now we add the following rules to the skin region. To simulate the rules of P_1 we need

$$r_{4k+8+j} : \bar{a}_i \to \beta^{k+1-i}_{1,j} \text{ for each rule } \alpha_{1,j} \to \beta_{1,j} \in P_1$$

where $\alpha_{1,j} = a_i$ for some $a_i \in \{a_1, a_2, \ldots a_k\}$ and $1 \leq j \leq m$.

For the simulation of P_2 we have

$$r_{4k+8+m+j} : \bar{\bar{a}}_i \to \beta^{k+1-i}_{2,j} \text{ for each rule } \alpha_{2,j} \to \beta_{2,j} \in P_2$$

where $\alpha_{2,j} = a_i$ for some $a_i \in \{a_1, a_2, \ldots a_k\}$ and $1 \leq j \leq m$.

After rewriting with the rules above, we have to use rules to count down with the indices of the objects that were produced until the last element of the alphabet is rewritten (similarly to the way we count down in the example). To achieve this, we need

$$r_{4k+8+2m+(i-1)k+n} : a^n_i \to a^{n-1}_i \text{ for } 2 \leq n \leq k, \text{ and}$$
$$r_{4k+8+2m+(i-1)k+1} : a^1_i \to a^{'}_i.$$

In order to stop the system, we use the rules

$$r_{4k+8+2m+k^2+i} : \bar{\bar{\bar{a}}}_i \to \delta_i \text{ where } \delta_i = a_i \text{ if } a_i \in T \text{ or } \delta_i = F \text{ if } a_i \notin T,$$

for $1 \leq i \leq k$ in the skin region. If the multiset in the skin region corresponding to the current sentential form of the simulated ET0L system contains primed versions of terminal only, that is, elements of the set $\{a'_1, \ldots, a'_l\}$, then these rules rewrite them to terminal objects. Otherwise, if nonterminal objects are present (that is, if the simulation should not be stopped), they introduce the nonterminal F which prevents the halting of the system because the rule

$$r_{5k+8+2m+k^2+1} : F \to F$$

is also present in the skin region.

To stop the system entirely, we also need to apply the rule $r_{k+4} : b \to d$ of $1L$. After we used this rule (and no F object is present in the skin region), the system shuts down because we never have d object in the skin region, and there are no rules applicable to d in $1L$, $1R$, or to the terminal objects in the skin region.

When Π halts, the result is a multiset over $T = \{a_1, a_2, \ldots, a_l\}$ which corresponds to a terminal string that can be generated by the ET0L system G.

To see that the P system Π cannot produce multsets that do not correspond to the Parikh set of a string generated by the ET0L system, observe the construction of Π.

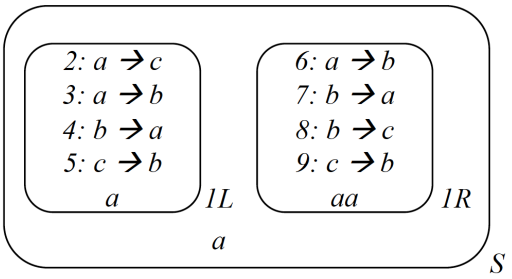


Fig. 3: The polymorphic P system Π_3 of Example 3.

Informally speaking, there are just a few points where the rule application in Π is not deterministic (apart from simulating the possibility when one symbol can be rewritten by more than one rule of the ET0L system). These nondeterministic steps involve the choice of the table of G to be simulated and the choice whether to finish or to continue the computation, so we might conclude that the behavior of the P system corresponds to an ET0L system derivation.

4 Polymorphic P systems with finite sets of instances of dynamic rules

Since there is no communication between the regions (as we consider P systems with "no ingredients"), the sequence of multisets appearing inside a given region as the computation proceeds only depends on the initial contents of the region itself. To formalize this idea, we introduce a successor relation defined on the contents of regions.

Let $\Pi = (O, T, \mu, w_s, \langle w_{1L}, w_{1R}\rangle, \ldots, \langle w_{nL}, w_{nR}\rangle, h_o)$ be a polymorphic P system, and let w_h for some $h \in \{s, 1L, 1R, \ldots, nL, nR\}$ denote the multiset contained by the region labelled by h after the jth step of the computation of Π for some $j \geq 0$. We say that w'_h is in the *successor set* of w_h, denoted as $w'_h \in \sigma_{j,h}(w_h)$, if w'_h can be obtained from w_h by the maximally parallel applications of the multiset rewriting rules associated to the region h, as can be deduced from the configuration of Π which is reached in the jth step of the computation.

If for the same w_h as above, we fix $\sigma^0_{j,h}(w_h) = w_h$ for any $j \geq 0$, and for $k \geq 0$ we have $\sigma^{k+1}_{j,h} = \sigma_{j+k,h}(\sigma^k_{j,h}(w_h))$, then we can define

$$\sigma^*_{j,h} = \bigcup_{k \geq 0} \sigma^k_{j,h}(w_h).$$

Definition 1. *Given a polymorphic P system Π as above, we say that a region h of Π is* finitely representable *or FIN-representable in short, if the set of successor multisets of the initial contents of h, w_h, is finite, that is, $\sigma^*_{0,h}(w_h)$ is finite.*

First we demonstrate the notion of FIN-representability on an example.

Example 3. Consider the polymorphic P system

$$\Pi_3 = (O, T, \mu, w_s, \langle w_{1L}, w_{1R}\rangle, \ldots, \langle w_{9L}, w_{9R}\rangle, s)$$

where $O = T = \{a, b, c\}$, and the membrane structure is $\mu = [\,[\ldots]_{1L}\,[\ldots]_{1R}\,]_s$, where the child membranes of 1L are $[\,]_{2L}\,[\,]_{2R}\ldots[\,]_{5L}\,[\,]_{5R}$ and the children of 1R are $[\,]_{6L}\,[\,]_{6R}\ldots[\,]_{9L}\,[\,]_{9R}$. Let

$$w_s = a,\ w_{1L} = a,\ w_{1R} = aa,$$

and using the simplified notation for static rules, let the rules corresponding to 1L be

$$r_2 : a \to c,\ r_3 : a \to b,\ r_4 : b \to a,\ r_5 : c \to b,$$

and the rules corresponding to 1R be

$$r_6 : a \to b,\ r_7 : b \to a,\ r_8 : b \to c,\ r_9 : c \to b,$$

as can also be seen in Figure 3.

In the following we show that both $1L$ and $1R$ are FIN-representable. Concerning $1L$, observe that $\sigma^*_{0,1L}(a) = \{a, b, c\}$ with $\sigma_{0,1L}(a) = \sigma_{j,1L}(a) = \{b, c\}$, $\sigma_{0,1L}(b) = \sigma_{j,1L}(b) = \{a\}$, and $\sigma_{0,1L}(c) = \sigma_{j,1L}(c) = \{b\}$ for all $j \geq 0$, which can be represented with the graph in Figure 4a.

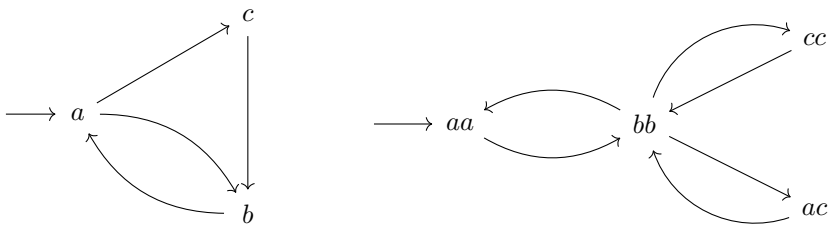
Considering $1R$, we have $\sigma_{0,1R}(aa) = \sigma_{j,1R}(aa) = \sigma_{0,1R}(cc) = \sigma_{j,1R}(cc) = \sigma_{0,1R}(ac) = \sigma_{j,1R}(ac) = \{bb\}$, and $\sigma_{0,1R}(bb) = \sigma_{j,1R}(bb) = \{aa, cc, ac\}$, thus, $\sigma^*_{0,1R}(aa) = \{aa, bb, cc, ac\}$, as can be seen on Figure 4b.

The skin region is not FIN-representable, as the dynamical rule r_1 given by the membranes labelled with $1L$ and $1R$ has more than one symbol on its right-hand side in each computational step, which means that the number of symbols in the skin region is increasing with each rule application.

Initially, $r_1 : a \to aa$, so $\sigma_{0,s}(a) = \{aa\}$. Then $r_1 \in \{c \to bb, b \to bb\}$, so it is not applicable, $\sigma^2_{0,s}(a) = \sigma_{1,s}(aa) = \{aa\}$. Then $r_1 \in \{b \to aa, b \to ac, b \to cc, a \to aa, a \to ac, a \to cc\}$, thus, $\sigma^3_{0,s}(a) = \sigma^2_{2,s}(aa) = \sigma_{3,s}(aa) = \{aaaa, aacc, aaac, cccc, ccac, acac\}$. As we see, the multiplicity of objects in the skin region keeps increasing, so this region cannot be FIN-representable according to Definition 1.

Given a non-cooperative polymorphic system as defined above, left-hand sides of rules have at most one symbol, so the membranes with labels $iL,\ 1 \leq i \leq n$, are always FIN-representable. However, the situation is different in the case of membranes $iR,\ 1 \leq i \leq n$. If at least one of the rules which is applicable (an arbitrary number of computational steps) during a computation inside some right-hand side membrane has more than one symbol on its right-hand side, then the corresponding dynamic rule has arbitrary many possible instances, and the region corresponding to its right-hand side cannot be FIN-representable. In general, any region with at least one applicable rule having more than one object on its right-hand side is not FIN-representable.

Let us denote by $NOP(polym1, ncoo, fin)$ the class of non-cooperative polymorphic membrane systems of degree $2n+1$ where all right-hand side regions, $iR,\ 1 \leq i \leq n$,



(a) region $1L$ (b) region $1R$

Fig. 4: Graphical demonstration the possible membrane contents of the P system of Example 3 with arrows indicating the initial contents and the successor relation between the multisets (which are given by their string representations).

are FIN-representable, and the skin region contains at most one rule that is non-constant (or dynamic), and let $\mathcal{L}(NOP(polym1, ncoo, fin))$ denote the languages that they generate. We can state the following theorem.

Theorem 2. $\mathcal{L}(NOP(polym1, ncoo, fin)) \subseteq PsET0L$.

Proof sketch. Let $\Pi = (O, T, \mu, w_s, \langle w_{1L}, w_{1R}\rangle, \ldots, \langle w_{nL}, w_{nR}\rangle, s)$ be a polymorphic P system, $\Pi \in NOP(polym1, ncoo, fin)$, and let us assume (without loss of generality) that the membranes that are directly contained in the skin region are labelled by the labels $1L, 1R, \ldots, kL, kR$, $k \leq n$, with $1L, 1R$ having non-constant contents.

Since both the left- and right-hand membranes $1L, 1R$ are FIN-representable, we can construct all possible sequences of rules that can be described by their contents in the particular steps of any possible sequence of computation steps that Π can perform.

Let us denote the rule $w_{1L} \to w_{1R}$ described in the initial configuration by the first pair of membranes by $r_1 : \alpha_1 \to \beta_1$. Now consider the rule set $R_{\sigma(1)} = \{u \to v \mid u \in \sigma_{0,1L}(w_{1L}),\ v \in \sigma_{0,1R}(w_{1R})\}$, and let us denote its elements by $r_{11} : \alpha_{11} \to \beta_{11}, \ldots, r_{1m} : \alpha_{1m} \to \beta_{1m}$ for $m = |R_{\sigma(1)}|$. Now consider a rule $r_{1j} \in R_{\sigma(1)}$, let $R_{\sigma(1j)} = \{u \to v \mid u \in \sigma_{1,1L}(\alpha_{1j}),\ v \in \sigma_{1,1R}(\beta_{1j})\}$, and let us denote the elements of $R_{\sigma(1j)}$ by $r_{1j1} : \alpha_{1j1} \to \beta_{1j1}, \ldots, r_{1jl} : \alpha_{1jl} \to \beta_{1jl}$ for $l = |R_{\sigma(1j)}|$.

Continuing this procedure, we get all possible rule sequences that is ever described by the membrane pair $1L, 1R$ during all possible computations performed by Π. Since both membranes are FIN-representable, all such sequences sooner or later produce $r_1 : \alpha_1 \to \beta_1$ (or $w_{1L} \to w_{1R}$ in the original notation), after which one of the other possible sequences are produced again.

Since all other rules corresponding to the skin membrane are constant, we can simulate the rewriting process with the tables of an ET0L system by including the rules corresponding to the skin membrane in different computational steps in different tables and making sure that these tables are applied in the order in which the instances of the dynamical rule appear in the above described sequences. This can be achieved by adding an extra nonterminal letter to the sentential forms of the ET0L system which can be labelled with different labels corresponding to different tables, and adding extra rules for

each table that either relabels this symbol for a table that should be applied in the next step, or introduces a trap symbol if the application of the "wrong" table is attempted.

If a configuration is halting, then no rule can be applied to the objects. This can also be checked by special tables which eliminate the special labelled nonterminal while introducing the trap symbol if there are letters present to which rules could be applied.

Corollary 1. $\mathcal{L}(NOP(polym1, ncoo, fin)) = PsET0L$.

Proof. By observing the proof of Theorem 1, we may see that for any ET0L system G, the right-hand regions of the P system $\Pi \in NOP^3(polym, ncoo)$ constructed to simulate G are FIN-representable, and the skin region contains exactly one dynamic rule. Combining this observation with Theorem 2 we obtain our statement.

Example 4. Consider the polymorphic P system of Example 3 which can be seen in Figure 3. Let us construct an ET0L system $G = (V, T', U, w)$ simulating this membrane system. The nonterminal alphabet of G includes O, a set of labelled symbols, and the trap symbol,

$$V \supseteq \{d_1, d_{11}, d_{12}, d_{111}, d_{112}, d_{113}, d_{121}, d_{122}, d_{123}, d_{1121}, d_{1122}, d_{1131}, d_{1132}, d_{1211}, d_{1212}, d_{12111}, d_{12112}, d_{12113}, d_{12114}, d_{12115}, d_{12116}, d_{12121}, d_{12122}, d_{12123}, d_{12124}, d_{12125}, d_{12126}, d_{12141}, F\}.$$

The terminals correspond to the terminal objects of the P system, $T' = \{a', b', c'\}$.

At the beginning, we construct the first table which contains the current instance of rule r_1, the relabelling rules, and additional rules for identical rewriting of b, c (because ET0L systems rewrite every symbol at every step) which we do not indicate in the tables below, for the sake of brevity. The construction of the relabelling rules is based on the number of different configurations that can be obtained after the next computational step. Since there are two possible configurations that the P system can reach, the symbol d_1 can be relabelled in two ways, either to d_{11} or d_{12}.

$$P_1 = \{d_1 \to d_x, d_y \to F \mid x \in \{11, 12\}, y \neq 1\} \cup \{a \to aa\}.$$

Possibly applicable tables following the first step can be constructed from table P_1. At the first step, in $1L$, rule $r_2 : a \to c$ or rule $r_3 : a \to b$ is applied. Depending on this choice, the content of $1L$ change and thus rule r_1. For this reason, the ET0L system has to choose between two options, so two tables have to be constructed for it. One table will be valid if rule r_2 is applied in the first step, the other table will be valid if rule r_3 is applied in the first step. In $1R$, rule $r_6 : a \to b$ is applied, so the right-hand side of rule r_1 changes to b. Moreover, since the number of configurations reachable by the next step is six (see the discussion in the next paragraph below the tables), the labelling of the d symbol can be extended in six possible ways,

$$P_{1,1} = \{d_{11} \to d_{11x}, d_y \to F \mid x \in \{1, 2, 3\}, y \neq 11\} \cup \{b \to bb\},$$
$$P_{1,2} = \{d_{12} \to d_{12x}, d_y \to F \mid x \in \{1, 2, 3\}, y \neq 12\} \cup \{c \to bb\}.$$

Now we construct the tables which can follow $P_{1,1}$. Two rules can be applied in $1R$ in the membrane system, rule $r_7 : b \to a$ and rule $r_8 : b \to c$. Depending on this choice,

the right-hand side of rule r_1 can be aa or cc or ac. Since the left-hand side of the rule can be an a symbol (due to rule r_4) three new tables have to be constructed from $P_{1,1}$, each demonstrating a possible rule application.

$$P_{1,1,1} = \{d_{111} \to d_x, d_y \to F \mid x \in \{11, 12\}, y \neq 111\} \cup \{a \to aa\},$$
$$P_{1,1,2} = \{d_{112} \to d_{112x}, d_y \to F \mid x \in \{1, 2\}, y \neq 112\} \cup \{a \to cc\},$$
$$P_{1,1,3} = \{d_{113} \to d_{113x}, d_y \to F \mid x \in \{1, 2\}, y \neq 113\} \cup \{a \to ac\}.$$

Table $P_{1,1,1}$ is similar to table P_1, therefore we don't need to build the tables further on this branch, we can go back to the level of P_1. Because of this, it can be seen that the label of d does not increase here, but changes to the same label as in P_1. The other two tables do not have similar "parents", so these branches must be continued.

Similarly, three new tables can be constructed from $P_{1,2}$ using rule r_5 to rewrite the left-hand side c to b.

$$P_{1,2,1} = \{d_{121} \to d_{1211}, d_y \to F \mid y \neq 121\} \cup \{b \to aa\},$$
$$P_{1,2,2} = \{d_{122} \to d_{1221}, d_y \to F \mid y \neq 122\} \cup \{b \to cc\},$$
$$P_{1,2,3} = \{d_{123} \to d_{1231}, d_y \to F \mid y \neq 123\} \cup \{b \to ac\}.$$

Now we continue with table $P_{1,1,2}$. Two tables are created from $P_{1,1,2}$ after the previous step.

$$P_{1,1,2,1} = \{d_{1121} \to d_x, d_y \to F \mid x \in \{121, 122, 123\}, y \neq 1121\} \cup \{c \to bb\},$$
$$P_{1,1,2,2} = \{d_{1122} \to d_x, d_y \to F \mid x \in \{111, 112, 113\}, y \neq 1122\} \cup \{b \to bb\}.$$

Table $P_{1,1,2,1}$ is similar to table $P_{1,2}$, therefore we don't need to build the tables further on this branch, just as in the case of table $P_{1,1,2,2}$ which is similar to $P_{1,1}$.

Continuing with table $P_{1,1,3}$, two tables are created.

$$P_{1,1,3,1} = \{d_{1131} \to d_x, d_y \to F \mid x \in \{121, 122, 123\}, y \neq 1131\} \cup \{c \to bb\},$$
$$P_{1,1,3,2} = \{d_{1132} \to d_x, d_y \to F \mid x \in \{111, 112, 113\}, y \neq 1132\} \cup \{b \to bb\}.$$

Table $P_{1,1,3,1}$ and $P_{1,1,3,2}$ are similar to table $P_{1,2}$ and $P_{1,1}$, therefore we don't need to build the tables further on this branch.

For tables $P_{1,2,1}$, $P_{1,2,2}$ and $P_{1,2,3}$, we need to continue all of them. It is an interesting case that when any of the three tables are applied, the same table result from each of them (this was reflected in the labels introduced).

$$P_{1,2,1,1} = \{d_{1211} \to d_{1211x}, d_y \to F \mid x \in \{1, 2, \ldots, 6\}, y \neq 1211\} \cup \{a \to bb\}.$$

From $P_{1,2,1,1}$ we have to construct six new tables, three of these already appear in the previous steps, so those branches are closed in the tree (these are the first three tables), the other three have to be continued.

$$P_{1,2,1,1,1} = \{d_{12111} \to d_{1211}, d_y \to F \mid y \neq 12111\} \cup \{b \to aa\},$$
$$P_{1,2,1,1,2} = \{d_{12112} \to d_{1221}, d_y \to F \mid y \neq 12112\} \cup \{b \to cc\},$$
$$P_{1,2,1,1,3} = \{d_{12113} \to d_{1231}, d_y \to F \mid y \neq 12113\} \cup \{b \to ac\},$$
$$P_{1,2,1,1,4} = \{d_{12114} \to d_{121141}, d_y \to F \mid y \neq 12114\} \cup \{c \to aa\},$$

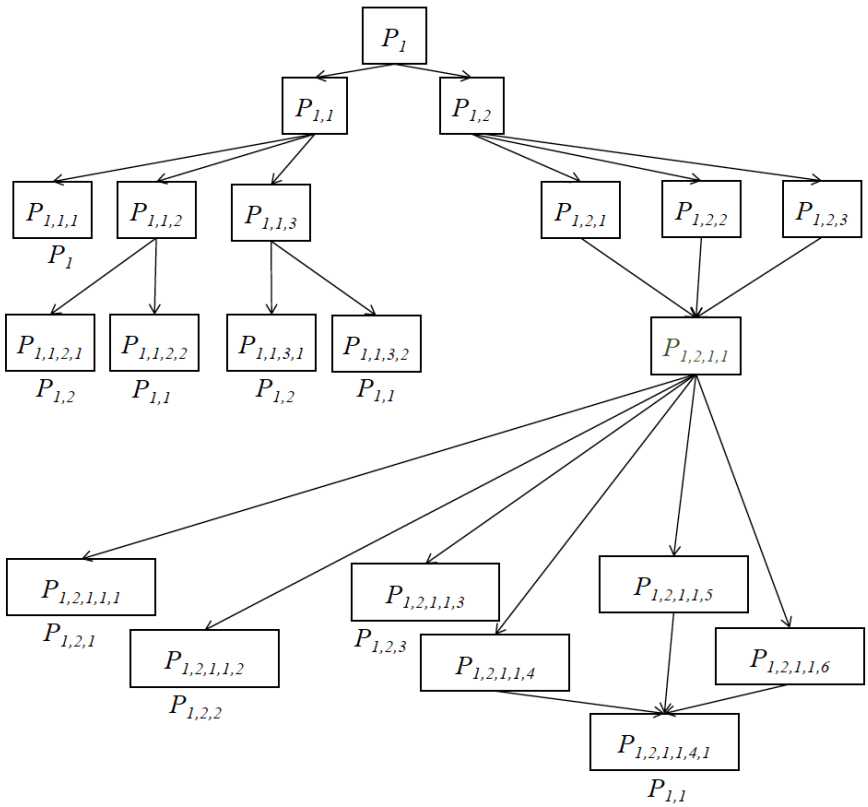


Fig. 5: The graph structure representing the order in which the tables constructed in Example 4 can be applied.

$$P_{1,2,1,1,5} = \{d_{12115} \to d_x, d_y \to F \mid x \in \{121141\}, y \neq 12115\} \cup \{c \to cc\},$$
$$P_{1,2,1,1,6} = \{d_{12116} \to d_x, d_y \to F \mid x \in \{121141\}, y \neq 12116\} \cup \{c \to ac\}.$$

There are only three tables left after the previous step that we still need to continue, these are $P_{1,2,1,1,4}$, $P_{1,2,1,1,5}$, and $P_{1,2,1,1,6}$. An interesting case occurs here as well, just like after step 3. All three tables will be followed by the same table (which is similar to $P_{1,1}$), so we can finish creating the tables on this branch as well.

$$P_{1,2,1,1,4,1} = \{d_{121141} \to d_x, d_y \to F \mid x \in \{111, 112, 113\}, y \neq 121141\} \cup \{b \to bb\}.$$

Figure 5 shows a graph which is based on the order in which the application of the tables constructed above can be applied. Nodes with no outgoing edges correspond to tables that are similar to another one which is already present in the graph.

All the objects in the P system of Example 3 are terminal, but the rules are constructed in such a way that the computations never halt, so they never produce any result at all.

If we add the rule $b \to e$ for the new object e (which cannot be further rewritten), then the computation could halt by applying the new rule. This means that all configurations where e is present in $1L$ are halting. To incorporate this feature in the simulating ET0L system, we need to follow a similar construction as above, but all tables that contain a rule with e on the left-hand side should be followed by a "finishing" table that identically rewrites a, b, c, but erases the marked nonterminals from the string.

5 Conclusion

We have shown how ET0L systems can be simulated by restricted variants of non-cooperative polymorphic P system of depth three, then showed that the simulation also works the other way around, so a precise characterization of Parikh sets of ET0L languages can be obtained in term of polymorphic P systems. Our work is intended to be an initial step in the investigation of the computing power of non-cooperative polymorphic systems with limited depth or FIN-representable regions.

References

1. Artiom Alhazov, Sergiu Ivanov, and Yurii Rogozhin. Polymorphic P systems. In Marian Gheorghe, Thomas Hinze, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 6501 of *Lecture Notes in Computer Science*, pages 81–94, Berlin, Heidelberg, 2011. Springer-Verlag.
2. Andrzej Ehrenfeucht, Grzegorz Rozenberg, and Sven Skyum. A relationship between ET0L and EDT0L languages. *Theoretical Computer Science*, 1(4):325–330, 1976.
3. Sergiu Ivanov. Polymorphic P systems with non-cooperative rules and no ingredients. In Marian Gheorghe, Grzegorz Rozenberg, Arto Salomaa, Petr Sosík, and Claudio Zandron, editors, *Membrane Computing*, volume 8961 of *Lecture Notes in Computer Science*, pages 258–273, Cham, 2014. Springer International Publishing.
4. Gheorghe Păun. *Membrane Computing: An Introduction*. Springer-Verlag, Berlin, Heidelberg, 2002.
5. Gheorghe Păun, Grzegorz Rozenberg, and Aarto Salomaa, editors. *The Oxford Handbook of Membrane Computing*. Oxford University Press, Oxford, 2010.
6. Grzegorz Rozenberg and Aarto Salomaa, editors. *Handbook of Formal Languages*. Springer-Verlag, Berlin Heidelberg, 1997.

Randomly walking with PDP systems

David Orellana-Martín[1,2][0000−0002−2892−6775], José A. Andreu-Guzmán[1][0000−0002−4109−7641], Carmen Graciani[1,2][0000−0002−3887−3494], Agustín Riscos-Núñez[1,2][0000−0002−5409−3578], and Mario J. Pérez-Jiménez[1,2][0000−0002−5055−0102]

[1] Research Group on Natural Computing, Department of Computer Science and Artificial Intelligence, Universidad de Sevilla, Avda. Reina Mercedes, s/n, 41012, Seville, Spain
{dorellana, jandreu, cgdiaz, ariscosn, marper}@us.es
[2] SCORE Laboratory, I3US, Universidad de Sevilla, Avda. Reina Mercedes, s/n, 41012, Seville Spain

Abstract. PDP systems have been widely used for real-life applications, such as systems biology, ecosystems, physics or economy, among others. Complex systems related with these areas are simulated in the framework of membrane computing by using objects and membranes that can represent entities or places in the real-life process. In physics, the study of a particle in different fluids, depending on their composition, is really interesting for several applications. A first approximation to this field is to think that particles move randomly in the available space, without any force that constrain their movements. This behaviour is known as random walk, and it is used not only in physics but in economics, genetics and ecology among other areas. In this paper, we introduce generic PDP systems for simulating the behaviour of particles, both for one-dimensional spaces and for two-dimensional spaces, using different simulators to analyze the computational resources consumed.

Keywords: Membrane Computing · PDP systems · Modelling · Random walk.

1 Introduction

Membrane Computing is a bio-inspired paradigm based on the structure and behavior of living cells. It was first introduced in [12], trying to give an alternative perspective to fields such as formal language theory and computability theory. The main devices within this framework are the so-called P systems. Several kinds of these systems have been defined, some of them are explained in [13,14]. Apart from theoretical results, such as computational completeness and efficiency, a wide range of applications have been found by specific types of P systems. As some of them can be found in [9,11,16], we want to stress the impact of probabilistic-like systems, called *PDP systems*, in the field of ecosystems. From the first successful implementation of a model for the endangered species *Gypaetus barbatus*, or bearded vultures, in the Pyrenean and Prepyrenean mountains of Catalonia [7], passing through the Pyrenean chamois [5] and the zebra mussel [8] in the fluvial reservoir of Riba-roja, to the Giant Panda conservation in China [16], it has been proved that this framework is plausible for the simulation of real-life processes. In fact, in [1,2], two simple models are defined to simulate two

classical physics problems such as the Stern-Gerlach experiment and the Uranium 238 decay. Some tools have been developed in order to simulate and validate these models, such as `P-Lingua` [17], `MeCoSim` [18] as well as some GPU based simulators in the `PMCGPU` project [19]. In this work, we want to prove the usefulness of these kinds of systems in the simulation of the dynamics of particles moving in a free manner. The paper is organized as follows: in the next section, some references of PDP systems are given. In Sections 3 and 4, two different models for one-dimensional and two-dimensional spaces are introduced. Next, we introduce some tools to graph the behaviour of particles in the space and to show a comparative of the time spent in the computations depending on the size of the space and the number of particles. The work will be closed with some conclusions and open research lines.

2 PDP systems

PDP systems are a variant of P systems inspired by the functioning of cells. Cells are able to run multiple processes in parallel in a perfectly synchronized manner, making them good candidates to be imitated for modeling complex problems. A PDP system can be viewed as a cellular tissue in which each cell is within a special compartment called environment. The cells have a particular structure hierarchy in which there is a skin membrane that defines and distinguishes the inside from the outside. In turn, inside a cell there are a number of hierarchically arranged membranes, where organelles or chemical substances capable of evolving according to specific reactions of the membrane may appear. PDP systems are probabilistic P systems, that is, the applications of their rules is commanded by a predefined probability on them. For a more exhaustive explanation of this model, see [6].

The key of software implementations of these systems are conflicts. If two or more rules compete for a resource, the algorithm has to take a strategy. The resolution of conflicts depends on the algorithm used to simulate the system. Some algorithms as the *Binomial Block Based simulation algorithm* (`BBB`) [3], the *Direct Non Deterministic distribution with Probabilities algorithm* (`DNDP`) [5] and the *Direct distribution based on Consistent Blocks Algorithm* (`DCBA`) [10] have been developed, each of them treating these conflicts in a different way. Thus, the state of the system at any time step is determined by the state of the system at the previous time step.

3 The one-dimensional model

In this model, particles going in a one-dimensional space are simulated. In this sense, particles have only two movements options: either going to the left or going to the right. Since the PDP system studied has a single environment and it is not used, we are going to define directly the behaviour of the P system under study. Let N be the number of particles simulated, and n_0 the space available. The corresponding P system is a tuple

$$\Pi = (\Gamma, \mu, \mathcal{M}_1, \mathcal{R}),$$

where:

1. $\Gamma = \{e_i \mid 0 \leq i \leq N-1\} \cup \{a_{i,j} \mid 0 \leq i \leq N-1, 0 \leq j \leq n_0 - 1\}$;
2. $\mu = [\quad]_1$;
3. $\mathcal{M}_1 = \{e_i \mid 0 \leq i \leq N-1\}$;
4. The set $\mathcal{R}$ contains the following rules:
   Objects e_i do not directly represent the particles, but they will be generated in the first step. In this case, the particles are generated in the one-dimensional space. We situate all the particles in the centre of the space. Object $a_{i,j}$ will represent that particle i will be present in the point j. The first subscript is very useful for identifying the particle from other particles in the same system. It can be used, for instance, for graphically describing the movement of the particle. Particles are not generated directly in the initial configuration since it will be useful for future research to be able to generate from the initial objects a different number of particles or to put them in a specific point of the space.
   $[\, e_i \to a_{i,j}\,]_1$ for $0 \leq i \leq N-1, j = \lfloor n_0/2 \rfloor$
   From the second configuration, we simulate the random movement of the particle. Particles have two possible actions: either they move to the left or they move to the right. Since the extreme points of the space are limits and they are not joint with each other, there are two exceptions to this rule: when a particle is at the leftmost point or at the rightmost point. In these cases, the only option for the particle is to go to the right or to the left, respectively.
   $$\left.\begin{array}{l}[\, a_{i,j}\,]_1 \overset{1/2}{\to} [\, a_{i,j+1}\,]_1 \\ [\, a_{i,j}\,]_1 \overset{1/2}{\to} [\, a_{i,j-1}\,]_1\end{array}\right\} \text{for } \begin{cases} 0 \leq i \leq N-1 \\ 1 \leq j \leq n_0 - 2 \end{cases}$$
   $$\left.\begin{array}{l}[\, a_{i,n_0-1} \to a_{i,n_0-2}\,]_1 \\ [\, a_{i,0} \to a_{i,1}\,]_1\end{array}\right\} \text{for } 0 \leq i \leq N-1$$

4 The two-dimensional model

In this model, the behaviour of a free movement of particles in a two-dimensional space is simulated. We simulate a two-dimensional space with "walls"; that is, particles cannot pass through the limits of the space. Let N be the number of particles simulated. Let n_0 and n_1 be the space available in the x axis and in the y axis, respectively; that is, if a particle is situated in the (i, j) coordinate, $0 \leq i \leq n_0 - 1$ and $0 \leq j \leq n_1 - 1$. The corresponding P system is a tuple

$$\Pi = (\Gamma, \mu, \mathcal{M}_1, \mathcal{R}),$$

where:

1. $\Gamma = \{e_i \mid 0 \leq i \leq N-1\} \cup$
   $\{a_{i,j,k} \mid 0 \leq i \leq N-1, 0 \leq j \leq n_0 - 1, 0 \leq j \leq n_0 - 1\}$;
2. $\mu = [\quad]_1$;
3. $\mathcal{M}_1 = \{e_i \mid 0 \leq i \leq N-1\}$;
4. The set $\mathcal{R}$ contains the following rules:
   Objects e_i will be transformed into objects $a_{i,j,k}$ in the first computational step. As in the one-dimensional case, we situate all the particles in the centre of the space.

Object $a_{i,j,k}$ will represent that particle i will be present in the point (j,k). It is interesting to keep the position as subscripts for having the possibility of creating higher-dimensional spaces with the same structure. Particles are not generated directly in the initial configuration since it will be useful for future research to be able to generate from the initial objects a different number of particles or to put them in a specific point of the space.

$[\, e_i \to a_{i,j,k} \,]_1$ for $0 \leq i \leq N-1, j = \lfloor n_0/2 \rfloor, k = \lfloor n_1/2 \rfloor$

From this point, the free movement of the particles will be simulated. In this case, instead of having two different actions, particles can move in two dimensions; that is, they have four possible actions (they are not able to move diagonally). There exist two exceptions to this rule are the following: First, when a particle is situated in an edge of the space. In this case, either they move in one of the two directions still being in the edge or they move in perpendicular directions to the edge (3 possible actions). Last, when a particle is situated in a corner of the space. In this case, it has the option to go to each of the edges that finish at that corner (2 possible options).

$$\left.\begin{array}{l} [\, a_{i,j,k} \,]_1 \xrightarrow{1/4} [\, a_{i,j-1,k} \,]_1 \\ [\, a_{i,j,k} \,]_1 \xrightarrow{1/4} [\, a_{i,j,k+1} \,]_1 \\ [\, a_{i,j,k} \,]_1 \xrightarrow{1/4} [\, a_{i,j+1,k} \,]_1 \\ [\, a_{i,j,k} \,]_1 \xrightarrow{1/4} [\, a_{i,j,k-1} \,]_1 \end{array}\right\} \text{for } \begin{cases} 0 \leq i \leq N-1 \\ 1 \leq j \leq n_0 - 2 \\ 1 \leq k \leq n_1 - 2 \end{cases}$$

$$\left.\begin{array}{l} [\, a_{i,j,0} \,]_1 \xrightarrow{1/3} [\, a_{i,j-1,0} \,]_1 \\ [\, a_{i,j,0} \,]_1 \xrightarrow{1/3} [\, a_{i,j,1} \,]_1 \\ [\, a_{i,j,0} \,]_1 \xrightarrow{1/3} [\, a_{i,j+1,0} \,]_1 \end{array}\right\} \text{for } \begin{cases} 0 \leq i \leq N-1 \\ 1 \leq j \leq n_0 - 2 \end{cases}$$

$$\left.\begin{array}{l} [\, a_{i,j,n_1-1} \,]_1 \xrightarrow{1/3} [\, a_{i,j-1,n_1-1} \,]_1 \\ [\, a_{i,j,n_1-1} \,]_1 \xrightarrow{1/3} [\, a_{i,j+1,n_1-1} \,]_1 \\ [\, a_{i,j,n_1-1} \,]_1 \xrightarrow{1/3} [\, a_{i,j,n_1-2} \,]_1 \end{array}\right\} \text{for } \begin{cases} 0 \leq i \leq N-1 \\ 1 \leq j \leq n_0 - 2 \end{cases}$$

$$\left.\begin{array}{l} [\, a_{i,n_0-1,k} \,]_1 \xrightarrow{1/3} [\, a_{i,n_0-2,k} \,]_1 \\ [\, a_{i,n_0-1,k} \,]_1 \xrightarrow{1/3} [\, a_{i,n_0-1,k+1} \,]_1 \\ [\, a_{i,n_0-1,k} \,]_1 \xrightarrow{1/3} [\, a_{i,n_0-1,k-1} \,]_1 \end{array}\right\} \text{for } \begin{cases} 0 \leq i \leq N-1 \\ 1 \leq k \leq n_1 - 2 \end{cases}$$

$$\left.\begin{array}{l} [\, a_{i,0,k} \,]_1 \xrightarrow{1/3} [\, a_{i,0,k+1} \,]_1 \\ [\, a_{i,0,k} \,]_1 \xrightarrow{1/3} [\, a_{i,1,k} \,]_1 \\ [\, a_{i,0,k} \,]_1 \xrightarrow{1/3} [\, a_{i,0,k-1} \,]_1 \end{array}\right\} \text{for } \begin{cases} 0 \leq i \leq N-1 \\ 1 \leq k \leq n_1 - 2 \end{cases}$$

$$\left.\begin{array}{l} [\, a_{i,0,n_1-1} \,]_1 \xrightarrow{1/2} [\, a_{i,1,n_1-1} \,]_1 \\ [\, a_{i,0,n_1-1} \,]_1 \xrightarrow{1/2} [\, a_{i,0,n_1-2} \,]_1 \end{array}\right\} \text{for } 0 \leq i \leq N-1$$

$$\left.\begin{array}{l} [\, a_{i,n_0-1,n_1-1} \,]_1 \xrightarrow{1/2} [\, a_{i,n_0-2,n_1-1} \,]_1 \\ [\, a_{i,n_0-1,n_1-1} \,]_1 \xrightarrow{1/2} [\, a_{i,n_0-1,n_1-2} \,]_1 \end{array}\right\} \text{for } 0 \leq i \leq N-1$$

$$\left.\begin{array}{l} [\, a_{i,0,0} \,]_1 \xrightarrow{1/2} [\, a_{i,0,1} \,]_1 \\ [\, a_{i,0,0} \,]_1 \xrightarrow{1/2} [\, a_{i,1,0} \,]_1 \end{array}\right\} \text{for } 0 \leq i \leq N-1$$

$$\left.\begin{array}{l} [\, a_{i,n_0-1,0}\,]_1 \overset{1/2}{\rightarrow} [\, a_{i,n_0-2,0}\,]_1 \\ [\, a_{i,n_0-1,0}\,]_1 \overset{1/2}{\rightarrow} [\, a_{i,n_0-1,1}\,]_1 \end{array}\right\} \text{for } 0 \leq i \leq N-1$$

5 Evolution of the systems

The evolution of the systems is easy to follow since each particle i is represented by an object $a_{i,j}$ (respectively, $a_{i,j,k}$) that represents that the particle i is in the point (j) (resp., (j,k)) of the one-dimensional (resp., two dimensional) space. In Figure 1 we can observe how 10 particles move through the space along all the computation. Take into account that the x axis represent the time steps and the y axis represent the point where the particles are placed.

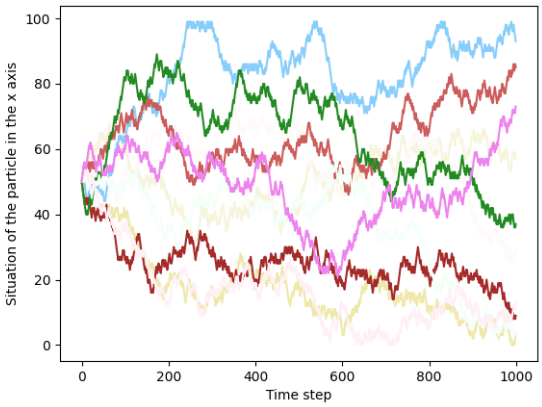


Fig. 1: Evolution of 10 particles in a one-dimensional space

6 Study of the complexity of the systems

We have made simulations for different number of particles and different sizes of space. One of the first interesting points to investigate is the impact of the algorithm used to simulate this program. We are comparing the three inference motors implemented in P-Lingua: BBB, DNDP4 [3] and DCBA. In Figures 2 and 3, we can see that BBB is, as expected, the most efficient algorithm, and that the time spent while using the DCBA algorithm drastically increments as the size or the number of particles increment. The DNDP4, instead, keeps a similar time independent of the size or the number of particles. While the first two figures are referred to one-dimensional spaces, Figures 4 and 5 give the

[3] This is the last version of the DNDP algorithm.

corresponding times for the two-dimensional spaces. These times have been calculated using the running time of the simulation of the systems in a Intel Core i5-8250U CPU @ 1.60GHz processor. In the future, we will try to update this comparative with parallel algorithms, such as the ABCD simulator [4]. This simulator is part of the PMC-GPU project [19], and it parallelizes the concepts included in the DNDP algorithm. It would be interesting to see if, for a big lattice or a high number of particles, the parallelism improves the behaviour of the compared algorithms.

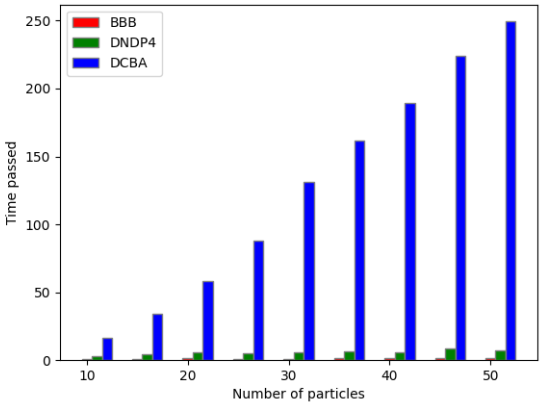


Fig. 2: Time spent in 1000 steps of computation of a one-dimensional space of size (50) depending on the number of particles

7 Conclusions and future work

In the framework of PDP systems, several applications have been found by using them as a modelling tool for real-life processes. In this sense, the study of particles moving in a space is interesting for Monte Carlo processes and different types of movements of particles in different fluids. Several research lines are open in this field. On the one hand, new models can arise to simulate the behaviour of different types of movement such as Brownian motion. On the other hand, different approaches depending on the simulators are interesting to study.

Virus machines [4] are interesting models of computation inspired by the spread and replication of viruses between hosts. In [15], authors introduce a variant of virus machines, called stochastic virus machines, where the instructions are connected to hosts instead of channels, and one of the channels going out of the host will be opened depending on probability functions associated to them. A software for this model can be

[4] https://sourceforge.net/projects/pmcgpu/files/ABCD-GPU/

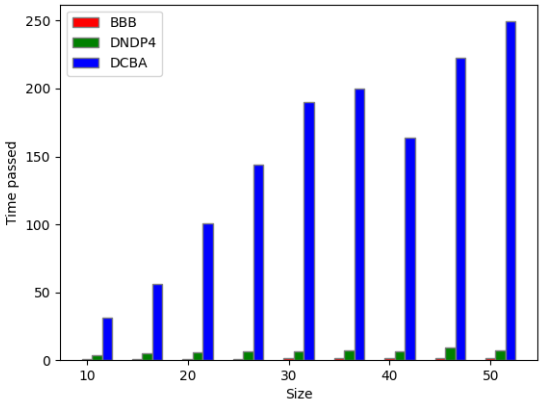


Fig. 3: Time spent in 1000 steps of computation of 50 particles in a one-dimensional space depending on the size

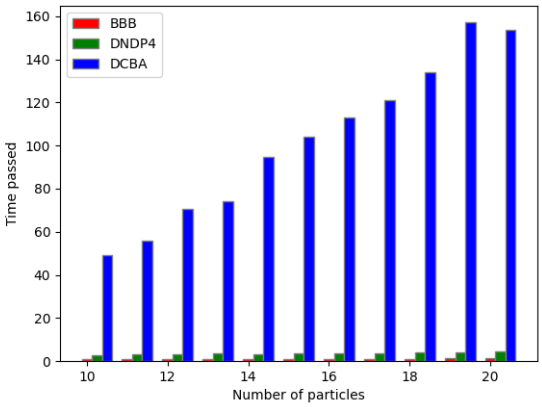


Fig. 4: Time spent in 1000 steps of computation of a two-dimensional space of size $(10, 10)$ depending on the number of particles

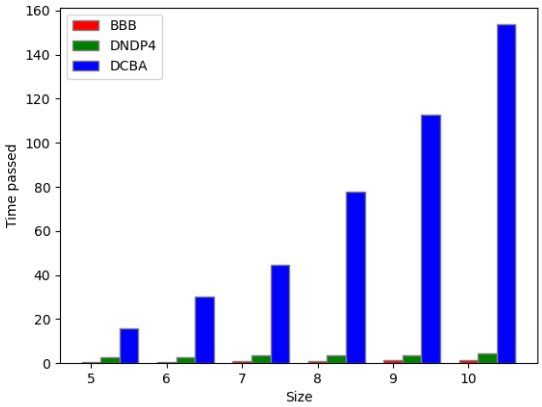


Fig. 5: Time spent in 1000 steps of computation of 20 particles in a two-dimensional space depending on the size

found in [20]. It would be interesting to see if virus machines are good to simulate the behaviour studied in this framework, and how the performance changes with respect to the framework of membrane computing.

Besides, authors have been working lately in a stochastic version of virus machines [4], called stochastic virus machines [15], that use probabilities besides weights in the channels between

We are working on new generators to automatically generate different systems depending on the number of dimensions, number of particles and so on.

Acknowledgements

The research described in this work is supported by the Zhejiang Lab BioBit Program (Grant No. 2022BCF05). D. Orellana-Martín acknowledges Contratación de Personal Investigador Doctor. (Convocatoria 2019) 43 Contratos Capital Humano Línea 2. Paidi 2020, supported by the European Social Fund and Junta de Andalucía.

References

1. M. Arazo, M. Barroso, O. De la Torre, L. Moreno, A. Ribes, P. Ribes, A. Ventura, D. Orellana-Martín. Stern-Gerlach Experiment. *Proceedings of the Fourteenth Brainstorming Week on Membrane Computing*, February 1 - 5, 2016, Sevilla, Spain, 101-112.
2. M. Arazo, M. Barroso, O. De la Torre, L. Moreno, A. Ribes, P. Ribes, A. Ventura, D. Orellana-Martín. Uranium-238 decay chain. *Proceedings of the Fourteenth Brainstorming Week on Membrane Computing*, February 1 - 5, 2016, Sevilla, Spain, 113-130.
3. M. Cardona, M.À. Colomer, A. Margalida, A. Palau, I. Pérez-Hurtado, M.J. Pérez-Jiménez, D. Sanuy. A computational modeling for real ecosystems based on P systems. *Natural Computing*, **10**, 1 (2011), 39-53.

4. X. Chen, M.J. Pérez-Jiménez, L. Valencia-Cabrera, B. Wang, X. Zeng. Computing with viruses. *Theoretical Computer Science*, **623** (2016), 146-159.
5. M.À. Colomer, S. Lavín, I. Marco, A. Margalida, I. Pérez-Hurtado, M.J. Pérez-Jiménez, D. Sanuy, E. Serrano, L. Valencia-Cabrera. Modeling population growth of Pyrenean Chamois (Rupicapra p. pyrenaica) by using P systems. *Membrane Computing, 11th International Conference, CMC 2010, Jena, Germany, August 24-27, 2010, Revised Selected Papers*. Lecture Notes in Computer Science, 6501 (2011), 144-159. A preliminary version in *Proceedings of the Eleventh International Conference on Membrane Computing*, Jena, Germany, 24-27 August 2010, Verlag ProBusiness Berlin, 2010, ISBN 978-3-86805-721-8, pp. 121-135.
6. M.À. Colomer, A. Margalida, M.J. Pérez-Jiménez. Population Dynamics P System (PDP) Models: A Standardized Protocol for Describing and Applying Novel Bio-Inspired Computing Tools. *PLOS ONE*, **8** (4): e60698 (2013).
7. M.À. Colomer, A. Margalida, D. Sanuy, M.J. Pérez-Jiménez. A bio-inspired computing model as a new tool for modeling ecosystems: The avian scavengers as a case study. *Ecological modelling*, **222**, 1 (2011), 33-47.
8. M.À. Colomer, A. Margalida, L. Valencia, A. Palau. Application of a computational model for complex fluvial ecosystems: The population dynamics of zebra mussel Dreissena polymorpha as a case study. *Ecological Complexity*, **20** (2014), 116-126.
9. P. Frisco, M. Gheorghe, M. J. Pérez-Jiménez. Applications of Membrane Computing in Systems and Synthetic Biology. *Emergence, Complexity and Computation Series*, 2014.
10. M.Á. Martínez, I. Pérez-Hurtado, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, Á. Romero-Jiménez, C. Graciani, A. Riscos-Núñez, M.À. Colomer, M.J. Pérez-Jiménez. DCBA: Simulating population dynamics P systems with proportional objects distribution. *Membrane Computing, 13th International Conference, CMC 2012 Budapest, Hungary, August 28-31, 2012, Revised Selected Papers*. Lecture Notes in Computer Science, 7762 (2013), 257-276. A preliminary version in *Proceedings of the 13th International Conference on Membrane Computing*, Budapest, Hungary, August 28-31, 2012, pp. 291-310.
11. L. Pan, Gh. Paun, M. J. Pérez-Jiménez, T. Song. Bio-inspired Computing: Theories and Applications. *Communications in Computer and Information Science Series*, 2014.
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. Gh. Păun. Membrane Computing: An introduction. *Springer Natural Computing Series*, 2002.
14. Gh. Paun, G. Rozenberg, A. Salomaa. The Oxford Handbook of Membrane Computing. *Oxford University Press*, Oxford, U.K., 2009.
15. A. Ramírez-de-Arellano, J.A. Rodríguez-Gallego, D. Orellana-Martín, S. Ivanov. Stochastic Virus Machines. *Proceedings of the Nineteenth Brainstorming Week on Membrane Computing*, January 24 - 27, 2023, Sevilla, Spain, 79-90.
16. G. Zhang, M. J. Pérez-Jiménez, M. Gheorghe. Real-life applications with Membrane Computing. *Emergence, Complexity and Computation Series*, 2017.
17. http://www.p-lingua.org/wiki/index.php/Main_Page
18. http://www.p-lingua.org/mecosim/
19. https://sourceforge.net/projects/pmcgpu/
20. https://github.com/RGNC/virusmachines

Solving the SAT problem using spiking neural P systems with coloured spikes and division rules*

Prithwineel Paul and Petr Sosík

Institute of Computer Science, Faculty of Philosophy and Science, Silesian University in Opava, Czech Republic
prithwineelpaul@gmail.com

Abstract. Spiking neural P systems (SNPS) are variants of the third-generation neural networks. In the last few decades, different variants of SNPS models have been introduced. In most of the SNPS models, spikes are represented using an alphabet with just one letter. In this paper we use a deterministic SNPS model with coloured spikes (i.e., the alphabet representing spikes contains multiple letters), together with neuron division rules to demonstrate an efficient solution to the SAT problem. As a result, we provide a simpler construction with significantly less class resources to solve the SAT problem in comparison to previously reported results using SNPSs.

Keywords: Spiking neural P system · Neuron division · Coloured spikes · SAT problem.

1 Introduction

Membrane computing is a well-known natural computing model. The computing models in membrane computing are inspired by the working of biological cells. In the last decades, many researchers have constructed different variants of membrane computing models inspired by different biological phenomena [14]. One such type of cell inspiring many computational models is the neuron. The structure and function of biological neurons communicating via sending impulses (spikes) was the main motivation behind the construction of a popular variant of membrane computing model known as the *spiking neural P system* (SNPS) [5,10]. Since SNNs (i.e., spiking neural networks) [3] belong to third-generation neural networks, SNPS models can also be considered third-generation neural networks.

Membrane computing models are mainly categorised into three types, i.e, (1) cell-like; (2) tissue-like; (3) neural-like. One of the most popular directions of research in membrane computing is solving computationally hard problems using different variants of P systems. A comprehensive survey on the use of variants of membrane computing models for solving NP-hard problems, i.e., NP-complete (SAT, SUBSET-SUM) and PSPACE-complete problems (QSAT, Q3SAT) can be found in [16,19]. For instance, a SNPS with pre-computed resources has been used to solve QSAT, Q3SAT problems [8],

*Supported by the Silesian University in Opava under the Student Funding Plan, project SGS/11/2023.

SUBSET-SUM problem [4,9,11], SAT & 3SAT problems [6,11] in a polynomial or even linear time. An SNPS with neuron division and budding can solve the SAT problem in a polynomial time with respect to the number of literals n and the number of clauses m. Moreover, these SNP systems can solve the problem in a deterministic manner [12,22]. Similarly, an SNPS with budding rules [7] and an SNPS with division rules [20] can solve the SAT problem in polynomial time and a deterministic manner. Furthermore, SNPSs with structural plasticity and time-free SNPS models can solve the Subset Sum problem in a feasible time [2,17]. In [1] it has been proved that SNPS with astrocytes producing calcium can solve the Subset Sum problem in a polynomial time. Recently, a linear time uniform solution for the Boolean SAT problem using self-adapting SNPS with refractory period and propagation delay is derived in [23]. Finally, SNPS models have been used to solve maximal independent set selection problems from distributed computing [21].

In this paper we construct a new variant of SNPS model, i.e., spiking neural P system with coloured spikes and division rules. The idea of coloured spikes was introduced by Song et. al. in [18] to simplify the construction of a SNPS solving complex tasks. Division rules were introduced for the first time in [13,12] to solve the SAT problem using SNPS with division rules and budding. Both the spiking rules and the division rules used here to solve the SAT problem are deterministic in nature. In this paper we show that, with this combined variant, we can solve the SAT problem efficiently using a lower number of steps as well as less amount of other resources.

The paper is organised in the following manner: in Section 2, we discuss the structure and function of SNP system with coloured spikes and neuron division. In Section 3 we describe a solution to the SAT problem using this variant of SNPS. In Section 3 we give a brief comparison of descriptional and computational complexity of two other SNPS models with division rules, in the amount of resources necessary to solve the SAT problem. Section 5 is conclusive in nature.

2 Spiking neural P system with coloured spikes and neuron division

In this section we introduce the new variant of SNPS model, i.e., spiking neural P systems with coloured spikes and neuron division, which we mentioned in Section 1. More precisely, this variant combines properties of two existing SNPS models, i.e., SNPS with neuron division [12] and SNPS with coloured spikes [18]. Division rules are used to obtain an exponential workspace in polynomial time to apply the strategy of trading space for time, while the use of coloured spikes allows for a simpler and more efficient construction of the SNPS. In the sequel we denote by $L(E)$ the language associated to E, where E is a regular expression over an alphabet S.

Definition 1. *A spiking neural P system with coloured spikes and neuron division of degree $m \geq 1$ is a construct of the form $\Pi = (S, H, syn, \sigma_1, \sigma_2, \ldots, \sigma_m, R, in, out)$ where*

- $m \geq 1$ *represents the initial degree, i.e., the number of neurons initially present in the system;*
- $S = \{a_1, a_2, \ldots, a_g\}, g \in \mathbb{N}$ *is the alphabet of spikes of different colours;*

- H *is the set containing labels of the neurons;*
- $syn \subseteq H \times H$ *represents the synapse dictionary between the neurons where* $(i,i) \notin syn$ *for* $i \in H$.
- $\sigma_1, \sigma_2, \ldots, \sigma_m$ *are neurons initially present in the system, with* $\{1,\ldots,m\} \subseteq H$, *where each neuron* $\sigma_i = \langle n_1^i, n_2^i, \ldots, n_g^i \rangle$, *for* $1 \leq i \leq m$, *contains initially* $n_j^i \geq 0$ *spikes of type* a_j $(1 \leq j \leq g)$;
- R *represents the set containing the rules of the system* Π. *Each neuron labelled* $i \in H$ *contains rules denoted by* $[r]_i$. *The rules in* R *are divided into three categories:*
  - ***Spiking rule:*** $[E/a_1^{n_1}a_2^{n_2}\ldots a_g^{n_g} \to a_1^{p_1}a_2^{p_2}\ldots a_g^{p_g}; d]_i$ *where* $i \in H$, E *is a regular expression over* S; $n_i \geq p_i \geq 0$ $(1 \leq i \leq g)$; $d \geq 0$ *is called delay. Furthermore,* $p_i > 0$ *for at least one* i, $1 \leq i \leq g$.
  - ***Forgetting rule:*** $[a_1^{t_1}a_2^{t_2}\ldots a_n^{t_n} \to \lambda]_i$ *where* $i \in H$, *and* $a_1^{t_1}a_2^{t_2}\ldots a_n^{t_n} \notin L(E)$ *for each regular expression* E *associated with any spiking rule present in the neuron* i;
  - ***Neuron division rule:*** $[E]_i \to []_j \,||\, []_k$ *where* E *is a regular expression over* S *and* $i,j,k \in H$.
- in *and* out *represent the input and output neurons, respectively.*

A *spiking rule* $[E/a_1^{n_1}a_2^{n_2}\ldots a_g^{n_g} \to a_1^{p_1}a_2^{p_2}\ldots a_g^{p_g}; d]_i$ is applicable when the neuron σ_i contains spikes $a_1^{c_1}a_2^{c_2}\ldots a_n^{c_n} \in L(E)$ and $c_j \geq n_j$ $(1 \leq j \leq g)$. After application of the rule, n_j copies of the spike a_j $(1 \leq j \leq g)$ are consumed while $(c_j - n_j)$ copies remains inside the neuron σ_i. Furthermore, p_j spikes $(1 \leq j \leq g)$ are sent to all neurons σ_i is connected to. These are either neurons σ_k such that $(i,k) \in syn$, or neurons to which σ_i inherited synapses during neuron division, as described bellow. If the delay $d = 0$, then the spikes leave the neuron i immediately. However, if $d \geq 1$ and the rule is applied at time t, then the neuron i will be closed at step $t, t+1, t+2, \ldots, t+d-1$. During this phase, the neuron cannot receive any spike from outside nor can apply any rule. At step $(t+d)$, the neuron spikes and it can also receive spikes from other neurons and apply a rule. The spiking rule $[E/a_1^{n_1}a_2^{n_2}\ldots a_g^{n_g} \to a_1^{p_1}a_2^{p_2}\ldots a_g^{p_g}; d]_i$ is simply written as $[a_1^{n_1}a_2^{n_2}\ldots a_g^{n_g} \to a_1^{p_1}a_2^{p_2}\ldots a_g^{p_g}; d]_i$ if $E = a_1^{n_1}a_2^{n_2}\ldots a_g^{n_g}$. If $d = 0$, then it becomes $[E/a_1^{n_1}a_2^{n_2}\ldots a_g^{n_g} \to a_1^{p_1}a_2^{p_2}\ldots a_g^{p_g}]_i$.

A *forgetting rule* $[a_1^{t_1}a_2^{t_2}\ldots a_n^{t_n} \to \lambda]_i$ is applicable when the neuron σ_i contains exactly the spikes $a_1^{t_1}a_2^{t_2}\ldots a_n^{t_n}$. Then all these spikes are consumed in the neuron σ_i.

A *division rule* $[E]_i \to []_j \,||\, []_k$ is applicable if a neuron σ_i contains spikes $a_1^{c_1}a_2^{c_2}\ldots a_n^{c_n} \in L(E)$. Then after application of the rule, two new neurons labelled j and k are created from σ_i and the spikes are consumed. Child neurons contain developmental rules from R labelled j and k, respectively, and initially they do not contain any spikes. Also, child neurons inherit synaptic connections of the parent neuron. If there exists any connection (σ_t, σ_i) (i.e., σ_t is connected to σ_i), then after division there exist connections (σ_t, σ_j) and (σ_t, σ_k). Similarly, if there exists a connection (σ_i, σ_t), then (σ_j, σ_t) and (σ_k, σ_t) will exist after division. Moreover, the child neurons will create synaptic connections which are provided by the synapse dictionary.

The configuration of the system provides information about the number of spikes present in the neurons, the synaptic connections with the other neurons and whether the neuron is open/ closed (i.e., whether it can receive/send spikes). The SNPS models

work as a parallel distributive computing model where the rules present in all neurons are applied in parallel synchronized manner and the configuration moves to the next configuration. Each computation starts from the initial configuration and stops at a final configuration (i.e. when no rules are further applicable in any of the neurons). If a neuron has more than one rule which is applicable in a given step, then one of these rules is non-deterministically chosen. However, in this work, such a situation never occurs and rules in the neurons are applied in a deterministic manner.

Since already the introductory variant of the SNPS presented in [5] is computationally universal in the Turing sense, and as our variant of the SNPS is an extension of this basic model, its computational universality follows by the results presented already in [5].

In the next section we use the deterministic SNPS with coloured spikes and neuron division to obtain a uniform solution to the SAT problem in a linear time.

3 A solution to the SAT problem

Many decision problems have been solved using spiking neural P systems in uniform as well as semi-uniform ways [2,6,8,9,11,12,17,20,22]. Let M be a decision problem. If the problem M is solved in a semi-uniform manner, then for each instance I of M, a spiking neural P system $\Pi_{I,M}$ is constructed in polynomial time (with respect to the size of the instance I). The structure and initial configuration of the system $\Pi_{I,M}$ depends on the instance I. Moreover, the system $\Pi_{I,M}$ halts (or it may spike a specified number of spikes within a time interval) if and only if I is a positive instance of the decision problem M. A uniform solution of M contains the family $\{\Pi_M(n)\}_{n\in\mathbb{N}}$ of SNP systems where for each instance $I \in M$ of size n, polynomial number of spikes based on n is introduced into a specified input neuron of $\Pi_{I,M}$. Moreover, the system $\Pi_{I,M}$ halts if and only if I is a positive instance of M. The uniform solutions are strictly associated with the structure of the problem instead of being associated with only one instance of the decision problem. This feature makes the uniform solutions to decision problems more preferable than the semi-uniform solutions. Also, in order to obtain a semi-uniform solution, the SNP system does not require any specific input neuron. However, in the case of uniform solutions, the system must have a specified input neuron(s) which receives the description of an instance of the decision problem in the form of a spike train.

The SAT problem (or the Boolean satisfiability problem) [15] is a well-known NP-complete decision problem. Each instance is a formula in propositional logic with variables obtaining values **TRUTH** or **FALSE**. The SAT decision problem is the problem of determining whether there exists an assignment of truth values to the variables such that the whole formula evaluates to **TRUTH**.

Let us consider an instance represented by the formula $\gamma_{n,m} = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ in the conjunctive normal form, where C_i $(1 \leq i \leq m)$ represent the clauses. Each clause is a disjunction of literals of the form x_j or $\neg x_j$, where x_j are logical variables, $1 \leq j \leq n$. Moreover, the class of SAT instances with n literals and m clauses is denoted by $SAT(n, m)$. So $\gamma_{n,m} \in SAT(n, m)$. In order to solve the SAT problem, at first we have to encode the instance $\gamma_{n,m}$ using spikes in the SNPS, so that the encoded instance could be sent to the input neuron. In this work, we consider the SNPS model with coloured spikes, i.e., different variables are encoded by different types of spikes.

The encoding of the instance $\gamma_{n,m}$ is as follows:

$$code(\gamma_{n,m}) = a^{n+1}(\alpha_{1,1}\alpha_{1,2}\dots\alpha_{1,n})a_c(\alpha_{2,1}\alpha_{2,2}\dots\alpha_{2,n})a_c\dots \\ \dots(\alpha_{m,1}\dots\alpha_{m,n})a_c a_f,$$

where

$$\alpha_{i,j} = \begin{cases} a_j, & \text{if } x_j \in C_i, \\ a'_j, & \text{if } \neg x_j \in C_i, \\ a, & \text{otherwise.} \end{cases}$$

In addition to $\alpha_{i,j}$, the encoding of the instance contains other auxiliary spikes. The term a^{n+1} is added at the beginning in order to give the system a necessary initial period during which it generates an exponential workspace with 2^n neurons. The encoding of each clause is separated by a_c and the end of the encoding is identified by a_f.

The spiking neural P system with coloured spikes and division rules solving the instances in $SAT(n,m)$ is described below.

3.1 The SNPS description

The structure of the SNPS with coloured spikes and division rules is as follows: $\Pi_{n,m} = (S, H, syn, \sigma_{1_0}, \sigma_{1'_0}, \sigma_{n+1}, \sigma_{n+2}, \sigma_{n+3}, \sigma_{n+4}, \sigma_{n+5}, R, in, out)$ where

- $S = \{a_i, a'_i \mid 1 \le i \le n\} \cup \{a, a_s, a_c, a_f\}$
- $H = \{i, i', i_0, i'_0 \mid i = 1, 2, \dots, n\} \cup \{n+1, n+2, n+3, n+4, n+5\}$ $\cup \{in, out\} \cup \{t_i, f_i \mid i = 1, 2, \dots, n\}$;
- $syn = \{(i, t_i), (i', f_i) \mid i = 1, 2, \dots, n\}$ $\cup \{(n+2, n+1), (n+2, n+3), (n+3, n+2), (n+4, n+2), (n+4, n+3)\}$ $\cup \{(in, 1_0), (in, 1'_0), (in, n+5), (n+5, t_1), (n+5, f_1)\} \cup \{(t_1, out), (f_1, out)\}$
- The initial configuration of the system contains neurons with labels $in, out, 1_0, 1'_0, n+1, n+2, n+3, n+4, n+5$. The neurons with labels $1_0, 1'_0, n+2, n+3$ and $n+4$ contain the spike a, the remaining neurons contain no spike.
- Rules in R are divided into four modules: (1) Generating module; (2) Input module; (3) Checking module; (4) Output module.

  (1) Rules in the generating module:
  $[a]_{i_0} \to []_i \,||\, []_{(i+1)_0}; i = 1, 2, \dots, n-2$
  $[a]_{(n-1)_0} \to []_{n-1} \,||\, []_n$
  $[a]_{i'_0} \to []_{i'} \,||\, []_{(i+1)'_0}; i = 1, 2, \dots, n-2$
  $[a]_{(n-1)'_0} \to []_{(n-1)'} \,||\, []_{n'}$
  $[a]_{t_i} \to []_{t_{i+1}} \,||\, []_{f_{i+1}}; i = 1, 2, \dots, n-1;$
  $[a]_{f_i} \to []_{t_{i+1}} \,||\, []_{f_{i+1}}; i = 1, 2, \dots, n-1;$
  $[a]_{n+1} \to []_{t_1} \,||\, []_{f_1}$
  $[a \to a]_{n+2}$
  $[a^2 \to \lambda]_{n+2}$
  $[a \to a]_{n+3}$

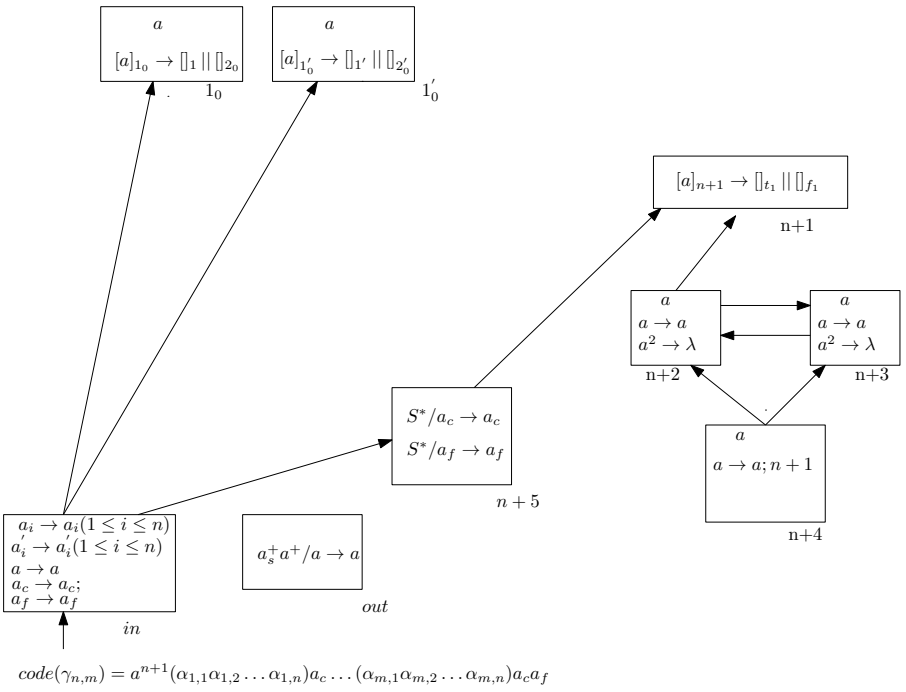


Fig. 1: Initial structure of the SNPS solving the SAT problem.

$[a^2 \to \lambda]_{n+3}$
$[a \to a; n+1]_{n+4}$
$[S^*/a_c \to a_c]_{n+5}$
$[S^*/a_f \to a_f]_{n+5}$

(2) Rules in the input module:
$[a_i \to a_i]_{in}; i = 1, 2, \ldots, n$
$[a'_i \to a'_i]_{in}; i = 1, 2, \ldots, n$
$[a \to a]_{in};$
$[a_c \to a_c]_{in};$
$[a_f \to a_f]_{in};$
$[S^*/a_i \to a]_i; i = 1, 2, \ldots, n$
$[S^*/a'_i \to a]_{i'}; i = 1, 2, \ldots, n$
Note: the last two rules are not seen in Fig. 1 as the neurons σ_i, $i = 1, 2, \ldots, n$, appear during the generating phase.

(3) Rules in the checking module:
$[a_s aa/a \to a_s]_{t_n}$

$[a_s a_c \to \lambda]_{t_n}$
$[a_s a_c a / a_c a \to a_s]_{t_n}$
$[a_s a_f \to a]_{t_n}$
$[a_s a a / a \to a_s]_{f_n}$
$[a_s a_c \to \lambda]_{f_n}$
$[a_s a_c a / a_c a \to a_s]_{f_n}$
$[a_s a_f \to a]_{f_n}$

(4) Rules in the output module:
$[a_s^+ a^+ / a \to a]_{out}$

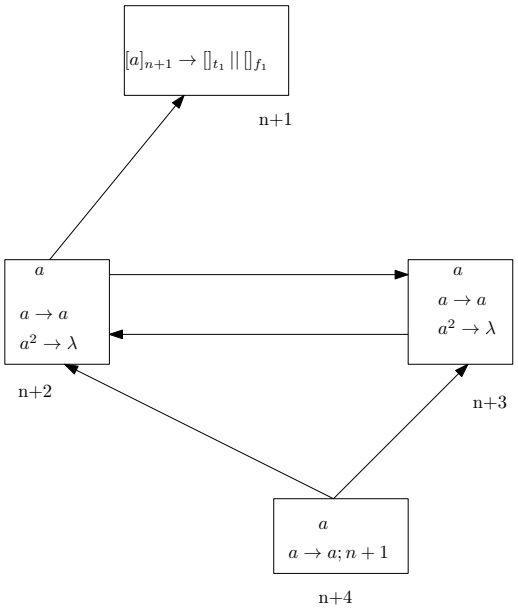


Fig. 2: Details of the generating module producing the exponential workspace of 2^n neurons in the first n computational steps of the SNPS.

The initial structure of the SNPS with coloured spikes and division rules solving an instance of the SAT problem using contains 9 neurons (see Fig. 1). Computation of the SNPS is divided into four stages: (1) generating; (2) input; (3) checking; (4) output.

In the generating stage, the division rules are used to create an exponential number of neurons which are further used during the input and checking stages. In the input stage, the input neuron receives the encoded instance of the SAT problem. This stage overlaps with the checking stage during which the SNPS verifies whether any assignment of values of the variables $x_1, x_2, \ldots, x_n$ satisfies all the clauses C_i $(1 \leq i \leq m)$ present

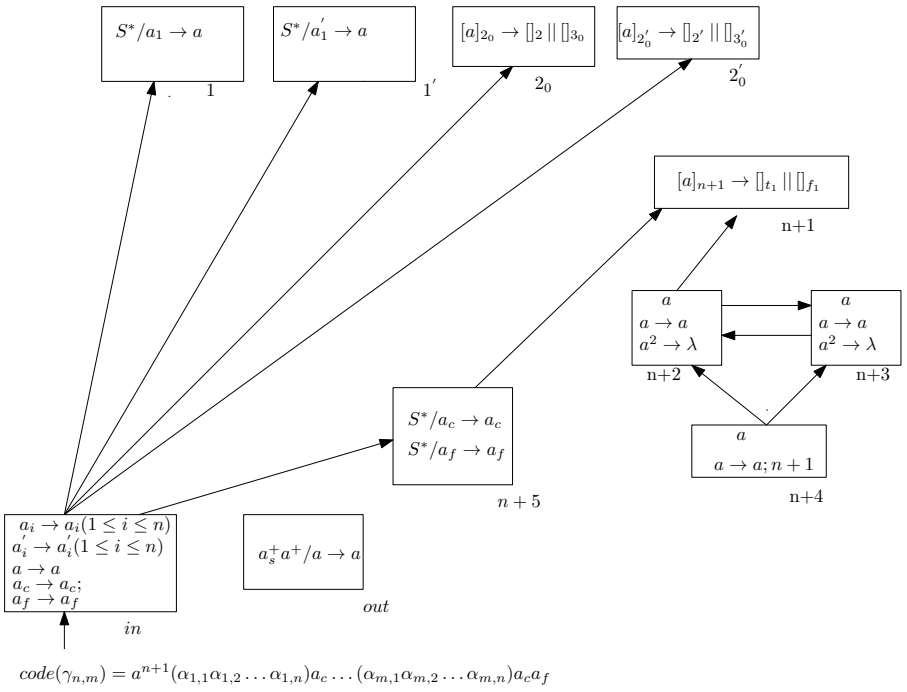


Fig. 3: Structure of the SNPS at time $t = 2$

in the proposition formula $\gamma_{n,m}$. Finally, if the output neuron spikes, it confirms that the formula $\gamma_{n,m}$ is satisfiable.

3.2 Generating stage

Neurons 1_0 and $1_0'$ contain initially the spike a and the rule $[a]_{1_0} \to []_1 \,||\, []_{2_0}$ and $[a]_{1_0'} \to []_{1'} \,||\, []_{2_0'}$, respectively. Thus, the neuron σ_{1_0} creates two neurons with labels 1 and 2_0, and the neuron $\sigma_{1_0'}$ creates two neurons with labels $1'$ and $2_0'$ at time $t = 1$.

The input neuron has synaptic connections to the neurons with labels 1_0 and $1_0'$ which are inherited by the neurons $1, 1', 2_0$ and $2_0'$. At times $i = 1, 2, \ldots, n+1$ the input neuron receives the spike a from the input spike train and at times $i = 2, 3, \ldots, n-1$ it sends the spike to neurons i_0 and i_0' $(2 \le i \le n-1)$. At times $i = n, n+1$ and $n+2$ the input neuron sends spikes, too, but no neurons with labels i_0 and i_0' $(n \le i \le n+2$ exist.)

Neurons i_0 and i_0' $(2 \le i \le n-2)$ contain the rule $[a]_{i_0} \to []_i \,||\, []_{(i+1)_0}$ and $[a]_{i_0'} \to []_{i'} \,||\, []_{(i+1)_0'}$, i.e., the neuron σ_{i_0} creates two neurons with labels i and $(i+1)_0$, and the neuron $\sigma_{i_0'}$ creates two neurons with labels i' and $(i+1)_0'$ $(2 \le i \le n-1)$.

Finally, at step $t = n - 1$, neurons with labels $(n-1)_0$ and $(n-1)'_0$ divide and create neurons with labels $n-1$, n, $(n-1)'$ and n'. So after application of all these division rules, there is a layer of neurons σ_i, $\sigma_{i'}$ $(1 \leq i \leq n)$ shown in Fig. 7, and the input neuron has synaptic connections to all of them.

Simultaneously, the neurons with labels $n+1, n+2, n+3$ and $n+4$ create subsequently 2^n neurons which will be used in the checking stage. The circuit controlling the generating stage is depicted in Figure 2. Initially, the neurons $n+2, n+3$ and $n+4$ contain one spike. At time $t = 1$, σ_{n+2} and σ_{n+3} spike and σ_{n+1} receives a spike at time $t = 2$. At $t = 2$, the rule $[a]_{n+1} \to []_{t_1} \,||\, []_{f_1}$ is applied. So at $t = 3$, two new neurons with labels t_1 and f_1 are created.

Note that the neuron σ_1 (resp. $\sigma_{1'}$) is connected to σ_{t_1} (resp. σ_{f_1}) using the synaptic connections from synapse dictionary. Moreover, σ_{t_1} and σ_{f_1} are connected to the output neuron with the synaptic connections from the synapse directory (see figure 4). The neuron σ_{t_1} contains the rule $[a]_{t_1} \to []_{t_2} \,||\, []_{f_2}$ and neuron σ_{f_1} contains $[a]_{f_1} \to []_{t_2} \,||\, []_{f_2}$.

At the same time $t = 3$, neurons σ_{t_1} and σ_{f_1} receive the spike a from σ_{n+2}. Hence, at $t = 4$, each of σ_{t_1} and σ_{f_1} is divided into σ_{t_2} and σ_{f_2}. Again, these neurons receive one spike from $\sigma_{(n+2)}$ and they are divided at the next step. This process will continue until $t = n + 2$. Newly created neurons t_i, f_i $(1 \leq i \leq n)$ will also get further synapses due to the synapse dictionary, as described above. After $(n+2)$ steps, we will have a system shown in Figure 7.

Finally, the rule $a \to a; n+1$ in σ_{n+4} is applied at time $t = n + 2$ and thus both σ_{n+2} and σ_{n+3} receive one spike at $t = n + 3$. Next, the rule $a^2 \to \lambda$ is applied and both spikes in σ_{n+2} and σ_{n+3} are consumed. So no spikes remain inside them and they are inactive from now on.

3.3 Input stage

Recall that the input neuron receives the encoding of an instance of the SAT problem with n variables and m clauses, i.e., $\gamma_{n,m}$, where

$$code(\gamma_{n,m}) = a^{n+1}(\alpha_{1,1}\alpha_{1,2}\ldots\alpha_{1,n})a_c \ldots (\alpha_{m,1}\alpha_{m,2}\ldots\alpha_{m,n})a_c a_f.$$

The encoding of each clause is ended by a_c and after the input is completely read, it ends with a_f. The initial buffer a^{n+1} is used to delay the input of the clause by $n+1$ steps, giving the SNPS enough time to generate 2^n checking neurons. The input neuron *in* has the following rules: (1) $a \to a$; (2) $a_i \to a_i$ $(1 \leq i \leq n)$; (3) $a'_i \to a'_i$ $(1 \leq i \leq n)$; (4) $a_c \to a_c$; (5) $a_f \to a_f$. Initially, the input neuron is empty and when it receives a, a_i, a'_i, a_c or a_f as input, it spikes and sends the same spike to neurons with labels i, i' $(1 \leq i \leq n)$ and $n+5$.

At time $t = n + 2$, the input neuron receives $\alpha_{1,1}$ as input. Since the input neuron contains the rules $[a_i \to a_i]_{in}; [a'_i \to a'_i]_{in}$ $(1 \leq i \leq n)$, it will spike immediately. The neurons with label i contain the rule $[S^*/a_i \to a]_i$ $(1 \leq i \leq n)$ and neurons with label i' contain the rule $[S^*/a'_i \to a]_{i'}$ $(1 \leq i \leq n)$. These rules are activated upon the existence of the literal x_i (resp. $\neg x_i$) in a clause. If σ_i spikes using the rule $[S^*/a_i \to a]_i$, it signals that the literal x_i is present in a clause. Similarly, the use of the rule $[S^*/a'_i \to a]_{i'}$ in $\sigma_{i'}$ signals the presence of $\neg x_i$ in a clause.

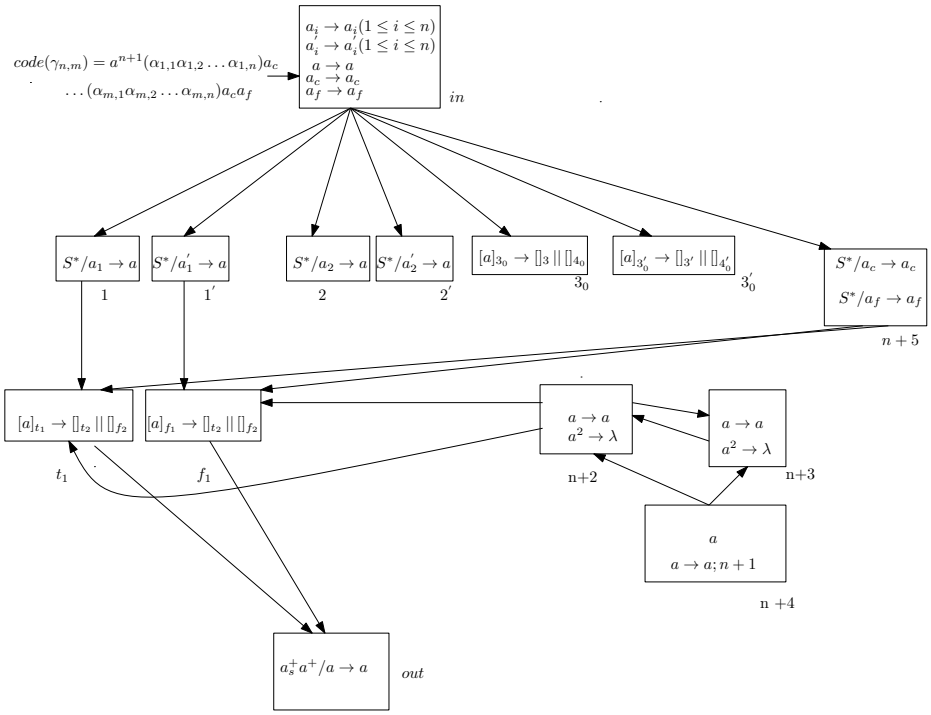


Fig. 4: Structure of the SNPS at time $t = 3$

3.4 Checking stage

After the generating stage, the input neuron is connected to neurons σ_i and $\sigma_{i'}$ $(1 \le i \le n)$. These neurons, in turn, are connected to the checking layer consisting of 2^n neurons labelled t_n or f_n, see Fig. 7. Each of the checking neurons σ_{t_n} or σ_{f_n} has exactly n incoming synapses from the input neurons σ_i or $\sigma_{i'}$ $(1 \le i \le n)$. These synapses represent one of the 2^n possible assignments of 'TRUTH' or 'FALSE' to the n variables of the formula. The structure of these synapses was created by either inherited synapses or by the synapse dictionary during the generating stage. The incoming synapses are indicated under checking neurons in Figures 4 to 7 by expressions in parentheses. Let us consider the assignment $t_1 t_2 f_3 \dots f_n$ (i.e., the first two variables have 'TRUTH' and the remaining ones have 'FALSE'). The corresponding checking neuron has synapses from neurons $\sigma_1, \sigma_2, \sigma_3', \dots, \sigma_n'$, in the input module.

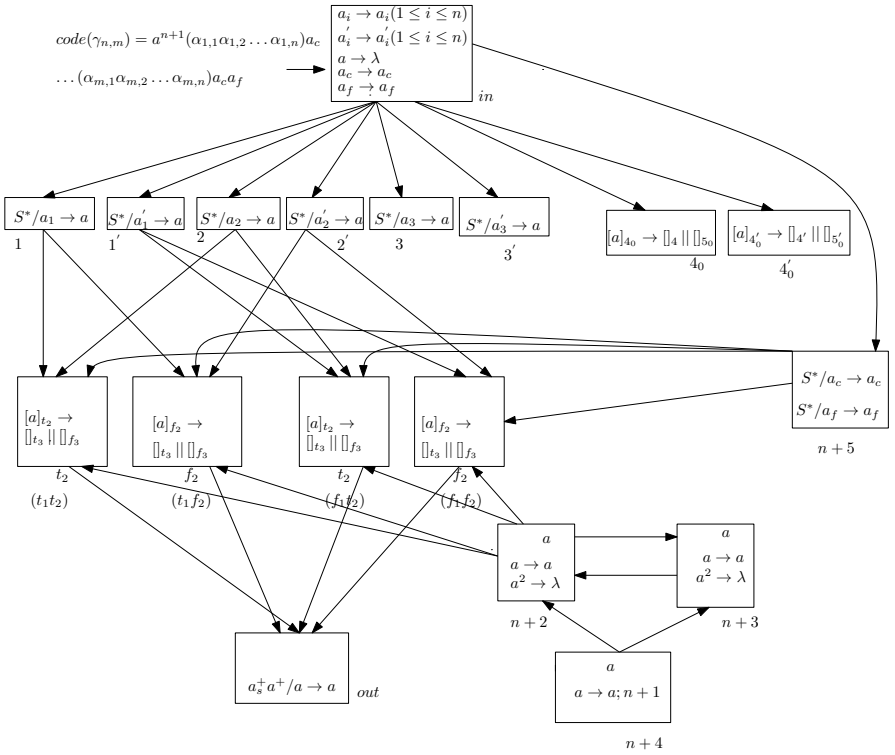


Fig. 5: Structure of the SNPS at time $t = 4$

The input module receives encoded clauses one-by-one. Spiking of the neuron σ_i in the input module signals the presence of the literal x_i in the clause, and spiking of the neuron $\sigma_{i'}$ signals the presence of the literal $\neg x_i$. Therefore, each of the neurons σ_{t_n} or σ_{f_n} obtains one or more spike a if its assignment satisfies the clause. These neurons contain rules $(1)a_s aa/a \to a_s;\ (2)a_s a_c \to \lambda;\ (3)a_s a_c a/a_c a \to a_s;\ (4)a_s a_f \to a$.

Initially, neurons in the checking module contain the spike a_s. If it receives the spike a from the input module, no rule can be applied. When another a spike comes, the rule $a_s aa/a \to a_s$ is applied and only one spike a remains in the neuron. When the encoding of the clause is read completely, the spike a_c is received from σ_{n+5}. If the assignment of a checking neuron satisfies the clause, the neuron now has the spikes $a_s a_c a$ and the rule $a_s a_c a/a_c a \to a_s$ is applied. Otherwise, the neuron contains the spike $a_s a_c$ and they are consumed by the rule $a_s a_c \to \lambda$. Since the neuron looses the spike a_s, no further computation will take place inside it.

3.5 Output stage

The output neuron can receive spikes a_s due to the application of the rules $a_s aa/a \to a_s$ and $a_s a_c a/a_c a \to a_s$ in checking neurons during the checking stage. However, these

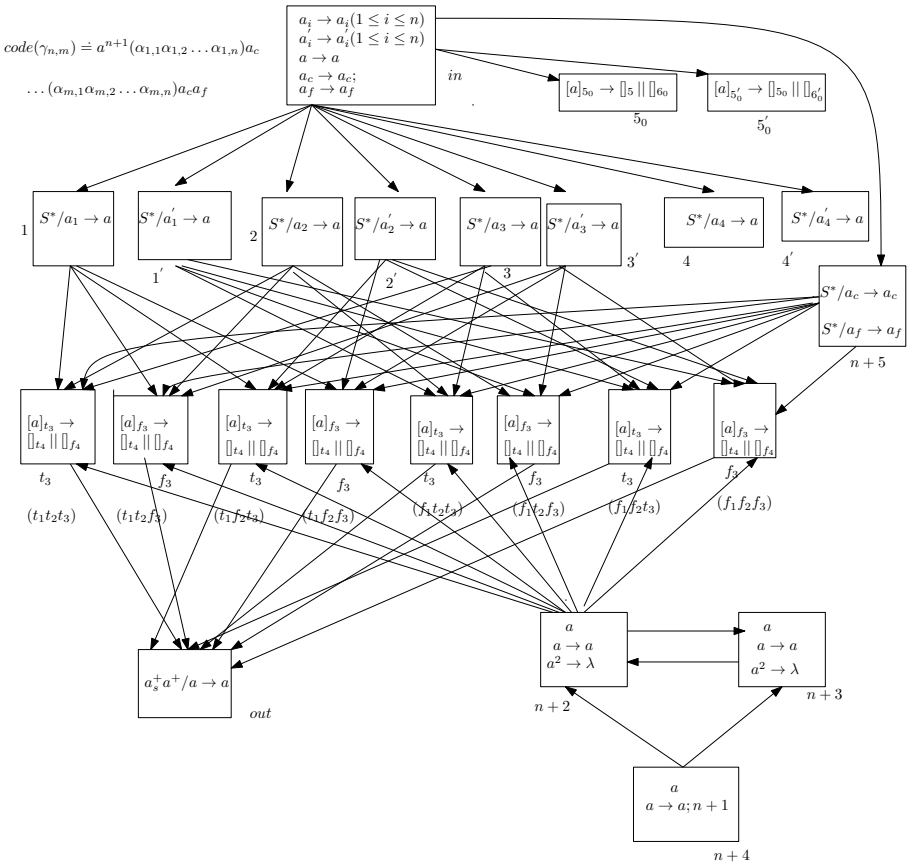
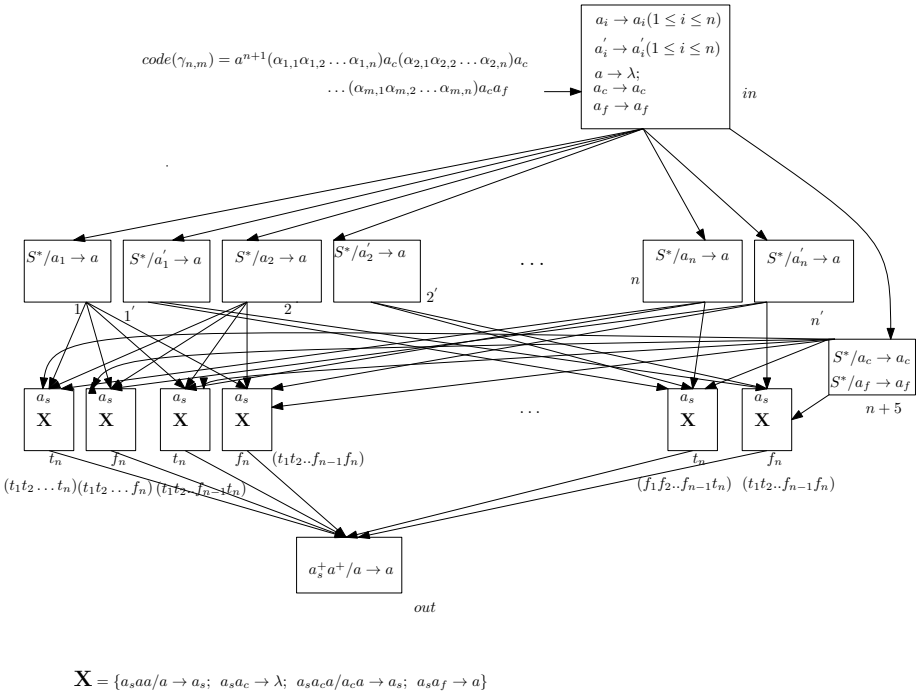


Fig. 6: Structure of the SNPS at time $t = 5$

spikes are ignored. Finally, when the formula is completely read, the checking neurons receive the spike a_f from σ_{n+5}. If any of them still has the spike a_s (meaning that its assignment satisfies all clauses), it spikes using the rule $a_s a_f \to a$ and the spike a is sent to the output neuron. Next, the rule $a_s^+ a^+/a \to a$ is applied in the output neuron, confirming that the formula is satisfiable.

4 Discussion

In this section we compare parameters of our solution to the SAT problem using a SNPS with coloured spikes and division rules with two other published papers [20] and [22] presenting similar solutions to the SAT with SNP systems. Both papers use neuron division rules, the authors of [22] use also neuron dissolution rules. We compare five parameters of descriptional complexity of the used SNP systems and also the running



$\mathbf{X} = \{a_s aa/a \to a_s;\ a_s a_c \to \lambda;\ a_s a_c a/a_c a \to a_s;\ a_s a_f \to a\}$

Fig. 7: Structure of the SNPS at time $t = n + 2$

time complexity. The results are summarized in the table below assuming a solution to an instance of $SAT(n, m)$, i.e., with m clauses and n variables.

It follows that all descriptional complexity parameters used in our model have significantly lower values than in the two compared papers. More specifically, the number of neurons, number of neuron labels, size of the synapse dictionary and number of rules used in our solution are significantly lower than in previous solutions and we are able to achieve this result using only 5 initial spikes. Especially, the number of rules is linear to n while in the previously reported solutions it is quadratic or even exponential. Note that the couloured spikes largely help to organize the work of the system and allow for these simplifications.

The only exception is the running time which is $\mathcal{O}(n + m)$ in [22], while our SNPS runs in time $\mathcal{O}(nm)$. The explanation is simple: the authors of [22] use m input neurons and a special encoding of the formula where each clause is sent in parallel to its designated input neuron. We conjecture that such an input module can be employed also in our case.

A more detailed analysis of the running time shows that the generation stage in our paper requires $(n+1)$ steps when the exponential workspace is created. The total number of steps required for the following input stage reading the encoded formula via the input neuron is $m(n + 1) + 1$., The checking stage largely overlaps with the input one and requires two more steps to complete. Finally the output stage required only one step.

Resources	Wang et. al. [20]	Zhao et. al. [22]	This paper
Initial number of neurons	11	$3n + 5$	9
Initial number of spikes	20	$2m + 3$	5
Number of neuron labels	$10n + 7$	$2^n + 11$	$6n + 7$
Size of synapse dictionary	$6n + 11$	$5n + 5$	$2n + 12$
Number of rules	$2n^2 + 26n + 26$	$n2^n + \frac{1}{3}(4^n - 1)$ + $9n + 5$	$8n + 16$
Time complexity	$4n + nm + 5$	$2n + m + 3$	$nm + n + m + 5$
Number of neurons generated throughout the computation	$2^n + 8n$	$2^{n+1} - 2$	$2^n + 2n$

5 Conclusion

We presented a deterministic spiking neural P system with coloured spikes and division rules which has been used to solve the SAT problem in linear time. We have shown that our model uses significantly less resources than those reported in [20] or [22] to solve the SAT problem. It is fair to note that we use a linear number of different spikes with respect to the number n of variables in SAT, while the two mentioned papers use just one type of spike. However, most types of spikes in our construction is used just to encode the input formula. We could use a similar input module as in [20] with only a few changes which would lower the number of different spikes in our model to just 5. This is left for future research. Furthermore, using another type of input module with multiple input neurons as that used in [22], we could possibly restrict our running time to $\mathcal{O}(n + m)$. As another promising future research direction one could focus on efficient solutions to PSPACE-complete problems using a similar SNPS model as that discussed in this paper.

References

1. Aman, B.: Solving Subset Sum by spiking neural P systems with astrocytes producing calcium. Natural Computing pp. 1–10 (2022)
2. Cabarle, F.G.C., Hernandez, N.H.S., Martínez-del Amor, M.Á.: Spiking neural P systems with structural plasticity: Attacking the Subset Sum problem. In: Membrane Computing: 16th International Conference, CMC 2015, Valencia, Spain, August 17-21, 2015, Revised Selected Papers 16. pp. 106–116. Springer (2015)
3. Ghosh-Dastidar, S., Adeli, H.: Spiking neural networks. International journal of neural systems **19**(04), 295–308 (2009)
4. Gutiérrez Naranjo, M.Á., Leporati, A.: Solving numerical NP-complete problems by spiking neural P systems with pre–computed resources. Proceedings of the Sixth Brainstorming Week on Membrane Computing, 193-210. Sevilla, ETS de Ingeniería Informática, 4-8 de Febrero, 2008 (2008)
5. Ionescu, M., Păun, G., Yokomori, T.: Spiking neural P systems. Fundamenta informaticae **71**(2-3), 279–308 (2006)
6. Ishdorj, T.O., Leporati, A.: Uniform solutions to SAT and 3-SAT by spiking neural P systems with pre-computed resources. Natural Computing **7**, 519–534 (2008)

7. Ishdorj, T.O., Leporati, A., Pan, L., Wang, J.: Solving NP-complete problems by spiking neural P systems with budding rules. In: Membrane Computing: 10th International Workshop, WMC 2009, Curtea de Arges, Romania, August 24-27, 2009. Revised Selected and Invited Papers 10. pp. 335–353. Springer (2010)
8. 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. Theoretical Computer Science **411**(25), 2345–2358 (2010)
9. Leporati, A., Gutiérrez-Naranjo, M.A.: Solving Subset Sum by spiking neural P systems with pre-computed resources. Fundamenta Informaticae **87**(1), 61–77 (2008)
10. Leporati, A., Mauri, G., Zandron, C.: Spiking neural P systems: main ideas and results. Natural Computing **21**(4), 629–649 (2022)
11. Leporati, A., Mauri, G., Zandron, C., Păun, G., Pérez-Jiménez, M.J.: Uniform solutions to SAT and Subset Sum by spiking neural P systems. Natural computing **8**(4), 681 (2009)
12. Pan, L., Păun, G., Pérez-Jiménez, M.J.: Spiking neural P systems with neuron division and budding. Science China Information Sciences **54**, 1596–1607 (2011)
13. Pan, L., Paun, G., Pérez Jiménez, M.J.: Spiking neural P systems with neuron division and budding. Proceedings of the Seventh Brainstorming Week on Membrane Computing, vol. II, 151-167. Sevilla, ETS de Ingeniería Informática, 2-6 de Febrero, 2009 (2009)
14. Paun, G., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., USA (2010)
15. Rintanen, J.: Planning and SAT. Handbook of Satisfiability **185**, 483–504 (2009)
16. Song, B., Li, K., Orellana-Martín, D., Pérez-Jiménez, M.J., Pérez-Hurtado, I.: A survey of nature-inspired computing: Membrane computing. ACM Computing Surveys (CSUR) **54**(1), 1–31 (2021)
17. Song, T., Luo, L., He, J., Chen, Z., Zhang, K.: Solving Subset Sum problems by time-free spiking neural P systems. Applied Mathematics & Information Sciences **8**(1), 327 (2014)
18. Song, T., Rodríguez-Patón, A., Zheng, P., Zeng, X.: Spiking neural P systems with colored spikes. IEEE Transactions on Cognitive and Developmental Systems **10**(4), 1106–1115 (2017)
19. Sosík, P.: P systems attacking hard problems beyond NP: a survey. Journal of Membrane Computing **1**, 198–208 (2019)
20. Wang, J., Hoogeboom, H.J., Pan, L.: Spiking neural P systems with neuron division. In: Membrane Computing: 11th International Conference, CMC 2010, Jena, Germany, August 24-27, 2010. Revised Selected Papers 11. pp. 361–376. Springer (2011)
21. Xu, L., Jeavons, P.: Simple neural-like P systems for maximal independent set selection. Neural Computation **25**(6), 1642–1659 (2013)
22. Zhao, Y., Liu, X., Wang, W.: Spiking neural P systems with neuron division and dissolution. PLoS One **11**(9), e0162882 (2016)
23. Zhao, Y., Liu, Y., Liu, X., Sun, M., Qi, F., Zheng, Y.: Self-adapting spiking neural P systems with refractory period and propagation delay. Information Sciences **589**, 80–93 (2022)

Detecting Android Malware Using Spiking Neural P Systems

Mihail-Iulian Pleșa*[1], Marian Gheoghe[2], Florentin Ipate[1], and Gexiang Zhang[3]

[1] Department of Computer Science, University of Bucharest, Bucharest, Romania
mihail-iulian.plesa@s.unibuc.ro
[2] School of Electrical Engineering and Computer Science, University of Bradford, Bradford, UK
[3] School of Automation, Chengdu University of Information Technology, Chengdu 610225, China

Abstract. Android is one of the most used operating systems for mobile platforms and also the most attacked by malicious actors. Currently, malware detection systems are based on signatures. The disadvantage of this approach is that the attacker can easily modify the malware to avoid detection. Machine learning algorithms can automatically analyze large datasets to detect patterns that can help to classify new entries. For this reason, machine learning represents a possible solution to the problem of malware detection. Spiking Neural P systems are third-generation neural networks that are much more energy efficient than the current ones. In this paper, we investigate the possibility of using Spiking Neural P systems to detect malware on Android platforms. We trained a Spiking Neural P system and several classic machine learning algorithms, among which an artificial neural network, on the same Android malware dataset. We show through experiments that the Spiking Neural P system can efficiently solve the malware detection problem with much fewer training epochs than an artificial neural network and obtains better performances than the other machine learning algorithms we studied.

Keywords: Malware detection · LSN P system · Information security

1 Introduction

Android is the most used mobile operating system installed on more than 75% of mobile devices [20]. Given this large amount of market share, Android platforms are the primary target for malicious actors. There are three basic steps taken by malware for infecting a mobile platform: installation, activation and deploying the payload [35]. Currently, there are two main approaches to detecting and analyzing malware: dynamic analysis and static analysis. Dynamic analysis involves running the infected Android Package (APK) in a controlled environment to evaluate the concrete actions taken by the malware. On the other side, static analysis implies analyzing the APK without running it. The main advantage of static analysis lies in the fact that it allows the creation of malware-specific signatures that can be used to detect it [22]. Unlike signature-based malware detection, machine learning algorithms can detect new malware by analyzing the data from either static or dynamic analysis.

The field of machine learning (ML) experienced a strong advance in the last year. In 2017 appeared the modern CNN architecture [15]. The model impressed the world

with its capabilities and size being able to classify 1000 different image classes with an error rate of only 15.3% [7]. Only five years later, OpenAI released ChatGPT, a large language model with extensive capabilities [3].

However, current deep learning algorithms have large energy consumption [32]. One reason why this happens is that the current deep learning algorithms are not inspired by the way the human brain works [23]. Spiking neural P systems (SN P systems) represent a possible solution to this problem [13]. SN P systems are neural networks inspired by how the neural cells work and interact. Unlike current neural networks, the neurons from an SN P system communicate using discrete spiking. In an SN P system, each neuron has a number of firing rules. When the firing condition is met, the neuron will emit one spike to all neurons with which it is connected. This raises the possibility to implement neural networks that are energy efficient. In this paper, we analyze the possibility of using the SN P system to detect malware on the Android platform. We compare an SN P system with a classical artificial neural network (ANN) and show that the SN P system is indeed more efficient. We also compare the SN P system with other classifiers e.g. logistic regression, linear and non-linear Support Vector Machine (SVM), and decision trees.

2 Related work and our contribution

In general, there are two main categories of machine learning algorithms used in malware detection: supervised and unsupervised. Supervised learning requires a labeled dataset used by the algorithm to learn how to classify new samples. On the other hand, the purpose of unsupervised learning algorithms, which do not require any labeled data, is to compute an approximate distribution of the dataset. In [17] the authors propose an ML system that detects Android malware by analyzing the permission usage. To detect the best classifier, they analyzed four different algorithms: AdaBoost, Naive Bayes, Decision Trees, and SVM [10]. In [16] it is presented a random forest classifier to detect malware based on more 377 features about Android APK. In [25] it is proposed a decision trees approach that classifies samples as malware or benign based on the opcode frequency. In [5] the authors presented DRACO, a machine-learning framework that detects Android malware based on static and dynamic features. Another idea of combining dimensionality reduction and a classifier for Android malware detection is presented in [29]. All of these ideas were based on classical machine learning algorithms. There are also other more recent methods for detecting Android malware using deep learning. In [26] the authors proposed an artificial neural network that detects Android malware based on dynamic features with more than 97% accuracy. The authors in [31] proposed MalNet, a deep learning architecture for malware detection with automatic feature selection. The idea is to treat the malware as a grayscale image and then use convolution neural networks for classification. In [12] the authors used deep belief networks to detect malware based on Application Programming Interface (API) calls of the APK.

There were also previous applications of SN P systems in machine learning [6]. In [34], SN P systems were used to construct a spiking convolutional neural network. In [19] a specific SN P system called DTN P system was used in medical image processing [4]. In [33] the authors proposed a general classifier based on the SN P system and test

it on the MNIST dataset [8]. Two comprehensive surveys on image processing using P systems are presented in [9] and [30].

Although there was previous work that proposed new machine learning algorithms based on the SN P system, from our knowledge, this is the first paper that studies a real cybersecurity problem i.e. android malware detection with the SN P system. We implement a concrete instantiation of the SN P system proposed by [33] and test it against an Android malware dataset. We compare the results obtained by the SN P system with results obtained by other machine learning algorithms e.g. artificial neural networks, logistic regression, linear and non-linear SVM, and decision trees. We prove experimentally that the SN P system classifier obtains better results than the other algorithms with respect to the analyzed metrics. The paper is organized as follows: in Sect. 3 we present the SN P system used to classify the malware i.e. the Layered Spiking Neural P system, in Sect. 4 we present the experiments while Sect. 5 is left for the conclusions and further directions of research.

3 Layered Spiking Neural P Systems

In this section we describe a concrete instance of an LSN P system described in [33]. The system is based on fuzzy values for which the following operations are defined accordingly to [27]:

1. $\vee$ is the OR operator which returns the maximum of the inputs.
2. $\oplus$ is the addition operator which returns the sum of the inputs.
3. $\otimes$ is the multiplication operator which returns the product of the inputs.

Definition 1. *A Layeblack Spiking Neural P system (LSN P system) is defined as the following construct:*

$$\Pi = (\{a\}, \{\sigma^1_{p_1}, \cdots \sigma^1_{p_k}, \sigma^3_{p_1}, \sigma^5_{p_1}\}, \{\sigma^2_{r_1}, \cdots \sigma^2_{r_n}, \sigma^4_{r_1}, \cdots \sigma^4_{r_n}\}, syn, IN, OUT)$$

where:

1. The symbol a denotes a spike
2. $\sigma^h_{r_j} = \{w^h_{ji}, r^h_j\}$ denotes a rule neuron where h is the layer label and:
   (a) w^h_{ji} denotes the weight on the synapse that connects the neuron $\sigma^h_{r_j}$ to the neuron $\sigma^{h+1}_{p_i}$. Here, $w^h_{ji} = 1, \forall 1 \leq i \leq k, 1 \leq j \leq n, h \in \{2, 4\}$.
   (b) r^h_j denotes a set of firing rules depending on the layer label:
      i. For the second layer, $r^2_j : E^2 : a^{\theta_j} \rightarrow a^{\theta_j}$, where $E^2 = \{\theta_j \geq 0\}$ and $\theta_j = (w_{1j} \otimes \alpha_1) \oplus (w_{2j} \otimes \alpha_2) \oplus \cdots \oplus (w_{kj} \otimes \alpha_k), 1 \leq j \leq n$.
      ii. For the fourth layer, $r^4_1 : E^4_j / a^{\theta_j} \rightarrow a; d_j$, where $E^4_j = \{\theta_j \geq o\}$ where the value of o is computed by the proposition neurons described below.
3. $\sigma^h_{p_i} = \{w^h_{ij}, r^h_i\}$ denotes a proposition neuron where h is the layer label and:
   (a) w^h_{ij} denotes the weight on the synapse that connects the neuron $\sigma^h_{p_i}$ to the neuron $\sigma^{h+1}_{r_j}$. Depending on the layer label, there are the following values:
      i. For the first layer, w^1_{ij} are chosen randomly from the range $(0, 1), \forall 1 \leq i \leq k$, $1 \leq j \leq n$.

ii. For the third layer, $w_{ij}^3 = 1, \forall 1 \leq i \leq k, 1 \leq j \leq n$.

(b) r_i^h denotes a set of firing rules. They can have the following form:

i. For the first layer, $r_i^1 : E^1/a^{\alpha_i} \rightarrow a^{\alpha_i}$, where $E_1 = \{\alpha_i \geq 0\}, \forall 1 \leq i \leq k$.

ii. For the third layer, $r^3 : E^3/a^o \rightarrow o$, where $E_3 = \{o \geq 0\}$ and $o = \theta_1 \vee \theta_2 \vee \cdots \vee \theta_n$.

iii. For the fifth layer, there is no firing condition, the rule $a \rightarrow a$ is always activated.

4. $syn = \left\{ \left(\sigma_{pi}^1, \sigma_{rj}^2\right), \left(\sigma_{rj}^2, \sigma_{p1}^3\right), \left(\sigma_{p1}^3, \sigma_{rj}^4\right), \left(\sigma_{rj}^4 \sigma_{p1}^5\right), \forall 1 \leq i \leq k, 1 \leq j \leq n \right\}$
5. $IN = \{\sigma_{pi}^1, \forall 1 \leq i \leq k\}$
6. $OUT = \{\sigma_{p1}^5\}$

The purpose of the weights in an LSN P system is to adjust the connection between two neurons. If neuron σ_{p_i} has n spikes and the weight between σ_{p_i} and σ_{r_j} is w_{ij}^1 then the neuron σ_{r_j} will receive $n \times w_{ij}^1$ spikes. The running steps of the system are the following:

1. The input is encoded using the encoded scheme described in Sect. 3.1 and provided to the input layer.
2. Each neuron in the input layer will send one spike to all of the neurons from the hidden layer. The number of spikes received by a neuron from the hidden layer is the weighted sum of the received spikes.
3. The neuron σ_{p1}^3 will send to the output layer the maximum number of spikes received from the hidden layer.
4. At each time step, one neuron corresponding to the maximum number of spikes will emit one spike. In this way, we identify the classification result.

Fig. 1 shows the LSN P system used in this paper.

3.1 Input encoding and learning function

The encoding scheme proposed in [33] is inspired by the fact that various biological phenomena that happen in the decision-making process involve nonlinear mixtures of variables [24]. Let $\mathbf{x}$ be the input features vector of length m. The first step is to scale all the elements of $\mathbf{x}$ into the range of $(0, 1)$ using (1):

$$T(\mathbf{x}) = \frac{\mathbf{x} - \mathbf{x}_{min}}{\mathbf{x}_{max} - \mathbf{x}_{min}} \tag{1}$$

where $\mathbf{x}_{max}$ and $\mathbf{x}_{min}$ are the maximum and minimum of the elements of the vector $\mathbf{x}$.

The second step is to encode the scaled input $\mathbf{x}$ using a 2^{nd}-degree Taylor expansion in m dimensions around zero as shown in Algorithm 1. Following the algorithm, the number of neurons of the input layer is $k = \frac{m(m+3)}{2}$.

The learning in an LSN P system is based on the Widrow-Hoff learning rule [28]. Let denote by $\tilde{t}$ the output of the system and by t the ground truth. Let $\mathbf{W}$ be the $2 \times n$ matrix of weights. Updating the weights is done by repeating the Widrow-Hoff:

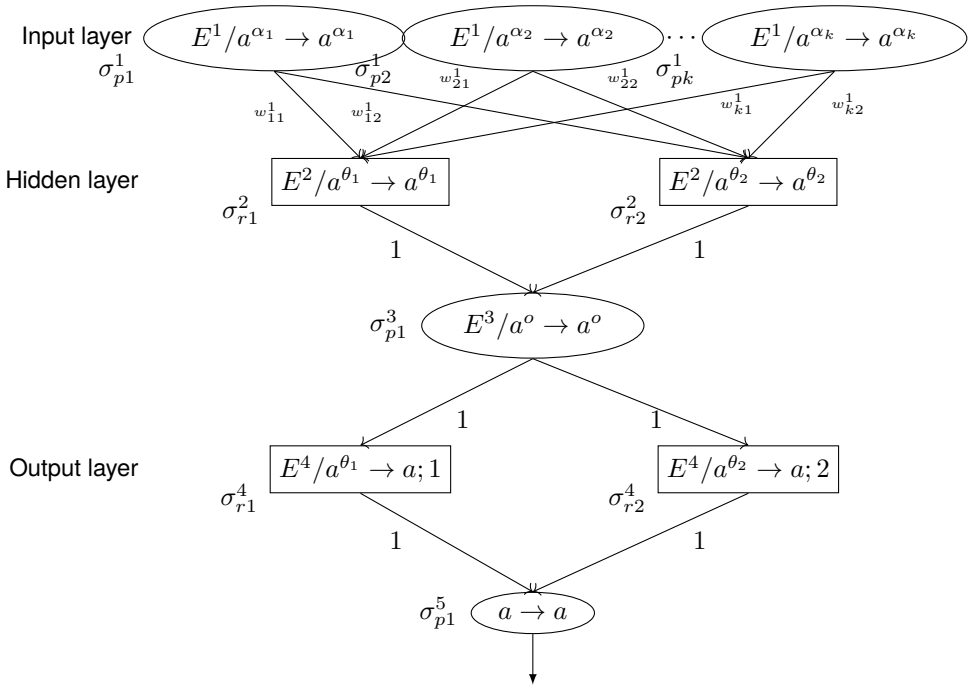


Fig. 1: LSN P system

Algorithm 1 The encoding algorithm

Input: x
Output: $\overline{\mathbf{x}}$

$k_1 = \binom{1}{m}$
for $i = 1\ i \leq k_1\ i = i + 1$ **do**
  $\overline{\mathbf{x}}[i] = \mathbf{x}[i]$
end for
k = 0
for $i = 1\ i \leq m\ i = i + 1$ **do**
  for $j = 1\ i \leq i\ j = j + 1$ **do**
    $\overline{\mathbf{x}}[k_1 + k] = \mathbf{x}[i] \cdot \mathbf{x}[j]$
    k = k + 1
  end for
end for

$$\mathbf{W} \Leftarrow \mathbf{W} + \eta \left(t - \tilde{t}\right) \boldsymbol{\alpha} \tag{2}$$

where η is the learning rate and $\boldsymbol{\alpha}$ is the output of the encoding algorithm.

The learning rule will be applied for several training epochs until satisfactory accuracy is obtained over the training set.

4 Experimental results

The experiments were performed on the Drebin-215 dataset [2]. Each sample in the dataset is characterized by 215 features from the following categories:

1. API call signatures e.g. onServiceConnected, android.os.Binder, attachInterface, etc.
2. Manifest permission e.g. SEND_SMS, READ_PHONE_STATE, RECEIVE_SMS, etc.
3. Intent e.g. android.intent.action.PACKAGE_REPLACED, android.intent.action.TIME_SET, android.intent.action.BATTERY_OKAY, etc.
4. Commands signature e.g. mount, remount, chown, etc.

The dataset is composed of 15036 samples of which 9476 are benign and 5560 are malware. For each experiment, the training set consists of 1000 samples collected randomly from the entire dataset. The test set represents the rest of the samples.

There are many metrics to evaluate a classifier [11]. We begin by defining some concepts that will be used to construct the metrics:

1. *True positive (TP)*: Represents a sample that is malware and it is classified as malware
2. *False positive (FP)*: Represents a sample that is benign but is classified as malware
3. *True negative (TN)*: Represents a sample that is benign and is classified as benign
4. *False negative (FN)*: Represents a sample that is malware and it is classified as benign

In this paper, we use the following metrics to evaluate the classifiers:

1. *Accuracy (Acc)*: Represents the ratio between the number of correctly classified samples and the total number of samples:

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

2. *Error rate (Err)*: Represents the ratio between the number of incorrectly classified samples and the total number of samples:

$$Err = \frac{FP + FN}{TP + TN + FP + FN} \tag{4}$$

3. *Precision (P)*: Represents the ratio between the number of samples correctly classified as malware and the total number of samples classified as malware:

$$P = \frac{TP}{TP + FP} \tag{5}$$

4. *Recall (R)*: Represents the ratio between the number of samples correctly classified as malware and the total number of malware samples:

$$R = \frac{TP}{TP + FN} \tag{6}$$

5. *F1 score (F1)*: Represents the harmonic mean between the precision and recall:

$$F1 = 2 \cdot \frac{P \cdot R}{P + R} \tag{7}$$

We first compare the LSN P system with a classifier based on an artificial neural network (ANN) since both approaches involve a neural network. The LSN P system has 23435 neurons on the hidden layer so we construct an ANN with one single hidden layer with 23435 neurons with ReLU activation function [1]. To train the ANN we used the Adam optimizer with a learning rate of 0.001. Because the LSN P system was trained over the entire dataset, we did not use batch training for the ANN. Fig. 2 shows the F1 score of the LSN P system and the ANN on the data during training with respect to the epoch.

Table 1 summarizes the comparison between the LSN P system and the ANN-based classifier. Although the LSN P system obtains slightly better performance with respect to the metrics, the main difference between the two models is that the LSN P system uses only 12 epochs for training, unlike the ANN which needs 146 epochs to be trained.

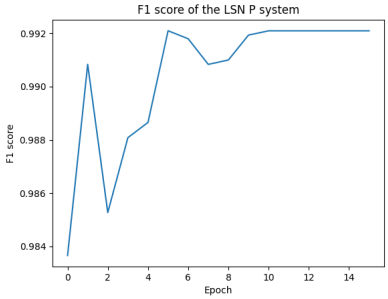
We also compare the LSN P system classifier with other classical classifiers: logistic regression, linear and non-linear SVM, and decision trees [18,21,14]. Table 2 presents the results. The LSN P system is $12\times$ more efficient than the classical ANN. Also, of all the algorithms used, the LSN P system achieves the highest accuracy. All the experiments were made on an Apple Mac M1 Max platform with 10 cores and 32 GB of RAM using the Python programming language.

Table 1: Comparison between LNS P system and ANN

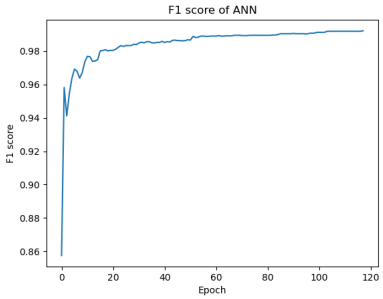
Model	**Number of epochs**	**Acc**	**Err**	**P**	**R**	**F1**
LSN P system	12	0.9930	0.0071	0.9939	0.9949	0.9944
ANN	146	0.9901	0.0099	0.9901	0.9944	0.9922

Table 2: Comparison between LNS P system and other classifiers

Model	**Accuracy**	**Error**	**Precision**	**Recall**	**F1**
LSN P system	0.9930	0.0071	0.9939	0.9949	0.9944
Logistic regression	0.9849	0.0150	0.9854	0.9904	0.9879
Linear SVM	0.9793	0.0206	0.9852	0.9814	0.9833
Non-linear SVM	0.9872	0.0127	0.9861	0.9940	0.9900
Decision Trees	0.9876	0.0123	0.9905	0.9898	0.9902



(a) Accuracy of LSN P system with respect to the training epoch



(b) Accuracy of ANN with respect to the training epoch

Fig. 2: The F1 Score of LSN P system and ANN with respect to the training epoch

5 Conclusions and further directions of research

In this paper, we studied the possibility of detecting malware on Android platforms using Spiking Neural P systems [13]. SN P systems are inspired by how biological neurons work and interact. Unlike current deep learning approaches this type of neural network is much more energy efficient. We prove experimentally that SN P systems obtain slightly better accuracy than the ANN but with much fewer training epochs.

There are multiple further directions of research. The first one is to compare SN P systems with other deep learning architectures e.g. Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM), etc. The second one is to analyze the confidence of SN P systems compared to the confidence of classical ANNs. In this paper, we used the model proposed in [33] which does not output the probability associated with each classification. To analyze the confidence of the model, it will have to be modified to output the probability as well. In this paper, we used a dataset composed of multiple manually crafted features. One possible direction of research is to apply the SN P system to classify malware directly from the opcode. Another direction of research is to implement the LSN P system on dedicated neuromorphic hardware and to asses its energy consumption efficiency.

Acknowledgements This research was supported by the European Regional Development Fund, Competitiveness Operational Program 2014-2020 through project IDBC (code SMIS 2014+: 121512).

References

1. Agarap, A.F.: Deep learning using rectified linear units (relu). arXiv preprint arXiv:1803.08375 (2018)

2. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: Effective and explainable detection of android malware in your pocket. In: Ndss. vol. 14, pp. 23–26 (2014)
3. Azaria, A.: Chatgpt usage and limitations
4. Bao, T., Zhou, N., Lv, Z., Peng, H., Wang, J.: Sequential dynamic threshold neural P systems. Journal of Membrane Computing **2**(4), 255–268 (2020)
5. Bhandari, S., Gupta, R., Laxmi, V., Gaur, M.S., Zemmari, A., Anikeev, M.: DRACO: DRoid analyst combo an android malware analysis framework. In: Proceedings of the 8th International Conference on Security of Information and Networks. pp. 283–289 (2015)
6. Chen, Y., Chen, Y., Zhang, G., Paul, P., Wu, T., Zhang, X., Rong, H., Ma, X.: A Survey of Learning Spiking Neural P systems and A Novel Instance. International Journal of Unconventional Computing **16** (2021)
7. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition. pp. 248–255. Ieee (2009)
8. Deng, L.: The mnist database of handwritten digit images for machine learning research. IEEE Signal Processing Magazine **29**(6), 141–142 (2012)
9. Díaz-Pernil, D., Gutiérrez-Naranjo, M.A., Peng, H.: Membrane computing and image processing: a short survey. Journal of Membrane Computing **1**(1), 58–73 (2019)
10. Géron, A.: Hands-on machine learning with scikit-learn and tensorflow: Concepts. Tools, and Techniques to build intelligent systems (2017)
11. Goodfellow, I., Bengio, Y., Courville, A.: Deep learning. MIT press (2016)
12. Hou, S., Saas, A., Ye, Y., Chen, L.: Droiddelver: An android malware detection system using deep belief network based on api call blocks. In: Web-Age Information Management: WAIM 2016 International Workshops, MWDA, SDMMW, and SemiBDMA, Nanchang, China, June 3-5, 2016, Revised Selected Papers 17. pp. 54–66. Springer (2016)
13. Ionescu, M., Păun, G., Yokomori, T.: Spiking neural P systems. Fundamenta Informaticae **71**(2-3), 279–308 (2006)
14. Kotsiantis, S.B.: Decision trees: a recent overview. Artificial Intelligence Review **39**, 261–283 (2013)
15. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Communications of the ACM **60**(6), 84–90 (2017)
16. Li, J., Wang, Z., Wang, T., Tang, J., Yang, Y., Zhou, Y.: An android malware detection system based on feature fusion. Chinese Journal of Electronics **27**(6), 1206–1213 (2018)
17. Li, J., Sun, L., Yan, Q., Li, Z., Srisa-An, W., Ye, H.: Significant permission identification for machine-learning-based android malware detection. IEEE Transactions on Industrial Informatics **14**(7), 3216–3225 (2018)
18. Menard, S.: Applied logistic regression analysis. No. 106, Sage (2002)
19. Mi, S., Zhang, L., Peng, H., Wang, J.: Medical image fusion based on DTNP systems and Laplacian pyramid. Journal of Membrane Computing **3**(4), 284–295 (2021)
20. Naldi, M.: Concentration in the mobile operating systems market. arXiv preprint arXiv:1605.04761 (2016)
21. Noble, W.S.: What is a support vector machine? Nature biotechnology **24**(12), 1565–1567 (2006)
22. Pan, Y., Ge, X., Fang, C., Fan, Y.: A systematic literature review of android malware detection using static analysis. IEEE Access **8**, 116363–116379 (2020)
23. Raichle, M.E., Gusnard, D.A.: Appraising the brain's energy budget. Proceedings of the National Academy of Sciences **99**(16), 10237–10239 (2002)
24. Rigotti, M., Barak, O., Warden, M.R., Wang, X.J., Daw, N.D., Miller, E.K., Fusi, S.: The importance of mixed selectivity in complex cognitive tasks. Nature **497**(7451), 585–590 (2013)

25. Santos, I., Brezo, F., Ugarte-Pedrero, X., Bringas, P.G.: Opcode sequences as representation of executables for data-mining-based unknown malware detection. information Sciences **231**, 64–82 (2013)
26. Singh, L., Hofmann, M.: Dynamic behavior analysis of android applications for malware detection. In: 2017 International Conference on Intelligent Communication and Computational Techniques (ICCT). pp. 1–7. IEEE (2017)
27. Wang, J., Shi, P., Peng, H., Pérez-Jiménez, M.J., Wang, T.: Weighted fuzzy spiking neural p systems. IEEE Transactions on Fuzzy Systems **21**(2), 209–220 (2012)
28. Wang, Z.Q., Manry, M.T., Schiano, J.L.: LMS learning algorithms: misconceptions and new results on converence. IEEE Transactions on Neural Networks **11**(1), 47–56 (2000)
29. Wei, L., Luo, W., Weng, J., Zhong, Y., Zhang, X., Yan, Z.: Machine learning-based malicious application detection of android. IEEE Access **5**, 25591–25601 (2017)
30. Yahya, R.I., Shamsuddin, S.M., Yahya, S.I., Hasan, S., Al-Salibi, B., Al-Khafaji, G.: Image segmentation using membrane computing: a literature survey. In: International Conference on Bio-Inspired Computing: Theories and Applications. pp. 314–335. Springer (2016)
31. Yan, J., Qi, Y., Rao, Q.: Detecting malware with an ensemble method based on deep neural network. Security and Communication Networks **2018** (2018)
32. Yang, T.J., Chen, Y.H., Emer, J., Sze, V.: A method to estimate the energy consumption of deep neural networks. In: 2017 51st asilomar conference on signals, systems, and computers. pp. 1916–1920. IEEE (2017)
33. Zhang, G., Zhang, X., Rong, H., Paul, P., Zhu, M., Neri, F., Ong, Y.S.: A layered spiking neural system for classification problems. International journal of neural systems **32**(08), 2250023 (2022)
34. Zhang, X., Liu, X.: Multiview Clustering of Adaptive Sparse Representation Based on Coupled P Systems. Entropy **24**(4), 568 (2022)
35. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: 2012 IEEE symposium on security and privacy. pp. 95–109. IEEE (2012)

On 2D P Colony Simulator

Daniel Valenta[1] and Miroslav Langer[2][0000−0001−5990−7780]

[1] Institute of Computer Science, Silesian University in Opava, Czech Republic
daniel.valenta@fpf.slu.cz
[2] Department of Applied Informatics, Faculty of Economics, VŠB-Technical University of Ostrava, Czech Republic
miroslav.langer@vsb.cz

Abstract. This paper presents the whole process of a development of a software system for a simulation of a theoretical computational model, the 2D P colony with a dynamic environment support.

Keywords: Membrane computing · 2D P colony · Simulation software development

1 Introduction

Membrane computing is a field of the computer science aiming to explore new computational models through the study of the biological cells, with a particular focus on the cell membranes. These computational models are called Membrane systems and are inspired by the structure and the functioning of the cells. Specifically, membrane computing deals with distributed and parallel computational models that process multiple sets of symbolic objects in a localized manner. One of the key features is the integration of the evolutionary rules that allow the encapsulation of evolving objects into partitions defined by the membranes. The communication between membranes and interaction with the environment plays a key role in these processes. These computational models are commonly referred to as P systems, named after Gheorghe Păun, the original creator of the model (see [12,13]).

The application scope of P systems is extensive and diverse. For instance, in a study by Xingqiao Deng et al., see [4], P systems were used to enhance the efficiency and accuracy of reducer lubrication. Liang Huang et al. proposed a new variant of tissue P system (TPS) to systematically optimize processes with multiple productive objectives, as discussed in their work [9]. Wang Bo et al. addressed the detection of malicious URLs using P systems, as described in their paper [1].
Additionally, P systems, specifically Optimization Spiking Neural P Systems (OSNPS), were utilized for solving combinatorial optimization problems, as demonstrated in various studies, such as in [5].

P Colony, as described in [10], is a very simple computational model derived from P systems. It draws an inspiration from the collective behavior of ant colonies. This computational model consists of a community of agents living in a shared environment, represented by a multiset of objects. Each agent represents a membrane that contains a specific number of objects, allowing for evolution into different objects or swapping them

with the environment. As research progressed on these simplified membrane systems, a 2-dimensional variant was introduced to explore their capabilities further. We call this extended model a 2D P colony. [3]

Unlike P colonies, agents of a 2D P colony live in a two-dimensional environment, and each agent is equipped with a set of programs consisting of a small number of simple rules that allow it to act and move in the environment.

This paper focuses on the creation of a novel simulator for 2D P colonies. Our ongoing research has led to new variants that include features such as a blackboard to simulate the GWO (agent-based optimization algorithm) or a dynamic environment to simulate ant colony optimization. As a result, we decided to develop a general-purpose simulator that offers good performance, a parametrically customizable simulation interface, and allows user-defined agent programs, as well as providing support for various modules including dynamic environments.

2 2D P colony with the Evolving Environment

The main goal of our application is to support not only 2D P colony simulator, but also allow the possibilities for the extensions such as support for dynamic environments, blackboard and other modules future in point. In this paper we focus on a modified definition of a 2D P colony with an evolving environment introduced in [11]. The only difference between 2D P colony and 2D P colony with the evolving environment is in a definition of an environment. The definition of the environment is only enriched by a set of evolution rules R. The evolving environment also affects the derivation step. In the definition of the 2D P colony, the changes in the environment can be done only by the agents using the communication rules.

Definition 1. *A 2D P colony with evolving environment, 2D$_{ev}$ P COL is a construct*

$$\Pi = (V, e, Env, A_1, \ldots, A_d, f), d \geq 1,$$

where:

- V *is the alphabet of the colony. The elements of the alphabet are called objects.*
- $e \in V$ *is the basic environmental object of the 2D P colony,*
- Env *is a triplet* $(m \times n, w_E, R)$*, where:*
  - $m \times n, m, n \in \mathbb{N}$ *is the size of the environment.*
  - w_E *is the initial contents of the environment, it is a matrix of size* $m \times n$ *of multiset of objects over* $V \setminus \{e\}$.
  - R *is a set of evolution rules. Each rule is of the form* $S \to T$*, where* S *is a multiset over the objects over* $V \setminus \{e\}$*, and where* T *is a multiset over the objects over* V*. We say, that the multiset* S *evolves into the multiset* T.
- $A_i,\ 1 \leq i \leq d$*, is an agent. The number* d *is called a degree of the colony. Each agent is a construct* $A_i = (O_i, P_i, [o, p]),\ 0 \leq o \leq m,\ 0 \leq p \leq n$*, where*
  - O_i *is a multiset over* V*, it determines the initial state (contents) of the agent,* $|O_i| = c, c \in \mathbb{N}$*. The number* c *is called a capacity of the colony.*

- $P_i = \{p_{i,1}, \ldots, p_{i,l_i}\}, l_i \geq 1, 1 \leq i \leq k$ *is a finite set of programs for each agent, where each program contains exactly* $h \in \mathbb{N}$ *rules,* h *is called a height. Each rule is in the following form:*
  - $a \rightarrow b, a, b \in V$ *is an evolution rule,*
  - $a \leftrightarrow b, a, b \in V$ *is a communication rule,*
  - $[a_{q,r}] \rightarrow s, a_{q,r} \in V, 0 \leq q, r \leq 2, s \in \{\Leftarrow, \Rightarrow, \Uparrow, \Downarrow\}$ *is a motion rule.* $[a_{q,r}]$ *is a matrix representing the vicinity of an agent.*
- $[o, p],\ 1 \leq o \leq m,\ 1 \leq p \leq n$, *is an initial position of agent* A_i *in the 2D environment,*

– $f \in V$ *is the final object of the colony.*

A configuration of the $2D_{ev}$ P COL is given by the state of the environment – the matrix of type $m \times n$ of multisets of objects over $V - \{e\}$, the states of all agents – the multisets of objects over V, and the coordinates of the agents. An initial configuration is given by the definition of the $2D_{ev}$ P colony.

A computational step of the $2D_{ev}$ P COL is a transition between two consecutive configurations. The computational step consists of four sub-steps. In the first sub-step, a set of the applicable programs of the agents is determined, according to the current configuration of the colony. In the second sub-step, for each agent, one program from this set is chosen. For the chosen programs, it must hold, that there is no collision between the communication rules of each two different programs. In the third sub-step, chosen programs are executed, the values of the environment are updated.

The fourth sub-step is the evolution of the environment. Let $E_{i,j}, 0 \leq i \leq m, 0 \leq j \leq n$ be a multiset representing the contents of the environment at the coordinates $[i, j]$. Let $A_{i,j}, 0 \leq i \leq m, 0 \leq j \leq n$ be a multiset of all the objects forming right sides of the communication rules of the programs chosen in the second sub-step for all the agents at the position $[i, j]$. Consider multisets $S^1_{i,j}, \ldots S^{o_{i,j}}_{i,j}, o_{i,j} \in \mathbb{N}$ such that $\cup_{k=0}^{o_{i,j}} S^k_{i,j} = E_{i,j} \setminus A_{i,j}$. Then the evolution of the environment in this particular derivation step is provided by the application of the evolution rules $S^k_{i,j} \rightarrow T \in R$.

Generally speaking, the objects of the environment, which were not changed by the actions of the agents, are modified by the evolution rules of the environment. The application of the rules of the agents has higher priority over evolution rules of the environment.

A computation is non-deterministic and maximally parallel. The non-determinism means that there is a uniform distribution on all the applicable rules. Hence, if an agent can apply more rules, or there can be applied more rules of the environment in one particular derivation step, only one rule is non-deterministically chosen, but each and every rule can be applied with the same probability. The computation ends by halting, when there is no agent that has an applicable program.

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

3 Analysis of System Requirements and Design of the Application

There are several key aspects to analyzing software development requirements in the context of a 2D P colony simulation. First, we need to specify the functional requirements

that describe the specific behaviors and functions that the software should have in order to accurately simulate the 2D P colony mode. This includes setting rules for agent interactions, updating the environment, and evolving objects in the colony. We require that each element is definable by the user.

Next, non-functional requirements such as expected performance, scalability, and the ability to efficiently handle large-scale simulations need to be considered as well. This will be achieved by using the standardized Matplotlib library for plotting the simulation. [8]

In addition, the user requirements play a crucial role in designing an intuitive and user-friendly interface to interact with the application, allowing users to configure various parameters, visualize the simulation, and analyze the results of the simulation. The Matplotlib library mentioned above allows us to plot agent positions in the environment, including options such as zoom and so on. The configuration of the agent should be viewed in a separate window if needed.

The aim is to create software that meets the real needs of users to use the software to simulate any 2D P colony they define and to provide a reliable and efficient platform for viewing the behavior of the model.

If we focus on specific functional requirements, we can see them in the use-case diagram in Figure 1, and they include the option to define the environment, the agents in it, the agent programs, and most importantly, the ability to run the simulation, adjust its parameters, and see a visualization of the progress of the simulation.

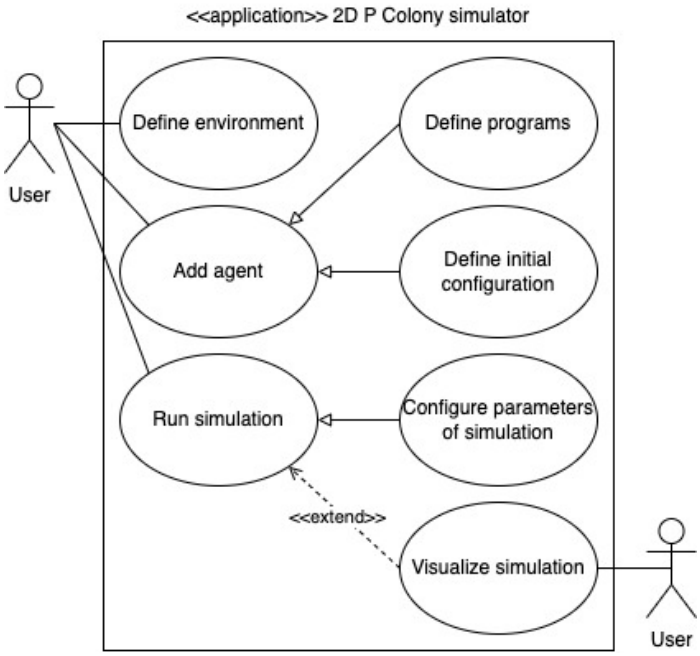


Fig. 1: Use-case diagram of the simulator application.

Let us describe the individual requirements more in detail:

- *Define environment* – option to define an environment of any size $m \times n$ with any content (multiset of objects).
- *Add agent* – Option to create any number of agents, each with its own programs and initial configuration.
  - *Define programs* – The program has to be defined using a file with an easy-to-learn syntax, it has to be also universal and support all types of 2D P colony rules. It has to be possible to define programs for each agent separately, but if multiple agents use the same set of programs, a set of identical agents has to be definable.
  - *Define initial configuration* – For each agent, it has to be possible to define an initial configuration – a multiset of internal objects and a default position in the environment. The initial position within the environment may be generated automatically if not specified.
- *Run simulation* – After initializing the environment and agents, it should be possible to start and pause the simulation and also modify its parameters such as speed, zooming in and out, display of visualized elements, and others.
  - *Configure parameters of simulation* – Customizable parameters should be the simulation speed (redrawing the configuration of all agents) in seconds, marking of significant objects to be colored in the environment, simulation run time in units of the number of programs applied by all agents or seconds.
  - *Visualize simulation* — The application must have a graphical interface that allows visualizing the objects such as the environment, the positions of agents in it, the positions of important objects in the environment, and others (specified by the user), with tools for changing perspective of view (zoom, move, etc.). In addition, it should also have a text interface where additional information such as current agent configurations and potential error messages will be available.

The non-functional requirements include in the first place the simulation speed, which should be smooth enough. More formally, the response time for redrawing a configuration of 100 agents with common programs in a 1000×1000 environment should take no more than 1 second on an average computer with an Intel i5 series processor or higher, unless the user adjusts the simulation speed to a slower speed deliberately using a relevant parameter in the configuration.

The simulation has to be also reliable and stable, which means that the application has to run without crashing and, if it does, appropriate error messages that guides the user to correct the error in the application configuration and continue running without problems has to be displayed.

Based on mentioned requirements, we have designed a software architecture for the application that ensures convenient access to all functionalities and necessary to display and control the elements. The software architecture of the application can be seen on Figure 2.

For the development of the application, we choose Python 3.10 programming language. Python is a great choice for simulator programming due to its simplicity, extensive libraries, scientific computing capabilities, community support, and cross-platform compatibility. In addition, the object-oriented nature of Python allows the

implementation of modular and reusable code, which facilitates the organization and maintenance of simulator components.

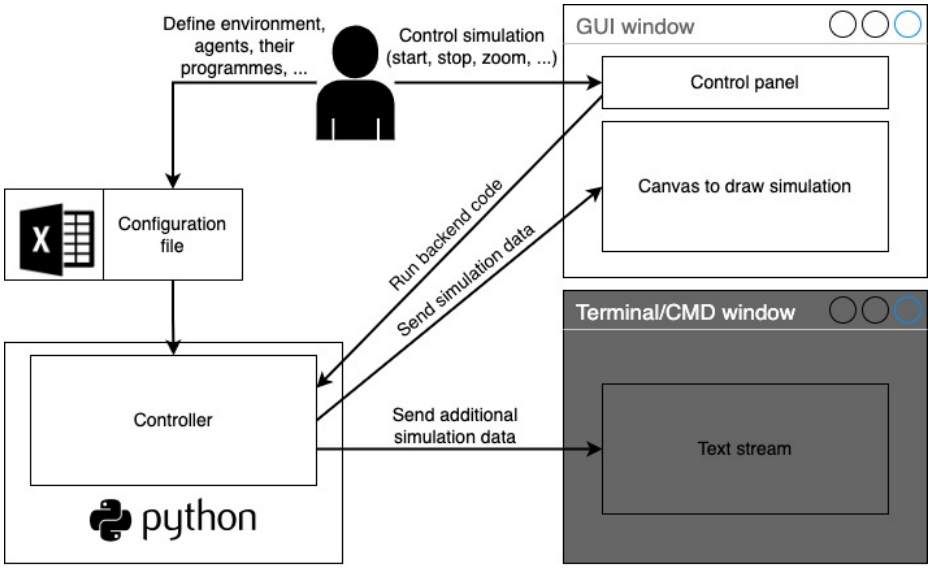


Fig. 2: Software architecture of the application.

In the next section, we will focus on the specifics and implementation of the individual components of the application.

4 Implementation

In this section, we describe the implementation of the application with a focus on the specifics and implementation of the individual components of the application. The simulator provides a computational tool for accurately replicating the behavior and dynamics of 2D P colonies and provides a platform for studying their complex processes. We discuss the technical details and strategies used during the process of the implementation to ensure the accuracy of the simulator in reproducing the key characteristics of 2D P colonies. With a focus on accuracy during implementation, we aim to create a reliable tool that researchers and practitioners can use to analyze and explore 2D P colony simulations in depth.

4.1 Input Parameters

The input parameters for the simulation, including the definitions of the environment, agents, their programs, and visualizations, should be customizable by the user. To fulfill

this requirement, we utilize a standard xslx file with a few sheets and a consistent structure, which can be edited in any spreadsheet software.

For file processing within the application, we create the parsing method *extractExcel* in which we utilize the openpyxl library for Python. This library enables fast and efficient retrieval of the values from individual cells and allows further manipulation with those values as variables.

The first sheet of this xlsx file, called *Parameters*, is reserved for the definition of the basic parameters of the simulation. Here we can define the following:

- *envRows* and *envColumns* – integer values which define the size of the environment,
- *envRules* – the number of environmental rules which will be defined in a separate xlsx file (see below),
- *steps* – integer value defining a termination criterion as the maximum number of computational steps (performed by each agent).
- *animationDelay* – integer value defining the redraw speed (pause between new iteration of computation) during visualization in seconds,
- *JokerSymbols* – block starting with keyword *JokerSymbolsBegin* and ending with keyword *JokerSymbolEnd* allows the definition of a substitute character for alphabet objects that share a similar meaning, such as objects $O_1, O_2, ..., O_n$ (these objects may express the same meaning but with varying "intensity".) The substitute character (e.g., "*", "+", etc.) is written in the second column, while the comma-separated objects represented by this character are written in the third column on the same row within this block.
- *ColorSettings* – block starting with keyword *ColorSettingsBegin* and ending with keyword *ColorSettingsEnd* enables the definition of colors for significant objects that should be displayed in the visualization. The object that should be displayed is written in the second column, and its color is written in the third column on the same row within this block.

To enhance clarity, we use the first column of the sheet mainly to define keywords that are descriptive and facilitate user orientation. The actual values are defined in the second column, or in the second and third columns if it is a definition block. In the case where multiple values need to be entered, we use a comma as a separator. This structure is clear and makes it easy to add additional parameters in the future.

The second sheet of the xlsx file, called *Environment* is used to define the environment. Here we define the multisets of objects in the x and y coordinates, each cell can contain several objects separated by a comma or can be empty (but the environmental symbol e is still available for the agent). The application operates only with an environment of a size determined by the variables *envRows* and *envColumns*, defined in the *Parameters* sheet. An example of a simple environment definition with a size of 3×3 and several objects can be seen in Table 1.

The next sheet of xlsx file, called *Environmental rules*, is reserved for defining the environmental rules. The syntax is as follows: in the first column, we enter the multiset of objects S (see 2D P colony definition above, it is the left-hand side of the rule); in the second column on the same row, we enter the rule type (currently only evolutionary rules are supported, represented by the symbol $>$); and in the third column, we enter the multiset of objects T (the right-hand side of the rule). The application only reads a

a,b,c	d	a
a	b,c	d
b	b,d,e	o,p

Table 1: 3×3 environment definition

specific number of rules, defined in the *Parameters* sheet. An example of two simple environment rule's definition can be seen in Table 2.

a,b	>	a
d	>	c

Table 2: Two environmental rules definition

The last sheet of the xlsx file, called *Agents*, contains the definition of the agents. Here the user can define individual agents in blocks starting with the keyword *AgentBegin* and ending with the keyword *AgentEnd*, which is written in the first column of the sheet. Within this block, one can specify the parameters of an agent in a sequential manner, beginning with the definition of the name of the agent using the keyword *ID*, followed by the initial contents (a multiset of objects) using the keyword *content*. Additionally, user can specify the position of the agent in the environment on the i-axis using the keyword *start i* and on the j-axis using the keyword *start j*. These parameters are entered sequentially on the same line following the *AgentBegin* keyword, with each keyword-value pair written in the subsequent columns. Next, within this block, user can also define programs using a nested block starting with the keyword *programBegin* and ending with the keyword *programEnd*. These keywords are defined in the second column of the sheet. Inside the program block, we write individual rules starting with the keyword "rule" on the same row within this block and continuing with defining the necessary configuration for activating the rule in the third column, the rule type (evolutionary, motion, ...) in the fourth column, and possibly the right side of the rule in the fifth column (for a motion rule it is not necessary, just specify the direction l as left, r as right, u as up, or d as down as the rule type). A simple example of an agent definition can be seen in Table 3.

AgentBegin	ID	Agent1	contents	e,x	start i	2	start j	2	copies	1
	programBegin									
	rule	e,a	u							
	rule	z	>	e						
	programEnd									
AgentEnd										

Table 3: Two environmental rules definition

The definition of an agent named *Agent1* is shown in Table 3. The agent contains internal objects e and a, and the initial position in the environment at position $2, 2$. Let us also note the attribute called *Copies*, which allows the creation of multiple copies of the same agent. In this case, the ID of the agent is generated using the index ID underscore the copy number.

4.2 2D P colony Controller

The implementation of the class Agent involves the definition of the necessary attributes and methods for representing and manipulating agents within the simulation system. The class agent serves as a template for creating individual instances of the object agent with specific properties and behaviors. Here is the list of the key attributes for the implementation of the class agent:

- *contents* – each agent contain a multiset of objects limited by the capacity of a particular 2D P colony,
- *programs* – each agent has its own set of programs that influence its behavior,
- *coordinates* – the current position of the agent in the environment (coordinates),
- *vicinityLength* – the environment of the agent that "sees",
- *colony* – each agent is a subclass of the class Colony,
- *jokerSymbols* – allows the agent to work with substitute symbols if they are defined.

Each of these attributes is assigned to the agent during initialization based on the defined values in the configuration file as described in the previous subsection.
The next key step is to implement methods that encapsulate the behavior and actions of the agent. These methods include following:

- *getVicinity* – this method reads the contents of the environment at all neighboring points around the agent, the method is mainly used in movement rules,
- *getEnvironmentContent* – this method reads the content of the environment at the current position of the agent,
- *getApplicablePrograms* – this method loops through all agent programs defined in the configuration file and creates a list of applicable programs corresponding to the current configuration,
- *applyProgram* – this method randomly selects one of the applicable programs (motion, evolving, or exchange) and applies it, which means that the agent (its position in the environment or internal content) or environment (its contents in the position where the agent is) configuration is changed.

Each of the agents is a subclass of the Colony class. This class groups the agents, takes care of updating the environment, and controls the computation. The computation of the colony is performed by calling the following methods:

- *initComputationalStep* – initializes the toConcat2env variable to collect the requests of the agent to write to the environment via an exchange rule,
- *agentsAct* – this method, in random order (a simulation of a maximal parallelism and non-determinisms), asks each agent to run one of the applicable programs.

- *evolveEnvironmet* – if defined, this method applies environment rules in all positions where possible,
- *add2environment* – this method ensures communication with the environment in case some of the agents uses the communication (exchange) rule, it ensures the insertion of the original agent object into the environment based on its request.

It should be noted that the specific behavior of a 2D P colony strictly depends on the specific definition specified via the input xlsx file.

4.3 GUI and Visualisation Tools

Graphical user interface (GUI) is an important part of the 2D P colony simulator. It serves as a visual representation of the simulated environment, agents, and their interactions, providing users with a clear and intuitive way to interact with the simulation. The GUI enables users to monitor the behavior of 2D P colony with specific programs of the agents, observe real-time changes, and analyze the outcomes of different scenarios.

As mentioned in the previous text, we used the Matplotlib library for Python to render the simulation. This provides all the necessary tools to control the graphical interface. In the following lines, we describe methods providing the data for rendering.

The key and first called method is *initPopulation*, which handles the initialization of the graphical window and its components. It takes the input information (parameters) about the size of the environment, position of the agent, and colors for significant elements to be displayed in the visualization. This information is obtained from the class Colony, the object in the controller of 2D P colony.
First, the *Pyplot* graphical window is created using the matplotlib library. The window displays the environment and its objects, such as agents. The *imshow* method from the matplotlib library is used for rendering, taking the environment size and agent positions as input arguments. By default, the canvas (environment) is gray, and agents are represented as the blue dots. Additional significant objects, for which colors are defined in the input xlsx file, are rendered using a separate function called *markInterest*. This function iterates through each position in the environment and, if it finds a significant object, colors its position.

Control tools, like movement, panning, zooming, taking screenshots, etc., are part of the pyplot canvas and do not require additional definitions. However, the application can be extended with additional elements from the standard Tkinter library, such as labels that provide information about the state of the simulation (running, current iteration, etc.) or buttons for simulation control, implemented from there. Additional elements in the application window are arranged using the *grid* geometry manager, which allows for organizing them in a tabular structure. In our case, we only add the necessary element, which is a label informing about the current program state.

To ensure user interaction during control simulation, the *keypress* events can be added to define keyboard shortcuts. For instance, in our case, the *c* key is used to terminate the application (sys.exit), and the *p* key pauses the simulation for 5 seconds (pyplot.pause(5)). It is important to note that certain keys may be reserved by the matplotlib library, see its documentation for more information [8].

At this point, we have initialized the graphical window with all the necessary components. The matplotlib canvas displays the environment with agents at their respective coordinates, along with any other significant objects that have been assigned specific colors.

For next updates (redraw to animate computation), it is not necessary to recreate these elements, but rather update them. To achieve this, we defined a method called *updatePlot*, which works as follows:

1. it retrieves the current information about the colony (environment and agent positions),
2. then it modifies the labels if needed, such as updating the iteration count label,
3. then it highlights significant symbols in the environment using the predefined colors,
4. and finally, it pauses the computation for a defined time period (as specified in the input xlsx file) to allow the user to observe the changes.

By implementing the *updatePlot* method, we can efficiently update and display the changes in real-time without recreating the entire plot. This method is called in the main application loop while the calculation is in progress.

When the simulation is completed, we call the termination method, *endOfVisualization*. This method updates the labels to display *End of simulation* and keeps the final configuration displayed on the canvas until a user presses any key to exit. This allows the user to review the last configuration before closing the visualization. The final GUI example can be seen in Figure 3.

5 The Function Main, Flow and Execution

The function main of the program orchestrates the execution flow and serves as the entry point for the application. The application is launched in the standard way using the Python interpreter.

As we already mentioned, our application utilizes an input file containing parameters for the 2D P colony and its simulation. This xlsx file needs to be loaded and parsed. The loading process is handled by the openpyxl library in our *extractExcel* method, which requires a specific file structure as described in the previous sections. This method takes a file as input, which can be specified as an absolute path or defined as an input attribute using sys.argv.

Once the file is parsed and all the values are loaded into the related variables of the newly created Colony object, the GUI with the visualizer can be initialized by the method *initPopulation*.

Afterwards, the main while loop of the application is initiated, within which the following steps are executed until the termination criterion (e.g. maximum number of steps) is reached:

1. firstly, the controller performs a computation step of the 2D P colony, during which the state of agents and the environment is updated.
2. then, the output is passed to the GUI for visualization through the updatePlot method.
3. additional information about agent configuration and possible error messages can be entered into the operating system's text console (command line or terminal).

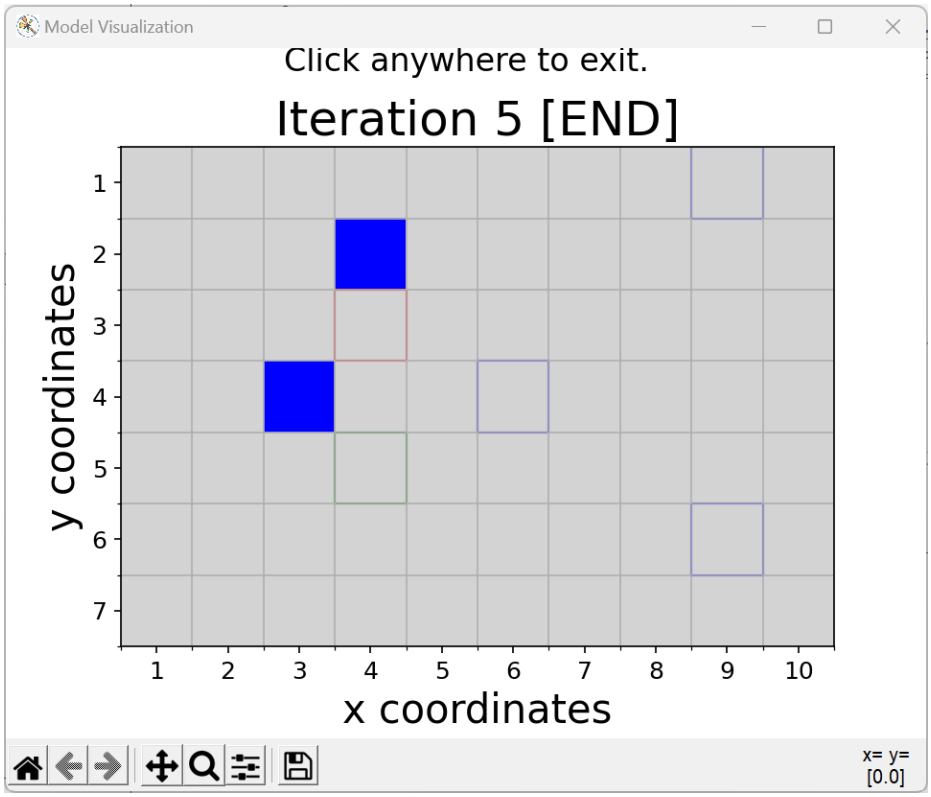


Fig. 3: The final GUI shows a 10×7 environment with two agents and several significant points highlighted in color.

Finally, when the computation is complete, the state at the last computation step remains in the GUI and the agent position information is written to the text console.

6 Conclusion

In this paper, we discussed the process of design and implementation of a 2D P colony simulator, which is a very valuable tool for studying and analyzing the dynamics of this model with specific programs of the agents. Through the use of agent-based modeling, customizable input parameters, and an intuitive graphical user interface, the simulator allows researchers and enthusiasts to explore the behavior and emergent properties of 2D P colonies in a simulated environment. Unlike our previous simulators, where the entire colony was hard-coded in the source code, this parametrical approach allows us to change the definition of a colony using only the parameters excel files. Recently, we continue in our research of implementation of the ant colony optimization, using our new simulator.

Acknowledgments

Research is partially supported by the Silesian University in Opava under the Student Funding Plan, project SGS/11/2023

References

1. Bo, W., Fang, Z. B., Wei, L. I., Cheng, Z. F., & Hua, Z. X. (2021). Malicious URLs detection based on a novel optimization algorithm. IEICE Transactions on Information and Systems, 104(4), 513–516. https://doi.org/10.1587/transinf.2020EDL8147. Released on J-STAGE April 01,. (2021). Online ISSN 1745–1361. Print ISSN, 0916–8532.
2. Buiu, C., et al. http://membranecomputing.net/
3. Cienciala, L., Ciencialová, L., Perdek, M. 2D P colonies. In: Csuhaj-Varjú E., Gheorghe M., Rozenberg G., Salomaa A., Vaszil Gy. (eds) Membrane Computing. CMC 2012. Lecture Notes in Computer Science, vol 7762. Springer, Berlin, Heidelberg, pp. 161–172 (2012), DOI: 10.1007/978-3-642-36751-9_12
4. Deng, X., Dong, J., Wang, S., Luo, B., Feng, H., Zhang, G. Reducer lubrication optimization with an optimization spiking neural P system, Information Sciences, Volume 604, 2022, Pages 28-44, ISSN 0020-0255, https://doi.org/10.1016/j.ins.2022.05.016.
5. Dong, J., Zhang, G., Luo, B., Yang, Q., Guo, D., Rong, H., Zhu, M., Zhou, K. A distributed adaptive optimization spiking neural P system for approximately solving combinatorial optimization problems, Information Sciences. Volume 596, 2022, Pages 1-14, ISSN 0020-0255, https://doi.org/10.1016/j.ins.2022.03.007.
6. Florea, A. G., Buiu, C. . Development of a software simulator for P colonies. Applications in robotics, International Journal of Unconventional Computing, Vol. 12, 2-3, pp. 189-205, 2016
7. Gheorghe, M., Stamatopoulou, I., Holcombe, M., Kefalas, P. Modelling Dynamically Organised Colonies of Bio-entities. In: Banâtre, JP., Fradet, P., Giavitto, JL., Michel, O. (eds) Unconventional Programming Paradigms. UPP 2004. Lecture Notes in Computer Science, vol 3566. Springer, Berlin, Heidelberg. DOI: 10.1007/11527800_17

8. Hunter, JD. Matplotlib: A 2D Graphics Environment. Computing in Science and Engineering, vol. 9, no. 3, pp.
9. Huang, L., Sun, L., Wang, N., Jin, X. Multiobjective Optimization of Simulated Moving Bed by Tissue P System, Chinese Journal of Chemical Engineering, Volume 15, Issue 5, 2007, Pages 683-690, ISSN 1004-9541, https://doi.org/10.1016/S1004-9541(07)60146-3.
10. Kelemen, J., Kelemenová, A., Păun, G. Preview of P colonies: A biochemically inspired computing model. In: Workshop and Tutorial Proceedings. Ninth International Conference on the Simulation and Synthesis of Living Systems (Alife IX). pp. 82–86. Boston, Massachusetts, USA (September 12-15 2004)
11. Langer, M., Valenta, D., On Evolving Environment of 2D P Colonies – Ant Colony Simulation. Journal of Membrane Computing. 2023; to appear
12. Păun, G. *Computing with membranes.* Journal of Computer and System Sciences 61, 2000, pp. 108–143.
13. Păun, G. *Introduction to membrane computing*. Applications of Membrane Computing. Springer BerlinHeidelberg, 2006.

Part III

Informal talks

SNP Systems with Astrocytes Producing Calcium: Power and Efficiency

Bogdan Aman[1,2] and Gabriel Ciobanu[1,2]

[1] Alexandru Ioan Cuza University of Iași, Romania
[2] Romanian Academy, Institute of Computer Science
{bogdan.aman,gabriel}@info.uaic.ro

Spiking neural P systems with astrocytes producing calcium [2] differ from the standard spiking neural P systems in several ways: we have a new type of resources called calcium unit alongside the standard spike, we have a new type of place called astrocyte alongside the standard neurons and we also have dedicated synapses for communicating calcium units alongside those used for communicating spikes.

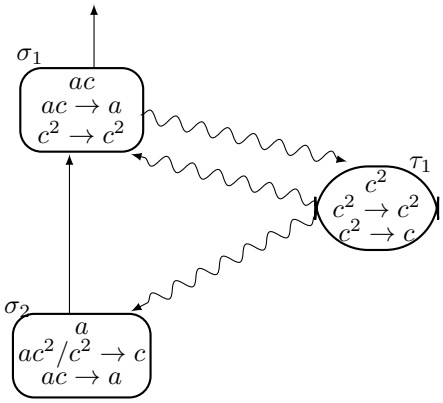


Fig. 1: A spiking neural P system with astrocytes producing calcium generating all even numbers without using either forgetting rules or delay in the evolution rules. We use rectangles and ellipses to depict the neurons and astrocytes, respectively. We also use straight and snake like arrows to depict the synapses that can send only spikes and calcium units, respectively. The straight arrow without a target leaving neuron σ_1 indicates that this is an output neuron that can send spikes into the environment.

Let $N_2(\Pi)$ be the set of numbers computed by Π, where the subscript 2 denotes the way in which the result of a computation is defined (namely the number of steps between the first 2 spikes). Additionally, $N_2SNP^k_{m,n}$ denotes the families of all sets $N_2(\Pi)$ computed by a spiking neural P system with astrocytes producing calcium, consisting of at most m neurons, n astrocytes and k rules in every neuron or astrocyte. Note that if one of the parameters k, m or n is not limited, then the symbol $*$ is used to substitute it.

Theorem 1 ([2]). $N_2SNP^4_{*,*} = NRE$.

The set of numbers accepted by the system Π is denoted by $N_{acc}(\Pi)$, where the subscript acc indicates that the system works in the accepting mode. We denote by $N_{acc}SNP^k_{m,n}$ the families of all sets $N_{acc}(\Pi)$ accepted by the spiking neural P system with astrocytes producing calcium containing at most m neurons, n astrocytes and k rules in every neuron or astrocyte of the system.

Theorem 2 ([2]). $N_{acc}SNP^4_{*,*} = NRE$.

Without delays or forgetting rules we have:

Theorem 3 ([2]). $N_2SNP^4_{*,*}(delay_0, forg_0) = NRE$.

Theorem 4 ([2]). $N_{acc}SNP^4_{*,*}(delay_0, forg_0) = NRE$.

Non-deterministic spiking neural P systems with astrocytes producing calcium that do not consider forgetting rules nor delay in the evolution rules, are powerful enough to provide polynomial-time solutions to the subset sum problem [1]. We provided four ways of constructing such a system: (i) semi-uniform: we constructed a spiking neural P system with astrocytes producing calcium for each instance of the subset problem and embedded the parameters into the constructed systems: in the rules and number of initial resources, in the rules and in the number of used neurons and astrocytes (Figure 2), and as number of resources (spikes and calcium units) and in the number of used neurons and astrocytes ; (ii) uniform: we constructed a spiking neural P system with astrocytes producing calcium for all instances of the same size of the subset problem and provided the parameters as number of spikes and calcium units.

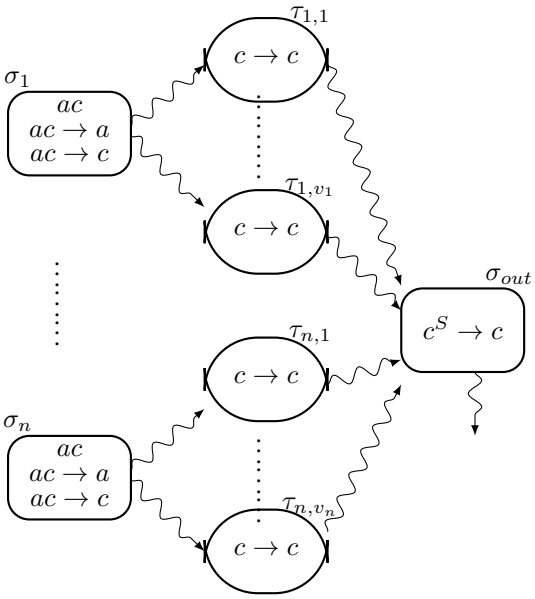


Fig. 2: A semi-uniform spiking neural P system with astrocytes producing calcium solving the subset sum problem without using either forgetting rules or delay in the evolution rules. The instance of the problem is encoded in the system in the rules and in the number of used places (neurons and astrocytes).

References

1. Aman, B.: Solving subset sum by spiking neural p systems with astrocytes producing calcium. Natural Computing **22**(1), 3–12 (2023). https://doi.org/10.1007/s11047-022-09900-7
2. Aman, B., Ciobanu, G.: Spiking neural p systems with astrocytes producing calcium. International Journal of Neural Systems **30**(12), 2050066 (2020). https://doi.org/10.1142/S0129065720500665

Communication Mechanisms in Networks of Reaction Systems

Erzsébet Csuhaj-Varjú and Pramod Kumar Sethy

Department of Algorithms and Their Applications
Faculty of Informatics, Eötvös Loránd University
Budapest, Hungary
{csuhaj,pksethy}@inf.elte.hu

In this talk, we discuss new variants of networks of reaction systems, with some new constraints for communication. To build the models, we take ideas from networks of reaction systems [1], communicating reaction systems with direct communication [3], and parallel communicating grammar systems [2].

Reaction systems were introduced by A. Ehrenfeucht and G. Rozenberg as a formal model of interactions between biochemical reactions. The main idea was to model the behavior of biological systems in which a large number of individual reactions interact with each other. Briefly, a reaction system consists of a finite set of objects that represent chemicals, called reactants, and a finite set of reactions. Each reaction is a triplet of three nonempty finite sets: the set of reactants, the set of inhibitors, and the set of products. Let T be a set of reactants. A reaction is enabled for T and it can be performed if all of its reactants are present in T and none of its inhibitors is in T. When the reaction is executed, then the set of its reactants is replaced by the set of its products. All enabled reactions are applied in parallel. The final set of products is the union of all single sets of products that were obtained by the reactions that were enabled for T.

A network of communicating reaction systems with direct communication is a virtual graph where in each node a reaction system and a finite set of reactants are located. The reaction systems, also called the components of the network, perform reactions on the set of reactants they have and after then send reactants or reactions to certain components according to the given communication protocol. The components work in a synchronized manner, governed by a global clock.

Two variants of networks of reaction systems with direct communication were introduced and studied in [3]. The first model, the cdcR(p) system, communicates products, that is every product of each reaction is associated with a set of target components to where a copy of the product is communicated. In the case of the second model, the cdcR(r) systems, target components are associated with the reaction itself. If the reaction is successfully performed, then a copy of it is sent to every associated target component. In [3] it was shown that these types of networks of reaction systems can be represented by single reaction systems and demonstrated that cdcR(p) systems simulate cdcR(r) systems.

These two types of systems communicate by command, i.e., after performing the reaction, communication is compulsory. The new models we introduce use communication by request. The condition for communication is formulated by a positive reply to a query, that is, by a logical constraint. These models are strongly motivated by parallel communicating grammar systems [2].

The first new model works as follows: before performing its enabled reactions, each component requests some additional elements (reactants) from certain given component(s). This action is called a query request. Once the component gets all requested elements, then these reactants are added to the set of reactants of the reaction. If the obtained reaction is enabled for the current set of reactants of the component, then it is performed. If the query is not satisfied or the new reaction is not enabled, then the reaction remains unchanged and is not allowed to be performed at that computation step. These constructs are called communicating reaction systems communicating by requests (cdcR(q) systems, in short). It can be shown that cdcR(q) systems can be represented by single reaction systems and cdcR(p) systems simulate cdcR(q) systems.

The second new model, called communicating reaction system communicating by partial requests (the cdcR(pq) system, in short), works analogously to cdcR(q) systems, except that the query only checks the presence or absence of the reactant at the other given component. The checked reactant from that node is not sent to the requesting node. If the query is satisfied and the reaction is enabled, then it is performed, otherwise, it cannot be performed at that computation step. Notice that this communication protocol introduces some kind of synchronization in the joint work of the components. As in the previous case, it can be shown that cdcR(pq) systems can be represented by single reaction systems and its state sequence can be represented as a map of the state sequence of a cdcR(p) system.

Finally, we propose some open problems for future research on the relationship between communication command and communication by request in networks of reaction systems.

References

1. Bottoni, P., Labella, A., Rozenberg, G.: Networks of reaction systems. Int. J. Found. Comput. Sci. **31**(1), 53–71 (2020)
2. Csuhaj-Varjú, E., Dassow, J., Kelemen, J., Păun, G.: Grammar Systems: A Grammatical Approach to Distribution and Cooperation. Gordon and Breach, London (1994)
3. Csuhaj-Varjú, E., Sethy, P.K.: Communicating reaction systems with direct communication. In: Freund, R., Ishdorj, T.O., Rozenberg, G., Salomaa, A., Zandron, C. (eds.) Membrane Computing -21st International Conference CMC 2020, Vienna, Austria, September 14-18, 2020, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12687, pp. 17–30. Springer (2021)

Author Index