13th Brainstorming Week on Membrane Computing

Sevilla, February 2 – 6, 2015

Luis F. Macías-Ramos
Gheorghe Păun
Agustín Riscos-Núñez
Luis Valencia-Cabrera
Editors

13th Brainstorming Week on Membrane Computing

Sevilla, February 2 – 6, 2015

Luis F. Macías-Ramos
Gheorghe Păun
Agustín Riscos-Núñez
Luis Valencia-Cabrera
Editors

RGNC REPORT 1/2015

Research Group on Natural Computing
Universidad de Sevilla

Fénix Editora, Sevilla, 2015

ISBN: 978-84-84-944366-2-8
Edita: Fénix Editora.
info@fenixeditora.com
Movil 620 98 36 94 - SEVILLA

Contents

Deterministic Non-cooperative P Systems with Strong Context Conditions

Artiom Alhazov[1], Rudolf Freund[2]

[1] Institute of Mathematics and Computer Science, Academy of Sciences of Moldova
Academiei 5, Chişinău MD-2028 Moldova
E-mail: artiom@math.md
[2] Faculty of Informatics, Vienna University of Technology
Favoritenstr. 9, 1040 Vienna, Austria
E-mail: rudi@emcc.at

Summary. We continue the line of research of deterministic parallel non-cooperative multiset rewriting with control. We here generalize control, i.e., rule applicability context conditions, from promoters and inhibitors checking presence or absence of certain object up to some bound, to regular and even stronger predicates, focusing at predicates over multiplicity of one symbol at a time.

1 Introduction

It is known, see [7], that non-cooperative P systems with atomic promoters or atomic inhibitors characterize $ET0L$, while using either one catalyst, see [6], [3], or promoters or inhibitors of weight 2, see [4], leads to the computational completeness of non-cooperative P systems. A question about the power of deterministic systems was posed in [5], inspired by the fact that all identical objects have the same behavior in the same context. This question was answered in [1]: deterministic non-cooperative P systems have weak behaviour, namely, only accepting finite number sets and their complements, even using generalized context conditions (except the sequential case, when they keep the computational completeness).

Generalized context conditions of rule applicability are defined as a list of pairs (p_i, F_i), $1 \leq i \leq k$, applicable to a rule if at least one condition applies, in the following way: p_i, called promoter, must be a submultiset of the current configuration (or the contents of the current region), and none of the elements of F_i, called inhibitors, are allowed to be submultisets of the current configuration (or the contents of the current region). A subsequent paper, [2], precisely characterized the power of priorities alone, as well as established how much power of promoters and inhibitors is actually needed to reach $NFIN \cup coNFIN$. Already in [1] it has been shown that generalized context conditions are equivalent to arbitrary predicates on boundings, i.e., all boolean combinations over conditions $< m$ (and,

hence, also $\geq m$, $> m$, $\leq m$, $= m$ and $\neq m$) for multiplicities of symbols. In other words, generalized context conditions are able to check exactly the multiplicities of symbols up to an arbitrary fixed bound m. In this paper we consider stronger context conditions.

2 Definitions

Let O be a finite alphabet. In this paper we will not distinguish between a multiset, its string representation (having as many occurrences of every symbol as its multiplicity in the multiset, the order in the string being irrelevant), and a vector of multiplicities (assuming that the order of enumeration of symbols from O is fixed). By O° we denote the set of all multisets over O. By a strong context in this paper we mean a language of multisets, i.e., a subset of O°.

Let $a \in O$ and $u \in O^\circ$, then $a \to u$ is a non-cooperative rule. The rules are applied in the maximally parallel way, which in the case of our interest, i.e., for deterministic non-cooperative P systems, correspond to replacing every occurrence of each symbol a by the corresponding multiset u from the right side of the applicable rule (if there is any; no competition between different rules can happen due to the determinism). Let region j of a membrane system contain multiset w.

Then rule $a \to u$ with a strong context condition $C \subset O^\circ$ (written $a \to u|C$) is applicable if and only if $|w|_a > 0$ and $w \in C$. Consider the following examples:

- a singleton atomic promoter $s \in O$ corresponds to the context $+(s) = \{w \in O^\circ \mid |w|_s > 0\}$; we denote this feature by $pro_{1,1}$;
- a singleton atomic inhibitor $s \in O$ corresponds to the complementary context condition: $-(s) = \{w \in O^\circ \mid |w|_s = 0\}$;
- a singleton promoter $s \in O^\circ$ of a higher weight corresponds to the context $+(s) = \{w \in O^\circ \mid s \subseteq w\}$;
- a singleton inhibitor $s \in O^\circ$ of a higher weight corresponds to the complementary context condition: $-(s) = \{w \in O^\circ \mid s \not\subseteq w\}$;
- a (finite) promoter-set $S \subset O^\circ$ corresponds to the context $+(S) = \bigcup_{s \in S} +(s)$, i.e., at least one promoter must be satisfied;
- a (finite) inhibitor-set $S \subset O^\circ$ corresponds to the complementary $-(S) = \bigcap_{s \in S} -(s)$, i.e., any inhibitor can forbid the rule;
- a promoter-set P and an inhibitor-set Q together are called a *simple context condition*, written (P, Q); it corresponds to the strong context condition $+(P) \cap -(Q)$;
- context conditions as considered in [1] and [2] constitute a finite collection of simple context conditions $(P_1, Q_1), \cdots, (P_m, Q_m)$, they correspond to the strong context condition $\bigcup_{1 \leq i \leq m} (+(P_i) \cap -(Q_i))$, and were shown to be equivalent to predicates on boundings[3];

[3] the meaning of a promoter-set in [3] is different, but the computational power results are equivalent up to the descriptional complexity parameters such as number of promoters/inhibitors and their weights

- a bounding b_k is an operation on a multiset, for any symbol preserving its multiplicity up to k, or "cropping" it down to k otherwise; a predicate on bounding can be specified by a finite set M of multisets with multiplicities not exceeding k; it corresponds to a strong context condition $\{w \in O^\circ \mid b_k(w) \in M\}$, and can express precisely all boolean combinations of conditions $|w|_a < j$, $a \in O$, $1 \leq j \leq k$;
- a regular strong context condition can be specified by a regular multiset language, or as a Parikh image of a regular string language; e.g., $\mathtt{Eq}(a, b) = \{w \in O^\circ \mid |w|_a = |w|_b\}$ is an example; we denote the family of such conditions by $ctxt(REG)$;
- if a strong context condition only depends on the multiplicities of k symbols from O (and all other symbols do not affect the applicability), we represent this property by a superscript k of $ctxt$; for instance, if we denote the symbols mentioned above by $S = \{s_1, \cdots, s_k\}$, then $ctxt^k(REG) = \{\{u \cup v \mid u \in L,\ v \in (O \setminus S)^\circ\} \mid L \subseteq S^\circ,\ L \in PsREG\}$; hence $\mathtt{Eq}(a, b) \in ctxt^2(REG)$; by $ctxt(\mathtt{Eq})$ we denote being able to compare the multiplicities of two symbols (for different pairs of symbols separately) for being equal, together with the complementary condition;
- to stay within Turing computability of the resulting P systems, in this paper we only consider recursive context conditions, i.e., multiset languages with decidable membership, denoted by $ctxt(REC)$;
- if a one-symbol strong context condition only depends on the multiplicity of one symbol, it can be specified by a predicate over $\mathbb{N}$; e.g., $\mathtt{Sq}(a) = \{w \in O^\circ \mid |w|_a = k^2,\ k \geq 0\}$ and $\mathtt{Sq}'(a) = \{w \in O^\circ \mid |w|_a = k^2,\ k \geq 1\}$ are examples; hence, $\mathtt{Sq}, \mathtt{Sq}' \in ctxt^1(REC)$; by $ctxt(\mathtt{Sq})$ or $ctxt(\mathtt{Sq}')$ we denote being able to test the multiplicities (of different symbols separately) for squares (including zero or not, respectively), together with the complementary condition.

3 Regular conditions

Theorem 1. $Ps_aDOP_1(ncoo, ctxt^2(REG)) =$
$Ps_aDOP_1(ncoo, ctxt(\mathtt{Eq})) = PsRE.$

Proof. Consider an arbitrary register machine M with m registers. For each working register i, $1 \leq i \leq m$, we represent its value by the difference of the multiplicities of associated objects a_i and b_i. Hence, increment can be performed by producing one copy of a_i, decrement can be performed by producing one copy of b_i, and zero can be distinguished from non-zero by the following regular conditions:

$$Z_i = \{w \in O^\circ \mid |w|_{a_i} = |w|_{b_i}\} = \mathtt{Eq}(a_i, b_i),\ 1 \leq i \leq m,$$
$$P_i = \{w \in O^\circ \mid |w|_{a_i} \neq |w|_{b_i}\} = O^\circ \setminus \mathtt{Eq}(a_i, b_i),\ 1 \leq i \leq m,$$

We construct the following P system:

$$\Pi = (O, \Sigma, \mu = [\ ]_1, w_1 = q_0, R_1), \text{ where}$$

$$
\begin{aligned}
O &= Q \cup T \cup \{a_i, b_i \mid 1 \leq i \leq m\},\\
\Sigma &\subseteq \{a_i \mid 1 \leq i \leq m\},\\
R_1 &= \{q \to a_i q' \mid q : (ADD(i), q') \in P\}\\
&\quad \cup \{q \to b_i q' | P_i,\ q \to q'' | Z_i \mid q : (SUB(i), q', q'') \in P\}.
\end{aligned}
$$

□

If only regular conditions over *one* symbol are allowed, then we expect the power of such P systems to be much more limited.

4 Stronger Conditions

Consider one-symbol context conditions that are even stronger than regular.

It is expected that, with recursively enumerable conditions over one number we get something like $NRE \cup coNRE$, so we look at intermediate cases. We look at ways of obtaining RE by encoding a number by a multiplicity of *one* object, say, a_i, in such a way that increment and decrement are reasonably simple to perform by non-removable objects. We propose the following encoding: "ignoring the greatest square", i.e., number $n = k^2 + t$ encodes t if $0 \leq t < 2k + 1$. In this way, zero-test becomes a test whether the encoding number is a perfect square. Increment is performed as increment of the encoding number, followed by addition of $2k + 1$ if the next perfect square, i.e., $(k + 1)^2$, is reached. Decrement can thus be done by adding $2k$ to the encoding number. The value k can be stored as the multiplicity of another non-removable object, say, b_i, whose multiplicity should be incremented each time the encoding number is increased by $2k$ or by $2k + 1$. Putting it all together, the following construction is obtained:

$$
\begin{aligned}
Z_i &= \{w \in O^\circ \mid |w|_{a_i} = k^2,\ k \geq 0\} = \mathtt{Sq}(a_i),\ P_i = O^\circ \setminus Z_i,\ 1 \leq i \leq m,\\
&\quad \text{We construct the following P system:}\\
\Pi &= (O, \Sigma, \mu = [\ ]_1, w_1 = q_0, R_1), \text{ where}\\
O &= Q \cup T \cup \{a_i, b_i \mid 1 \leq i \leq m\},\\
\Sigma &\subseteq \{a_i \mid 1 \leq i \leq m\},\\
R_1 &= \{q \to a_i \tilde{q},\ \tilde{q} \to q' | P_i,\ \tilde{q} \to \hat{q} | Z_i,\ \hat{q} \to a_i b_i q',\ b_i \to a_i a_i b_i |_{\hat{q}}\\
&\quad \mid q : (ADD(i), q') \in P\}\\
&\quad \cup \{q \to q'' | Z_i,\ q \to \hat{q} | P_i,\ \hat{q} \to b_i q',\ b_i \to a_i a_i b_i |_{\hat{q}}\\
&\quad \mid q : (SUB(i), q', q'') \in P\}.
\end{aligned}
$$

Yet there is a major drawback of this result established above in comparison with the result from Theorem 1, as the input has to be encoded: given a number n_i for input register i, we have to compute numbers $n_i + k_i^2$ and k_i, such that $k_i^2 \leq n_i \leq k_i^2 + 2k_i$. But this is an algorithm which is not difficult to be implemented;

also our context condition for testing a number to be a perfect square does not require a difficult algorithm.

Hence, we have just shown the following result, where the index wa instead of a in $Ps_{wa}DOP_1(ncoo, pro_{1,1}, ctxt(\mathtt{Sq}))$ indicates weak computational completeness as for having to encode the input:

Theorem 2. $Ps_{wa}DOP_1(ncoo, ctxt^1(REC)) = PsPs_{wa}DOP_1(ncoo, pro_{1,1}, ctxt(\mathtt{Sq})) = PsRE.$

Adding rules $a_i \to \lambda|_{q_f}$, $b_i \to \lambda|_{q_f}$ and $q_f \to \lambda$ for $1 \leq i \leq m$, where q_f is the final state of the simulated register machines, we even obtain the clean result, i.e., halting without additional objects, still preserving determinism.

We can strengthen the claim of Theorem 2 by showing **strong** computational completeness (in the sense of deterministic acceptance and even deterministic way of computing functions). Without restricting the power of register machines, we assume that in the simulated register machine, the output registers are never decremented. Then, for the output registers, we replace the simulation of each increment instructions with a single rule $q \to a_i q'$, where $q : (ADD(i), q') \in P$ and i is an output register. In this way, the output will be produced without encoding.

It remains to show that P systems with strong context conditions over one symbol can simulate register machines where also the **input** is not encoded. We use the following idea. To represent the input N of a register in the way the P system constructed in the proof of Theorem 2 needs it, we first describe how to get two numbers x_N and y_N such that N is a function of x_N and y_N, and, moreover, by computing these two numbers from N, we get their representation in the form we need them as for the P system constructed in Theorem 2.

First we explain the algorithm how to obtain x_N and y_N: Starting with N represented by N copies of an object c_N, the multiplicity of these input objects is incremented until it becomes a perfect square (counting the increments, thus finally obtaining x_N), and then incrementing it (again counting the increments, thus finally obtaining y_N) until it again becomes a perfect square. From these two numbers x_N and $y_N$we can regain N by the formula computed in the following:

Given input N, the next perfect squares are $k_N^2 = N + x_N$ $(x_N \geq 0)$ and $(k_N + 1)^2 = N + x_N + y_N$, then $y_N = 2k_N + 1$, so $k_N = (y_N - 1)/2$, and $N = k_N^2 - x_N = (y_N - 1)^2/4 - x_N$. Of course, the function $f(x_N, y_N) = (y_N - 1)^2/4 - x_N$ decoding N from x_N and y_N can be implemented by a register machine and simulated by a P system as described in Theorem 2.

In the following example we specify more formally the precomputing block mentioned above.

Example 1. Encoding the input number N.

Let the input N be given as a multiplicity of symbol c_i, and we want to obtain values x_N and y_N described above in auxiliary registers j and l, respectively, but represented already in the way we need their contents x_N and y_N implemented

with the corresponding number of symbols a_j and b_j as well as a_l and b_l. We also use an additional starting object s_i and in sum the following rules:

$$
\begin{aligned}
&s_i \to c_i a_j \tilde{s}_i | P_i',\ \tilde{s}_i \to s' | P_j,\ \tilde{s}_i \to \hat{s}_i | Z_j,\ \hat{s}_i \to a_j b_j s',\ b_j \to a_j a_j b_j |_{\hat{s}_i},\\
&s_i \to c_i t_i | Z_i',\\
&t_i \to c_i a_l \tilde{t}_i | P_i',\ \tilde{t}_i \to t_i' | P_l,\ \tilde{t}_i \to \hat{t}_i | Z_l,\ \hat{t}_i \to a_l b_k t',\ b_k \to a_l a_l b_l |_{\hat{t}_i},\\
&t_i \to q_0^{(i)} | Z_i',\ \text{where}\\
&Z_i' = \{w \in O^\circ \mid |w|_{c_i} = k^2,\ k \geq 0\} = \mathtt{Sq}(c_i),\ P_i' = O^\circ \setminus Z_i'.
\end{aligned}
$$

Essentially, the rules above are exactly like increment instructions from Theorem 2, tracking how many times the multiplicity of the input object c_i has to be incremented to reach a perfect square and the next perfect square.

In the next phase of the encoding procedure, the P system should simulate a register machine which starts in state $q_0^{(i)}$ and computes the function $f(x_N, y_N) = (y_N - 1)^2/4 - x_N$, given x_N in register j and y_N in register l, producing the result (i.e., the value N of the input register i to be represented) in register i, represented by symbols a_i and b_i and thus in a suitable way to be the input for the P system constructed in Theorem 2.

Theorem 3. $Ps_aDOP_1(ncoo, ctxt^1(REC)) = Ps_aDOP_1(ncoo, pro_{1,1}, ctxt(\mathtt{Sq})) = PsRE.$

Proof. Clearly, any input vector can be processed accordingly in the way described in Example 1, and then a simulation of the register machine on these inputs as outlined in Theorem 2 completes the explanation of the following result. □

The construction in Theorem 3 may be adjusted so that we never rely on multiplicities of symbols a_i being zero, i.e., when starting with a value 0 in a register, we start with encoding it by 1. Moreover, testing for the appearance of a symbol which never appears more than once (which we needed for the symbols corresponding to the states of the simulated register machine) corresponds with testing for a perfect square of positive integers. Hence, for each checking set from $\mathtt{Sq}'$ (or its complement) or each singleton promoter used in the previous construction we can use a set from $\mathtt{Sq}'$ (or its complement) only. In sum we get:

Corollary 1. $Ps_aDOP_1(ncoo, ctxt(\mathtt{Sq}')) = PsRE.$

5 Conclusions

It was known that generalized context conditions are equivalent to predicates on boundings, and that using them in deterministic maximally parallel non-cooperative P systems still leaves their accepting power as low as $NFIN \cup coNFIN$. We have shown that regular context conditions yield computational

completeness of deterministic maximally parallel non-cooperative P systems, expecting that the power of P systems with regular context conditions over one symbol is still quite limited. However, we have shown computational completeness using a simple stronger one-symbol context condition, namely, $\{w \in O^\circ \mid |w|_{a_i} = k^2,\ k \geq 0\}$.

References

1. A. Alhazov, R. Freund: Asynchronous and Maximally Parallel Deterministic Controlled Non-Cooperative P Systems Characterize $NFIN$ and $coNFIN$. In: E. Csuhaj-Varjú, M. Gheorghe, G. Rozenberg, A. Salomaa, Gy. Vaszil: *Membrane Computing – 13th International Conference*, CMC13, Budapest, Revised Selected Papers, Lecture Notes in Computer Science **7762**, Springer, 2013, 101–111.
2. A. Alhazov, R. Freund: Priorities, Promoters and Inhibitors in Deterministic Non-Cooperative P Systems. In: M. Gheorghe, G. Rozenberg, A. Salomaa, P. Sosík, C. Zandron: *Membrane Computing - 15th International Conference*, CMC 2014, Prague, Revised Selected Papers, Lecture Notes in Computer Science **8961**, Springer, 2014, 86–98.
3. A. Alhazov, R. Freund, S. Verlan: Promoters and Inhibitors in Purely Catalytic P Systems. In: M. Gheorghe, G. Rozenberg, A. Salomaa, P. Sosík, C. Zandron: *Membrane Computing - 15th International Conference*, CMC 2014, Prague, Revised Selected Papers, Lecture Notes in Computer Science **8961**, Springer, 2014, 126–138.
4. A. Alhazov, D. Sburlan: Ultimately Confluent Rewriting Systems. Parallel Multiset-Rewriting with Permitting or Forbidding Contexts. In: G. Mauri, Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg, A. Salomaa: *Membrane Computing, 5th International Workshop*, WMC 2004, Milan, Revised Selected and Invited Papers, Lecture Notes in Computer Science **3365**, Springer, 2005, 178–189.
5. M. Gheorghe, Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg: Research Frontiers of Membrane Computing: Open Problems and Research Topics. *International Journal of Foundations of Computer Science* **24** (5), 2013, 547–624.
6. M. Ionescu, D. Sburlan: On P Systems with Promoters/Inhibitors. *Journal of Universal Computer Science* **10** (5), 2004, 581–599.
7. D. Sburlan: Further Results on P Systems with Promoters/Inhibitors. *International Journal of Foundations of Computer Science* **17** (1), 2006, 205–221.

Polarizationless P Systems with One Active Membrane

Artiom Alhazov[1], Rudolf Freund[2]

[1] Institute of Mathematics and Computer Science, Academy of Sciences of Moldova
Academiei 5, Chişinău MD-2028 Moldova
E-mail: artiom@math.md
[2] Faculty of Informatics, Vienna University of Technology
Favoritenstr. 9, 1040 Vienna, Austria
E-mail: rudi@emcc.at

Summary. The aim of this paper is to study the computational power of P systems with one active membrane without polarizations. For P systems with active membranes, it is known that computational completeness can be obtained with either of the following combinations of features: 1)two polarizations, 2)membrane creation and dissolution, 3)four membranes with three labels, membrane division and dissolution, 4)seven membranes with two labels, membrane division and dissolution.

Clearly, with one membrane only object evolution rules and send-out rules are permitted. Two variants are considered: external output and internal output.

1 Introduction

Membrane computing is a theoretical framework of parallel distributed multiset processing. It has been introduced by Gheorghe Păun in 1998, and has been an active research area since then, see [10] for the comprehensive bibliography and [6],[8] for a systematic survey. Membrane systems are also called P systems.

It has been shown in [4] (some results being improvements of the results from [1] and [3]) that the following P systems with active membranes are computationally complete: 1) with one membrane and two polarizations, as acceptors, 2) polarizationless ones with membrane creation and dissolution, 3) polarizationless ones starting with four membranes and three labels, 4) polarizationless ones starting with seven membranes and two labels.

The object of study of this paper is the family of P systems with one active membrane without polarizations. Similar questions for non-cooperative transitional P systems without any additional features have been addressed in [2].

2 Definitions

2.1 Formal Language Preliminaries

Consider a finite set V. The set of all words over V is denoted by V^*, the concatenation operation is denoted by $\bullet$ (which is written only when necessary) and the empty word is denoted by λ. Any set $L \subseteq V^*$ is called a language. For a word $w \in V^*$ and a symbol $a \in V$, the number of occurrences of a in w is written as $|w|_a$. The permutations of a word $w \in V^*$ are $\texttt{Perm}(w) = \{x \in V^* \mid |x|_a = |w|_a \text{ for all a } \in \text{V}\}$. We denote the set of all permutations of the words in L by $\texttt{Perm}(L)$, and we extend this notation to families of languages. We use FIN, REG, LIN, CF, MAT, CS, RE to denote finite, regular, linear, context-free, matrix without appearance checking and with erasing rules, context-sensitive, and recursively enumerable families of languages, respectively. The family of languages generated by extended (tabled) interactionless L systems is denoted by $E(T)0L$. The family of sets of numbers generated by forbidden random context multiset grammars is denoted by $NfRC$. For more formal language preliminaries, we refer the reader to [9].

Throughout this paper we use string notation to denote the multisets. When speaking about membrane systems, keep in mind that the order in which symbols are written is irrelevant, unless we speak about the symbols sent to the environment. In particular, speaking about the contents of some membrane, when we write $a_1^{n_1} \cdots a_m^{n_m}$ (or any permutation of it), we mean a multiset consisting of n_i instances of symbol a_i, $1 \leq i \leq m$.

2.2 P systems with One (Active) Membrane

We present the definition of a *P system with active membranes*, simplified for studying the generative power in case of one membrane.

$\Pi = (O, \mu = [\ ]_1, w_1, R_1, i_0)$, where

O is a finite set of objects,

w_1 is the initial multiset in region 1,

R_1 is the set of rules associated to membrane 1,

i_0 is the output region; when languages are considered, $i_0 = 0$ is assumed.

The rules of a membrane system have the forms $(a_0)\ [\ a \to u\ ]_1$ (evolution of an object), and $(c_0)\ [\ a\ ]_1 \to [\ ]_1 b$ (sending an object out, possibly renaming it), where $a, b \in O$ and $u \in O^*$.

The rules are applied in maximally parallel way: no further rule should be applicable to the idle objects, except rules of type (c_0) may be applied to at most one object at any step.

A *catalytic P system* (with one membrane) is a construct

$\Pi = (O, C, \mu = [\ ]_1, w_1, R_1, i_0)$, where

O is a finite set of objects,

C is a special subset of Owhose elements are called catalysts,

w_1 is the initial multiset in region 1,

R_1 is the set of rules associated to membrane 1,

i_0 is the output region; when languages are considered, $i_0 = 0$ is assumed.

The rules in R are either non-cooperative rules of the form $a \to (b_1, tar_1) \cdots (b_k, tar_k)$ with a and the b_i, $1 \leq i \leq k$, being from $O \setminus C$ and the $tar_i \in \{here, out\}$ being the targets for the corresponding symbols b_i, or catalytic rules of the form $ca \to c(b_1, tar_1) \cdots (b_k, tar_k)$ with $c \in C$.

A configuration of a P system is a construct which contains the information about the contents of the skin membrane as well as the sequence of objects sent out. A sequence of transitions between the configurations is called a computation. The computation halts when such a configuration is reached that no rules are applicable. In case of external output ($i_0 = 0$), as the result of a (halting) computation we may consider the *sequence* of objects sent to the environment; we denote it by $L(\Pi)$. Both in case of internal output ($i_0 = 1$) and in case of external output, we may consider as the result the vector of multiplicities of objects in region i_0, we denote it by $Ps(\Pi)$, or the total number of objects in region i_0, which we denote by $N(\Pi)$.

The family of P systems with one polarizationless active membrane may be denoted by $OP_1(a_0, c_0)$. The class of sets of numbers/vectors/words generated by a family $\mathbf{F}$ of P system is denoted by $N\mathbf{F}$, $Ps\mathbf{F}$ and $L\mathbf{F}$, respectively. We use a superscript *int* or *ext* when speaking about internal and external output, respectively, and we may omit subscript *ext* in the case of generating languages, i.e., external output is assumed for $L\mathbf{F}$.

Moreover, we may use a subscript T to denote terminal filtering of the result; in this case, a subset $T \subset O$ is additionally specified for Π, and the objects not belonging to T are not considered in the result. For example, the family of sets of vectors of non-negative integers generated internally by P systems with one polarizationless active membrane with terminal filtering are denoted by $Ps_T^{int}OP_1(a_0, c_0)$.

Example 1. To illustrate generation, consider the following P system:

$$\begin{aligned}\Pi &= (O = \{S, a, b, c, d, f\}, \mu = [\ ]_1, w_1 = a, R_1, i_0),\\ R_1 &= \{[\ S \to Sabcd\ ]_1,\ [\ S \to f\ ]_1,\\ &\quad [\ a\ ]_1 \to [\ ]_1 a,\ [\ b\ ]_1 \to [\ ]_1 b,\ [\ c\ ]_1 \to [\ ]_1 c\}.\end{aligned}$$

Object S produces objects a, b, c, d in arbitrary but equal amounts. Objects a, b, c are sent out in arbitrary order. Hence, if $i_0 = 1$ then $N(\Pi) = \mathbb{N}_1$ (i.e., the set of all positive integers), and if $i_0 = 0$ then $L(\Pi) = \bigcup_{n \geq 0} \mathtt{Perm}(a^n b^n c^n) = \{w \in \{a, b, c\}^* \mid |w|_a = |w|_b = |w|_c\}$.

P systems can be also viewed as acceptors. In that case, an input subalphabet Σ is additionally specified in the tuple defining P system before μ, and $i_0 = 1$ is the input region. An input multiset over Σ is additionally placed inside the membrane before the computation starts, and it is accepted if and only if the computation halts. The result $Ps_{acc}(\Pi)$ is the set of all accepted inputs, and the family of vector sets accepted by P systems with one active membrane is $Ps_{acc}OP_1(a_0, c_0)$.

3 Comparison with a Transitional Model: Catalytic P Systems with One Catalyst

The model of P systems with active membranes, for the case of one membrane, can be compared to the following case of transitional P systems: non-distributed P systems

with one catalyst. Indeed, for each P system with one active membrane, there exists a 1-catalytic non-distributed P system with the same behavior, as non-cooperative rules work equivalently in both models: $[\ A \to u\ ]_h$ is equivalent to $A \to u$, and sending out corresponds to particular rules with one catalyst, i.e., $[\ A\ ]_h \to [\ ]_h a$ corresponds with $cA \to c(a, out)$, or, if without restricting generality we assume the set of symbols that may appear inside the system to be disjoint from the set of symbols that may be sent to the environment, simply with $cA \to c(a, here)$.

Notice that for P system with external output, we may ignore the objects remaining inside the system when it halts (as explained in the next section), while for P systems with internal output, we should ignore the objects sent out. In this way, for the case of internal output, sending out corresponds to a catalytic erasing, while for the case of external output sending out corresponds to a catalytic renaming of a non-terminal symbol into a terminal symbol.

Hence, we can immediately conclude that

$$X^{\alpha}_{\beta}OP_1(a_0, c_0) \subseteq X_{\beta}OP_1(ncoo, cat_1) \text{ for } X \in \{N, Ps, L\},\ \alpha \in \{int, ext\},\ \beta \in \{-, T\},$$

where $\beta = -$ stands for not specifying a subscript.

One-catalytic P systems were investigated in [5], where some subclasses of P systems with one catalyst are defined and certain results on their generative power are presented. In particular, it was shown in [5] that $N_{-c}OP_1(wsepcat_1) = NREG$ and $N_{-c}OP_1(complcat_1) \subseteq NfRC$. Clearly, the corresponding restrictions might also be considered for polarizationless P systems with one active membrane, and such results can be claimed as upper bounds for the corresponding restrictions, e.g.,

$$NOP_1(wsep(a_0, c_0)) = NREG,$$

where the restriction of the *weak separation* can be reformulated for the model with active membranes as follows: the set O of objects is divided into three **disjoint** subsets O', O'' and O''', such that

- objects $a \in O'$ have no associated rules (they cannot evolve or be sent out, so if they are produced, they remain idle inside the system),
- objects $a \in O''$ have associated send-out rules, but no evolution rules,
- objects $a \in O'''$ have associated evolution rules, but no send-out rules.

It is worth mentioning that the additional requirement from [5] that the objects produced by a catalytic rule cannot undergo a non-cooperative rule is *automatically* satisfied after translation into the active membrane case, so the only restriction remaining in the case of weak separation is that a rule of type (a_0) and a rule of type (c_0) are not allowed to compete for the same object. This restriction means, for instance, that all objects that have associated send-out rules cannot evolve inside the system, they simply wait there until they are chosen to be sent out.

A different restriction considered in [5] is *complete* P systems (mentioned above as $complcat_1$). It can be reformulated in the model of polarizationless P systems with active membranes as follows: there is no object having associated rules of type (c_0) and no rules of type (a_0). This restriction means that no object is allowed to be temporarily idle; if it is not sent out, then it either evolves immediately, or remains idle throughout the computation. It follows that

$$NREG \subseteq NOP_1(compl(a_0, c_0)) \subseteq NfRC.$$

It is interesting to note that weak separation and completeness are, in some sense, two opposite requirements. While the latter one requires that *all* objects which can be sent out must evolve if they are not chosen to be sent out, the first special case requires that *no* objects which can be sent out are allowed to evolve. Of course, in the most general case there can be both kinds of objects which can be sent out.

4 External output

The first goal of this section is to present a reduction of any P system with one active membrane without polarizations and external output to an equivalent normal form. Then we will use this normal form to prove an upper bound result. We require the normal form mentioned above to satisfy the following conditions:

- Every object appears on the left side of some rule.
- The only erasing rule allowed is for the initial object; if so, the initial object does not appear on the right side of any rule. (If we have an initial multiset w, then we add the rule $S \to w$ where S is a new symbol now being the initial object.)

We approach this goal in a few stages. First, we remark that, without restricting generality, we may assume that no objects may remain inside the system when it halts. Indeed, let O_λ be the set of all objects that do not have associated rules. By adding rules $R_\lambda = \{[\ a \to \lambda\ ]_1 \mid a \in O_\lambda\}$, we make sure that there are no objects that do not have associated rules. On the other side, adding rules R_λ does not affect the result of a P system with external output, since preserving/erasing objects from O_λ has no alternatives, and it does not affect the environment.

Second, we remark that, without restricting generality, we may assume that the initial multiset consists of only one object, say S, which does not appear in the right side of any rule. Indeed, for a P system starting with a multiset (represented by) w, consider an equivalent P system starting with a multiset consisting of a new object S, and adding $R_S = \{[\ S \to w\ ]_1\}$ to R_1.

Third, we claim that for any P system satisfying the assumptions mentioned above, there exists a P system without erasing rules (except, possibly, for S).

Proof. Indeed, let us first add rules $R_t = \{[\ a \to \#\ ]_1 \mid ([\ a \to \lambda\ ]_1) \in R_1 \text{ or } a = \#\}$, where $\#$ is a new symbol, shared for all such reductions, so if it appears in a configuration, the system will never halt, and will therefore not produce any result. This transformation will certainly not affect the result of the system, since every new computation branch will not be productive, while the existing branches will not be affected (since by construction, one can *always* apply some other rule to a instead of trapping).

Second, compute the set O_λ of erasable objects as follows:

- Set O_λ to $\{a \in O \mid [\ a \to \lambda\ ]_1 \in R_1$,
- If $[\ a \to u\ ]_1$ is in R_1 and $u \in O_\lambda^*$, then add a to O_λ,
- Iterate the previous procedure until no more elements can be added to O_λ.

Third, replace each rule $[\ a \to u\ ]_1$ by rules $[\ a \to u'\ ]_1$, where the u' are obtained from u by removing (in all possible combinations) some objects from O_λ. This will again yield an equivalent system, because every symbol that could eventually be deleted does not have to be produced in the first place.

Fourth, remove all erasing rules. We claim that the resulting P system is still equivalent to the original P system. Indeed, any object (other than S) that should be erased, could be "pre-erased" by not producing it in the first place. However, any object that should evolve can evolve by other rules, and any object that should be sent out can be sent out (unless some competing object is sent out, in which case the simulation would not be correct, so the computation is discarded by producing symbol #). □

Corollary 1. $LOP_1(a_0, c_0) \subseteq CS$.

Proof. Indeed, the total number of objects (inside and outside the membrane) never decreases throughout the computation (except, possibly, for the empty word, generated in one step), and the length of the result matches the total number of objects when the system halts. □

We now proceed with the lower bound result.

Theorem 1. $LOP_1(a_0, c_0) \supseteq REG \bullet \texttt{Perm}(REG)$.

Proof. Consider an alphabet T and two arbitrary regular languages over T. Then there exist reduced regular grammars $G_1 = (N_1, T, P_1, S_1)$ and $G_2 = (N_2, T, P_2, S_2)$ generating them, such as $N_1 \cap N_2 = \emptyset$. We construct the following P system:

$$
\begin{aligned}
\Pi &= (O = N_1 \cup N_2 \cup T \cup T', \mu = [\ ]_1, w_1 = S_1, R_1),\\
T' &= \{a' \mid a \in T\},\\
R_1 &= \{[\ A \to aB\ ]_1 \mid (A \to aB) \in P_1\} \cup \{[\ A \to S_2\ ]_1 \mid (A \to \lambda) \in P_1\}\\
&\quad \cup \{[\ A \to a'B\ ]_1 \mid (A \to aB) \in P_2\} \cup \{[\ A \to \lambda\ ]_1 \mid (A \to \lambda) \in P_2\}\\
&\quad \cup \{[\ a' \to a'\ ]_1 \mid a \in T\} \cup \{[\ a\ ]_1 \to [\ ]_1 a,\ [\ a'\ ]_1 \to [\ ]_1 a \mid a \in T\}.
\end{aligned}
$$

The P system constructed above generates $L(G_1) \bullet L(G_2)$, except the symbols generated by the second grammars are produced in a primed form, and may undergo trivial rewriting for an arbitrarily long time before they are sent out, which ensures that after generating a word from $L(G_1)$, any permutation of a word from $L(G_2)$ may be generated. □

We now present a few closure properties.

Lemma 1. *The family $LOP_1(a_0, c_0)$ is closed under renaming morphisms.*

Proof. The statement follows from applying the renaming morphism to the send-out rules. □

Theorem 2. *$LOP_1(a_0, c_0)$ is closed under union.*

Proof. The closure under union follows from adding a new axiom and productions of non-deterministic choice between multiple axioms. □

5 Internal output

In this case the environment is no longer relevant: it does not matter which symbol is written in the right side of a send-out rule. The object sent out no longer affects the result, so sending out is equivalent to a sequential version of erasing.

Of course, we can generate $PsREG$ with rules of type (a_0) corresponding to the rules of a reduced regular grammar. Hence,

$$Ps^{int}OP_1(a_0, c_0) \supseteq PsREG.$$

Is it an *open question* whether non-semilinear number sets can be generated, see also the partial results transferred from the one-catalytic model, recalled in Section 3.

6 P systems with input

In this section we show that, not very surprisingly, for P systems with one polarizationless active membrane, their accepting power is even smaller than their generative power. More exactly, unless such a P system accepts all allowed inputs, it only accepts specific finite sets. We start by establishing some useful facts (we remind that we use $\subseteq$ to denote the submultiset relation, $\cup$ to denote the union of multisets, and $\setminus$ to denote the difference of multisets).

Lemma 2. *Let $\Pi \in OP_1(a_0, c_0)$ be a P system with alphabet O, let $[\ u\ ]_1 \Rightarrow [\ v\ ]_1\alpha$ in Π ($\alpha \in O \cup \{\lambda\}$) Then for every multiset $u' \subseteq u$, either $[\ u'\ ]_1$ is already a halting configuration, or there exists a multiset $v' \subseteq v$ and $\beta \in O \cup \{\lambda\}$ such that $[\ u'\ ]_1 \Rightarrow [\ v'\ ]_1\beta$ in Π.*

Proof. In a transition $[\ u\ ]_1 \Rightarrow [\ v\ ]_1\alpha$, one of three possible cases happen for every (copy of) object a in u:

- a is rewritten by some rule of Π into a (possibly empty) multiset, contributing to v;
- a is sent out by some rule of Π as α;
- a remains idle, contributing to v.

Note that v consists exactly of the resulting objects from the first case and the objects of the third case. More precisely, let the union of multisets of the right side rules for all copies of rewritten objects be v_r, and let the multiset of idle objects be v_i; then, $v = v_r \cup v_i$. By definition of the model, the second case was applied to at most one (copy of) an object in u. Also by definition of the model, for each object in the third case, there exist no rules to evolve it, except, possibly, send-out rules, in which case $\alpha \neq \lambda$.

We recall that u' may be obtained from u by erasing some (copies) of objects. Fix some correspondence of (copies of) objects in u' to objects in u, and consider a transition from u' by the same behavior of objects in u' as of objects in u:

- rewritten objects will yield some submultiset v'_r of v_r;
- β' will be produced in the environment, $\beta' = \alpha$ or $\beta' = \lambda$;
- idle objects will yield some submultiset v'_i of v_i.

It is obvious that these rules are applicable, and that $v'_r \cup v'_i \subseteq v$. Maximality also holds, except in one special situation: when $\alpha \neq \lambda$, but it was produced from a (copy of) an object not in u', while there exists at least one object b that was idle in a transition $[\ u\ ]_1 \Rightarrow [\ v\ ]_1\alpha$.

In this situation, one object b, instead of being idle, should be sent out as β, and the resulting multiset in the skin is $v' = v'_r \cup v'_i \setminus b$ (if this situation does not happen, we take $\beta = \beta'$ and $v' = v'_r \cup v'_i$).

Therefore, $[\ u'\ ]_1 \Rightarrow [\ v'\ ]_1\beta$ in Π if at least one (copy) of object from u' fell into the first or the second case, and otherwise $[\ u'\ ]_1$ is already a halting configuration. □

Lemma 3. *If $n \in N(\Pi)$, then also $n' \in N(\Pi)$ for any non-negative integer $n' \leq n$.*

Proof. Let the alphabet of Π be O, let the initial contents of the skin membrane of Π be w_1, and let the input subalphabet of Π be Σ. By definition of acceptance, a number n is accepted if there exists a halting computation in Π starting from configuration $[\ u\ ]_1$, for some $u \in w_1\Sigma^n$.

Consider the "sub-input" of only n' objects, i.e., $u' \in w_1\Sigma^n$ such that $u' \subseteq u$. If $[\ u\ ]_1$ is already halting, then so is $[\ u'\ ]_1$, so the statement of the lemma holds; now we assume the contrary: $[\ u\ ]_1 \Rightarrow [\ v\ ]_1\alpha$. By the previous lemma, in one step, either the computation with u' in the skin will immediately halt (and the statement of the lemma again holds), or there is a one-step transition $[\ u'\ ]_1 \Rightarrow [\ v'\ ]_1\beta$ with $v' \subseteq v$.

Iterating the application of the previous lemma, by induction, we conclude that there exists a computation starting from $[\ u'\ ]_1$ that will halt in at most as many step as the halting computation starting from $[\ u\ ]_1$ that we considered. Hence $n' \in N(\Pi)$. □

It follows that the accepted set of numbers is either $\mathbb{N}$, or empty, or it contains all integers less than or equal to the maximal accepted number, so accepting P systems with one polarizationless active membrane cannot be computationally complete, and P systems with one polarizationless active membrane are obviously weaker as acceptors than as generators:

$$N_{acc}OP_1(a_0, c_0) \subseteq \{\emptyset, \mathbb{N}\} \cup \{\{k \mid 0 \leq k \leq n\} \mid n \in \mathbb{N}\}.$$

In the rest of the section we show, by all necessary examples, that this inclusion is an equality:

$$\begin{aligned}
&\Pi_\emptyset = (O = \{a\}, \Sigma = \{a\}, \mu = [\ \ ]_1, w_1 = a, R_1 = \{[\ a \to a\ ]_1\}, i_0 = 1).\\
&\Pi_\mathbb{N} = (O = \{a\}, \Sigma = \{a\}, \mu = [\ \ ]_1, w_1 = \lambda, R_1 = \{[\ a \to \lambda\ ]_1\}, i_0 = 1).\\
&\Pi_n = (O = \{a_i \mid 0 \leq i \leq n\}, \Sigma = \{a_0\}, \mu = [\ \ ]_1, w_1 = \lambda, R_1, i_0 = 1), \text{ where}\\
&R_1 = \{[\ a_i \to a_{i+1}\ ]_1,\ [\ a_i\ ]_1 \to [\ \ ]_1 a_0 \mid 0 \leq i < n\} \cup \{[\ a_n \to a_n\ ]_1\}.
\end{aligned}$$

Clearly, $\Pi_\emptyset$ accepts nothing, since with any input it starts with at least one object, and carries out an infinite computation. On the other end of the spectrum, system $\Pi_\mathbb{N}$ accepts any input, by erasing it in one step and halting. Finally, we claim that system Π_n accepts exactly set $\{k \mid 0 \leq k \leq n\}$. Indeed, any object increments its index every step, unless the object is sent out, or the index reaches n (forcing an infinite computation). It is easy to see that at most n input objects may be sent out in this way; the system with input $(a_0)^k$ has a halting computation if and only if $k \leq n$.

Overall, we have established the following results:

$$\begin{aligned} REG \bullet \texttt{Perm}(REG) \subseteq LOP_1(a_0, c_0) &\subseteq CS, \\ Ps^{int}OP_1(a_0, c_0) &\subseteq PsREG, \\ N\alpha OP_1(wsep(a_0, c_0)) &= NREG,\ \alpha \in \{int, ext\}, \\ NREG \subseteq N\alpha OP_1(compl(a_0, c_0)) &\subseteq NfRC,\ \alpha \in \{int, ext\}, \\ N_{acc}OP_1(a_0, c_0) &= \{\{k \mid 0 \leq k \leq n\} \mid n \in \mathbb{N}\} \cup \{\emptyset, \mathbb{N}\}. \end{aligned}$$

7 Conclusions

In this paper we have considered the family of languages generated by polarizationless P systems with one active membrane. A normal form was given for external output case. It was than shown that the family of generated languages lies between $REG \bullet \texttt{Perm}(REG)$ and CS, and is closed under union and renaming morphisms. The exact characterization is an *open question*, but polarizationless P systems with one active membrane can be simulated by (and are, therefore, *at most* as powerful as) P systems with one catalyst, transferring two results on the generative power of two restricted classes, independently from the output region.

Then we also considered sets of vectors or numbers generated internally, as well as sets of vectors or numbers accepted by polarizationless P systems with one active membrane. Several questions about the families of these sets are still *open*, too.

Another possible generalization that can be considered is to also allow rules of type (b_0) to bring objects from the environment back to the skin. Note that such systems would still correspond to a subclass of 1-catalytic P systems, but some definitions would have to be revised, as well as all related results.

We have proved that accepting P systems with one polarizationless active membrane are not computationally complete, unlike those with two polarizations or like those with membrane creation and dissolution, or with multiple membranes and membrane dissolution.

The questions about the computational power of polarizationless P systems with active membranes with 2 and 3 membranes in the initial configuration are still *open*, as well as of polarizationless systems with less than 7 membranes and two labels, or of all polarizationless systems with only one label.

References

1. A. Alhazov: P Systems without Multiplicities of Symbol-Objects. *Information Processing Letters* **100**, 3, 2006, 124–129.
2. A. Alhazov, C. Ciubotaru, Yu. Rogozhin, S. Ivanov: The Family of Languages Generated by Non-Cooperative Membrane Systems. In: Gh. Păun, M.J. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, A. Salomaa: *Membrane Computing, 11th International Conference*, CMC11, Jena, Revised Selected Papers, Lecture Notes in Computer Science **6501**, 2011, 65–79.

3. A. Alhazov, R. Freund, Gh. Păun: Computational Completeness of P Systems with Active Membranes and Two Polarizations. In: M. Margenstern: *Machines, Computations, and Universality*, 4th International Conference, MCU 2004, Saint Petersburg, Revised Selected Papers, Lecture Notes in Computer Science **3354**, Springer, 2005, 82–92.
4. A. Alhazov, R. Freund, A. Riscos-Núñez: Membrane Division, Restricted Membrane Creation and Object Complexity in P Systems. *International Journal of Computer Mathematics* **83**, 7, 2006, 529–548.
5. R. Freund:
Special Variants of P Systems with One Catalyst in One Membrane. In: H. Leung, G. Pighizzini: 8th International Workshop on *Descriptional Complexity of Formal Systems* - DCFS 2006, Las Cruces, New Mexico, 2006. Proceedings, 2006, 250–258.
6. Gh. Păun: Membrane Computing. An Introduction, Springer, 2002.
7. Gh. Păun, G. Rozenberg, A. Salomaa: Membrane Computing with an External Output. *Fundamenta Informaticae* **41**, 3, 2000, 313–340.
8. Gh. Păun, G., Rozenberg, A. Salomaa, A. (eds.): *The Oxford Handbook of Membrane Computing*, Oxford University Press, 2010.
9. G. Rozenberg, A. Salomaa (eds.): *Handbook of Formal Languages*, 1-3 vol., Springer, 1997.
10. P systems webpage. `http://ppage.psystems.eu/`

Variants of P Systems with Toxic Objects

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

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

[2] Faculty of Informatics, Vienna University of Technology
Favoritenstr. 9, 1040 Vienna, Austria
Email: rudi@emcc.at

[3] Université Paris Est, France
Email: sergiu.ivanov@u-pec.fr

Summary. Toxic objects have been introduced to avoid trap rules, especially in (purely) catalytic P systems. No toxic object is allowed to stay idle during a valid derivation in a P system with toxic objects. In this paper we consider special variants of toxic P systems where the set of toxic objects is predefined – either by requiring all objects to be toxic or all catalysts to be toxic or all objects except the catalysts to be toxic. With all objects staying inside and being toxic, purely catalytic P systems cannot go beyond the finite sets, neither as generating nor as accepting systems. With allowing the output to be sent to the environment, exactly the regular sets can be generated. With non-cooperative systems with all objects being toxic we can generate exactly the Parikh sets of languages generated by extended Lindenmayer systems. Catalytic P systems with all catalysts being toxic can generate at least $PsMAT$.

1 Definitions

We assume the reader to be familiar with the underlying notions and concepts From formal language theory, e.g., see [16], as well as from the area of P systems, e.g., see [13, 14, 15]; we also refer the reader to [18] for actual news.

1.1 Prerequisites

The set of integers is denoted by $\mathbb{Z}$, and the set of non-negative integers by $\mathbb{N}$. Given an alphabet V, a finite non-empty set of abstract symbols, the free monoid generated by V under the operation of concatenation is denoted by V^*. The elements of V^* are called strings, the empty string is denoted by λ, and $V^* \setminus \{\lambda\}$ is denoted by V^+. For any string $w \in V$, by $alph(w)$ we denote the set of symbols

occurring in w; moreover, the set of all strings which are obtained by permuting the symbols of w is denoted by $Perm(w)$; for a set of stings L, we define $Perm(L) = \{Perm(w) \mid w \in L\}$.

For an arbitrary alphabet $V = \{a_1, \cdots, a_n\}$, 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 $|x| = \sum_{a_i \in V} |x|_{a_i}$. A (finite) multiset over a (finite) alphabet $V = \{a_1, \cdots, a_n\}$ is a mapping $f : V \to \mathbb{N}$ and can be represented by $\left\langle a_1^{f(a_1)}, \cdots, a_n^{f(a_n)} \right\rangle$ or by any string x for which $(|x|_{a_1}, \cdots, |x|_{a_n}) = (f(a_1), \cdots, f(a_n))$. We will denote the vector $(f(a_1), \cdots, f(a_n))$ by $\Psi(f)_V$. The families of regular and recursively enumerable string languages are denoted by REG and RE, respectively.

1.2 Finite Automata

The regular languages in REG are exactly the languages accepted by finite automata. A *finite automaton* is a quintuple $M = (Q, T, \delta, q_0, F)$, where Q is the set of *states*, T is the *input alphabet*, $\delta \subseteq (Q \times T \times Q)$ is the transition function, $q_0 \in Q$ is the *initial state*, and $F \subseteq Q$ is the set of final states. The language over T accepted by M is denoted by $L(M)$. A finite automaton is called *deterministic*, if for every pair (q, a) with $q \in Q$ and $a \in P$ there exists exactly one state $p \in Q$ such that $(q, a, p) \in \delta$.

A *finite automaton with output*, also called *generalized sequential machine* or *gsm* for short, is a construct $M = (Q, T, \Sigma, \delta, q_0, F)$, where Q is the set of *states*, T is the *input alphabet*, Σ is the *output alphabet*, $\delta \subseteq (Q \times T \times Q \times \Sigma^*)$ is the finite transition function, $q_0 \in Q$ is the *initial state*, and $F \subseteq Q$ is the set of final states. M called *deterministic*, if for every pair (q, a) with $q \in Q$ and $a \in P$ there exists exactly one pair $(q, w) \in Q \times \Sigma^*$ such that $(q, a, p, w) \in \delta$. A (deterministic) gsm defines a relation (function) $T^* \to \Sigma^*$, called *(deterministic) gsm mapping*. The sets of all relations (functions) defined by (deterministic) gsm mappings are denoted by $RelREG$ and $FunREG$, respectively.

We also consider a special variant of finite automata which resembles the idea of input-driven push-down automata (for an overview, see [11]), also called visibly push-down automata (for example, see [3]). Hence, we call this variant where the next state only depends on the input symbol *input-driven finite automata*, i.e., for any two triples $(q, a, p), (q', a, p') \in \delta$ with $q, p, q', p' \in Q$ and $a \in T$ we have $p = p'$. In the following, the subclass of regular languages accepted by input-driven finite automata will be denoted by $IDREG$.

A gsm is called input-driven if for any two tuples $(q, a, p, w), (q', a, p', w') \in \delta$ with $q, p, q', p' \in Q$ and $a \in T$ we have $p = p'$ as in the case of finite automata; such a gsm is called deterministic if we even have $(p, w) = (p', w')$. The subclasses of (deterministic) gsm mappings defined by input-driven finite automata with output are denoted by $RelIDREG$ and $FunIDREG$, respectively.

1.3 *ET0L* Systems

An $ET0L$ system is a construct $G = (V, T, P_1, \cdots, P_m, w)$ where $m \geq 1$, V is an alphabet, $T \subseteq V$ is the terminal alphabet, the P_i, $1 \leq i \leq m$, are finite sets (*tables*) of non-cooperative rules over V, and $w \in V^*$ is the *axiom*. In a derivation step in G, all the symbols present in the current sentential form are rewritten using one table. The language generated by G, denoted by $L(G)$, consists of all terminal strings $w \in T^*$ which can be generated by a derivation in G starting from the axiom w. The family of languages generated by $ET0L$ systems and by $ET0L$ systems with at most k tables is denoted by $ET0L$ and ET_k0L, respectively. If only one table is used, we omit the T.

1.4 Register Machines

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

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

A *configuration* of a register machine is described by the contents (i.e., by the number stored in the register) of each register and by the current label, which indicates the next instruction to be executed. Computations start by executing the instruction l_0 of P, and terminate with reaching the HALT-instruction l_h.

In order to deal with strings, this basic model of register machines can be extended by instructions for reading from an input tape and writing to an output tape containing strings over an input alphabet T_{in} and an output alphabet T_{out}, respectively:

- $l_1 : (read(a), l_2)$, with $l_1 \in B \setminus \{l_h\}$, $l_2 \in B$, $a \in T_{in}$.
  Reads the symbol a from the input tape and jumps to instruction l_2.
- $l_1 : (write(a), l_2)$, with $l_1 \in B \setminus \{l_h\}$, $l_2 \in B$, $a \in T_{out}$.
  Writes the symbol a on the output tape and jumps to instruction l_2.

Such a register machine working on strings we call a *register machine with input and output tape*, and we write $M = (m, B, l_0, l_h, P, T_{in}, T_{out})$. If no output is written, we omit T_{out}.

As is well known (e.g., see [10]), for any recursively enumerable set of natural numbers there exists a register machine with (at most) three registers accepting the numbers in this set. Register machines with an input tape, can simulate the computations of Turing machines with two registers and thus characterize RE. All these results are obtained with deterministic register machines, where the ADD-instructions are of the form $l_1 : (ADD\,(r)\,, l_2)$, with $l_1 \in B \backslash \{l_h\}$, $l_2 \in B$, $1 \leq j \leq m$.

Partially blind register machines with d registers use instructions $q_i : (ADD(r), q_j, q_k)$ and $q_i : (SUB(r), q_j)$. Moreover, the result is produced in the first m registers, while in a successful computation registers $m+1, \cdots, d$ are required to be empty in the end (and we assume that the output registers are never decremented).

2 P systems

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

Yet there are a lot of other variants of rules; for example, if on the right-hand side of a rule we add the symbol δ, the surrounding membrane is dissolved whenever at least one such rule is applied, at the same moment all objects inside this membrane (the objects of this membrane region together with the whole

inner membrane structure) are released to the surrounding membrane, and the rules assigned to the dissolved membrane region get lost.

Another option is to add *promoters* $p_1, \cdots, p_m \in O^+$ and *inhibitors* $q_1, \cdots, q_n \in O^+$ to a rule and write $u \to v|_{p_1,\cdots,p_m,\neg q_1,\cdots,\neg q_n}$, which rule then is only applicable if the current contents of the membrane region includes any of the promoter multisets, but none of the inhibitor multisets; in most cases promoters and inhibitors are rather taken to be singleton objects than multisets.

For all these variants of P systems defined above, the variants of toxic objects defined later in this paper can be defined, too. As this paper is just a starting point of such investigations, in the following we shall restrict ourselves to P systems containing only non-cooperative rules and/or catalytic rules (see definitions given below).

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

$$\Pi = (O, \mu, w_1, \cdots, w_m, R_1, \cdots, R_m, f_I, f_O)$$

where O is the alphabet of *objects*, μ is the *membrane structure* (with m membranes), $w_1, \cdots, w_m$ are multisets of objects present in the m regions of μ at the beginning of a computation, $R_1, \cdots, R_m$ are finite sets of *evolution rules*, associated with the membrane regions of μ, and f_O/f_I is the label of the membrane region where the outputs are put in/from where the inputs are taken. ($f_O/f_I = 0$ indicates that the output/input is taken sent to/taken from the environment).

If a rule $u \to v$ has at least two objects in u, then it is called *cooperative*, otherwise it is called *non-cooperative*. In *catalytic P systems* we use non-cooperative as well as *catalytic rules* which are of the form $ca \to cv$, where c is a special object which never evolves and never passes through a membrane (both these restrictions can be relaxed), but it just assists object a to evolve to the multiset v. In a *purely catalytic P system* we only allow catalytic rules. For a catalytic as well as for a purely catalytic P system Π, in the description of Π we replace "O" by "O, C" in order to specify those objects from O which are the catalysts in the set C. As already explained above, cooperative and non-cooperative as well as catalytic rules can be extended by adding promoters and/or inhibitors, thus yielding rules of the form $u \to v|_{p_1,\cdots,p_m,\neg q_1,\cdots,\neg q_n}$.

All the rules defined so far can be used in different derivation modes: in the *sequential* mode (*sequ*), we apply exactly one rule in every derivation step; in the *asynchronous* mode (*asyn*), an arbitrary number of rules is applied in parallel; in the *maximally parallel* (*maxpar*) derivation mode, in any computation step of Π we choose a multiset of rules from the sets $R_1, \cdots, R_m$ in a non-deterministic way such that no further rule can be added to it so that the obtained multiset would still be applicable to the existing objects in the membrane regions $1, \cdots, m$.

The membranes and the objects present in the compartments of a system at a given time form a *configuration*; starting from a given *initial configuration* and using the rules as explained above, we get *transitions* among configurations; a sequence of transitions forms a *computation* (we often also say *derivation*). A computation is *halting* if and only if it reaches a configuration where no rule can

be applied any more. With a halting computation we associate a *result generated* by this computation, in the form of the number of objects present in membrane f_O in the halting configuration. The set of multisets obtained as results of halting computations in Π working in the derivation mode $\delta \in \{sequ, asyn, maxpar\}$ is denoted by $mL_{gen,\delta}(\Pi)$, the set of natural numbers obtained by just counting the number of objects in the multisets of $mL_{gen,\delta}(\Pi)$ by $N_{gen,\delta}(\Pi)$, and the set of (Parikh) vectors obtained from the multisets in $mL_{gen,\delta}(\Pi)$ by $Ps_{gen,\delta}(\Pi)$. If we first project the results in $mL_{gen,\delta}(\Pi)$ to a terminal alphabet O_T, then we add the superscript T to N and Ps.

Yet we may also start with some additional input multiset w_{input} over an *input alphabet* Σ in membrane f_I, i.e., in total we there have $w_{f_I} w_{input}$ in the initial configuration, and *accept* this input w_{input} if and only if there exists a halting computation with this input; the set of multisets accepted by halting computations in

$$\Pi = (O, \Sigma, \mu, w_1, \cdots, w_m, R_1, \cdots, R_m, f_I)$$

working in the derivation mode δ is denoted by $mL_{acc,\delta}(\Pi)$, the corresponding sets of natural numbers and of (Parikh) vectors are denoted by $N_{acc,\delta}(\Pi)$ and $Ps_{acc,\delta}(\Pi)$, respectively.

For the input being taken from the environment, i.e., for $f_I = 0$, we need an additional target indication *come* as, for example, used in a special variant of communication P systems introduced by Petr Sosík (e.g., see [17]) where no objects are generated or deleted, but may only pass through membranes; $(a, come)$ on the right-hand side of a rule applied in the skin membrane means that the object a is taken into the skin membrane from the environment (all objects there are assumed to be available in an unbounded number). The multiset of all objects taken from the environment during a halting computation then is the multiset accepted by this accepting P system, which in this case we shall call a *P automaton*; the idea of *P automata* was first published in [4] and considered at the same time under the notion of *analysing P systems* in [8]. The set of non-negative integers and the set of (Parikh) vectors of non-negative integers accepted by halting computations in Π are denoted by $N_{aut}(\Pi)$ and $Ps_{aut}(\Pi)$, respectively.

The family of sets $Y_{\gamma,\delta}(\Pi)$, $Y \in \{N, Ps\}$, $\gamma \in \{gen, acc, aut\}$ computed by P systems with at most m membranes working in the derivation mode δ and with rules of type X is denoted by $Y_{\gamma,\delta}OP_m(X)$. If we first project the results in $mL_{gen,\delta}(\Pi)$ to a terminal alphabet O_T, then we add the superscript T to N and Ps.

A P system Π can also be considered as a system computing a partial recursive function (in the deterministic case) or even a partial recursive relation (in the non-deterministic case), with the input being given in a membrane region $f_I \neq 0$ as in the accepting case or being taken from the environment as in the automaton case. The corresponding functions/relations computed by halting computations in Π are denoted by $ZY_\alpha(\Pi)$, $Z \in \{Fun, Rel\}$, $Y \in \{N, Ps\}$, $\alpha \in \{acc, aut\}$.

For example, it is well known (for example, see [12]) that for any $m \geq 1$, for the types of non-cooperative ($ncoo$) and cooperative (coo) rules we have

$$NREG = N_{gen,maxpar}OP_m\,(ncoo) \subset N_{gen,maxpar}OP_m\,(coo) = NRE.$$

For $\gamma \in \{gen, acc, aut\}$ and $\delta \in \{sequ, asyn, maxpar\}$, the family of sets $Y_{\gamma,\delta}\,(\Pi)$, $Y \in \{N, Ps\}$, computed by (purely) catalytic P systems with at most m membranes and at most k catalysts is denoted by $Y_{\gamma,\delta}OP_m\,(cat_k)$ and $Y_{\gamma,\delta}OP_m\,(pcat_k)$, respectively; from [5] we know that, with the results being sent to the environment (which means taking $f_O = 0$), we have

$$Y_{gen,maxpar}OP_1\,(cat_2) = Y_{gen,maxpar}OP_1\,(pcat_3) = YRE.$$

Remark 1. Here we have to add a remark which is important for the rest of this paper. Originally, Gheorghe Păun used an internal elementary membrane to obtain clean results without having to count the catalysts. Hence, sending out the results also uses a second membrane region, thus, from a topological point of view, there in fact is no difference between using the outer region or an inner membrane region without rules to be applied there. In sum, specifying the number of membranes is not sufficient to capture all subtle features of complexity. Hence, in the following, we will write $P_{1,ext}$ to indicate that, besides the single membrane, we also use the environment as a second membrane region. Thus, the result for (purely) catalytic P systems now will be written as

$$Y_{gen,maxpar}OP_{1,ext}\,(cat_2) = Y_{gen,maxpar}P_{1,ext}\,(pcat_3) = YRE.$$

In the general case, we will also use the notation $P_{m,ext}$ for P systems with m membranes and external output, and to contrast this, we will use $P_{m,int}$ for systems with internal output to make a clear difference to the normal notations P_m which might mean both of these cases.

Finally we remark that P systems with internal output still could (mis)use the environment to let objects vanish, yet we will assume that such symbols will be erased instead of being sent out, so for such P systems, without loss of generality, we can assume that there is no communication with the environment at all.

Remark 2. In order to avoid counting the catalysts in the results, we can also make a projection erasing them. Whereas in general we would write $Y_{gen,maxpar}OP_1^T\,(cat_2)$, instead we now would write $Y_{gen,maxpar}OP_{1,int}^{-cat}\,(cat_2)$. In this case, we really use one membrane only, as only one membrane region itself is needed to obtain the results.

Remark 3. Usually, catalytic P systems and many other variants of P systems can be flattened to one membrane, see [6]. Yet in general, flattening means that we have to make a terminal projection to get the results or to use external output for that purpose, i.e., with catalytic P systems flattened to one membrane, clean

results cannot be obtained without using external out or terminal extraction. In fact, all results in the P systems area should be carefully inspected with respect to these subtle details of complexity definitions.

As we shall see in the following sections, the way how to obtain the results in many cases will have a significant influence on the computational power of several variants of P systems with toxic objects.

Remark 4. As in this paper we will only consider P systems using the maximally parallel derivation mode, in the following the subscript *maxpar* will be omitted.

Finally, P systems can also be considered as mechanisms for generating and accepting string languages as well as for computing any partial recursive function $f : \Sigma^* \to \Gamma^*$ on strings. Here the input string consists of the sequence of symbols taken in from the environment during a halting computation and the output string is formed by the sequence of symbols sent out to the environment; hence, the P system works like in the automaton style, but the input and output streams of symbols are interpreted as strings. In general, any number of symbols can be taken in and sent out in one computation step, and any possible sequence of those symbols has to be taken into account as a substring to be concatenated with the strings already computed by the preceding computation steps – thus, not only one input and one output string may result from a successful halting computation.

The string relation computed by halting computations in a P system Π is denoted by $L_{com}(\Pi)$. If we only consider the symbols taken in from the environment, $L_{com}(\Pi)$ can be seen as an automaton accepting the strings computed by the sequences of symbols taken in during halting computations and we also write $L_{aut}(\Pi)$; if no symbols are taken from the environment, $L_{com}(\Pi)$ describes a string language generated by Π and we also write $L_{gen}(\Pi)$. By $L_{\delta}OP_m(X)$, $\delta \in \{gen, aut\}$, as well as by $RelL_{com}OP_m(X)$ and $FunL_{com}OP_m(X)$ we denote the families of string languages generated and accepted as well as the families of string relations and functions computed by P systems with at most m membranes using rules of type X. With $FunRE$ and $RelRE$ denoting the class of partial recursive string functions and relations, respectively, the following results can be derived from the results proved in [5] (for the generating case, also see [15], Theorem 4.17):

Theorem 1. *For any $\delta \in \{gen, aut\}$, $Z \in \{Fun, Rel\}$*

$$RE = L_{\delta}OP_1(cat_2) = L_{\delta}OP_1(pcat_3)$$

as well as

$$ZRE = ZL_{com}OP_1(cat_2) = ZL_{com}OP_1(pcat_3).$$

3 Toxic Objects in P Systems

We specify a specific subset O_{tox} of O as *toxic* objects. Toxic objects must not stay idle as otherwise the computation is abandoned without yielding a result.

In a successful computation, in any computation step continuing a derivation, we always have to apply multisets of rules evolving all toxic objects. On the other hand, if no rule can be applied any more and thus the system halts, toxic objects do no harm and we take out the results in the usual way depending on the specific definition for the systems under consideration.

A P system with toxic objects is only allowed to continue a computation from a configuration $\mathbb{C}$ by using an applicable multiset of rules covering all copies of objects from O_{tox} occurring in $\mathbb{C}$; moreover, if every non-empty multiset of applicable rules is not covering all toxic objects, the whole computation having yielded the configuration $\mathbb{C}$ is abandoned, i.e., no results can be obtained from this computation.

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

3.1 Variants of P Systems with Toxic Objects

We may distinguish the following variants:

- all symbols are toxic, i.e., we write $Y_\gamma O_{tox\,all} P_m(X)$;
- in catalytic P systems, exactly the catalysts are toxic, i.e., we write $Y_\gamma O_{tox\,cat} P_m(X)$;
- at least the catalysts are toxic, i.e., we write $Y_\gamma O_{tox \geq cat} P_m(X)$;
- all except the catalysts are toxic, i.e., we write $Y_\gamma O_{tox - cat} P_m(X)$.

In all these notations, we may add the superscript T to indicate terminal extraction or the superscript $-cat$ to indicate that the catalysts are not taken into account for the results; moreover, we replace P_m by $P_{m,ext}$ or $P_{m,int}$ in order to explicitly specify that the system uses external or internal output, respectively.

Remark 5. The results established in the following implicitly may assume the P system to be flattened to one membrane, but in the sense of the previous remarks, we have to be very careful whether we have internal output, so that toxicity of symbols matters, or else we have external output, in which case we assume that the objects sent out do not affect the work of the system any more.

4 Purely Catalytic P Systems with All Objects Being Toxic

We first consider the specific variants of P systems which in any step only allow for a bounded number k of rules to be applied, for example, purely catalytic P systems. Obviously, in this case, as until the end of a computation every symbol has to be affected by a rule, at most k symbols can evolve in any computation

step, which of course bounds the number of possible configurations by a constant number, too.

For the generative case with internal output, we show that we get precisely all finite sets, while for the accepting case (i.e., internal input), the power is also quite limited – everything which is not finite is arbitrary, and, moreover, only specific finite sets are accepted.

Lemma 1. *For all $m \geq 1$ and all $k \geq 1$,*

$$Ps_{gen}O_{toxall}P^{-cat}_{m,int}\left(pcat_k\right) = PsFIN.$$

Proof. For the forward inclusion, we notice that the initial configuration is fixed, and the size of a vector we can generate by halting in any non-initial configuration is bounded by the maximal sum of the right-hand sides of rules over different catalysts.

For the converse inclusion, it is enough to mention that for any finite set F of d-dimensional vectors of non-negative integers, there exists a P system Π of type $O_{toxall}P^{-cat}_{1,int}\left(pcat_1\right)$ such that

$$\Pi = \left(O = \{c_1, a\} \cup T, C = \{c_1\}, \mu = \left[\ \right]_1, w_1 = c_1 a, R_1, 1\right),$$

where $\{c_1, a\} \cap T = \emptyset$, $|T| = d$ with T being written $\langle a_1, \cdots, a_d \rangle$ as an ordered set, and R_1 consists of precisely one rule $c_1 a \to c_1 a_1^{k_1} \cdots a_d^{k_d}$ for every element $(k_1, \cdots, k_d) \in F$, so each element of F is generated in one step; for $F = \emptyset$, we simply take the rule $c_1 a \to c_1 a$, which causes an infinite computation. For both cases, we can define R_1 as follows:

$$R_1 = \left\{ c_1 a \to c_1 a_1^{k_1} \cdots a_d^{k_d} \mid (k_1, \cdots, k_d) \in F \right\} \cup \{c_1 a \to c_1 a\}.$$

□

Lemma 2. *For all $m \geq 1$ and all $k \geq 1$,*

$$\begin{aligned} N_{acc}O_{toxall}P_m\left(pcat_k\right) = &\{\{d\} \mid 0 \leq d \leq k-1\} \\ &\cup \{\{0, k'\} \mid 0 \leq k' \leq k\} \cup \{\emptyset, \mathbb{N}\}. \end{aligned}$$

Proof. We proceed with the forward inclusion. Take an arbitrary P system Π of type $O_{toxall}P_m\left(pcat_k\right)$, where

$$\Pi = (O, C = \{c_1, \cdots, c_{k'}\}, \Sigma \subseteq O \setminus C, \mu, w_1, \cdots, w_m, R_1, \cdots, R_m, i_0),$$

where $k' \leq k$. Before the computation starts, input $w_0 \in \Sigma^*$ is added to w_{i_0}. Only two cases are possible that do *not* lead to a computation which is not abandoned immediately: either the P system halts immediately, or all objects from $w_1, \cdots, w_m$ as well as all the objects from $w_0 \in \Sigma^*$ additionally placed in region i_0 participate in catalytic rules in the first step, hence the number of catalysts must be equal to

the number of non-catalysts. In the second case, it follows that the size $|w_0|$ of the input w_0 must be equal to $k' - \sum_{i=1}^{m} |w_i|$. In the first case, it is easy to see that if Π immediately halts on some *non-empty* input, then it must also immediately halt on the *empty* input. If we have at least one rule for every input symbol in Σ, then immediate halting happens *only* on the empty input. If, however, there exists at least one symbol from Σ that does not appear in the left side of any rule from R_{i_0}, then *any* number of these symbols (let us call them "passive") would be accepted.

We now put it all together. For the same system, having both immediate halting case and later halting case is only possible if besides the input, the initial system has *only* catalysts. This yields exactly $\{0, k'\}$, $k' \leq k$ if there are no passive objects in Σ, or the entire set $\mathbb{N}$ otherwise. Only immediate halting yields $\{0\}$ and $\mathbb{N}$, depending on the presence of passive objects in Σ. Finally, only later halting yields $\{d\}$ for $0 \leq d < k$ (the last inequality is strict since at least one non-catalyst is needed besides the input to reject 0). And of course, we may have a P system with no halting computations, accepting $\emptyset$. The family of all sets mentioned above is $\{\{d\} \mid 0 \leq d \leq k-1\} \cup \{\{0, k'\} \mid 0 \leq k' \leq k\} \cup \{\emptyset, \mathbb{N}\}$, which proves the forward inclusion of the claim of the lemma.

For the converse inclusion, it is enough to exhibit P systems for each of these sets; in every case, the input alphabet is $\Sigma = \{a\}$.

$$\begin{aligned}
&\Pi_{\emptyset} = \left(O = \{c_1, a\}, C = \{c_1\}, \Sigma = \{a\}, \mu = \left[\ \right]_1, w_1 = c_1 a, R_1, i_0 = 1\right), \\
&R_1 = \{c_1 a \to c_1 a\};
\end{aligned}$$

$$\begin{aligned}
&\Pi_{\mathbb{N}} = \left(O = \{c_1, a\}, C = \{c_1\}, \Sigma = \{a\}, \mu = \left[\ \right]_1, w_1 = c_1, R_1, i_0 = 1\right), \\
&R_1 = \emptyset;
\end{aligned}$$

$$\begin{aligned}
&\Pi_{d,0} = \left(O = C \cup \{a\}, C = \{c_1, \cdots, c_d\}, \Sigma = \{a\}, \mu = \left[\ \right]_1, w_1, R_1, i_0 = 1\right), \\
&w_1 = c_1 \cdots c_d, \\
&R_1 = \{c_i a \to c_i \mid 1 \leq i \leq d\},\ 0 \leq d \leq k;
\end{aligned}$$

$$\begin{aligned}
&\Pi_{d} = \left(O = C \cup \{a\}, C = \{c_1, \cdots, c_{d+1}\}, \Sigma = \{a\}, \mu = \left[\ \right]_1, w_1, R_1, i_0 = 1\right), \\
&w_1 = c_1 \cdots c_{d+1} a, \\
&R_1 = \{c_i a \to c_i \mid 1 \leq i \leq d\},\ 0 \leq d \leq k-1.
\end{aligned}$$

Indeed, the only rule of $\Pi_{\emptyset}$ forces an infinite loop on the empty input, while for a non-empty input the computation is blocked because more than one toxic object a cannot be simultaneously taken by c_1. On the other end of the spectrum, $\Pi_{\mathbb{N}}$ accepts any input by immediate halting, because the catalyst always stays idle as there is no rule in the system. P system $\Pi_{d,0}$ either halts immediately with no input, or halts after one step, erasing the input of exactly d objects, $d \leq k$. Finally, the P system Π_d halts after one step, erasing the input of exactly d objects, $d < k$.

These observations conclude the proof. □

Note 1. Of course, characterizing sets of vectors in the accepting case would be more tedious for the following reason. Without passive objects, P system would accept *some* subset of $\Sigma^{\leq k-1}$, as well as a subset of $\Sigma^{\leq k}$ containing *some* vectors of weight $d \leq k$ and a vector of zeros, and the empty set. With passive objects, *any* number of them is allowed for the case of immediate halting, while the projection of the accepted vectors onto the non-passive objects should form a set containing *some* vectors of weight $d \leq k$ and a vector of zeros. The meaning of the word *some* throughout this note can be made more precise by analyzing exactly which multisets of weight d can be consumed by d catalysts, depending on the rules of the system (whereas in the case of accepting numbers, only the total weight of such multisets was taken into consideration).

While Lemma 1 characterized $PsFIN$ by P systems with internal output, in the case of external output their power becomes exactly $PsREG$, as the following theorem shows:

Lemma 3. *For all $m \geq 1$ and all $k \geq 1$,*

$$Ps_{gen}O_{toxall}P_{m,ext}\left(pcat_k\right) = PsREG.$$

Proof. Let $M = (Q, T, \delta, q_0, F)$ be a deterministic finite automaton. Then we construct the P system Π which generates $Ps\left(L\left(M\right)\right)$:

$$\begin{aligned}
\Pi &= \left(O = C \cup Q \cup T, C = \{c_1\}, \mu = \left[\ \right]_1, w_1 = c_1 q_0, R_1, i_0 = 0\right), \\
R_1 &= \{c_1 p \to c_1 q(a, out) \mid (p, a, q) \in \delta, p, q \in Q, a \in T\} \\
&\quad \cup \{c_1 p \to c_1 \mid p \in F\}.
\end{aligned}$$

We conclude that $PsREG \subseteq Ps_{gen}O_{toxall}P_{1,ext}\left(pcat_1\right)$.

The converse inclusion can be argued as follows: In any successful computation step with k catalysts, there must be exactly k non-catalysts, and a computation stops with having yielded a result if all objects inside the system including the catalysts are idle; hence, this finite set of useful configurations is finite and constitutes the set of states of a finite automaton simulating the computations of the P system. Since every rule in such a system involves one catalyst and one non-catalyst, for a configuration C to allow some derivation $C \Rightarrow C'$ it is necessary (although not sufficient) that the number of catalysts equals the number of non-catalysts inside the system. Hence, for a P system

$$\Pi = (O, C, \mu, w_1, \cdots, w_{m'}, R_1, \cdots R_{m'}, i_0)$$

having fixed the set of objects O, the membrane structure μ of $m' \leq m$ membranes, and the set of catalysts C, with the number $k' \leq k$ of catalysts, the set Q of configurations containing a total of exactly k' objects from $O \setminus C$ in m' regions of the P system is bounded. Moreover, the set Q'' of all configurations reachable from Q in one step is also bounded. Finally, we define $Q' = Q'' \cup \{q_0\}$ where q_0 is

the initial configuration, as well as $Q_h \subseteq Q'$ to be the set of halting configurations (in which no rule can be applied any more).

Hence, a P system with external output generating vectors of natural numbers can be modeled by a finite automaton $M = (Q', T, \delta, q_0, Q_h)$ having Q' as the set of states, T contains d symbols for the generation of d-dimensional vectors, δ contains the triple (p, v, q) for any transition from a configuration $p \in Q'$ to a configuration $q \in Q'$ sending out v; the set of the final states is precisely Q_h.

In sum, with all objects being toxic, purely catalytic P systems with external output can exactly generate the regular sets of vectors. □

The statement of Lemma 3 can be generalized to languages, as well as to P automata and P transducers. Indeed, in case of external input (P automaton case) and/or external output, the finite number of different configurations can serve as the finite state set of a finite automaton for the input specified by $(a, come)$ in the rules and/or for the output specified by (a, out) in the rules.

Lemma 4. *For all $m \geq 1$ and all $k \geq 1$,*

$$L_{gen}O_{toxall}P_m\,(pcat_k) = REG.$$

Proof. Using similar arguments as already pointed out in the previous proof, we can easily argue that purely catalytic P systems with all objects being toxic can generate any regular language L; the only difference now is that any sequence of symbols sent out during a successful computation is interpreted as string.

Now we consider the converse, i.e., as in the previous proof, a P system with external output generating strings can be modeled by the finite automaton having Q' as the set of states and Q_h as the set of the final states as constructed there, but now for any transition from a configuration $p \in Q'$ to a configuration $q \in Q'$ sending out v, δ contains the triple (p, v', q) for all $v' \in Perm(v)$. □

Lemma 5. *For all $m \geq 1$ and $k \geq 2$,*

$$L_{aut}O_{toxall}P_m\,(pcat_k) = REG,$$
$$Rel_{aut}O_{toxall}P_m\,(pcat_k) = RelREG.$$

Proof. For a P automaton or a P transducer Π, again take Q' and Q_h as constructed in the proof of Lemma 4.

A P automaton can be modeled by a finite automaton having Q' as the set of states. A transition from configuration $p \in Q'$ to a configuration $q \in Q'$ while having u brought from the environment is simulated by rules (p, u', q) for all $u' \in Perm(u)$; clearly, $|u| \leq k'$. The set of the final states is precisely Q_h.

A P automaton with external output can be modeled by a finite transducer having Q' as the set of states. A transition from configuration $p \in Q'$ to a configuration $q \in Q'$ while having u brought in and v sent out is simulated by rules

$(p, u'/v', q)$, $u' \in Perm(u)$, $v' \in Perm(v)$; clearly, $|u| \leq k'$ and $|v| \leq k'$. The set of the final states is precisely Q_h.

For proving the converse inclusion, take an arbitrary finite automaton $M = (Q, T, \delta, q_0, F)$; without loss of generality, we assume that M has at least one outgoing transition from any non-final state.

A P automaton

$$\Pi = \left(O = \{c_1, c_2, b\} \cup T, C = \{c_1, c_2\}, \Sigma = T, \mu = \left[\ \right]_1, w_1 = c_1 c_2 b, R_1, i_0 = 1\right)$$

$$R_1 = \{c_1 p \to c_1 q(a, come) \mid (p, a, q) \in \delta, p, q \in Q, a \in T\}$$
$$\cup \{c_2 x \to c_2 \mid x \in \{b\} \cup T\} \cup \{c_2 q \to c_2 \mid q \in F\}$$

can simulate M, using two catalysts c_1, c_2. Each transition $(p, a, q) \in \delta$ can be simulated by rule $c_1 p \to c_1 q(a, come)$, but also rule the $c_2 a \to c_2$ is needed to erase the symbol previously brought in. The initial contents of a single membrane is $c_1 c_2 q_0 b$, where besides q_0, one additional non-catalyst $b \notin \{c_1, c_2\} \cup T$ is used to keep c_2 busy in the first step. Halting can be simulated by rule $c_1 q \to c_1$ for each final state q; in the same step c_2 deletes the last symbol brought in.

A P automaton with external output can simulate a finite transducer in the same way as a P automaton without output simulates a finite automaton. The only difference is that now the simulated transitions have the form $(p, a/u, q)$, and the corresponding simulating rules are $c_1 p \to c_1 q(a, come)(u, out)$, the rest of the construction being exactly the same as in the previous paragraph. □

Two catalysts are needed for simulating an arbitrary finite automaton by a P automaton, since both the state symbol and the symbol brought in from the environment have to be processed in parallel. For the case of only one catalyst, the object brought in from the environment itself has to serve as a state. However, in this way, the last object brought inside completely determines the set of possible objects that can be brought inside in the next step, which considerably reduces the generality of finite automata. Having this in mind, we characterize input-driven finite automata:

Lemma 6. *For all $m \geq 1$,*

$$L_{aut} O_{toxall} P_m (pcat_1) = IDREG,$$
$$Rel_{aut} O_{toxall} P_m (pcat_1) = RelIDREG.$$

Proof. The inclusion that at most $IDREG/RelIDREG$ is generated/computed with one catalyst follows from the fact that exactly one non-catalyst may appear in any non-halting non-blocking configuration, and, except the initial configuration, this is precisely the symbol taken from the environment in the previous step. In any successful computation, the only rules applied in any step, possibly except in the last step, are erasing one non-catalyst while bringing in another one instead.

For the inclusion that we can generate/compute the entire families $IDREG/RelIDREG$, we use a construction similar to that of the previous lemma,

except instead of erasing the object a brought in the previous step by c_2, this object is used instead of the state object: each transition (q_a, b, q_b) now is simulated by the rule $c_1 a \to c_1(b, come)$. The initial contents of a single membrane is $c_1 q_0$, where the initial state q_0 is an additional symbol not in the input alphabet. Halting can be simulated by the rules $c_1 a \to c_1$ for all final states q_a.

For the case of a P automaton with external output, the simulation is the same as above, except that now we also have an output: the transitions to be simulated have the form $(q_a, b/u, q_b)$, and the corresponding simulating rules are $c_1 a \to c_1(b, come)(u, out)$; the rest of the construction is exactly the same as in the previous paragraph. □

Finally, the domain of relations computed with internal input (with either output region) corresponds to the sets accepted with internal input, see Lemma 2. Similarly, the range of relations computed with internal output and with internal input corresponds to the sets generated with internal output, see Lemma 1. The nature of these relations always results from a finite-state behavior, but we are not going into further details here; another question to be answered in the future is the exact characterization of P automata with internal output.

5 Non-Cooperative P Systems with All Objects Being Toxic

In this section we consider P systems without catalysts and with only non-cooperative rules, yet with all objects being toxic.

5.1 Connection to L Systems

Example 1. Take the following P system with all objects being toxic.

$$\Pi_{int} = \left(O = \{a, b\}, \mu = \left[\ \right]_1, w_1 = a, R_1 = \{a \to aa,\ a \to b\}, i_0 = 1\right).$$

In n computation steps we obtain a^{2^n} and in a final step b^{2^n}. Only in this last step we may apply the rule $a \to b$ introducing the toxic symbol b for which no rule exists. Hence, the generated set is $N_{gen}(\Pi_{int}) = \{2^n \mid n \geq 0\}$.

Example 2. The same set is accepted by the P automaton Π_{aut} and generated by the P system Π_{ext} with external output:

$$N_{aut}(\Pi_{aut}) = N_{gen}(\Pi_{ext}) = N_{gen}(\Pi_{int}) = \{2^n \mid n \geq 0\} \text{ where}$$
$$\Pi_{aut} = \left(O = \{a, b\}, \mu = \left[\ \right]_1, w_1 = a, R_1 = \{a \to aa,\ a \to (b, come)\}, i_0 = 1\right);$$
$$\Pi_{ext} = \left(O = \{a, b\}, \mu = \left[\ \right]_1, w_1 = a, R_1 = \{a \to aa,\ a \to b(b, out)\}, i_0 = 0\right).$$

Indeed, the behavior of Π_{aut} and Π_{ext} is the same as that of Π_{int}, except producing b inside the membrane is replaced by bringing in b from the environment, or accompanied by sending out b to the environment. Again, if both rules are simultaneously applied, then the toxic objects b will block the computation, but still we will get a result if only symbols b are present in the final configuration.

We now compare non-cooperative P systems with all objects being toxic to L systems.

Lemma 7. *For* $(\gamma, \alpha) \in \{(gen, int), (gen, ext), (aut, -)\}$,

$$PsE0L \subseteq Ps_{\gamma}O_{toxall}P_{1,\alpha}\,(ncoo)\,.$$

Proof. Let $G = (V, T, P, w)$ be an $E0L$ system. We recall that for each symbol $a \in V$, P has to contain some rule with a on the left side. For every $a \in T$, we replace a by N_a throughout all the rules of P. Moreover, we take $N_a \to a$ into R for every $a \in T$. These terminal rules are exactly applied in the last step of a derivation in the P system Π, whereas the other rules in R simulate the rules from P. As the final step with using the rules $N_a \to a$ is an additional derivation step, instead of rules $a \to \lambda$ erasing a symbol a we instead have to use the rules $N_a \to E$ and $E \to \lambda$ where E is a new symbol representing the erased symbol for one step.

Hence, we construct the corresponding non-cooperative P system Π with all symbols being toxic as follows (for the automaton case, we have to insert $\Sigma = T$):

$$\begin{aligned}
\Pi &= (O = V \cup \{N_a \mid a \in T\} \cup \{E\}, \mu = [\ ]_1\ , w_1 = w, R_1, i_0 = 1),\\
R_1 &= \{h(a) \to h(u) \mid a \to u \in P,\ a \in V\} \cup \{N_a \to E \mid a \to \lambda \in P,\ a \in T\}\\
&\quad \cup \{N_a \to a \mid a \in T\} \cup \{E \to \lambda\}\,,
\end{aligned}$$

where $h : V \cup \{E\} \to V \setminus T \cup \{N_a \mid a \in T\} \cup \{E\}$ is the morphism given by $h(a) = a$ for $X \in V \setminus T \cup \{E\}$ and $h(a) = N_a$ for $a \in T$.

By construction, every object from $\{N_a \mid a \in T\} \cup \{V \setminus T\}$ can evolve by rules from R_1, while objects in T cannot. If a computation in Π ends up with a configuration in which the skin contains both objects from T and objects not from T, then the computation is blocked without yielding any result. Therefore, the only derivations of Π which will not be discarded are those in which Π simulates a derivation of G up to some configuration $h(w)$, $w \in (T \cup \{E\})^*$, and then applies the rules $N_a \to a$ and eventually the rule $E \to \lambda$, and only those, to transform $h(w)$ into w.

To show the same result for external output, it suffices to set i_0 to 0 and replace every rule $N_a \to a$ by $N_a \to a(a, out)$. Alternatively, to show the same result for external input, it suffices to set i_0 to 0 and replace every rule $N_a \to a$ by $N_a \to a(a, come)$. □

In the case of P systems with internal output (without terminal filtering) and only one membrane, we can directly show the converse inclusion.

Lemma 8. $Ps_{gen}O_{toxall}P_{1,int}\,(ncoo) \subseteq PsE0L$.

Proof. Let $\Pi = (O, \mu = [\ ]_1\ , w_1, R_1, i_0 = 1)$ be a non-cooperative P system with all objects being toxic. We construct the corresponding $E0L$ system G as follows:

$$G = (V = O \cup \{\#\}, T, P, w = w_1),$$
$$T = \{a \in O \mid \text{there exists no rule } a \to u \in R \text{ with } u \in O^*\},$$
$$P = R_1 \cup \{a \to \# \mid a \in T \cup \{\#\}\}.$$

We immediately observe that, whenever G introduces a terminal symbol, it will be rewritten into a trap symbol in the next step. Thus, the only way for G to produce a terminal string is to move from a string over $V \setminus T$ to a string over T in a single step. But this exactly corresponds to the way in which Π evolves, because rewriting a terminal a into # in G corresponds to discarding the derivation of Π in which a is produced alongside non-terminals. □

Corollary 1. $Ps_{gen}O_{toxall}P_{1,int}(ncoo) = PsE0L$.

Proof. The result follows from Lemma 8 in combination with Lemma 7 for the case of P systems with internal output and only one membrane. □

In case of multiple membranes or terminal filtering or both, however, there is a problem: symbols that represent objects in non-output regions do not contribute to the output. Yet, since $E0L$ is known to be closed under arbitrary morphisms (see, e.g., [16] volume 1 page 34), the result can be strengthened as follows:

Theorem 2. *For all* $m \geq 1$,

$$\begin{aligned} PsE0L &= Ps_{gen}O_{toxall}P^T_{m,int}(ncoo) \\ &= Ps_{gen}O_{toxall}P_{m,int}(ncoo) \\ &= Ps_{gen}O_{toxall}P_{m,ext}(ncoo) \\ &= Ps_{aut}O_{toxall}P_m(ncoo). \end{aligned}$$

Proof. We only have to show that any P system with internal output, eventually even with terminal extraction, or else with external output (terminal extraction need not be considered in this case, as any non-wanted symbol need not be sent out) or external input can be simulated by an $E0L$-system.

First, we can flatten the given P system Π with internal output to only one membrane, yet keeping in mind that then we have to use terminal extraction to obtain the results in a clean form. Hence, in this case, we simply apply the construction from Lemma 8 to the flattened P system Π' thus obtaining an $E0L$-system G generating a set of strings which exactly represent the multisets generated by the P system Π'. In order to obtain the original results, we have to apply a projection h_T erasing all non-terminals only yielding strings/multisets over T. As $E0L$ is closed under arbitrary morphisms (see, e.g., [16] volume 1 page 34), from G we can construct an $E0L$-system G' directly generating the desired results. If the original P system Π used terminal extraction to a terminal alphabet Σ, we can to apply another projection from T to Σ to obtain the desired results (by constructing a corresponding $E0L$-system G'').

If we have a P system Π using external output, we instead first flatten the system to an equivalent P system Π' with only one membrane, but still having external output. Then, from this P system Π' with one membrane and external output we construct an equivalent P system Π'' with only one membrane region having internal output in the skin membrane, but with terminal filtering. Instead of sending a symbol a out by using (a, out) on the right side of a rule in Π, we replace any occurrence of (a, out) by N_a in the rules of the skin membrane in Π''. In that way, the P system Π'' keeps each symbol a sent out by Π in the new inner output membrane $i_0 = 1$ as N_a. In the skin membrane, the rules $N_a \to N_a$ keep these symbols alive until the application of the rules $N_a \to a$ allows the system to halt with yielding the desired result if exactly with the application of these terminal rules no non-terminal afterwards remains in the whole system. As a subtle detail we have to mention that we again, as in the proof of Lemma 7, have to be careful with λ-rules $a \to \lambda$ (again we use the rules $a \to E$ and $E \to \lambda$ instead), but now also with any other "passive" object b which cannot evolve any more - such a symbol has to be treated like a terminal symbol, going to an intermediate symbol N_b before it finally goes to b, and then each of these symbols is projected to λ by using the terminal extraction.

Hence, we conclude that computations in Π'' and in Π yield the same results. According to the construction given above, we can obtain an $E0L$-system G'' such that $Ps\left(L\left(G''\right)\right) = Ps\left(\Pi''\right) = Ps\left(\Pi\right)$.

In the automaton case, we can use similar ideas as in the previous case: instead of $(a, come)$ for terminal symbols from the input alphabet Σ on the right side of rules of a P automaton Π we use $N_a N'_a$ in a flattened P system Π'' with internal output and terminal extraction. Then N'_a is used instead of a in the rules of Π'' used instead of the corresponding rules of Π, and we also add the rules $N_a \to N_a$ and the terminal rules $N_a \to a$. Moreover, any "passive" object b which cannot evolve any more has to be treated as already explained before; the same holds for the dealing with λ-rules $a \to \lambda$. Hence, finally projecting all terminal symbols on themselves and all other symbols on λ with the projection h_Σ, we have got a P system with internal input and terminal extraction Π'' such that $h_\Sigma(Ps(\Pi'')) = Ps(\Pi)$. □

5.2 Internal Input

However, the accepting power of P systems with internal input is much lower, namely subregular.

Lemma 9. $Ps_{acc}O_{tox\,all}P_m(ncoo) \subsetneq PsREG$.

Proof. First, if a P system Π accepts *any* non-empty input over the input alphabet Σ, then also *the empty input* is accepted. Indeed, take an arbitrary P system Π accepting some multiset $w_{in} \in \Sigma^*$, say in m steps. Each of the objects, both initial ones and input ones, initially being in the system, will produce some (possibly

empty) multiset of objects which cannot further evolve by rules of Π. Clearly, replacing w_{in} by λ and following exactly the same evolution of all initial objects, we will get an accepting computation of at most m steps.

Second, for a similar reason, if w_{in} is accepted, than any submultiset of w_{in} is accepted.

Third, if Π accepts some input w_{in} containing *at most* one occurrence of any symbol in Σ, then it also accepts every input $w'_{in} \in (alph\,(w_{in}))^*$, i.e., *any* multiset over $\{a \in \Sigma \mid |w_{in}|_a > 0\}$. Indeed, consider the accepting computation of w_{in}, say of m steps. In this computation, every input object a from w_{in} is either erased in at most m steps, or produces in exactly m steps some non-empty multiset of objects that cannot evolve by rules of Π; let us call such symbols "passive". Replacing each input object by an arbitrary number of its copies, following the same evolution as in the accepting computation before, we again get a computation where every input object is either erased in at most m steps, or produces in exactly m steps some non-empty multiset of passive objects. This computation will either erase everything in at most m steps, or halt in exactly m steps.

Therefore, non-cooperative P systems with internal input with all symbols being toxic can accept *at most* all possible unions of sets from $\{T^* \mid T \subseteq \Sigma\}$. □

It can be shown that last statement from the proof given above actually is an equality.

Theorem 3. *For all $m \geq 1$,*

$$Ps_{acc}O_{toxall}P_m(ncoo) = \left\{ Ps\left(\bigcup_{i=1}^{n} T_i^*\right) \mid \Sigma \text{ alphabet}, T_i \subseteq \Sigma,\ 1 \leq i \leq n,\ n \geq 0 \right\}.$$

Proof. The inclusion $\subseteq$ follows from the proof of the previous theorem, so it suffices to prove that all such sets can indeed be accepted. For $n = 0$, $\emptyset$ can be accepted by the P system

$$\Pi = \left(O = \{a, b\}, \Sigma = \{a\}, \mu = [\ ]_1, w_1 = b, R_1 = \{b \to b\}, i_0 = 1\right).$$

Take arbitrary numbers $n > 0$ and $k > 0$, an input alphabet $\Sigma = \{a_{j,0} \mid 1 \leq j \leq k\}$ and sets $T_i \subseteq \Sigma$, $1 \leq i \leq n$. We construct a P system accepting precisely all inputs from $\bigcup_{i=1}^{n} T_i^*$.

$$\begin{aligned}
\Pi &= \left(O, \Sigma, \mu = [\ ]_1, w_1 = \lambda, R_1, i_0 = 1\right),\\
O &= \{a_{j,i} \mid 0 \leq i \leq n+1,\ 1 \leq j \leq k\} \cup \{b\},\\
R_1 &= \{a_{j,i} \to a_{j,i+1} \mid 0 \leq i \leq n,\ 1 \leq j \leq k\}\\
&\quad \cup \{a_{j,i} \to b \mid 1 \leq i \leq n,\ 1 \leq j \leq k, a_{j,0} \in T_i\}\\
&\quad \cup \{a_{j,n+1} \to a_{j,n+1} \mid 1 \leq j \leq k\}.
\end{aligned}$$

Indeed, every input object $a_{j,0}$ either enters an infinite loop after $n+1$ steps, or evolves into b (that cannot further evolve by the rules of Π) after some number i

of steps if $a_{j,0} \in T_i$. The only way a non-empty input is accepted is if all objects evolve into b in the same number i of steps, which is possible if and only if the input is contained in $\bigcup_{i=1}^{n} T_i^*$. □

5.3 Describing Languages

In the previous subsection we have studied numbers and vectors described by non-cooperative P systems with all objects being toxic. We have shown that they characterize very limited subregular behavior in the case of internal input, while in the cases of external input, internal output or external output their power is strongly related to that of $E0L$ systems. A natural question arises – what can we say about languages? We now illustrate the difficulty of this problem by a few examples, generating non-context-free languages and illustrating the power of synchronization emerging from halting with toxic objects.

Example 3. Our first example shows how a simple non-context-free language can be accepted using external input:

$$\Pi = \left(O = \{S, a, b, c\}, \mu = \left[\ \right]_1, w_1 = S, R_1, i_0 = 0\right),$$
$$R_1 = \{S \to S(a, come),\ S \to a,\ a \to a,\ a \to (bc, come)\}.$$

This P system accepts the language $L_{a(bc)} = \bigcup_{n \geq 1} \{a^n\}\, Perm\,(b^n c^n)$, since the multiplicities of symbols a, b and c brought in are the same, and all objects b and c must be brought inside in the same (i.e., in the last) step of the computation, which must be after all objects a have been brought inside.

Note that, without toxicity, the result would still be some non-context-free language consisting of the same number of symbols a, b and c, but it would also contain strings which are not of the form $\{a\}^* \{b, c\}^*$.

It is no longer surprising that replacing $(x, come)$ by $x(x, out)$ for all $x \in \{a, b, c\}$ in the system above, we get a P system with external output *generating* the same language $L_{a(bc)}$.

Example 4. If, in the previous example, we replace each rule $a \to (bc, come)$ by the two rules $a \to (b, come)$ and $b \to (c, come)$, we get a P automaton accepting language $L_{a(b)(c)} = \{a^n b^n c^n \mid n \geq 1\}$, because to halt without being blocked, all objects c must be brought inside in the same step, and therefore also all objects b must have been brought inside just one step before that, and hence all objects a must have already been brought in by then. Clearly, replacing $(x, come)$ by $x(x, out)$ for all $x \in \{a, b, c\}$ throughout all the rules, we get a P system with external output again *generating* the language $L_{a(b)(c)}$.

In the rest of this section we show that non-cooperative P systems can generate rather complicated languages even without making use of the synchronization power of toxicity, and taking all objects to be toxic does not change the language.

The following example of a difficult language generated by a non-cooperative P system with external output is taken from [2]. This language is considerably more "difficult" than languages in $REG \cdot Perm(REG)$, which informally can be explained as follows: besides permutations of symbols sent out at the same time, it exhibits another kind of non-context-freeness, although this second source of "difficulty" alone, however, could be captured as the intersection of two linear languages.

Example 5. Consider the non-cooperative P system with external output

$$\begin{aligned}\Pi_D &= \left(O = \{D, D', a, b, c, a', b', c'\}, \mu = \left[\ \right]_1, w_1 = DD, R_1, i_0 = 0\right),\\ R_1 &= \big\{D \to (a, out)(b, out)(c, out)D'D',\ D \to (a, out)(b, out)(c, out),\\ &\quad D' \to (a', out)(b', out, (c', out)DD,\ D' \to (a', out)(b', out)(c', out)\big\}.\end{aligned}$$

The contents of region 1 is a population of objects D, initially 2, which are primed if the step is odd. Assume that there are k objects inside the system. In each step, every symbol D is either erased or doubled (and primed or deprimed), so the next step the number of objects inside the system will be any even number between 0 and $2k$. In addition to that, the output during that step is $Perm\left((abc)^k\right)$, primed if the step is odd. Hence, the generated language can be described as

$$\begin{aligned}L(\Pi_D) = \bigcup\nolimits_{k_0=1,\ 0\le k_i\le 2k_{i-1},\ 1\le i\le 2t+1,\ t\ge 0}\ & Perm\left((abc)^{2k_0}\right) Perm\left((a'b'c')^{2k_1}\right)\cdots\\ & Perm\left((abc)^{2k_{2t}}\right) Perm\left((a'b'c')^{2k_{2t+1}}\right).\end{aligned}$$

To give an idea of how complex a language generated by a non-cooperative membrane system can be, imagine that the skin may contain populations of multiple symbols that (like D in the example above) can be erased or multiplied (with different periods), and also be rewritten into each other. The same, of course, happens in usual context-free grammars, but since the terminal symbols in P systems with external output are collected from the derivation tree level by level instead of from left to right as in context-free grammars, the effect is quite different.

We finally again mention that the generated language remains the same even if all objects are toxic. Moreover, by replacing all outputs of the form (x, out) by $(x, come)$ and adding rules $x \to \lambda$ we can convert this P system with external output into a P automaton defining the same language.

6 Catalytic P Systems with Exactly the Catalysts being Toxic Generate at Least $PsMAT$

In this section we investigate catalytic P systems where precisely the catalysts are toxic, i.e., the computation is aborted if any of them is not used in some step

before the computation halts. We prove that at least all Parikh sets of matrix languages can be generated in this setting, using the fact that partially blind register machines generate precisely $PsMAT$.

Theorem 4. *For all $m \geq 1$,*

$$Ps_{gen}O_{toxcat}P_{m,int}(cat_*) \supseteq PsMAT.$$

Proof. Let $M = (d, B, l_0, l_h, P)$ be a partially blind register machine, with the first k registers being its output registers. Let $e = f_{k+1} \cdots f_d$ and let $e(r)$ be e without f_r. Without loss of generality, we assume that the first instruction labeled by l_0 is an ADD-instruction. We then construct the following P system.

$$\begin{aligned}
&\Pi = (O, C, \mu = [\ ]_1, w_1 = l_0 e f_{d+1}, R_1, i_0 = 1), \text{ where}\\
&C = \{c_r \mid k+1 \leq r \leq d\} \cup \{c_p\},\\
&O = C \cup B \cup \{f_r \mid m+1 \leq r \leq d+1\} \cup \{o_r \mid 1 \leq r \leq d\},\\
&R_1 = \{c_p l_i \to c_p l_j o_r e,\ c_p l_i \to c_p l_k o_r e \mid l_i : (\texttt{ADD}(r), l_j, l_k) \in P\}\\
&\quad \cup \{c_p l_i \to c_p l_j e(r) \mid l_i : (\texttt{SUB}(r), l_j) \in P\}\\
&\quad \cup \{c_r o_r \to c_r,\ c_r f_r \to c_r,\ c_r f_{d+1} \to c_r \# \mid k+1 \leq r \leq d\}\\
&\quad \cup \{x \to \# \mid x \in B \cup \{\#\} \cup \{f_r \mid k+1 \leq r \leq d\}\}\\
&\quad \cup \{c_p l_h \to c_p e\} \cup \{c_p f_{d+1} \to c_p\}.
\end{aligned}$$

The catalyst c_r for the working register r, $k+1 \leq r \leq d$, is kept busy by the single copy of the symbol f_r; only in the case of a SUB-instruction on r, in the next step the rule $c_r o_r \to c_r$ should be applied, hence, $e(r)$ is taken instead of e, thus leaving the catalyst c_r free for an object o_r. On the other hand, if such a symbol o_r in that case is not present, due to maximal parallelism the rule $c_r f_{d+1} \to c_r \#$ introducing the trap symbol # has to be applied. The catalyst c_p is used for guiding the computation according to the program given by P. When M reaches the final halting instruction labeled by l_h, then for a last time e is generated, but no instruction label is generated any more, hence, in the last step of a successful computation of Π the rule $c_p f_{d+1} \to c_p$ will be applied together with the rules $c_r f_r \to c_r$ for $k+1 \leq r \leq d$. The computation in Π will then only stop with yielding a result if no rule can be applied any more, i.e., if no trap symbol has been introduced during the computation, and moreover, at the end no symbol o_r for $k+1 \leq r \leq d$ is present, i.e., if all working registers are empty.

The lack of the symbol f_r when a SUB-instruction on register r is carried out guarantees that the computation is trapped if no register symbol o_r, $k+1 \leq r \leq d$, is present by the enforced application of the rule $c_r f_{d+1} \to c_r \#$. If the final rule $c_p f_{d+1} \to c_p$ is applied too early, i.e., as long as some $l \in B$ is present, then the introduction of the trap symbol # by the rule $l \to \#$ is enforced by the maximal parallelism.

We finally observe that only the catalysts are toxic here, so any number of symbols o_r with $1 \leq r \leq d$ can be generated during any successful computation. □

The preceding theorem shows that a partially blind register machine with $m-1$ working registers can be simulated by a P system with internal output in the single membrane with at most m catalysts these being exactly the toxic objects.

7 Conclusion and Future Research

In this paper we have introduced multiple variants of P systems with toxic objects, depending on which objects are toxic. It is important to note that so far in P systems toxic objects and the concept of synchronized halting with toxic objects has not been used in the literature, and thus toxic objects also have not been used to change the computational power of P systems, but rather to decrease the rule complexity of P systems (toxic objects in that case being equivalent to the usual ones, additionally having rules rewriting them into a trap symbol thus forcing the computation to never halt). In this paper, the results are quite different. The most visible one is the non-cooperative case, where toxicity boosts the computational power from $PsREG$ to $PsE0L$. On the other side, requiring certain, or even all, objects to be toxic can also bring limitations to the computational power. The most dramatic limitation is the power of purely catalytic P systems, where complete toxicity lowers the power of internal output from $PsRE$ all the way down to $PsFIN$, and the power of internal input from NRE down to either accepting all numbers, or accepting very restricted finite sets.

Most results we have obtained here describe the computational power of P systems with all objects being toxic or precisely all catalysts being toxic, depending on the kinds of rules used (e.g., non-cooperative, purely catalytic or catalytic), the number of membranes, the output region, in terms of number sets, vector sets or languages, or even computing relations. We repeat the results we obtained in this paper, for easier comparison.

For all $m \geq 1$, we have shown that

$$
\begin{aligned}
Ps_{gen}O_{toxall}P_{m,int}\,(pcat_k) &= PsFIN,\ k \geq 1,\\
N_{acc}O_{toxall}P_m\,(pcat_k) &= \{\{d\} \mid 0 \leq d \leq k-1\}\\
&\quad \cup \{\{0,k'\} \mid 0 \leq k' \leq k\} \cup \{\emptyset, \mathbb{N}\},\ k \geq 1,\\
L_{gen}O_{toxall}P_m\,(pcat_k) &= REG,\ k \geq 1,\\
L_{aut}O_{toxall}P_m\,(pcat_k) &= REG,\ k \geq 2,\\
Rel_{aut}O_{toxall}P_m\,(pcat_k) &= RelREG,\ k \geq 2,\\
L_{aut}O_{toxall}P_m\,(pcat_1) &= IDREG,\\
Rel_{aut}O_{toxall}P_m\,(pcat_1) &= RelIDREG,\\
Ps_{gen}O_{toxall}P^T_{m,int}\,(ncoo) &= PsE0L,\\
Ps_{gen}O_{toxall}P_{m,int}\,(ncoo) &= PsE0L,\\
Ps_{gen}O_{toxall}P_{m,ext}\,(ncoo) &= PsE0L,\\
Ps_{aut}O_{toxall}P_m\,(ncoo) &= PsE0L,\\
Ps_{acc}O_{toxall}P_m(ncoo) &= \{Ps\,(\textstyle\bigcup_{i=1}^{n} T_i^*) \mid \Sigma \text{ alphabet}, T_i \subseteq \Sigma,\ 1 \leq i \leq n,\ n \geq 0\},\\
Ps_{gen}O_{toxcat}P_{m,int}(cat_*) &\supseteq PsMAT.
\end{aligned}
$$

Non-cooperative P systems with all objects being toxic have been shown to be quite versatile, for example, we can easily generate $\{a^n b^n c^n \mid n \geq 1\}$.

Multiple problems remain open. We find particularly interesting the following ones:

- Characterize $Ps_{gen}O_{toxcat}P_{m,int}(cat_*)$ and all other possible variants of restricting the set of toxic objects not yet covered by the results obtained in this paper.
- *There still remains the open problem how to characterize the families of sets of (vectors of) natural numbers generated by [purely] catalytic P systems with only one [two] catalyst[s].*

References

1. A. Alhazov, B. Aman, R. Freund: P systems with anti-matter. In: [9], 66–85.
2. A. Alhazov, C. Ciubotaru, Yu. Rogozhin, S. Ivanov: The family of languages generated by non-cooperative membrane systems. In: Gh. Păun, M.J. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, A. Salomaa: *Membrane Computing, International Conference*, CMC11, Jena, Lecture Notes in Computer Science **6501**, 2011, 65–79.
3. R. Alur, P. Madhusudan: Visibly pushdown languages. In: L. Babai (Ed.): *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, Chicago, IL, USA, June 13-16, 2004, 202–211, ACM, 2004.
4. E. Csuhaj-Varjú, Gy. Vaszil: P automata or purely communicating accepting P systems. In: Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron (Eds.): *Membrane Computing, International Workshop, WMC-CdeA 2002, Curtea de Argeş, Romania, August 19–23, 2002, Revised Papers*. Lecture Notes in Computer Science **2597**, Springer, 2003, 219–233.
5. R. Freund, L. Kari, M. Oswald, P. Sosík: Computationally universal P systems without priorities: two catalysts are sufficient. *Theoretical Computer Science* **330** (2), 251–266 (2005).
6. R. Freund, A. Leporati, G. Mauri, A. E. Porreca, S. Verlan, C. Zandron: Flattening in (tissue) P systems. In: A. Alhazov, S. Cojocaru, M. Gheorghe, Yu. Rogozhin, G. Rozenberg, A. Salomaa (Eds.): Membrane Computing - 14th International Conference, CMC 2013, Chişinău, Republic of Moldova, August 20-23, 2013, Revised Selected Papers, Lecture Notes in Computer Science **8340**, Springer, 2014, 173–188.
7. R. Freund, I. Pérez-Hurtado, A. Riscos-Núñez , S. Verlan: A formalization of membrane systems with dynamically evolving structures. *International Journal of Computer Mathematics* **90** (4) (2013), 801-815.
8. R. Freund, M. Oswald: A short note on analysing P systems. *Bulletin of the EATCS* **78**, 2002, 231–236.
9. M. Gheorghe, G. Rozenberg, A. Salomaa, P. Sosík, C. Zandron: 15th International Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers. Lecture Notes in Computer Science **8961**, Springer, 2014.
10. M. L. Minsky: *Computation: Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1967.
11. A. Okhotin, K. Salomaa: Complexity of input-driven pushdown automata. *SIGACT News* **45** (2), 47–67 (2014).

12. Gh. Păun: Computing with Membranes. *J. Comput. Syst. Sci.* **61**, 108–143 (2000); also see TUCS Report 208, 1998, `www.tucs.fi`.
13. Gh. Păun: Computing with Membranes. *Journal of Computer and System Sciences* **61** (1) (2000), 108–143 (and Turku Center for Computer Science-TUCS Report 208, November 1998, `www.tucs.fi`).
14. Gh. Păun: *Membrane Computing. An Introduction.* Springer, 2002.
15. Gh. Păun, G. Rozenberg, A. Salomaa (Eds.): *The Oxford Handbook of Membrane Computing.* Oxford University Press, 2010.
16. G. Rozenberg, A. Salomaa (Eds.): *Handbook of Formal Languages,* 3 volumes. Springer, 1997.
17. P. Sosík, J. Matyšek: Membrane computing: when communication is enough. In: C. Calude, M. Dinneen, F. Peper (Eds.): *Unconventional Models of Computation 2002*, Lecture Notes in Computer Science **2509**, Springer, 2002, 264–275.
18. The P Systems Website: `http://ppage.psystems.eu`.

Extended Spiking Neural P Systems with White Hole Rules

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

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

[2] Faculty of Informatics, Vienna University of Technology
Favoritenstr. 9, A-1040 Wien, Austria
E-mail: {rudi,marion}@emcc.at

[3] Université Paris Est, France
E-mail: {sergiu.ivanov,verlan}@u-pec.fr

Summary. We consider extended spiking neural P systems with the additional possibility of so-called "white hole rules", which send the complete contents of a neuron to other neurons, and we show how this extension of the original model allow for easy proofs of the computational completeness of this variant of extended spiking neural P systems using only one actor neuron. Using only such white hole rules, we can easily simulate special variants of Lindenmayer systems.

1 Introduction

Based on the biological background of neurons sending electrical impulses along axons to other neurons, several models were developed in the area of neural computation, e.g., see [15], [16], and [10]. In the area of P systems, the model of *spiking neural P systems* was introduced in [14]. Whereas the basic model of membrane systems reflects hierarchical membrane structures, the model of tissue P systems considers cells to be placed in the nodes of a graph. This variant was first considered in [23] and then further elaborated, for example, in [9] and [17]. In spiking neural P systems, the cells are arranged as in tissue P systems, but the contents of a cell (neuron) consists of a number of so-called *spikes*, i.e., of a multiset over a single object. The rules assigned to a cell allow us to send information to other neurons in the form of electrical impulses (also called spikes) which are summed up at the target cell; the application of the rules depends on the contents of the neuron and in the general case is described by regular sets. As inspired from biology, the cell sending out spikes may be "closed" for a specific time period corresponding

to the refraction period of a neuron; during this refraction period, the neuron is closed for new input and cannot get excited ("fire") for spiking again.

The length of the axon may cause a time delay before a spike arrives at the target. Moreover, the spikes coming along different axons may cause effects of different magnitude. All these biologically motivated features were included in the model of extended spiking neural P systems considered in [1], the most important theoretical feature being that neurons can send spikes along the axons with different magnitudes at different moments of time. In this paper, we will further extend the model of extended spiking neural P systems by using so-called "white hole rules", which allow us to use the whole contents of a neuron and send it to other cells, yet eventually multiplied by some constant rational number.

In the literature, several variants how to obtain results from the computations of a spiking neural P system have been investigated. For example, in [14] the output of a spiking neural P system was considered to be the time between two spikes in a designated output cell. It was shown how spiking neural P systems in that way can generate any recursively enumerable set of natural numbers. Moreover, a characterization of semilinear sets was obtained by spiking neural P system with a bounded number of spikes in the neurons. These results can also be obtained with even more restricted forms of spiking neural P systems, e.g., no time delay (refraction period) is needed, as it was shown in [13]. In [4], the generation of strings (over the binary alphabet 0 and 1) by spiking neural P systems was investigated; due to the restrictions of the original model of spiking neural P systems, even specific finite languages cannot be generated, but on the other hand, regular languages can be represented as inverse-morphic images of languages generated by finite spiking neural P systems, and even recursively enumerable languages can be characterized as projections of inverse-morphic images of languages generated by spiking neural P systems. The problems occurring in the proofs are also caused by the quite restricted way the output is obtained from the output neuron as sequence of symbols 0 and 1. The strings of a regular or recursively enumerable language could be obtained directly by collecting the spikes sent by specific output neurons for each symbol.

In the extended model considered in [1], a specific output neuron was used for each symbol. Computational completeness could be obtained by simulating register machines as in the proofs elaborated in the papers mentioned above, yet in an easier way using only a bounded number of neurons. Moreover, regular languages could be characterized by finite extended spiking neural P systems; again, only a bounded number of neurons was needed.

In this paper, we now extend this model of extended spiking neural P systems by also using so-called "white hole rules", which may send the whole contents of a neuron along its axons, eventually even multiplied by a (positive) rational number. In that way, the whole contents of a neuron can be multiplied by a rational number, in fact, multiplied with or divided by a natural number. Hence, even one single neuron is able to simulate the computations of an arbitrary register machine.

The idea of consuming the whole contents of a neuron by white hole rules is closely related with concept of the exhaustive use of rules, i.e., an enabled rule is applied in the maximal way possible in one step; P systems with the exhaustive use of rules can be used in the usual maximally parallel way on the level of the whole system or in the sequential way, for example, see [27] and [26]. Yet all the approaches of spiking neural P systems with the exhaustive use of rules are mainly based on the classic definitions of spiking neural P systems, whereas the spiking neural P systems with white hole rules as investigated in this paper are based on the extended model as introduced in [1].

The rest of the paper is organized as follows: In the next section, we recall some preliminary notions and definitions from formal language theory, especially the definition and some well-known results for register machines. In section 3 we recall the definitions of the extended model of spiking neural P systems as considered in [1] as well as the most important results established there. Moreover, we show a new result for extended spiking neural P systems – such systems with only one actor neuron have exactly the same computational power as register machines with only one register that can be decremented.

In section 4, we define the model of extended spiking neural P systems extended by the use of white hole rules. Besides giving some examples, for instance showing how Lindenmayer systems can be simulated by extended spiking neural P systems only using white hole rules, we prove that the computations of an arbitrary register machine can be simulated by only one single neuron equipped with the most powerful variant of white hole rules. In that way we can show that extended spiking neural P systems equipped with white hole rules are even more powerful than extended spiking neural P systems, which need (at least) two neurons to be able to simulate the computations of an arbitrary register machine. Finally, in section 5 we give a short summary of the results obtained in this paper and discuss some future research topics for extended spiking neural P systems with white hole rules, for example, variants with inhibiting neurons or axons.

2 Preliminaries

In this section we recall the basic elements of formal language theory and especially the definitions and results for register machines; we here mainly follow the corresponding section from [1].

For the basic elements of formal language theory needed in the following, we refer to any monograph in this area, in particular, to [5] and [25]. We just list a few notions and notations: V^* is the free monoid generated by the alphabet V under the operation of concatenation and the empty string, denoted by λ, as unit element; for any $w \in V^*$, $|w|$ denotes the number of symbols in w (the *length* of w). $\mathbb{N}_+$ denotes the set of positive integers (natural numbers), $\mathbb{N}$ is the set of non-negative integers, i.e., $\mathbb{N} = \mathbb{N}_+ \cup \{0\}$, and $\mathbb{Z}$ is the set of integers, i.e., $\mathbb{Z} = \mathbb{N}_+ \cup \{0\} \cup -\mathbb{N}_+$. The interval of non-negative integers between k and m is

denoted by $[k..m]$, and $k \cdot \mathbb{N}_+$ denotes the set of positive multiples of k. Observe that there is a one-to-one correspondence between a set $M \subseteq \mathbb{N}$ and the one-letter language $L(M) = \{a^n \mid n \in M\}$; e.g., M is a regular (semilinear) set of non-negative integers if and only if $L(M)$ is a regular language. By $FIN\left(\mathbb{N}^k\right)$, $REG\left(\mathbb{N}^k\right)$, and $RE\left(\mathbb{N}^k\right)$, for any $k \in \mathbb{N}$, we denote the sets of subsets of $\mathbb{N}^k$ that are finite, regular, and recursively enumerable, respectively.

By REG ($REG(V)$) and RE ($RE(V)$) we denote the family of regular and recursively enumerable languages (over the alphabet V, respectively). By $\Psi_T(L)$ we denote the Parikh image of the language $L \subseteq T^*$, and by $PsFL$ we denote the set of Parikh images of languages from a given family FL. In that sense, $PsRE(V)$ for a k-letter alphabet V corresponds with the family of recursively enumerable sets of k-dimensional vectors of non-negative integers.

2.1 Register Machines

The proofs of the results establishing computational completeness in the area of P systems often are based on the simulation of register machines; we refer to [18] for original definitions, and to [7] for definitions like those we use in this paper:

An *n-register machine* is a construct $M = (n, P, l_0, l_h)$, where n is the number of registers, P is a finite set of instructions injectively labelled with elements from a given set $Lab(M)$, l_0 is the initial/start label, and l_h is the final label.

The instructions are of the following forms:

- $l_1 : (ADD(r), l_2, l_3)$ (ADD instruction)
  Add 1 to the contents of register r and proceed to one of the instructions (labelled with) l_2 and l_3.
- $l_1 : (SUB(r), l_2, l_3)$ (SUB instruction)
  If register r is not empty, then subtract 1 from its contents and go to instruction l_2, otherwise proceed to instruction l_3.
- $l_h : halt$ (HALT instruction)
  Stop the machine. The final label l_h is only assigned to this instruction.

A (non-deterministic) register machine M is said to generate a vector $(s_1, \cdots, s_\beta)$ of natural numbers if, starting with the instruction with label l_0 and all registers containing the number 0, the machine stops (it reaches the instruction $l_h : halt$) with the first β registers containing the numbers $s_1, \cdots, s_\beta$ (and all other registers being empty).

Without loss of generality, in the succeeding proofs we will assume that in each ADD instruction $l_1 : (ADD(r), l_2, l_3)$ and in each SUB instruction $l_1 : (SUB(r), l_2, l_3)$ the labels l_1, l_2, l_3 are mutually distinct (for a short proof see [9]).

The register machines are known to be computationally complete, equal in power to (non-deterministic) Turing machines: they generate exactly the sets of vectors of non-negative integers which can be generated by Turing machines, i.e., the family $PsRE$.

Based on the results established in [18], the results proved in [7] and [8] immediately lead to the following result:

Proposition 1. *For any recursively enumerable set $L \subseteq \mathbb{N}^\beta$ of vectors of non-negative integers there exists a non-deterministic $(\beta+2)$-register machine M generating L in such a way that, when starting with all registers 1 to $\beta+2$ being empty, M non-deterministically computes and halts with n_i in registers i, $1 \leq i \leq \beta$, and registers $\beta+1$ and $\beta+2$ being empty if and only if $(n_1, ..., n_\beta) \in L$. Moreover, the registers 1 to β are never decremented.*

When considering the generation of languages, we can use the model of a *register machine with output tape*, which also uses a tape operation:

– $l_1 : (write\,(a)\,, l_2)$
  Write symbol a on the output tape and go to instruction l_2.

We then also specify the output alphabet T in the description of the register machine with output tape, i.e., we write $M = (n, T, P, l_0, l_h)$.

The following result is folklore, too (e.g., see [18] and [6]):

Proposition 2. *Let $L \subseteq T^*$ be a recursively enumerable language. Then L can be generated by a register machine with output tape with 2 registers. Moreover, at the beginning and at the end of a successful computation generating a string $w \in L$ both registers are empty, and finally, only successful computations halt.*

3 Extended Spiking Neural P Systems

The reader is supposed to be familiar with basic elements of membrane computing, e.g., from [21] and [24]; comprehensive information can be found on the P systems web page [28]. Moreover, for the motivation and the biological background of spiking neural P systems we refer the reader to [14]. The definition of an *extended spiking neural P system* is mainly taken from [1], with the number of spikes k still be given in the "classical" way as a^k; later on, we simple will use the number k itself only instead of a^k.

3.1 The Definition of ESNP Systems

The definitions given in the following are taken from [1].

Definition 1. *An* extended spiking neural P system *(of degree $m \geq 1$) (in the following we shall simply speak of an* ESNP system*) is a construct*

$$\Pi = (m, S, R)$$

where

- *m is the number of* cells *(or* neurons*); the neurons are uniquely identified by a number between* 1 *and m (obviously, we could instead use an alphabet with m symbols to identify the neurons);*
- *S describes the* initial configuration *by assigning an initial value (of spikes) to each neuron; for the sake of simplicity, we assume that at the beginning of a computation we have no pending packages along the axons between the neurons;*
- *R is a finite set of* rules *of the form $\left(i, E/a^k \to P; d\right)$ such that $i \in [1..m]$ (specifying that this rule is assigned to cell i), $E \subseteq REG\left(\{a\}\right)$ is the* checking set *(the current number of spikes in the neuron has to be from E if this rule shall be executed), $k \in \mathbb{N}$ is the "number of spikes" (the energy) consumed by this rule, d is the* delay *(the "refraction time" when neuron i performs this rule), and P is a (possibly empty) set of* productions *of the form (l, w, t) where $l \in [1..m]$ (thus specifying the target cell), $w \in \{a\}^*$ is the* weight *of the energy sent along the axon from neuron i to neuron l, and t is the time needed before the information sent from neuron i arrives at neuron l (i.e., the* delay along the axon*). If the checking sets in all rules are finite, then Π is called a* finite ESNP system.

Definition 2. *A* configuration *of the ESNP system is described as follows:*

- *for each neuron, the actual number of spikes in the neuron is specified;*
- *in each neuron i, we may find an "activated rule" $\left(i, E/a^k \to P; d'\right)$ waiting to be executed where d' is the remaining time until the neuron spikes;*
- *in each axon to a neuron l, we may find pending packages of the form (l, w, t') where t' is the remaining time until $|w|$ spikes have to be added to neuron l provided it is not closed for input at the time this package arrives.*

A transition *from one configuration to another one now works as follows:*

- *for each neuron i, we first check whether we find an "activated rule" $\left(i, E/a^k \to P; d'\right)$ waiting to be executed; if $d' = 0$, then neuron i "spikes", i.e., for every production (l, w, t) occurring in the set P we put the corresponding package (l, w, t) on the axon from neuron i to neuron l, and after that, we eliminate this "activated rule" $\left(i, E/a^k \to P; d'\right)$;*
- *for each neuron l, we now consider all packages (l, w, t') on axons leading to neuron l; provided the neuron is not closed, i.e., if it does not carry an activated rule $\left(i, E/a^k \to P; d'\right)$ with $d' > 0$, we then sum up all weights w in such packages where $t' = 0$ and add this sum of spikes to the corresponding number of spikes in neuron l; in any case, the packages with $t' = 0$ are eliminated from the axons, whereas for all packages with $t' > 0$, we decrement t' by one;*
- *for each neuron i, we now again check whether we find an "activated rule" $\left(i, E/a^k \to P; d'\right)$ (with $d' > 0$) or not; if we have not found an "activated rule", we now may apply any rule $\left(i, E/a^k \to P; d\right)$ from R for which the current number of spikes in the neuron is in E and then put a copy of this rule as "activated rule" for this neuron into the description of the current configuration; on the other hand, if there still has been an "activated rule" $\left(i, E/a^k \to P; d'\right)$ in the*

neuron with $d' > 0$, then we replace d' by $d' - 1$ and keep $\big(i, E/a^k \to P; d' - 1\big)$ as the "activated rule" in neuron i in the description of the configuration for the next step of the computation.

After having executed all the substeps described above in the correct sequence, we obtain the description of the new configuration. A computation *is a sequence of configurations starting with the initial configuration given by S. A computation is called* successful *if it halts, i.e., if no pending package can be found along any axon, no neuron contains an activated rule, and for no neuron, a rule can be activated.*

In the original model introduced in [14], in the productions (l, w, t) of a rule $\big(i, E/a^k \to \{(l, w, t)\}; d\big)$, only $w = a$ (for *spiking rules*) or $w = \lambda$ (for *forgetting rules*) as well as $t = 0$ was allowed (and for forgetting rules, the checking set E had to be finite and disjoint from all other sets E in rules assigned to neuron i). Moreover, reflexive axons, i.e., leading from neuron i to neuron i, were not allowed, hence, for (l, w, t) being a production in a rule $\big(i, E/a^k \to P; d\big)$ for neuron i, $l \neq i$ was required. Yet the most important extension is that different rules for neuron i may affect different axons leaving from it whereas in the original model the structure of the axons (called synapses there) was fixed. In [1], the sequence of substeps leading from one configuration to the next one together with the interpretation of the rules from R was taken in such a way that the original model can be interpreted in a consistent way within the extended model introduced in that paper. As mentioned in [1], from a mathematical point of view, another interpretation would have been even more suitable: whenever a rule $\big(i, E/a^k \to P; d\big)$ is activated, the packages induced by the productions (l, w, t) in the set P of a rule $\big(i, E/a^k \to P; d\big)$ activated in a computation step are immediately put on the axon from neuron i to neuron l, whereas the delay d only indicates the refraction time for neuron i itself, i.e., the time period this neuron will be closed. The delay t in productions (l, w, t) can be used to replace the delay in the neurons themselves in many of the constructions elaborated, for example, in [14], [23], and [4]. Yet as in (the proofs of computational completeness given in) [1], we shall not need any of the delay features in this paper, hence we need not go into the details of these variants of interpreting the delays in more details.

Depending on the purpose the ESNP system is to be used, some more features have to be specified: for generating k-dimensional vectors of non-negative integers, we have to designate k neurons as *output neurons*; the other neurons then will also be called *actor neurons*. There are several possibilities to define how the output values are computed; according to [14], we can take the distance between the first two spikes in an output neuron to define its value. As in [1], also in this paper, we take the number of spikes at the end of a successful computation in the neuron as the output value. For generating strings, we do not interpret the spike train of a single output neuron as done, for example, in [4], but instead consider the sequence of spikes in the output neurons each of them corresponding to a specific terminal symbol; if more than one output neuron spikes, we take any permutation of the corresponding symbols as the next substring of the string to be generated.

Remark 1. As already mentioned, there is a one-to-one correspondence between (sets of) strings a^k over the one-letter alphabet $\{a\}$ and the corresponding non-negative integer k. Hence, in the following, we will consider the checking sets E of a rule $\left(i, E/a^k \to P; d\right)$ to be sets of non-negative integers and write k instead of a^k for any $w = a^k$ in a production (l, w, t) of P. Moreover, if no delays d or t are needed, we simply omit them. For example, instead of $\left(2, \left\{a^i\right\}/a^i \to \left\{(1, a, 0), \left(2, a^j, 0\right)\right\}; 0\right)$ we write $(2, \{i\}/i \to \{(1,1), (2,j)\})$.

3.2 ESNP Systems as Generating Devices

As in [1], we first consider extended spiking neural P systems as generating devices. The following example gives a characterization of regular sets of non-negative integers:

Example 1. Any semilinear set of non-negative integers M can be generated by a finite ESNP system with only two neurons.

Let M be a semilinear set of non-negative integers and consider a regular grammar G generating the language $L(G) \subseteq \{a\}^*$ with $N(L(G)) = M$; without loss of generality we assume the regular grammar to be of the form $G = (N, \{a\}, A_1, P)$ with the set of non-terminal symbols N, $N = \{A_i \mid 1 \leq i \leq m\}$, the start symbol A_1, and P the set of regular productions of the form $B \to aC$ with $B, C \in N$ and $A \to \lambda$. We now construct the finite ESNP system $\Pi = (2, S, R)$ that generates an element of M by the number of spikes contained in the output neuron 1 at the end of a halting computation: we start with one spike in neuron 2 (representing the start symbol A_1 and no spike in the output neuron 1, i.e., $S = \{(1,0), (2,1)\}$. The production $A_i \to aA_j$ is simulated by the rule $(2, \{i\}/i \to \{(1,1), (2,j)\})$ and $A_i \to \lambda$ is simulated by the rule $(2, \{i\}/i \to \emptyset)$, i.e., in sum we obtain

$$\begin{aligned}
\Pi &= (2, S, R), \\
S &= \{(1,0), (2,1)\}, \\
R &= \{(2, \{i\}/i \to \{(1,1), (2,j)\}) \mid 1 \leq i, j \leq m, A_i \to aA_j \in P\} \\
&\quad \cup \{(2, \{i\}/i \to \emptyset) \mid 1 \leq i \leq m, A_i \to \lambda \in P\}.
\end{aligned}$$

Neuron 2 keeps track of the actual non-terminal symbol and stops the derivation as soon as it simulates a production $A_i \to \lambda$, because finally neuron 2 is empty. In order to guarantee that this is the only way how we can obtain a halting computation in Π, without loss of generality we assume G to be reduced, i.e., for every non-terminal symbol A from N there is a regular production with A on the left-hand side. These observations prove that we have $N(L(G)) = M$.

The following results were proved in [1]:

Lemma 1. *For any ESNP system where during any computation only a bounded number of spikes occurs in the actor neurons, the generated language is regular.*

Theorem 1. *Any regular language L with $L \subseteq T^*$ for a terminal alphabet T with $card(T) = n$ can be generated by a finite ESNP system with $n+1$ neurons. On the other hand, every language generated by a finite ESNP system is regular.*

Corollary 1. *Any semilinear set of n-dimensional vectors can be generated by a finite ESNP system with $n+1$ neurons. On the other hand, every set of n-dimensional vectors generated by a finite ESNP system is semilinear.*

Theorem 2. *Any recursively enumerable language L with $L \subseteq T^*$ for a terminal alphabet T with $card(T) = n$ can be generated by an ESNP system with $n+2$ neurons.*

Corollary 2. *Any recursively enumerable set of n-dimensional vectors can be generated by an ESNP system with $n+2$ neurons.*

Besides these results already established in [1], we now prove a characterization of languages and sets of (vectors of) natural numbers generated by ESNPS with only one neuron. Roughly speaking, having only one actor neuron corresponds with, besides output registers, having only one register which can be decremented.

Lemma 2. *For any ESNP system with only one actor neuron we can effectively construct a register machine with output tape and only one register that can be decremented, generating the same language, respectively a register machine with one register that can be decremented, generating the same set of (vectors of) natural numbers.*

Proof. First we notice that the delays would not matter: the overall system is sequential, and therefore it is always possible to pre-compute what happens until the actor neuron re-opens; the weight of all pending packages is also bounded. All the details of storing and managing all these features by the finite control of the register machines are tedious, but very much straightforward. In the following, we therefore assume that the ESNPS is given as:

$$\begin{aligned}
&\Pi = (n+1, S, R),\\
&S = \{(1, m_1), \cdots, (n, m_n), (n+1, m_{n+1})\},\\
&R = \{(n+1, E_r/i_r \rightarrow \{(1, p_{r,1}), \cdots, (n, p_{r,n}), (n+1, p_{r,n+1})\}) \mid 1 \leq r \leq q\}.
\end{aligned}$$

Thus, given n, Π can be specified by the following non-negative integers: the number q of rules, initial spikes $m_1, \cdots, m_n, m_{n+1}$, and, for every rule r, the following ingredients: the number i_r of consumed spikes, the numbers $p_{r,1}, \cdots, p_{r,n+1}$ of produced spikes, and the regular sets E_r of numbers. Note that, as it will be obvious later, it is enough to only consider the case $m_1 = \cdots = m_n = 0$, because otherwise placing the initial spikes can be done by a 1-register machine in a preparatory phase, before switching to the instruction corresponding to starting the simulation.

The main challenge of the construction is to remember the actual "status" of the regular checking sets. It is known that every regular set E of numbers

is semilinear, and it is possible to write $E_r = \bigcup_{j=1}^{l_r}(k_r\mathbb{N} + d_{r,j}) \cup D_r$, i.e., all the linear sets constituting E_r can be reduced to a common period k_r, and an additional finite set. Then, we can take a common multiple k of periods k_r, and represent each checking set as $E_r = \left(k\mathbb{N}_+ + \{d'_{r,j} \mid 1 \le j \le l'_r\}\right) \cup D'_r$, where D'_r is finite.

Finally, take a number M such that M is a multiple of k, that M is larger than any element of D_r, $1 \le r \le q$, that M is larger than any number $d'_{r,j}$, $1 \le j \le l'_r$, $1 \le r \le q$, that M is larger than any of i_r and $p_{r,n+1}$, $1 \le r \le q$. Then, if neuron $n+1$ has N spikes, the following properties hold:

- rule r is applicable if and only if $N \in E_r$ in case when $i_r \le N < M$, and if and only if $M + (N \text{mod M}) \in \text{E}_\text{r}$ in case when $N \ge M$,
- the difference between the number of spikes in neuron $n+1$ in two successive configurations is not larger than M.

For neuron $n+1$, $Mk + j$ spikes (where $0 \le j \le M-1$) will be represented by value k of register 1 and state j.

We simulate Π by a register machine R with one register and an output tape of m symbols. Before we proceed, we need to remark that, without restricting the generality, we may have an arbitrary set of "next instructions" instead of $\{l_2, l_3\}$ in $l_1 : (ADD(r), l_2, l_3)$, and arbitrary sets of "next instructions" instead of $\{l_2\}$ and $\{l_3\}$ in $l_1 : (SUB(r), l_2, l_3)$. Indeed, non-determinism between choice of multiple instructions can be implemented by an increment followed by a decrement in each case, as many times as needed for the corresponding set of "next instructions". Clearly, $l_1 : (ADD(r), \{l_2\})$ is just a shorter form of $l_1 : (ADD(r), l_2, l_2)$.

Finally, besides instructions $ADD(r)$, $SUB(r)$, $write(a)$ and $halt$, we introduce the notation of NOP, meaning only a switch to a different instruction without modifying the register. This will greatly simplify the construction below, and such a notation can be reduced to either compressing the rules (by substituting the instruction label with the label of the next instruction in all other instructions), or be simulated by an $ADD(1)$ instruction, followed by a $SUB(1)$ instruction.

We take $b(m_{n+1} \text{mod M})$ as the starting state of R, and the starting value of register 1 is $m_{n+1} \text{div M}$.

For every class modulo M, $0 \le j \le M-1$, we define sets

$$L_{j,0} = \{l_{r,0} \mid 1 \le r \le q,\ j \in E_r,\ i_r \ge j\},$$
$$L_{j,+} = \{l_{r,+} \mid 1 \le r \le q,\ j + M \in E_r\}$$

of applicable rules corresponding to remainder j, subscripts 0 and + represent cases of having less than M spikes, and at least M spikes, respectively. Let us redefine any of these sets to $\{l_h\}$ if the expression above is empty.

We proceed with the actual simulation. A rule

$$(n+1, E_r/i_r \to \{(1, p_{r,1}), \cdots, (n, p_{r,n}), (n+1, p_{r,n+1})\})$$

can be simulated by the following rules of R:

$b(j) : (S(1), L_{j,+}, L_{j,0}),\ l_r \in L_{j,0}$;
$l_{r,\alpha} : \cdots,$ (a sequence of p_{r_1} instructions $write(a_1), \cdots,$)
$\cdots,$ (p_{r_n} instructions $write(a_n)$),
$\cdots l'_{r,\alpha}$, (and $p_{r_{n+1}}$ instructions $ADD(1)$), $\alpha \in \{0, +\}$;
$l'_{r,+} : (NOP, \{b((j - i_r + p_{r,n+1})\text{mod M})\}),$ if $\mathrm{j - i_r + p_{r,n+1} < 0}$;
$l'_{r,+} : (ADD(1), \{l'_{r,0}\}),$ if $j - i_r + p_{r,n+1} < M$;
$l'_{r,0} : (NOP, \{b((j - i_r + p_{r,n+1})\text{mod M})\}),$ if $\mathrm{j - i_r + p_{r,n+1} < M}$;
$l'_{r,0} : (ADD(1), \{b((j - i_r + p_{r,n+1})\text{mod M})\}),$ if $\mathrm{j - i_r + p_{r,n+1} \geq M}$;
$l_h : halt.$

Indeed, instruction $b(j)$ corresponds to checking whether neuron $n+1$ has at least M spikes, transitioning into the halting instruction, or into the set of instructions associated with the corresponding applicable rules, in the context of the result of the checking mentioned above. Sending spikes to output neurons is simulated by writing the corresponding symbols on the tape. This goal is obtained, knowing values j, i_r, $p_{r,n+1}$, and whether neuron 1 had at least M spikes or not, by transitioning to instruction $b((j - i_r + p_{r,n+1})\text{mod M})$ after incrementing register 1 the needed number of times (0, 1 or 2), which is equal to $(j - i_r + p_{r,n+1}\text{div M}) + d$, where $d = 0$ if neuron 1 had at least M spikes, and $d = 1$ otherwise (to compensate for the subtraction done by instruction $b(j)$ in the initial checking). The simulation of instructions continues until we reach the situation where no rules of the underlying spiking system are applicable, transitioning to some $L_{j,\alpha} = \{l_h\}$.

Finally, let us formally describe the instruction sequences from $l_{r,\alpha}$ to $l'_{r,\alpha}$. For the sake of simplicity of notation, we do not mention subscripts r, α in the notation of the intermediate instructions, keeping in mind that these are different instructions for different r, α. The difficulty for generating the string languages is that, by the definition, all permutations are to be considered if spikes are sent to multiple neurons $1, \cdots, m$.

$l_{r,\alpha} : (NOP, \{s(p_{r_1}, \cdots, p_{r_n})\});$
$s(i_1, \cdots, i_n) : (NOP, \{s^k(i_1, \cdots, i_n) \mid i_k > 0,\ 1 \leq k \leq n\}),$
$0 \leq i_j \leq p_{r_j},\ 1 \leq j \leq n, (i_1, \cdots, i_n) \neq (0, \cdots, 0);$
$s^{(k)}(i_1, \cdots, i_n) : (write(a_k), \{s(i'_1, \cdots, i'_n)\}),$
$i'_k = i_k - 1,$ and $i'_j = i_j,\ 1 \leq j \leq n,\ j \neq k,$
$0 \leq i_j \leq p_{r_j},\ 1 \leq j \leq n, (i_1, \cdots, i_n) \neq (0, \cdots, 0);$
$s(0, \cdots, 0) : (NOP, \{t(p_{r_{n+1}})\});$
$t(i) : (ADD(n+1), t(i-1)),\ 1 \leq i \leq p_{r_{n+1}};$
$t(0) : (NOP, l'_{r,\alpha}).$

The rules above describe precisely the following behavior: to produce any sequence with the desired numbers of occurrences of symbols $a_1, \cdots, a_n$, a symbol is non-

deterministically chosen (out of those, the current desired number of occurrences of which is positive) and written, iterating until all desired symbols are written.

Next, the register is incremented the needed number of times. This finishes the explanation of the instruction sequences from $l_{r,\alpha}$ to $l'_{r,\alpha}$, as well as the explanation of the simulation.

Therefore, the class of languages generated by ESNP systems with only one neuron containing rules and n output neurons is included in the class of languages generated by 1-register machines with an output tape of n symbols.

Applying Parikh mapping to both classes, just replacing *write*-instructions by *ADD*-instructions on new registers associated with these symbols, it follows that the class of sets of vectors generated by ESNP systems with only one neuron containing rules and n output neurons is included in the class of sets of vectors generated by $n+1$-register machines where all registers except one are restricted to be increment-only. These observations conclude the proof. □

The inclusions formulated at the end of the proof given above are actually characterizations, as we can also prove the opposite inclusion.

Lemma 3. *For any register machine with output tape with only one register that can be decremented respectively for any register machine with only one register that can be decremented we can effectively construct an ESNP system generating the same language respectively the same set of (vectors of) natural numbers.*

Proof. By definition, output registers can only be incremented, so the main computational power lies in the register which can also be decremented. The decrementable register can be simulated together with storing the actual state by storing the number $dn + c_i$ where: n is the actual contents of the register, c_i is a number encoding the i-th instruction of the register machine, and d is a number bigger than all c_i. Then incrementing this first register by an instruction c_i and jumping to c_j means consuming c_i and adding $d + c_j$ in the actor neuron, provided the checking set guarantees that the actual contents is an element of $d\mathbb{N} + c_i$. Decrementing means consuming $d + c_i$ and adding c_j in the actor neuron, provided the checking set guarantees that the actual contents is an element of $d\mathbb{N}_+ + c_i$; if $n = 0$, then c_i is consumed and c_k is added in the actor neuron with c_k being the instruction to continue in the zero case. At the same time, with each of these simulation steps, the output neurons can be incremented in the exact way as the output registers; in the case of register machines with output tape, a spike is sent to the output neuron representing the symbol to be written. Further details of this construction are left to the reader. □

4 ESNP Systems with White Hole Rules

In this section, we extend the model of extended spiking neural P systems, introduced in [1] and described in the previous section, by white hole rules. We

will show that with this new variant of extended spiking neural P systems, computational completeness can already be obtained with only one actor neuron, by proving that the computations of any register machines can already be simulated in only one neuron equipped with the most general variant of white hole rules. Using this single actor neuron to also extract the final result of a computation, we even obtain weak universality with only one neuron.

As already mentioned in Remark 1, we are going to describe the checking sets and the number of spikes by non-negative integers. The following definition is an extension of Definition 1:

Definition 3. *An* extended spiking neural P system with white hole rules *(of degree $m \geq 1$) (in the following we shall simply speak of an* EESNP system*) is a construct*

$$\Pi = (m, S, R)$$

where

- *m is the number of* cells *(or* neurons*); the neurons are uniquely identified by a number between* 1 *and m;*
- *S describes the* initial configuration *by assigning an initial value (of spikes) to each neuron;*
- *R is a finite set of* rules *either being a* white hole rule *or a rule of the form as already described in Definition 3 $(i, E/k \to P; d)$ such that $i \in [1..m]$ (specifying that this rule is assigned to cell i), $E \subseteq REG(\mathbb{N})$ is the* checking set *(the current number of spikes in the neuron has to be from E if this rule shall be executed), $k \in \mathbb{N}$ is the "number of spikes" (the energy) consumed by this rule, d is the* delay *(the "refraction time" when neuron i performs this rule), and P is a (possibly empty) set of* productions *of the form (l, w, t) where $l \in [1..m]$ (thus specifying the target cell), $w \in \mathbb{N}$ is the* weight *of the energy sent along the axon from neuron i to neuron l, and t is the time needed before the information sent from neuron i arrives at neuron l (i.e., the* delay along the axon*). A* white hole rule *is of the form $(i, E/all \to P; d)$ where all means that the whole contents of the neuron is taken out of the neuron; in the productions (l, w, t), either $w \in \mathbb{N}$ as before or else $w = (all + p) \cdot q + z$ with $p, q, z \in \mathbb{Q}$; provided $(c + p) \cdot q + z$, where c denotes the contents of the neuron, is non-negative, then $\lfloor (c + p) \cdot q + z \rfloor$ is the number of spikes put on the axon to neuron l.*
  If the checking sets in all rules are finite, then Π is called a finite EESNP system.

Allowing the white hole rules having productions being of the form $w = (all + p) \cdot q + z$ with $p, q, z \in \mathbb{Q}$ is a very general variant, which can be restricted in many ways, for example, by taking $z \in \mathbb{Z}$ or omitting any of the rational numbers $p, q, z \in \mathbb{Q}$ or demanding them to be in $\mathbb{N}$ etc.

Obviously, every ESNPS also is an EESNPS, but without white hole rules, and a finite EESNPS also is a finite ESNPS, as in this case the effect of white hole rules

is also bounded, i.e., even with allowing the use of white hole rules, the following lemma as a counterpart of Lemma 1 is still valid:

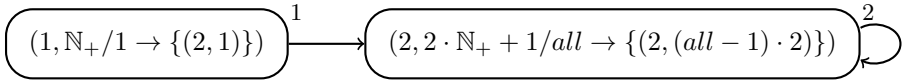
Lemma 4. *For any EESNP system where during any computation only a bounded number of spikes occurs in the actor neurons, the generated language is regular.*

Hence, in the following our main interest is in EESNPS which really make use of the whole power of white hole rules.

4.1 Examples for EESNPS

EESNPS can also be used for computing functions, not only for generating sets of (vectors of) integer numbers. As a simple example, we show how the function $n \mapsto 2^{n+1}$ can be computed by a deterministic EESPNS, which only has exactly one rule in each of its two neurons; the output neuron 2 in this case is not free of rules.

Example 2. Computing $n \mapsto 2^{n+1}$

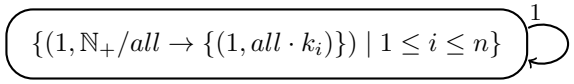


Initial value $= n$ Initial value $= 2$

The rule $(2, 2 \cdot \mathbb{N}_+ + 1/all \to \{(2, (all-1) \cdot 2)\})$ could also be written as $(2, 2 \cdot \mathbb{N}_+ + 1/all \to \{(2, (all) \cdot 2 - 2)\})$. In both cases, starting with the input number n (of spikes) in neuron 1, with each decrement in neuron 1, the contents of neuron 2 (not taking into account the enabling spike from neuron 1) is doubled. The computation stops with 2^{n+1} in neuron 1, as with 0 in neuron 1 no enabling spike is sent to neuron 2 any more, hence, the firing condition is not fulfilled any more. We finally remark that with the initial value 1 in neuron 2 we can compute the function $n \mapsto 2^n$.

Example 3. Pure White Hole Model of EESNPS for DT0L Systems

Let $G = (\{a\}, P, a^s)$ be a Lindenmayer system with the axiom a^s and the finite set of tables P each containing a finite set of parallel productions of the form $a \to a^k$. Such a system is called a tabled Lindenmayer system, abbreviated $T0L$ system, and it is called deterministic, abbreviated $DT0L$ system, if each table contains exactly one rule. Now let $G = (\{a\}, P, a^s)$ be a $DT0L$ system with $P = \{\{a \to a^{k_i}\} \mid 1 \leq i \leq n\}$. Then the following EESNPS using only white hole rules computes the same set of natural numbers as are represented by the language generated by G, with the results being taken with *unconditional halting*, i.e., taking a result at every moment (see [2]).



Initial value $= s$

If we want to generate with normal halting, we have to add an additional output neuron 2 and an additional rule $\{(1, \mathbb{N}_+/all \to \{(2, all \cdot 1)\})\}$ in neuron 1 which at the end moves the contents of neuron 1 to neuron 2.

4.2 Universality with EESNPS

Lemma 5. *The computation of any register machine can be simulated in only one single actor neuron of an EESPNS.*

Proof. Let $M = (n, P, l_0, l_h)$ be an n-register machine, where n is the number of registers, P is a finite set of instructions injectively labelled with elements from a set of labels $Lab(M)$, l_0 is the initial label, and l_h is the final label.

Then we can effectively construct an EESNPS $\Pi = (m, S, R)$ simulating the computations of M by encoding the contents n_i of each register i, $1 \leq i \leq n$, as $p_i^{n_i}$ for different prime numbers p_i. Moreover, for each instruction (label) j we take a prime number q_j, of course, also each of them being different from each other and from the p_i.

The instructions are simulated as follows:

- $l_1 : (ADD(r), l_2, l_3)$ (ADD instruction)
  This instruction can be simulated by the rules
  $\{(1, q_{l_1} \cdot \mathbb{N}_+/all \to \{(1, all \cdot q_{l_i} p_r/q_{l_1})\}) \mid 2 \leq i \leq 3\}$
  in neuron 1.
- $l_1 : (SUB(r), l_2, l_3)$ (SUB instruction)
  This instruction can be simulated by the rules
  $(1, q_{l_1} p_r \cdot \mathbb{N}_+/all \to \{(1, all \cdot q_{l_2}/(q_{l_1} p_r))\})$
  and
  $(1, q_{l_1} \cdot \mathbb{N}_+ \setminus q_{l_1} p_r \cdot \mathbb{N}_+/all \to \{(1, all \cdot q_{l_2}/q_{l_1})\})$
  in neuron 1; the first rule simulates the decrement case, the second one the zero test.
- $l_h : halt$ (HALT instruction)
  This instruction can be simulated by the rule
  $(1, q_{l_h} \cdot \mathbb{N}_+/all \to \{(1, all \cdot 1/q_{l_h})\})$
  in neuron 1.
  In fact, after the application of the last rule, we end up with $p_1^{m_1} \cdots p_n^{m_n}$ in neuron 1, where $(m_1, \cdots, m_n)$ is the vector computed by M and now, in the prime number encoding, by Π as well.

All the checking sets we use are regular, and the productions in all the white hole rules even again yield integer numbers. □

Remark 2. As the productions in all the white hole rules of the EESNPS constructed in the preceding proof even again yield integer numbers, we could also interpret this EESNPS as an ESPNS with exhaustive use of rules:

The white hole rules in the EESNPS constructed in the previous proof are of the general form

$(1, q \cdot \mathbb{N}_+ / all \to \{(1, all \cdot p/q)\})$
with p and q being natural numbers. Each of these rules can be simulated in a one-to-one manner by the rule
$(1, q \cdot \mathbb{N}_+ / q \to p)$
used in an ESNPS with one neuron in the exhaustive way.

Theorem 3. *Any recursively enumerable set of n-dimensional vectors can be generated by an ESNP system with $n+1$ neurons.*

Proof. We only have to show how to extract the results into the additional output neurons from the single actor neuron which can do the whole computational task as exhibited in Lemma 5. Yet this is pretty easy:

When the actor neuron reaches the halting state, the desired result m_i for output neuron $i+1$ is stored as factor in this one number stored in the actor neuron within the prime number encoding, i.e., as $q_i^{m_i}$, for $1 \leq i \leq n$. Instead of using the final rule $(1, q_{l_h} \cdot \mathbb{N}_+ / all \to \{(1, all \cdot 1/q_{l_h})\})$ in neuron 1 we now take the rule $(1, q_{l_h} \cdot \mathbb{N}_+ / all \to \{(1, all \cdot r_1/q_{l_h})\})$.

With the rules $(1, r_i q_i \mathbb{N}_+ / all \to \{(1, all \cdot 1/k_i), (i+1, 1)\})$, we can decode the factor $q_i^{m_i}$ to m_i into output neuron $i+1$, with the instruction code (prime number) r_i for $1 \leq i \leq n$. If the contents of the actor neuron is not dividable by q_i any more, we switch to the next instruction code r_{i+1} by the rule $(1, r_i \cdot \mathbb{N}_+ \setminus r_i q_i \cdot \mathbb{N}_+ / all \to \{(1, all \cdot r_{i+1}/r_i)\})$. At the end, we can end up with 0 in the actor neuron after having used the rule $(1, r_i \cdot \mathbb{N}_+ \setminus r_i q_i \cdot \mathbb{N}_+ / all \to \emptyset)$ and then stop with m_i in output neuron $i+1$, $1 \leq i \leq n$. □

Theorem 4. *Any recursively enumerable language L with $L \subseteq T^*$ for a terminal alphabet T with $card(T) = n$ can be generated by an ESNP system with $n+1$ neurons.*

Proof. In the case of generating strings, we have to simulate a register machine with output tape; hence, in addition to the simulating rules already described in Lemma 5, we have to simulate the tape rule $l_1 : (write(a), l_2)$, which in the EESNPS means sending one spike to the output neuron $N(a)$ representing the symbol a. This task is accomplished by the rule $(1, l_1 \cdot \mathbb{N}_+ / all \to \{(1, all \cdot l_2/l_1), (N(a), 1)\})$. The rest of the construction and of the proof is similar to that what we have done in the proof of Lemma 5. □

5 Summary and Further Variants

In this paper, we have extended the model of extended spiking neural P systems from [1] by white hole rules. With this new variant of extended spiking neural P systems, computational completeness can already be obtained with only one actor neuron, as the computations of any register machine can already be simulated in only one neuron equipped with the most general variant of white hole rules. Using

this single actor neuron to also extract the final result of a computation, we even obtain weak universality with only one neuron.

A quite natural feature found in biology and also already used in the area of spiking neural P systems is that of inhibiting neurons or axons between neurons, i.e., certain connections from one neuron to another one can be specified as inhibiting ones – the spikes coming along such inhibiting axons then close the target neuron for a time period given by the sum of all inhibiting spikes, e.g., see [3]. Such variants can also be considered for extended spiking neural P systems with white hole rules.

References

1. A. Alhazov, R. Freund, M. Oswald, M. Slavkovik: Extended spiking neural P systems. In: [11], 123–134.
2. M. Beyreder, R. Freund: Membrane systems using noncooperative rules with unconditional halting. In: D. W. Corne, P. Frisco, Gh. Păun, G. Rozenberg, A. Salomaa (Eds.): Membrane Computing. 9th International Workshop, WMC 2008, Edinburgh, UK, July 28-31, 2008, Revised Selected and Invited Papers. Lecture Notes in Computer Science *5391*, Springer, 2009, 129–136.
3. A. Binder, R. Freund, M. Oswald, L. Vock: Extended spiking neural P systems with excitatory and inhibitory astrocytes. In: M.A. Gutiérrez-Naranjo, Gh. Păun, A. Romero-Jiménez, A. Riscos-Núñez (Eds.): *Fifth Brainstorming Week on Membrane Computing*, RGNC REPORT 01/2007, Research Group on Natural Computing, Sevilla University, Fénix Editora, 2007, 63–72.
4. H. Chen, R. Freund, M. Ionescu, Gh. Păun, M.J. Pérez-Jiménez: On string languages generated by spiking neural P systems. In: [11], 169–194
5. J. Dassow, Gh. Păun: *Regulated Rewriting in Formal Language Theory*. Springer, 1989.
6. H. Fernau, R. Freund, M. Oswald, K. Reinhardt: Refining the nonterminal complexity of graph-controlled, programmed, and matrix grammars. *Journal of Automata, Languages and Combinatorics*, **12** (1-2) (2007), 117–138.
7. R. Freund, M. Oswald: P systems with activated/prohibited membrane channels. In: Gh. Păun , G. Rozenberg, A. Salomaa, C. Zandron (Eds.): *Membrane Computing. International Workshop WMC 2002*, Curtea de Argeş, Romania. Lecture Notes in Computer Science **2597**, Springer, Berlin, 2002, 261–268.
8. R. Freund, Gh. Păun: From regulated rewriting to computing with membranes: collapsing hierarchies. *Theoretical Computer Science* **312**, 143–188.
9. R. Freund, Gh. Păun, M.J. Pérez-Jiménez: Tissue-like P systems with channel states. *Theoretical Computer Science* **330** (2004), 101–116.
10. W. Gerstner, W. Kistler: *Spiking Neuron Models. Single Neurons, Populations, Plasticity*. Cambridge Univ. Press, 2002.
11. M.A. Gutiérrez-Naranjo, Gh. Păun, A. Riscos-Núñez, F.J. Romero-Campero (Eds.): *Fourth Brainstorming Week on Membrane Computing*, Vol. I RGNC REPORT 02/2006, Research Group on Natural Computing, Sevilla University, Fénix Editora, 2006.

12. M.A. Gutiérrez-Naranjo, Gh. Păun, A. Riscos-Núñez, F.J. Romero-Campero (Eds.): *Fourth Brainstorming Week on Membrane Computing*, Vol. II RGNC REPORT 02/2006, Research Group on Natural Computing, Sevilla University, Fénix Editora, 2006.
13. O.H. Ibarra, A. Păun, Gh. Păun Gh, A. Rodríguez-Patón, P. Sosík, S. Woodworth: Normal forms for spiking neural P systems. In: [12], 105–136, 2006.
14. M. Ionescu, Gh. Păun, T. Yokomori: Spiking neural P systems. *Fundamenta Informaticae* **71** (2 –3) (2006), 279–308.
15. W. Maass: Computing with spikes. *Special Issue on Foundations of Information Processing of TELEMATIK* **8** (1) (2002), 32–36.
16. W. Maass, C. Bishop (Eds.): *Pulsed Neural Networks*. MIT Press, Cambridge, 1999.
17. Martín-Vide C, Pazos J, Păun Gh, Rodríguez-Patón A (2002) A new class of symbolic abstract neural nets: Tissue P systems. In: *Proceedings of COCOON 2002*, Singapore, Lecture Notes in Computer Science **2387**, Springer-Verlag, Berlin, 290–299
18. M. L. Minsky: *Computation: Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1967.
19. L. Pan, Gh. Păun: Spiking Neural P Systems with Anti-Matter. *International Journal of Computers, Communications & Control* **4** (3), 273–282 (2009).
20. Gh. Păun: Computing with Membranes. *Journal of Computer and System Sciences* **61** (1) (2000), 108–143 (and Turku Center for Computer Science-TUCS Report 208, November 1998, `www.tucs.fi`).
21. Gh. Păun: *Membrane Computing. An Introduction.* Springer, 2002.
22. Păun Gh, M.J. Pérez-Jiménez, G. Rozenberg: Spike trains in spiking neural P systems. *Intern. J. Found. Computer Sci.* **17** (2006), 975–1002.
23. Gh. Păun, Y. Sakakibara, T. Yokomori: P systems on graphs of restricted forms. *Publicationes Mathematicae Debrecen* **60** (2006), 635–660.
24. Gh. Păun, G. Rozenberg, A. Salomaa (Eds.): *The Oxford Handbook of Membrane Computing.* Oxford University Press, 2010.
25. G. Rozenberg, A. Salomaa (Eds.): *Handbook of Formal Languages,* 3 volumes. Springer, 1997.
26. X. Zhang, B. Luo, X. Fang, L. Pan: Sequential spiking neural P systems with exhaustive use of rules. *BioSystems* **108** (2012), 52–62.
27. X. Zhang, X. Zeng, L. Pan: On String Languages Generated by Spiking Neural P Systems with Exhaustive Use of Rules. *Natural computing* **7** (4) (2002), 535–549.
28. The P Systems Website: `www.ppage.psystems.eu`.

Simulating Membrane Systems and Dissolution in a Typed Chemical Calculus

Bogdan Aman[1], Péter Battyányi[2], Gabriel Ciobanu[1], and György Vaszil[2]

[1] Romanian Academy, Institute of Computer Science
Blvd. Carol I no.8, 700505 Iaşi, Romania
baman@iit.tuiasi.ro
gabriel@info.uaic.ro

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

Summary. We present a transformation of membrane systems, possibly with promoter/inhibitor rules, priority relations, and membrane dissolution, into formulas of the chemical calculus such that terminating computations of membranes correspond to terminating reduction sequences of formulas and vice versa. In the end, the same result can be extracted from the underlying computation of the membrane system as from the reduction sequence of the chemical term. The simulation takes place in a typed chemical calculus, but we also give a short account of the untyped case.

1 Introduction

In the present paper we continue the investigations started in [2, 3] concerning the possibility of defining the semantics of membrane systems with rewriting logic [1, 2] in order to obtain a logical description of membrane system computations.

The direct precedent of our work is [7] where a logical description of simple membrane systems was given using the γ-calculus of Banâtre and Le Métayer from [6] (see also [4] for more details). Their aim was to free the expression of algorithms from the sequentiality which is not inherently present in the problem to be solved, that is, the sequentiality which is implied by the structure of the computational model on which the given algorithm is to be performed. They called their calculus chemical calculus, and the underlying computational paradigm the chemical paradigm of computation while the execution model behind them closely resembles the way chemical reactions take place in chemical solutions. A chemical "machine" can be thought of as a symbolic chemical solution where data can be seen as molecules and operations as chemical reactions. If some molecules satisfy a reaction condition, they are replaced by the result of the reaction. If no reaction is possible, the program terminates. Chemical solutions are represented by multisets. Molecules interact freely according to reaction rules which results in an implicitly parallel, non-deterministic, distributed model.

In what follows, using a slightly modified variant of the operational semantics of membrane systems presented in [3], we show how to transform a membrane system

with rules using promoters/inhibitors (see [8]), priorities, and also the possibility of membrane dissolution (introduced already in [9]), into formulas of the chemical calculus, such that terminating computations of the membrane system correspond to terminating reduction sequences of formulas and vice versa.

2 Preliminaries

In this section we present the basic notions and notations we are going to use. For a comprehensive treatment of membrane systems ranging from the basic definitions to their computational power, see the the monographs [10, 11], for more information on the chemical calculus, we refer to [4, 5].

A finite multiset over an alphabet V is a mapping $m : V \to \mathbb{N}$ where $\mathbb{N}$ denotes the set of non-negative integers, and $m(a)$ for $a \in V$ is said to be the multiplicity of a in V. We say that $m_1 \subseteq m_2$ if for all $a \in V$, $m_1(a) \leq m_2(a)$. The union or sum of two multisets over V is defined as $(m_1 + m_2)(a) = m_1(a) + m_2(a)$, the difference is defined for $m_2 \subseteq m_1$ as $(m_1 - m_2)(a) = m_1(a) - m_2(a)$ for all $a \in V$. The multiset m can also be represented by any permutation of a string $w = a_1^{m(a_1)} a_2^{m(a_2)} \dots a_n^{m(a_n)} \in V^*$, where if $m(x) \neq 0$, then there exists j, $1 \leq j \leq n$, such that $x = a_j$. The set of all finite multisets over an alphabet V is denoted by $\mathcal{M}(V)$, the empty multiset is denoted by $\emptyset$ as in the case of the empty set.

2.1 Membrane systems

A membrane system, or P system is a structure of hierarchically embedded membranes, each having a label and enclosing a region containing a multiset of objects and possibly other membranes. The unique out-most membrane is called the skin membrane. The membrane structure is denoted by a sequence of matching parentheses where the matching pairs have the same label as the membranes they represent. We assume the membranes are labelled by natural numbers $\{1, \dots, n\}$, and we use the notation m_i for the membrane with label i. Each membrane m_i, except for the skin membrane, has its parent membrane, which we denote by $\mu(m_i)$. As an abuse of notation μ stands for both the membrane structure and both for the function determining the parent membrane of a membrane. To facilitate presentation we assume that $\mu(m_j) = m_i$ implies $i < j$.

The evolution of the contents of the regions of a P system is described by rules associated to the regions. The system performs a computation by passing from one configuration to another one, applying the rules synchronously in each region. In the variant we consider in this paper, the rules are multiset rewriting rules given in the form of $u \to v$ where u, v are multisets, and they are applied in the maximal parallel manner, that is, as many rules are applied in each region as possible. The end of the computation is defined by the following halting condition: A P system halts when no more rules can be applied in any of the regions; the result is a number, the number of objects in a membrane labelled as output.

A *P system* of degree $n \geq 1$ is a construct

$$\Pi = (O, \mu, w_1, \dots, w_n, R_1, \dots, R_n, \rho_1, \dots, \rho_n)$$

where

- O is an alphabet of objects,
- μ is a membrane structure of n membranes,
- $w_i \in \mathcal{M}(O),\ 1 \leq i \leq n$, are the initial contents of the n regions,
- $R_i,\ 1 \leq i \leq n$, are the sets of evolution rules associated to the regions; they are of the form $u \to v$ where $u \in \mathcal{M}(O)$ and $v \in \mathcal{M}(O \times tar)$ where $tar = \{here, out\} \cup \{in_j \mid 1 \leq j \leq n\}$, and
- $\rho_1, \ldots, \rho_n$ are the priority rules associated with membranes $m_1, \ldots, m_n$.

The evolution rules of the system are applied in the non-deterministic, maximally parallel manner to the n-tuple of multisets of objects constituting the configuration of the system. A configuration is the sequence $C = (v_1, \ldots v_n, \mu_C)$ where $v_i \in O^*,\ 1 \leq i \leq n$ are the contents of the membranes, and μ_C is the current membrane structure. For two configurations $C_1 = (u_1, \ldots, u_n, \mu_{C_1})$ and $C_2 = (v_1, \ldots, v_n, \mu_{C_2})$, we can obtain C_2 from C_1, denoted as $C_1 \Rightarrow C_2$, by applying the rules of $R_1, \ldots, R_n$. Let $\mathcal{R} = R_1 \cup R_2 \cup \cdots \cup R_n$, where $R_i = \{r_{i1}, \ldots, r_{ik_i}\}$ is the set of rules corresponding to membrane m_i. The application of $u \to v \in R_i$ in the region i means to remove the objects of u from u_i and add the new objects specified by v to the system. The rule application in each region takes place in a non-deterministic and maximally parallel manner. This means that the rule application phase finishes, if no rule can be applied anymore in either region. As a result, each region where rule application took place, is possibly supplied with elements of the set $O \times tar$. We call a configuration which is a multiset over $O \cup O \times tar$ an intermediate configuration. If we want to emphasize that $C = (w_1, \ldots, w_n, \mu)$ consists of multisets over O, we say that C is a proper configuration. Rule applications can be preceded by priority check, if priority relations are present. Let $\rho_i \subseteq R_i \times R_i\ 1 \leq i \leq n$ be the (possibly empty) priority relations. Then $r \in R_i$ is applicable only if no $r' \in R_i$ can be applied with $(r', r) \in \rho_i$. We may also denote the relation $(r', r) \in \rho_i$ by $r' > r$.

In the next phase the objects coming from v should be added to the regions as specified by the target indicators associated to them. If v contains a pair $(a, here) \in O \times tar$, then a is placed in region i, the region where the rule is applied. If v contains $(a, out) \in O \times tar$, then a is added to the contents of the parent region of region i; if v contains $(a, in_j) \in O \times tar$ for some region j which is contained inside the region i (so region i is the parent region of region j), then a is added to the contents of region j.

The symbol δ marks a region for dissolution. When it is introduced in the membrane by a rule, after having finished the maximal parallel and communication steps, the actual membrane disappears. Its objects move to the parent membrane and its rules can not be applied anymore.

We can render promoter/inhibitor sets, *prom*/*inhib*, to each rule $r = (u \to v) \in \mathcal{R}_i$. The promoter/inhibitor sets belonging to r are subsets of O. When r is going to be applied they act as follows: r can be applied to the content w_i of membrane m_i only if every element of *prom* is present in w and no element of *inhib* can be found in w.

2.2 The chemical calculus

We give a brief summary of the chemical calculus following the presentation in [4] and [5]. Chemical programming is the formal equivalent of Gamma programming, which is a higher order multiset manipulating program language. Like Gamma

programming, the chemical calculus is also based on the chemical metaphor: data are represented by γ-terms, which are called molecules, and reactions between them are represented by rewrite rules. We begin with the basic definitions. The syntactical elements of molecules, reaction conditions, and patterns, denoted by M, C and P, respectively, are defined as follows.

$$M := x \mid (M_1, M_2) \mid \langle M \rangle \mid \gamma(P)[C].M$$

where x is a variable standing for any molecule, (M_1, M_2) is a compound molecule built with the commutative and associative "," constructor operator, $\langle M \rangle$ is called a solution, and $\gamma(P)[C].M$ is called a γ-abstraction with pattern P, reaction condition C, result M. The γ-abstraction encodes a rewriting rule: when the pattern P is respected and the condition C is met, a substituted variant of M is created as a result. A pattern is

$$P := x \mid (P_1, P_2) \mid \langle P \rangle,$$

where x matches any molecule, (P_1, P_2) matches a compound molecule, and $\langle P \rangle$ matches an inert solution, that is, a solution where no reaction can occur: it consists entirely of solutions or entirely of γ-abstractions. (The contained solutions can still be active, however.)

The solution $\langle M \rangle$ encapsulates the molecule M which is inside the solution, and thus, insulated from molecules outside the solution. The contents of solutions can only be changed by reactions which occur inside the solution.

Now we define how patterns are matched, which requires the notion of substitution. A *substitution* is a mapping ϕ from the set of variables to the set of molecules. We can define the application of a substitution to as follows:

$$\begin{aligned} \phi x &= \phi(x) \\ \phi(M_1, M_2) &= \phi M_1, \phi M_2 \\ \phi\langle M \rangle &= \langle \phi M \rangle \\ \phi(\gamma(P)[C].M) &= \gamma(P)[C].\phi' M, \end{aligned}$$

where ϕ' is obtained from ϕ by removing from the domain all the variables which occur in P.

The result of a match is an assignment of molecules to variables. The first argument of *match* is a pattern, the second one is a molecule, its value is a substitution. Let x denote a variable, P a pattern, and M a molecule. Then we define

$$\begin{aligned} match(x, M) &= \{x \mapsto M\} \\ match((P_1, P_2), (M_1, M_2)) &= match(P_1, M_1) \circ match(P_2, M_2) \\ match(\langle P \rangle, \langle M \rangle) &= match(P, M) \text{ provided } inert(M) \\ match(P, M) &= \boldsymbol{fail} \text{ in every other case,} \end{aligned}$$

where $\circ$ denotes the operation of function composition.

The reaction rule is defined as

$$\gamma(P)[C].M, N \to \phi M,$$

where $match(P, N) = \phi$ assigns values to variables in such a way that $\phi(C)$ holds in the typed case or reduces to *true* in the untyped case. In this case *true* can be a special constant defined in advance, for example, $true \rightleftharpoons \gamma\langle x \rangle[x].x$.

We can define an operator *replace* (cf. [5]) which does not vanish in the course of the reduction:

$$replace\ P\ by\ M\ if\ C \rightleftharpoons let\ rec\ f = \gamma(P)[C].M, f\ in\ f.$$

Then the new operator obeys the following reduction rule:

$$replace\ P\ by\ M\ if\ C, N \rightarrow replace\ P\ by\ M\ if\ C, \phi(M),$$

where $match(P, N) = \phi$ and either $\phi(C)$ is true or it reduces to *true*.

At this point we should mention that the simulation takes place in the typed γ-calculus ([5]), because it is more convenient to talk about equality and comparison of integer values, than to check whether the conditional part of an untyped γ-expression reduces to *true* (which is, in fact, undecidable in the general case). We could, however, restrict the γ-expressions taking part in the simulation in such a way that their conditional parts form a fragment of the γ-calculus that is decidable with respect to equality. (We can take, e. g., the γ-calculus equivalents of Church numerals and define Boolean operations on them.)

3 Results

First we introduce molecules for the description of membrane system configurations.

Notation 1 *Let* $[x, y] = (\langle x \rangle, y)$, *and* $[x_1, \ldots, x_n, x_{n+1}] = [[x_1, \ldots, x_n], x_{n+1}]$.

Remark 1. Let $P = [x_1, x_2, \ldots, x_l]$ be a pattern in the sense of the previous section, and $M = [s_1, s_2, \ldots, s_l]$, where $s_1, \ldots, s_l$ are arithmetical expressions, i.e. expressions composed of natural numbers, variables and arithmetical operations. If we assume that none of the x_i appears among the free variables of $s_1, \ldots, s_l$, then $match(P, M) = \Phi \neq fail$ implies $\Phi = [x_1/s_1, x_2/s_2, \ldots, x_l/s_l]$, where Φ is the simultaneous substitution formed by the substitutions $[x_1/s_1], \ldots, [x_l/s_l]$. In other words, in this special case, the molecule $[x_1, x_2, \ldots, x_l]$ behaves as an ordered tuple.

If we use a, b as variables for elements of O and r as a rule variable, respectively, then we say that a rule $r = u \rightarrow v \in R_i$ is valid with respect to the configuration $(w_1, \ldots, w_n, \mu)$ if the following conditions hold:

1. membrane structure μ contains membrane m_i,
2. $(\forall a \in prom_r)\ (w_i(a) \geq 1)$,
3. $(\forall a \in inhib_r)\ (w_i(a) = 0)$, and
4. $(\forall a \in O)(\forall\ 1 \leq j \leq n)\ (v(a, in_j) \geq 1)$ implies that μ contains the membrane m_j (m_j is not dissolved) and $\mu(m_j) = m_i$, namely m_i is the parent membrane of m_j.

where $prom_r \subseteq O$ and $inhib_r \subseteq O$ denotes the set of promoters and inhibitors associated to rule r, respectively.

A description of a membrane system configuration as above is a molecule of the form

$$\begin{aligned} Descr = [&c_{11}, \ldots, c_{1k}, \ldots, c_{n1}, \ldots, c_{nk},\\ &\overline{c}_{11}, \ldots, \overline{c}_{1k}, \ldots, \overline{c}_{n1}, \ldots, \overline{c}_{nk},\\ &d_1, \ldots, d_n,\\ &p_{11}, \ldots, p_{1k_1}, \ldots, p_{n1}, \ldots, p_{nk_n}], \end{aligned}$$

where c_{ij} and $\overline{c}_{ij}$ are natural numbers ($1 \leq i \leq n, 1 \leq j \leq k$), $d_i \in \{0,1\}$ ($1 \leq i \leq n$) and $p_{ik_j} \in \{0,1\}$ ($1 \leq i,j \leq n$). If N is a description we denote by c_{ij}, $\overline{c}_{ij}$, etc. the respective parts of N.

Let $C = (\mu, w_1, \ldots, w_n)$ be an (intermediate) configuration. A description $Descr(C)$ corresponding to C is a description, where $c_{ij} = w_i(a_j)$ and $\overline{c}_{ij} = w_i(a_j, here) + \sum_{p \neq i, \mu(m_i) = m_p} w_p(a_j, in_i) + \sum_{\mu(p)=i} w_p(a_j, out)$ with $(1 \leq i,p \leq n)$ and $(1 \leq j \leq k)$. Here $\mu(p)$ denotes the parent membrane of m_p, and recall that $w(a)$ denotes the number of elements a in the multiset w. Intuitively, c_{ij} stands for the number of occurrences of a_j in m_i, and $\overline{c}_{ij}$ denotes the location of the targeted elements of O. Moreover, $d_i = 1$ iff m_i is dissolved or under dissolution and p_{ik_j} describes the validity of rules: rule r_{ik_j} is valid iff $p_{ik_j} = 1$, if C is a proper configuration. If $C \rightarrow^* C'$, and C' is an intermediate configuration and there are no proper configurations in the reduction sequence other than C, then $p'_{ik_j} = 1$ in the description of C' iff $p_{ik_j} = 1$ in the description of C. Observe that if C is a proper configuration then $\overline{c}_{ij} = 0$ for every possible i and j. When a configuration is proper, $d_i = 1$ implies $w_i = 0$.

A pattern for a description is a tuple of the form

$$\begin{aligned} S = [&x_{m_1a_1}, \ldots, x_{m_1a_k}, \ldots, x_{m_na_1}, \ldots, x_{m_na_k},\\ &\overline{x}_{m_1a_1}, \ldots, \overline{x}_{m_1a_k}, \ldots, \overline{x}_{m_na_1}, \ldots, \overline{x}_{m_na_k},\\ &x_{d_1}, \ldots, x_{d_n}, x_{r_{1k_1}}, \ldots, x_{r_{nk_n}}]. \end{aligned} \tag{1}$$

Let $\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n, \rho_1, \ldots, \rho_n)$ be a P system, and let $C' = (w'_{k_1}, \ldots, w'_{k_j}, \mu')$ be a proper configuration obtained from the initial configuration in a finite number of computational steps, where $1 \leq k_1 < \ldots < k_j \leq n$. Then the description of C' relative to μ is the description obtained from $Descr(C')$ when we set $d_i = 1$ for $i \notin \{k_1, \ldots, k_j\}$ and $c_{ij} = 0$ $(1 \leq j \leq k)$ and $p_{il} = 0$ for every rule $r_{il} \in R_i$. That is, we supplement $Descr(C')$ as if it were a description of an n-ary membrane system by treating the missing membranes as empty membranes. We denote the description of a configuration C' relative to μ by $Descr_\mu(C')$.

Because a description should also contain information about the structure of the original P system itself, we append a representation of the function μ at the end of each description. Let Π be a P system of order n as before. Then a tuple $[p_2, \ldots, p_n]$ of length $n-1$ is appended to every description in the simulation with the following meaning: if membrane m_j has membrane m_i as its parent, then $p_j = i$. Since the $Skin$ has no parent membrane, numbering begins with 2. Likewise, a description pattern is expanded with the tuple $[x_{p_2}, \ldots, x_{p_n}]$. Since the structure of the original P system remains the same in the course of the simulation process, we do not indicate the appended values for μ, they are implicitly understood to be there.

With this in hand we are able to define the molecule in charge for deciding rule validity. Let $r = u \rightarrow v \in R_i$, and S be a description pattern . Then let

$$
\begin{aligned}
Cond(r) = (x_{d_i} = 0 \wedge & \qquad (2)\\
& \bigwedge_{1\le j\le k} (a_j \in prom_r \supset x_{m_i a_j} \ge 1) \wedge \\
& \bigwedge_{1\le j\le k} (a_j \in inhib_r \supset x_{m_i a_j} = 0)) \wedge \\
& \bigwedge_{1\le l\le k} \bigwedge_{1\le j\le n} \big(v(a_l, in_j) \ge 1 \supset x_{d_j} = 0 \wedge \\
& \big(\bigvee_{l_0=i>l_1>\ldots>l_{s-1}>j=l_s} \big(\bigwedge_{1\le t\le s} x_{p_t} = l_{t-1} \wedge \bigwedge_{1\le q\le s-1} x_{d_q} = 1\big)\big)\big)
\end{aligned}
$$

The last row expresses the fact that either m_i is the parent of m_j, or m_i is an ancestor of m_j and all the intermediate parent membranes have been dissolved in the construction.

Now rule validity can be expressed as

$$Val(r) = replace\ [S,0]\ by\ [S[x_r/1],0]\ if\ Cond(r)$$

where the value 0 plays a role of synchronization to be specified later on. We remark that if a rule r is determined to be valid in this phase of the simulation, then r remains valid in the course of the simulation of a maximal parallel step.

Discussion 1 *At this point, we can also incorporate in the simulation of a membrane system the priority rules, if present. Let $(\rho_1, \ldots, \rho_n)$ be the tuple prescribing the priority relations in the membranes of the given P system. We define molecules determining the validity of rules when priority is present. Assume $r \in R_i$. We distinguish two cases:*

- *There does not exist $r' \in R_i$ such that $r' > r$. Then $Val_\rho(r)$ is defined as $Val(r)$ above.*
- *There are rules $r_1, \ldots, r_j \in R_i$ such that $r_l > r$ $(1 \le l \le j)$. Let S be a description pattern and denote by $Cond(r)$ the conditional part of $Val(r)$ defined in Equation (2). Then*

$$
\begin{aligned}
Val_\rho(r) = \big(& replace\ [S,0]\ by\ [S[x_r/1],0] \\
& if\ (Cond(r) \wedge \bigwedge_{1\le l\le j} x_{r_l} = 0), \\
& replace\ [S,0]\ by\ [S[x_r/0],0] \\
& if\ (x_r = 1 \wedge (\bigvee_{1\le l\le j} x_{r_l} = 1)\big).
\end{aligned}
$$

Now we can turn to the main part of the simulation. The conditions of rule application must reflect now the fact that the rule is executable together with the conditions that make it valid.

Definition 1. *Let $r = u \to v \in R_i$, and let S be a description pattern. Then the molecule describing the effect of an execution of r is defined as*

$$
\begin{aligned}
App(r) = replace\ [S,1]\ by\ [\,apply(S,r),1]\ if \\
\big(x_r = 1 \wedge \bigwedge_{1\le j\le k} (u(a_j) \le x_{m_i,a_j}),
\end{aligned}
$$

where

$$apply(S,r)(x_{m_sa_t}) = \begin{cases} x_{m_sa_t} - u(a_t) & \text{if } s = i, \\ x_{m_sa_t} & \text{otherwise}, \end{cases}$$

$$apply(S,r)(\overline{x}_{m_sa_t}) = \begin{cases} \overline{x}_{m_sa_t} + v(a_t, here) & \text{if } s = i, \\ \overline{x}_{m_sa_t} + v(a_t, in_j) & \text{if } s = j \neq i, \\ \overline{x}_{m_sa_t} + v(a_t, out) & \text{if } s = \mu(i), \end{cases}$$

$$apply(S,r)(x_{d_j}) = \begin{cases} 1 & \text{if } v(\delta) = 1, \\ x_{d_j} & \text{otherwise}, \end{cases}$$

$$apply(S,r)(x_r) = x_r.$$

Here we made use of the implicit stipulation that S is of the form as in Equation (1), which is indeed the case if we ignore variable renaming.

The next group of rules is the set of communication rules. In what follows, we define the chemical calculus equivalents of communication steps.

Definition 2.

$$Msg = replace\ [S,2]\ by\ [msg(S),2]\ if \\ \big(\bigvee_{1\leq i\leq n}\bigvee_{1\leq j\leq k} \overline{x}_{m_ia_j} \geq 1\big),$$

where

$$msg(S)(x_{m_ia_j}) = x_{m_ia_j} + \overline{x}_{m_ia_j}, \text{ for } 1 \leq i \leq n \text{ and } 1 \leq j \leq k$$

and

$$msg(S)(\overline{x}_{m_ia_j}) = 0, \text{ for } 1 \leq i \leq n \text{ and } 1 \leq j \leq k.$$

At this point, we simulate the effects of membrane dissolving. We have to drive the elements leaving the actual membranes by applications of in_j or *out* rules or elements of membranes freshly dissolved into membranes remaining existent after performing of the maximal parallel step. To this end, we define the following molecule.

Definition 3.

$$Dis_i = replace\ [S,3]\ by\ [dis_i(S),3]\ if \\ \big(x_{d_i} = 1 \wedge \\ \big(\bigvee_{1\leq j\leq k} x_{m_ia_j} \geq 1\big)\big),$$

where

$$dis_i(S)(x_{m_ja_l}) = \begin{cases} x_{m_ja_l} + x_{m_ia_l} & \text{if } j = \mu(i), \\ 0 & \text{if } j = i, \\ x_{m_ja_l} & \text{otherwise.} \end{cases}$$

We also need some auxiliary molecules to set the values indicating the validity of rules to zero, in order to start a new maximal parallel step. Thus

Definition 4.

$$RemVal(r) = replace\ [S,4]\ by\ [S[x_r/0],4]\ if\ x_r = 1.$$

Now we are in a position to determine the molecule leading us through the simulation process. Let

$$\begin{aligned}
Val_\rho &= \bigcup\{Val_\rho(r) \mid r \in \mathcal{R}\},\\
App &= \bigcup\{App(r) \mid r \in \mathcal{R}\},\\
Dis &= \bigcup\{Dis_i \mid i \in \{1,\ldots,n\}\},\\
RemVal &= \bigcup\{RemVal(r) \mid r \in \mathcal{R}\},\\
Sync &= replace\ \langle [S, x_{sync}], Val_\rho, App, Msg, Dis, RemVal\rangle\ by\\
&\quad \langle [S, x_{sync} + 1\ mod\ 5], Val_\rho, App, Msg, Dis, RemVal\rangle\ if\\
&\quad \bigvee_{1\le i\le n} x_{r_i} = 1,\\
&\quad \text{where } S \text{ is a description pattern.}
\end{aligned}$$

Notation 2 *Let N be a molecule and let*

$$M(N) = (\langle N, Val_\rho, App, Msg, Dis\rangle, Sync).$$

If C is a configuration of Π such that $C \Rightarrow^* C'$ for some C' and $i \in \{0,1,2\}$, then we write

$$M(C', i) = M([Descr_\mu(C'), i]).$$

The terms of the chemical calculus, and also the configurations of membrane systems can be considered as rewriting systems. A rewriting system, as used in this paper, is a pair $\mathcal{A} = \{\Sigma, (\rightarrow_i)_{i\in I}\}$, where Σ is a set and $(\rightarrow_i)_{i\in I}$ is a set of binary relations defined on Σ. The relations $(\rightarrow_i)_{i\in I}$ are called reduction relations. It is supposed that a reduction relation $\rightarrow_i$ is compatible with the term formation rules. Moreover, if $\rightarrow_i$ is a reduction relation, we denote by $\rightarrow_i^*$ its reflexive, transitive closure. We may use the notation $\rightarrow = \cup_{i\in I}(\rightarrow_i)$, too. In the following, the set Σ is the set of configurations of a P system or, in the case of the chemical formalism, the set of γ-terms, and $\rightarrow_i$ are the binary relations rendering configurations to configurations or terms to terms, respectively. We say that $m \in \Sigma$ is in *normal form*, if there is no $n \in \Sigma$, such that $m \rightarrow n$. Moreover, an $m \in \Sigma$ is *strongly normalizable*, if every reduction sequence starting from m is finite, or *weakly normalizable*, if there exists a finite reduction sequence starting from m. We say that a molecule or a membrane M is $\rightarrow_i$-irreducible, if there is no M' such that $M \rightarrow_i M'$. In what follows, to conform to the usual membrane system notation, we use $\Rightarrow$ to denote $\rightarrow$ when we speak of a rewriting step in a membrane computation.

Theorem 1. *(1) Let $\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n, \rho_1, \ldots, \rho_n)$ be a P system of order n with membrane dissolving, promoter/inhibitor sets for rules and priority relations. Assume*

$$C_0 = (\mu, w_1, \ldots, w_n) \Rightarrow^* C_1 = (\mu', w'_{n_1}, \ldots, w'_{n_i}),$$

where $1 \le n_1 \le \ldots \le n_i \le n$. Then

$$M(C_0, 0) \rightarrow^* M(C_1, 0).$$

If the computation starting from C_0 contains at least one step, then the reduction sequence starting from $M(C_0, 0)$ is non-empty either.

(2) Let Π be a P system as above. Assume

$$M(C_0, 0) \to^* M([N, 0]), \text{ and}$$

assume that $\bar{c}_{ij} = 0$ for $(1 \leq i \leq n)$ and $(1 \leq j \leq k)$ in N and $[N, 0]$ is Val_ρ irreducible. Then there exists a configuration $C_1 = (\mu', w'_{n_1}, \ldots, w'_{n_i})$ of Π such that $M([N, 0]) = M(C_1, 0)$ and

$$C_0 \Rightarrow^* C_1.$$

Moreover, if the length of $M_\mu(C_0) \to^* M([N, 0])$ is at least one, then the length of the computation starting from C_0 is non-zero.

We work our way to the proof of the theorem by stating several auxiliary lemmas.

As formulated in [2], a computational step starting from a configuration C_0 of Π consists of a maximal parallel step (mpr), a step for removing the directions from the targeted elements (tar) and a step for accomplishing membrane dissolution (δ). In notation, if C_0 is a configuration of Π and $C_0 \Rightarrow C_1$, then there are C'_0 and, if δ is present, C''_0 such that

$$C_0 \Rightarrow^*_{mpr} C'_0 \Rightarrow_{tar} C''_0 \Rightarrow_\delta C_1.$$

In the present paper, instead of $\Rightarrow_{tar}$, we choose a sequential relation (msg) defined in Definition 5 for removing messages instead of parallel communication rules, which equally suffices for our purposes. In what follows, if $C \Rightarrow_s C'$ by an intermediate step, we denote by $s \in mpr$ ($s \in msg, s \in \delta$) the fact whether s is a maximal parallel, message removing, or membrane dissolving step, respectively.

We verify the lemmas simultaneously by induction on the number of intermediate steps in a computational step of the P system and on the number of reductions in the chemical calculus.

Notation 3 *Let C' be an (intermediate) configuration, where $C \Rightarrow^* C'$. Let $Descr_\mu(C')$ be the description of C' relative to μ. Then we use the notation below to extract the corresponding values from $M(C', l)$:*

$$\begin{array}{ll}
\lfloor M(C', l) \rfloor_{c_{ij}} = Descr_\mu(C')_{i \cdot k + j} & (0 \leq i \leq n-1,\ 1 \leq j \leq k), \\
\lfloor M(C', l) \rfloor_{\bar{c}_{ij}} = Descr_\mu(C')_{(n+i) \cdot k + j} & (1 \leq i \leq n,\ 1 \leq j \leq k), \\
\lfloor M(C', l) \rfloor_{d_i} = Descr_\mu(C')_{2n \cdot k + i} & (1 \leq i \leq n), \\
\lfloor M(C', l) \rfloor_{r_{ij}} = Descr_\mu(C')_{(2k+1) \cdot n + k_1 + \ldots + k_{i-1} + j} & (1 \leq j \leq k_i,\ 1 \leq i \leq n).
\end{array}$$

The following claims can be verified easily. Below, let D denote a description.

Claim. Let $M([D, 0]) \to^*_{Val} M'$. Then $M' = M([D', 0])$, where D' is a description.

Claim. Let $M([D, 1]) \to^*_{App} M'$. Then $M' = M([D', 1])$, where D' is a description.

Claim. Let $M([D, 2]) \to^*_{Msg} M'$. Then $M' = M([D', 2])$, where D' is a description.

Claim. Let $M([D, 3]) \to^*_{Dis} M'$. Then $M' = M([D', 3])$, where D' is a description.

In the following, we assume that every possible configuration is the result of some computational sequence starting from a fixed configuration C of the P system $\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n, \rho_1, \ldots, \rho_n)$ of order n with membrane dissolving, promoter/inhibitor sets for rules, and priority relations.

We prove the two parts of the theorem by simultaneous induction on the number of reduction steps in the chemical calculus and computational steps in the P system, respectively.

Lemma 1. *(1) If $C' \Rightarrow^*_{mpr} C''$, then $M(C', 1) \rightarrow^*_{App} M(C'', 1)$, and conversely,*

(2) if we assume $M(C', 1) \rightarrow^*_{App} M''$, then there is C'' such that $C' \Rightarrow^*_{mpr} C''$ and $M'' = M(C'', 1)$.

Proof. We prove the lemma by simultaneous induction on the lengths of the reduction sequences. Assume we know the result for reduction sequences of lengths at most s.

(1) Let $C \Rightarrow^* C'$, assume $C' = (\mu', w'_1, \ldots, w'_n)$. Suppose $C' \Rightarrow^s_{mpr} C''' \Rightarrow_r C''$, $C'' = (\mu'', w''_1, \ldots, w''_n)$, $C''' = (\mu''', w'''_1, \ldots, w'''_n)$ and $r = u \rightarrow v \in \mathcal{R}_i$. Since r is applicable to C''', we have $\lfloor M(C''', 1) \rfloor_r = 1$ and $u(a_j) \leq w_i(a_j)$, which means $u(a_j) \leq \lfloor M(C''', 1) \rfloor_{c_{ij}}$. These together imply that $App(r)$ can be applied to $M(C''', 1)$ yielding $M([apply(Descr_\mu(C'''), r), 1])$.

- Let $\lfloor M([apply(Descr_\mu(C'''), r), 1]) \rfloor_{cl_j} = s_{lj}$. Then $s_{lj} = \lfloor M(C''', 1) \rfloor_{cl_j} - u(a_j) = w'''_i(a_j) - u(a_j)$, if $l = i$, and $s_{lj} = \lfloor M(C''', 1) \rfloor_{cl_j} = w'''_i(a_j)$ otherwise.
- Let $\lfloor M([apply(Descr_\mu(C'''), r), 1]) \rfloor_{\bar{c}_{lj}} = t_{lj}$. Then $t_{lj} = \lfloor M(C''', 1) \rfloor_{\bar{c}_{lj}} + v(a_j, here)$, if $l = i$, $t_{lj} = \lfloor M(C''', 1) \rfloor_{\bar{c}_{lj}} + v(a_j, in_h)$, if $l = h \neq i$ and $\mu(m_h) = m_i$, and $t_{lj} = \lfloor M(C''', 1) \rfloor_{\bar{c}_{lj}} + v(a_j, out)$, if $l = \mu'''(i)$. Taking all these into account, $t_{ij} = w''_i(a_j, here) + \sum_{p \neq i, \mu(m_i) = m_p} w''_p(a_j, in_i) + \sum_{\mu(p) = i} w''_p(a_j, out)$ remains valid.
- If $v(\delta) = 1$, then $\lfloor M(C'', i) \rfloor_{d_i}$ is set to 1.

(2) Let $C \Rightarrow^* C'$, and $M(C', 1) \rightarrow^*_{App} M''$. It is enough to prove the result for the case $M(C', 1) \rightarrow_{App(r)} M''$, where $r = u \rightarrow v$. By Claim 3, $M'' = M([D'', 1])$. Since r is applicable to $M(C', 1)$, we have, by $\lfloor M(C', 1) \rfloor_r = 1$, that $r = u \rightarrow v \in \mathcal{R}_i$ is valid for some fixed i depending on r. Moreover, $u(a_j) \leq \lfloor M(C', 1) \rfloor_{c_{lj}} = w'_i(a_j)$, for every $1 \leq j \leq k$, which makes r applicable to C'. From this point on, we can show by a reasoning similar to that of the previous point that $D'' = M(C'', 1)$, where $C' \Rightarrow_r C''$. We omit the details. □

Instead of parallel communication as defined in [2] we choose the simpler way which is equally suitable to our present purposes and we define $\Rightarrow_{msg}$ as the following set of sequential multiset transformations.

Definition 5. *Let $C = (w_1, \ldots, w_n, \mu)$ and $C' = (w'_1, \ldots, w'_n, \mu,)$. Then $C \Rightarrow^*_{tar} C'$ holds iff one of the following cases is valid.*

1. *Assume that $w_i(a_j, here) > 0$. Then $w'_i(a_j) = w_i(a_j) + w_i(a_j, here)$ and $w'_i(a_j, here) = 0$. All the other values remain unchanged.*
2. *Assume $w_i(a_j, in_l) > 0$. Then $w'_l(a_j) = w_l(a_j) + w_i(a_j, in_l)$ and $w'_i(a_j, in_l) = 0$. All the other values remain unchanged.*
3. *Assume $w_i(a_j, out) > 0$ and $l = \mu(i)$ is defined. Then $w'_l(a_j) = w_l(a_j) + w_i(a_j, out)$ and $w'_i(a_j, out) = 0$. If $i = Skin$, then $w'_i(a_j, out) = 0$. All the other values remain unchanged.*

Lemma 2. *(1) Let $C' \Rightarrow^*_{msg} C''$, and assume that C'' is msg-irreducible. Then $M(C',2) \rightarrow_{Msg} M(C'',2)$.*

(2) Conversely, assume $M(C',2) \rightarrow_{Msg} M''$. Then there is C'' such that $C' \Rightarrow_{msg} C''$, C'' is msg-irreducible, and $M'' = M(C'',1)$.

Proof. We prove by induction on the number of steps in $C \Rightarrow_{msg} C'$ that, for every $1 \leq i \leq n$ and $1 \leq j \leq k$,

$$Descr_\mu(C)_{c_{ij}} + Descr_\mu(C)_{\overline{c}_{ij}} = Descr_\mu(C')_{c_{ij}} + Descr_\mu(C')_{\overline{c}_{ij}}. \tag{3}$$

To this end, we show that, if $C \rightarrow_{msg} C'$ and $C = (\mu, w_1, \ldots, w_n)$ and $C = (\mu, w'_1, \ldots, w'_n)$, then

$$\begin{aligned} &w_i(a_j) + w_i(a_j, here) + \sum_{p \neq i} w_p(a_j, in_i) + \sum_{\mu(p)=i} w_p(a_j, out) = \\ &w'_i(a_j) + w'_i(a_j, here) + \sum_{p \neq i} w'_p(a_j, in_i) + \sum_{\mu(p)=i} w'_p(a_j, out). \end{aligned} \tag{4}$$

We treat Point 2 of Definition 5, the remaining cases can be handled similarly. Let $C \Rightarrow_{msg} C'$ by Point 2 of Definition 5. Assume $w_i(a_j, in_l) > 0$. Let us consider only the case $i = l$ in Equation 4, since for all the other cases the equation trivially holds. But in this case the left hand side contains $w_l(a_j) + w_i(a_j, in_l)$, and the right hand side contains the corresponding $w'_l(a_j) + w'_i(a_j, in_l)$, which, by definition, are equal.

($\Rightarrow$) Let $C \Rightarrow_{msg} C'$, assume that C' is msg-irreducible. A msg-irreducible P system with the *Skin* membrane as the outermost membrane contains no messages, thus, by Equation 3, $M(C,2) \rightarrow_{Msg} M(C',2)$.

($\Leftarrow$) Let $M(C,2) \rightarrow_{Msg} N'$. Then, by Claim 3, $N' = M(D',2)$ for some description D'. Let $C \Rightarrow_{msg} C'$ such that C' is msg-irreducible. Then C' is message free, which, by Equation 4, entails $D' = Descr_\mu(C')$.

□

Now, following [1], we define the skeleton of a configuration $(\mu, w_1, \ldots, w_n)$ as $U' = (u'_1, \ldots, u'_n)$, where $u'_i = *$, if membrane i is dissolved or under dissolution (that is, $u_i(\delta) = 1$ and $i \neq Skin$) and $u'_i = 0$ otherwise. Let

$$\begin{aligned} &\mu^0(i) = i, \\ &\mu^j(i) = \mu(\mu^{j-1}(i)) \quad \text{for } j > 0. \end{aligned}$$

Let $\mu_{U'}(i) = min\{j \mid \mu^k(i) = j \wedge u'_j \neq * \wedge u'(\mu^l(i)) = * \text{ for } 0 \leq l \leq k-1\}$. That is, $\mu_{U'}(i)$ is the smallest membrane containing membrane i which exists or does not disappear. Let $C' \Rightarrow_\delta C''$, assume $w'_l(\delta) = 1$ for at least one membrane m_l. We define the effect of the dissolution rule as follows: $(\mu', w'_1, \ldots, w'_n) \Rightarrow_\delta (\mu'', w''_1, \ldots, w''_n)$, where $w''_i = *$ provided $u'_i = *$, and $w''_i(a_j) = w'_i(a_j) + \sum\{w'_l(a_j) \mid \mu_{U'}(l) = i, w'_l(\delta) = 1\}$, if $u'(i) = 0$.

Lemma 3. *(1) If $C' \Rightarrow_\delta C''$, then $M(C',3) \rightarrow^*_{Dis} M(C'',3)$.*

(2) Conversely, assume that $M(C',3) \rightarrow^*_{Dis} M''$, and M'' is Dis-irreducible. Then there exists a proper configuration C'' with $C' \Rightarrow_\delta C''$ and $M'' = M(C'',3)$.

Proof. (1) Let $C' = (\mu, w'_1, \ldots, w'_n) \Rightarrow_\delta C''$, and assume that $w'_i(\delta) = 1$. Then $\lfloor M(C', 3) \rfloor_{d_i} = 1$. Let $\lfloor M(C', 3) \rfloor_{c_{ij}} > 1$ for some $1 \leq j \leq k$. Then $M(C', 3) \rightarrow_{Dis_i} M([dis_i(Descr_\mu(C')), 3])$. Let $p = \mu_{U'}(i)$, where U' is the skeleton of C'. Let $D' = Descr_\mu(C')$ and $D'' = dis_i(Descr_\mu(C'))$. Let us denote by $D'_{c_{ij}}$ and $D''_{c_{ij}}$ the values of the descriptions pertaining to the coordinates (i, j). It follows immediately, by Definition 3, that

$$D'_{c_{pj}} + \sum \{D'_{c_{lj}} \mid \mu_{U'}(l) = p, D'_{d_l} = 1\} = \tag{5}$$
$$D''_{c_{pj}} + \sum \{D''_{c_{lj}} \mid \mu_{U'}(l) = p, D''_{d_l} = 1\}.$$

In other words, for every $1 \leq j \leq k$, the sums of the occurrences of elements a_j in the dissolved or to be dissolved descendants of membrane m_p plus the multiplicity of a_j in m_p remain the same at a dissolution step in the chemical calculus. Let $M(C', 3) \rightarrow_{Dis} M([D, 3])$ such that $M([D, 3])$ is irreducible with respect to Dis. Then $D_{d_i} = 1$ implies $D_{c_{ij}} = 0$, which, by Equation 5, involves that $D = Descr_\mu(C'')$.

(2) Let $M(C', 3) \rightarrow^*_{Dis} M''$. By Claim 3, there exists a description D'' such that $M'' = M([D'', 3])$. Since M'' is Dis irreducible, $D''_{d_i} = 1$ implies $D''_{c_{ij}} = 0$. This means, there is a proper configuration C'' such that $Descr_\mu(C'') = D''$. Assume $w'_s(\delta) = 1$ holds in C'. Let $D' = Descr_\mu(C')$. Let U' be the skeleton of D' and $p = \mu_{U'}(s)$. Since M'' is Dis irreducible, Equation 5 simplifies to

$$D'_{c_{pj}} + \sum \{D'_{c_{lj}} \mid \mu_{U'}(l) = p, D'_{d_l} = 1\} = D''_{c_{pj}}.$$

Taking the corresponding configurations, this amounts to $C' \Rightarrow_\delta C''$. □

Proof of Theorem 1.

($\Rightarrow$) Let $\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n, \rho_1, \ldots, \rho_n)$ be a P system of order n with membrane dissolving, promoter/inhibitor sets for rules and priority relations. Assume $C_0 \Rightarrow^t C_1$. We prove by induction on t that $M(C_0, 0) \rightarrow^* M(C_1, 0)$. Let $C \Rightarrow^{t-1} C_2 \Rightarrow C_1$. Assume $C_2 = (\mu'', w''_{n_1}, \ldots, w''_{n_i})$, $C_1 = (\mu', w'_{n_1}, \ldots, w'_{n_i})$. Assume there exists N'' such that $M(C'', 0) \rightarrow_{Val} N''$. But $Val_\rho(r)$ is applicable iff r is valid and no rule r' with $(r', r) \in \rho$ is valid, this means $N'' = M(C'', 0)$ and $M(C'', 0)$ is Val_ρ irreducible. In this case

$$M(C'', 0) \rightarrow_{Sync} M(C'', 1).$$

Putting Lemmas 1, 2 and 3 together, taking into account the fact that $M(E, i) \rightarrow_{Sync} M(E, i + 1 \ mod \ 5)$ whenever $M(E, i)$ is irreducible for the corresponding reduction, we obtain that there exists a configuration $\widetilde{C}$ and a description D such that

$$M(C'', 0) \rightarrow^*_{RemVal} M([D, 4]) \rightarrow_{Sync} M([D, 0]),$$

where D is the description $Descr_\mu(\widetilde{C})$ except for the values $D_r = 0$. A transition in $Val_\rho(r)$ is applicable at most twice for every rule r. This means there is a description D' such that

$$M([D, 0]) \rightarrow^*_{Val_\rho} M([D', 0])$$

and $M([D', 0])$ is Val_ρ irreducible. But then $D' = Descr_\mu(\widetilde{C})$.

($\Leftarrow$) Follows in a way similar to the above part from Lemmas 1, 2 and 3, but this time applying the other directions of the lemmas.

Corollary 1. *Let* $\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n, \rho_1, \ldots, \rho_n)$ *and let* $C = (\mu, w_1, \ldots, w_n)$. *Then* Π *is strongly (resp. weakly) normalizing iff* $M(C, 0)$ *is strongly (resp. weakly) normalizing. Moreover, the halting computations starting from* C *provide the same results as those supplied by the terminating reduction sequences of* $M(C, 0)$.

References

1. O. Agrigoroaiei, G. Ciobanu, Flattening the transition P systems with dissolution. In: M. Gheorghe, T. Hinze, Gh. Păun, G. Rozenberg, A. Salomaa (editors), *Membrane Computing, 11th International Conference, CMC 2010, Jena, Germany, 2010, Revised Selected Papers.* Volume 6501 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg (2011) 53–64.
2. O. Andrei, G. Ciobanu and D. Lucanu, Structural operational semantics of P systems. In: R. Freund, Gh. Păun, G. Rozenberg, A. Salomaa (editors), *Membrane Computing, 6th International Workshop, WMC 2005, Vienna, Austria, 2005, Revised Selected and Invited Papers.* Volume 3850 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg (2006) 32–49.
3. O. Andrei, G. Ciobanu and D. Lucanu, A rewriting logic framework for operational semantics of membrane systems. *Theoretical Computer Science*, 373 (2007) 163–181.
4. J.P. Banâtre, P. Fradet, Y. Radenac, Principles of chemical computing. *Electronic Notes in Theoretical Computer Science* 124 (2005) 133–147.
5. J.P. Banâtre, P. Fradet, Y. Radenac, Generalized multisets for chemical programming. *Mathematical Structures in Computer Science* 16 (2006) 557–580.
6. J.P. Banâtre, D. Le Métayer, A new computational model and its discipline of programming. *Technical Report RR0566*, INRIA (1986).
7. P. Battyányi, Gy. Vaszil, Describing membrane computations with a chemical calculus. *Fundamenta Informaticae*, 134 (2014) 39–50.
8. P. Bottoni, C. Martín-Vide, G. Păun, G. Rozenberg, Membrane systems with promoters/inhibitors. *Acta Informatica*, 38 (2002) 695–720.
9. G. Păun, Computing with membranes. *Journal of Computer and System Sciences*, 61(1) (2000) 108–143.
10. G. Păun, *Membrane Computing. An Introduction.* Springer, 2002.
11. G. Păun, G. Rozenberg, A. Salomaa (eds), *The Oxford Handbook of Membrane Computing.* Oxford University Press, 2010.

Notes on Spiking Neural P Systems and Finite Automata

Francis George C. Cabarle[1], Henry N. Adorna[1], Mario J. Pérez-Jiménez[2]

[1]Department of Computer Science
University of the Philippines Diliman
Diliman, 1101, Quezon city, Philippines;
[2]Department of Computer Science and AI
Universidad de Sevilla
Avda. Reina Mercedes s/n, 41012, Sevilla, Spain
fccabarle@up.edu.ph, hnadorna@dcs.upd.edu.ph, marper@us.es

Summary. Spiking neural P systems (in short, SNP systems) are membrane computing models inspired by the pulse coding of information in biological neurons. SNP systems with standard rules have neurons that emit at most one spike (the pulse) each step, and have either an input or output neuron connected to the environment. SNP transducers were introduced, where both input and output neurons were used. More recently, SNP modules were introduced which generalize SNP transducers: extended rules are used (more than one spike can be emitted each step) and a set of input and output neurons can be used. In this work we continue relating SNP modules and finite automata: (i) we amend previous constructions for DFA and DFST simulations, (ii) improve the construction from three neurons down to one neuron, (iii) DFA with output are simulated, and (iv) we generate automatic sequences using results from (iii).

Key words: Membrane computing, Spiking neural P systems, Finite automata, Automatic sequences

1 Introduction

Spiking neural P systems (in short, SNP systems) introduced in [7], incorporated into membrane computing the idea of pulse coding of information in computations using spiking neurons (see for example [10][11] and references therein for more information). In pulse coding from neuroscience, pulses known as *spikes* are not distinct, so information is instead encoded in their multiplicity or the time they are emitted.

On the computing side, SNP systems have *neurons* processing only one object (the spike symbol a), and neurons are placed on nodes of a directed graph. Arcs between neurons are called *synapses*. SNP systems are known to be universal in

both generative (an output is given, but not an input) and accepting (an input is given, but not an output) modes. SNP systems can also solve hard problems in feasible (polynomial to constant) time. We do not go into such details, and we refer to [7][8][9][16] and references therein.

SNP systems with standard rules (as introduced in their seminal paper) have neurons that can emit at most one pulse (the spike) each step, and either an input or output neuron connected to the environment, but not both. In [15], SNP systems were equipped with both an input and output neuron, and were known as *SNP transducers*. Furthermore, extended rules were introduced in [3] and [14], so that a neuron can produce more than one spike each step. The introduced *SNP modules* in [6] can then be seen as generalizations of SNP transducers: more than one spike can enter or leave the system, and more than one neuron can function as input or output neuron.

In this work we continue investigations on SNP modules. In particular we amend the problem introduced in the construction of [6], where SNP modules were used to simulate deterministic finite automata and state transducers. Our constructions also reduce the neurons for such SNP modules: from three neurons down to one. Our reduction relies on more involved superscripts, similar to some of the constructions in [12].

We also provide constructions for SNP modules simulating DFA with output. Establishing simulations between DFA with output and SNP modules, we are then able to generate automatic sequences. Such class of sequences contain, for example, a common and useful automatic sequence known as the Thue-Morse sequence. The Thue-Morse sequence, among others, play important roles in many areas of mathematics (e.g. number theory) and computer science (e.g. automata theory). Aside from DFA with output, another way to generate automatic sequences is by iterating morphisms. We invite the interested reader to [1] for further theories and applications related to automatic sequences.

This paper is organized as follows: Section 2 provides our preliminaries. Section 3 provides our results. Finally, section 4 provides our final remarks.

2 Preliminaries

It is assumed that the readers are familiar with the basics of membrane computing (a good introduction is [13] with recent results and information in the P systems webpage[1] and a recent handbook [17]) and formal language theory (available in many monographs). We only briefly mention notions and notations which will be useful throughout the paper.

2.1 Language theory and string notations

We denote the set of natural (counting) numbers as $\mathbb{N} = \{0, 1, 2, \ldots\}$. Let V be an alphabet, V^* is the set of all finite strings over V with respect to concatenation and

[1] `http://ppage.psystems.eu/`

the identity element λ (the empty string). The set of all non-empty strings over V is denoted as V^+ so $V^+ = V^* - \{\lambda\}$. We call V a singleton if $V = \{a\}$ and simply write a^* and a^+ instead of $\{a\}^*$ and $\{a\}^+$. If a is a symbol in V, then $a^0 = \lambda$, A regular expression over an alphabet V is constructed starting from λ and the symbols of V using the operations union, concatenation, and $+$. Specifically, (i) λ and each $a \in V$ are regular expressions, (ii) if E_1 and E_2 are regular expressions over V then $(E_1 \cup E_2)$, E_1E_2, and E_1^+ are regular expressions over V, and (iii) nothing else is a regular expression over V. The length of a string $w \in V^*$ is denoted by $|w|$. Unnecessary parentheses are omitted when writing regular expressions, and $E^+ \cup \{\lambda\}$ is written as E^*. We write the language generated by a regular expression E as $L(E)$. If V has k symbols, then $[w]_k = n$ is the base-k representation of $n \in \mathbb{N}$.

2.2 Deterministic finite automata

Definition 1. *A deterministic finite automaton (in short, a DFA) D, is defined by the 5-tuple $D = (Q, \Sigma, q_1, \delta, F)$, where:*

- $Q = \{q_1, \ldots, q_n\}$ *is a finite set of states,*
- $\Sigma = \{b_1, \ldots, b_m\}$ *is the input alphabet,*
- $\delta : Q \times \Sigma \to Q$ *is the transition function,*
- $q_1 \in Q$ *is the initial state,*
- $F \subseteq Q$ *is a set of final states.*

Definition 2. *A deterministic finite state transducer (in short, a DFST) with accepting states T, is defined by the 6-tuple $T = (Q, \Sigma, \Delta, q_1, \delta', F)$, where:*

- $Q = \{q_1, \ldots, q_n\}$ *is a finite set of states,*
- $\Sigma = \{b_1, \ldots, b_m\}$ *is the input alphabet,*
- $\Delta = \{c_1, \ldots, c_t\}$ *is the output alphabet,*
- $\delta' : Q \times \Sigma \to Q \times \Delta$ *is the transition function,*
- $q_1 \in Q$ *is the initial state,*
- $F \subseteq Q$ *is a set of final states.*

Definition 3. *A deterministic finite automaton with output (in short, a DFAO) M, is defined by the 6-tuple $M = (Q, \Sigma, \delta'', q_1, \Delta, \tau)$, where:*

- $Q = \{q_1, \ldots, q_n\}$ *is a finite set of states,*
- $\Sigma = \{b_1, \ldots, b_m\}$ *is the input alphabet,*
- $\delta'' : Q \times \Sigma \to Q$ *is the transition function,*
- $q_1 \in Q$ *is the initial state,*
- $\Delta = \{c_1, \ldots, c_t\}$ *is the output alphabet,*
- $\tau : Q \to \Delta$ *is the output function.*

A given DFAO M defines a function from Σ^* to Δ, denoted as $f_M(w) = \tau(\delta''(q_1, w))$ for $w \in \Sigma^*$. If $\Sigma = \{1, ..., k\}$, denoted as Σ_k, then M is a *k-DFAO.*

Definition 4. *A sequence, denoted as* $\mathbf{a} = (a_n)_{n\geq 0}$*, is* k*-automatic if there exists a* k*-DFAO,* M*, such that given* $w \in \Sigma_k^*$*,* $a_n = \tau(\delta''(q_1, w))$*, where* $[w]_k = n$*.*

Example 1. (Thue-Morse sequence) The Thue-Morse sequence $\mathbf{t} = (t_n)_{n\geq 0}$ counts the number of 1's (mod 2) in the base-2 representation of n. The 2-DFAO for $\mathbf{t}$ is given in Fig. 1. In order to generate $\mathbf{t}$, the 2-DFAO is in state q_1 with output 0, if the input bits seen so far sum to 0 (mod 2). In state q_2 with output 1, the 2-DFAO has so far seen input bits that sum to 1 (mod 2). For example, we have $t_0 = 0$, $t_1 = t_2 = 1$, and $t_3 = 0$.

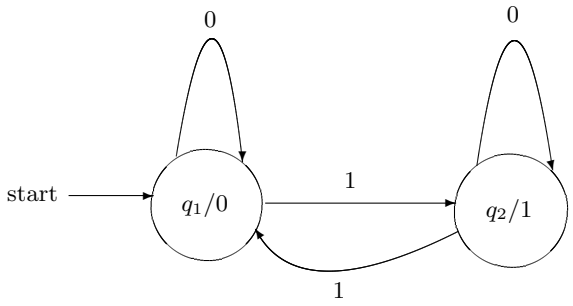


Fig. 1. 2-DFAO generating the Thue-Morse sequence.

2.3 Spiking neural P systems

Definition 5. *A spiking neural P system (in short, an SNP system) of degree* $m \geq 1$*, is a construct of the form* $\Pi = (\{a\}, \sigma_1, \ldots, \sigma_m, syn, in, out)$

where:

- $\{a\}$ is the singleton alphabet (a is called *spike*);
- $\sigma_1, \ldots, \sigma_m$ are *neurons* of the form $\sigma_i = (n_i, R_i), 1 \leq i \leq m$, where:
  - $n_i \geq 0$ is the *initial number of spikes* inside σ_i;
  - R_i is a finite *set of rules* of the general form: $E/a^c \rightarrow a^p; d$, where E is a regular expression over $\{a\}$, $c \geq 1$, with $p, d \geq 0$, and $c \geq p$; if $p = 0$, then $d = 0$ and $L(E) = \{a^c\}$;
- $syn \subseteq \{1, \ldots, m\} \times \{1, \ldots, m\}$, with $(i, i) \notin syn$ for $1 \leq i \leq m$ (*synapses*);
- $in, out \in \{1, \ldots, m\}$ indicate the *input* and *output* neurons, respectively.

A rule $E/a^c \rightarrow a^p; d$ in neuron σ_i (we also say neuron i or simply σ_i if there is no confusion) is called a *spiking rule* if $p \geq 1$. If $p = 0$, then $d = 0$ and $L(E) = \{a^c\}$, so that the rule is written simply as $a^c \rightarrow \lambda$, known as a *forgetting rule*. If a spiking rule has $L(E) = \{a^c\}$, we simply write it as $a^c \rightarrow a^p; d$. The systems from the original paper [7], with rules of the form $E/a^c \rightarrow a; d$ and $a^c \rightarrow \lambda$, are referred to

as *standard* systems with *standard rules*. The extended rules (i.e. $p \geq 1$) used in this work are referred to as SNP systems with extended rules in other literature, e.g. [6], [14], [16].

The rules are applied as follows: If σ_i contains k spikes, $a^k \in L(E)$ and $k \geq c$, then the rule $E/a^c \rightarrow a^p; d \in R_i$ with $p \geq 1$, is enabled and can be applied. Rule application means consuming c spikes, so only $k - c$ spikes remain in σ_i. The neuron produces p spikes (also referred to as *spiking*) after d time units, to every σ_j where $(i, j) \in syn$. If $d = 0$ then the p spikes arrive at the same time as rule application. If $d \geq 1$ and the time of rule application is t, then during the time sequence $t, t + 1, \ldots, t + d - 1$ the neuron is *closed*. If a neuron is closed, it cannot receive spikes, and all spikes sent to it are lost. Starting at times $t + d$ and $t + d + 1$, the neuron becomes *open* (i.e., can receive spikes), and can apply rules again, respectively. Applying a forgetting rule means producing no spikes. Note that a forgetting rule is never delayed since $d = 0$.

SNP systems operate under a global clock, i.e. they are *synchronous*. At every step, every neuron that can apply a rule must do so. It is possible that at least two rules $E_1/a^{c_1} \rightarrow a^{p_1}; d_1$ and $E_2/a^{c_2} \rightarrow a^{p_2}; d_2$, with $L(E_1) \cap L(E_2) \neq \emptyset$, can be applied at the same step. The system *nondeterministically* chooses exactly one rule to apply. The system is *globally parallel* (each neuron can apply a rule) but is *locally sequential* (a neuron can apply at most one rule).

A *configuration* or state of the system at time t can be described by $C_t = \langle r_1/t_1, \ldots, r_m/t_m \rangle$ for $1 \leq i \leq m$: Neuron i contains $r_i \geq 0$ spikes and it will open after $t_i \geq 0$ time steps. The initial configuration of the system is therefore $C_0 = \langle n_1/0, \ldots, n_m/0 \rangle$, where all neurons are initially open. Rule application provides us a *transition* from one configuration to another. A *computation* is any (finite or infinite) sequence of transitions, starting from a C_0. A halting computation is reached when all neurons are open and no rule can be applied.

If σ_{out} produces i spikes in a step, we associate the symbol b_i to that step. In particular, the system (using rules in its output neuron) generates strings over $\Sigma = \{p_1, \ldots, p_m\}$, for every rule $r_\ell = E_\ell/a^{j_\ell} \rightarrow a^{p_\ell}; d_\ell, 1 \leq \ell \leq m$, in σ_{out}. From [3] we can have two cases: associating b_0 (when no spikes are produced) with a symbol, or as λ. In this work and as in [6], we only consider the latter.

Definition 6. *A spiking neural P module (in short, an SNP module) of degree $m \geq 1$, is a construct of the form* $\Pi = (\{a\}, \sigma_1, \ldots, \sigma_m, syn, N_{in}, N_{out})$

where

- $\{a\}$ is the singleton alphabet (a is called *spike*);
- $\sigma_1, \ldots, \sigma_m$ are *neurons* of the form $\sigma_i = (n_i, R_i), 1 \leq i \leq m$, where:
  - $n_i \geq 0$ is the *initial number of spikes* inside σ_i;
  - R_i is a finite *set of rules* of the general form: $E/a^c \rightarrow a^p$, where E is a regular expression over $\{a\}$, $c \geq 1$, and $p \geq 0$, with $c \geq p$; if $p = 0$, then $L(E) = \{a^c\}$
- $syn \subseteq \{1, \ldots, m\} \times \{1, \ldots, m\}$, with $(i, i) \notin syn$ for $1 \leq i \leq m$ (*synapses*);

- $N_{in}, N_{out} (\subseteq \{1, 2, \ldots, m\})$ indicate the *sets of input* and *output* neurons, respectively.

In [15], SNP transducers operated on strings over a binary alphabet as well considering b_0 as a symbol. SNP modules, first introduced in [6], are a special type of SNP systems with extended rules, and generalize SNP transducers.

SNP modules behave in the usual way as SNP systems, except that spiking and forgetting rules now both contain no delays. In contrast to SNP systems, SNP modules have the following *distinguishing feature*: at each step, each input neuron $\sigma_i, i \in N_{in}$, takes as input *multiple copies* of a from the environment (in short, Env); Each output neuron $\sigma_o, o \in N_{out}$, produces p spikes to Env, if a rule $E/a^c \to a^p$ is applied in σ_o; Note that $N_{in} \cap N_{out}$ is not necessarily empty.

3 Main results

In this section we amend and improve constructions given in [6] to simulate DFA and DFST using SNP modules. Then, k-DFAO are also simulated with SNP modules. Lastly, SNP modules are related to k-automatic sequences.

3.1 DFA and DFST simulations

We briefly recall the constructions from theorem 8 and 9 of [6] for SNP modules simulating DFAs and DFSTs. The constructions for both DFAs and DFSTs have a similar structure, which is shown in Fig. 2. For neurons 1 and 2 in Fig. 2, the spikes and rules for DFA and DFST simulation are equal, so the constructions only differ for the contents of neuron 3. Let $D = (Q, \Sigma, \delta, q_1, F)$ be a DFA, where $\Sigma = \{b_1, \ldots, b_m\}$, $Q = \{q_1, \ldots, q_n\}$. The construction for theorem 8 of [6] for an SNP Module Π_D simulating D is as follows:

$$\Pi_D = (\{a\}, \sigma_1, \sigma_2, \sigma_3, syn, \{3\}, \{3\}),$$

where

- $\sigma_1 = \sigma_2 = (n, \{a^n \to a^n\})$,
- $\sigma_3 = (n, \{a^{2n+i+k}/a^{2n+i+k-j} \to a^j | \delta(q_i, b_k) = q_j\})$,
- $syn = \{(1,2), (2,1), (1,3)\}$.

The structure for Π_D is shown in Fig. 2. Note that $n, m \in \mathbb{N}$, are fixed numbers, and each state $q_i \in Q$ is represented as a^i spikes in σ_3, for $1 \leq i \leq n$. For each symbol $b_k \in \Sigma$, the representation is a^{n+k}. The operation of Π_D is as follows: σ_1 and σ_2 interchange a^n spikes at every step, while σ_1 also sends a^n spikes to σ_3.

Suppose that D is in state q_i and will receive input b_k, so that σ_3 of Π_D has a^i spikes and will receive a^{n+k} spikes. In the next step, σ_3 will collect a^n spikes from σ_1, a^{n+k} spikes from Env, so that the total spikes in σ_3 is a^{2n+i+k}. A rule in σ_3 with $L(E) = \{a^{2n+i+k}\}$ is applied, and the rule consumes $2n + i + k - j$ spikes,

therefore leaving only a^j spikes. A single state transition $\delta(q_i, b_k) = q_j$ is therefore simulated.

With a 1-step delay, Π_D receives a given input $w = b_{i_1}, \ldots, b_{i_r}$ in Σ^* and produces a sequence of states $z = q_{i_1}, \ldots, q_{i_r}$ (represented by $a^{i_1}, \ldots, a^{i_r}$) such that $\delta(q_{i_\ell}, b_{i_\ell}) = q_{i_{\ell+1}}$, for each $\ell = 1, \ldots, r$ where $q_{i_1} = q_1$. Then, w is accepted by D (i.e. $\delta(q_1, w) \in F$) iff $z = \Pi_D(w)$ ends with a state in F (i.e. $q_{i_r} \in F$). Let the language accepted by Π_D be defined as:

$$L(\Pi_D) = \{w \in \Sigma^* | \Pi_D(w) \in Q^* F\}.$$

Then, the following is theorem 8 from [6]

Theorem 1. *(Ibarra et al [6]) Any regular language L can be expressed as $L = L(\Pi_D)$ for some SNP module Π_D.*

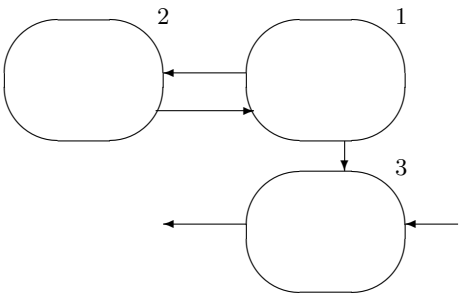


Fig. 2. Structure of SNP modules from [6] simulating DFAs and DFSTs.

The simulation of DFSTs requires a slight modification of the DFA construction. Let $T = (Q, \Sigma, \Delta, \delta', q_1, F)$ be a DFST, where $\Sigma = \{b_1, \ldots, b_k\}$, $\Delta = \{c_1, \ldots, c_t\}$, $Q = \{q_1, \ldots, q_n\}$. We construct the following SNP module simulating T:

$$\Pi_T = (\{a\}, \sigma_1, \sigma_2, \sigma_3, syn, \{3\}, \{3\}),$$

where:

- $\sigma_1 = \sigma_2 = (n, \{a^n \to a^n\})$,
- $\sigma_3 = (n, \{a^{2n+i+k+t}/a^{2n+i+k+t-j} \to a^{n+s} | \delta'(q_i, b_k) = (q_j, c_s)\})$,
- $syn = \{(1,2), (2,1), (1,3)\}$.

The structure for Π_T is shown in Fig. 2. Note that $n, m, t \in \mathbb{N}$ are fixed numbers. For $1 \leq i \leq n, 1 \leq s \leq t, 1 \leq k \leq m$: each state $q_i \in Q$, each input symbol $b_k \in \Sigma$, and each output symbol $c_s \in \Delta$, is represented by a^i, a^{n+t+k}, and a^{n+s}, respectively.

The operation of Π_T given an input $w \in \Sigma^*$ is in parallel to the operation of Π_D; the difference is that the former produces a $c_s \in \Delta$, while the latter produces

a $q_i \in Q$. From the construction of Π_T and the claim in Theorem 1, the following is Theorem 9 from [6]:

Theorem 2. *(Ibarra et al[6]) Any finite transducer T can be simulated by some SNP module Π_T.*

The previous constructions from [6] on simulating DFAs and DFSTs have however, the following technical problem:

Suppose we are to simulate DFA D with at least two transitions, (1) $\delta(q_i, b_k) = q_j$, and (2) $\delta(q_{i'}, b_{k'}) = q_{j'}$. Let $j \neq j', i = k'$, and $k = i'$. The SNP module Π_D simulating D then has at least two rules in σ_3: $r_1 = a^{2n+i+k}/a^{2n+i+k-j} \rightarrow a^j$, (simulating (1)) and $r_2 = a^{2n+i'+k'}/a^{2n+i'+k'-j'} \rightarrow a^{j'}$ (simulating (2)).

Observe that $2n+i+k = 2n+i'+k'$, so that in σ_3, the regular expression for r_1 is exactly the regular expression for r_2. We therefore have a nondeterministic rule selection in σ_3. However, D being a DFA, transitions to two different states q_j and $q_{j'}$. Therefore, Π_D is a nondeterministic SNP module that can, at certain steps, incorrectly simulate the DFA D. This nondeterminism also occurs in the DFST simulation. An illustration of the problem is given in example 2.

Example 2. We modify the 2-DFAO in Fig. 1 into a DFA in Fig. 3 as follows: Instead of $\Sigma = \{0, 1\}$, we have $\Sigma = \{1, 2\}$; We maintain $n = m = 2$, however, the transitions are swapped, so in Fig. 3 we have the following two (among four) transitions: $\delta(q_1, 2) = q_2$, and $\delta(q_2, 1) = q_1$. These two transitions cause the nondeterministic problem for the SNP module given in Fig. 4. The problem concerns the simulation of the two previous transitions using rules $a^7/a^5 \rightarrow a^2$ and $a^7/a^6 \rightarrow a$ in σ_3, which can be nondeterministically applied: if σ_3 contains a^2 spikes and receives a^3 from Env (representing input 1 for the DFA), at the next step σ_3 will have a^7 spikes, allowing the possibility of an incorrect simulation.

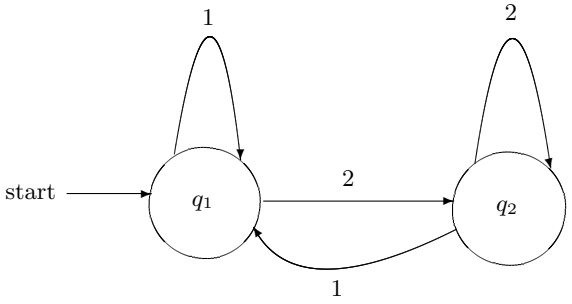


Fig. 3. DFA with incorrect simulation by the SNP module in Fig. 4.

Next, we amend the problem and modify the constructions for simulating DFAs and DFSTs in SNP modules. Given a DFA D, we construct an SNP module Π'_D simulating D as follows:

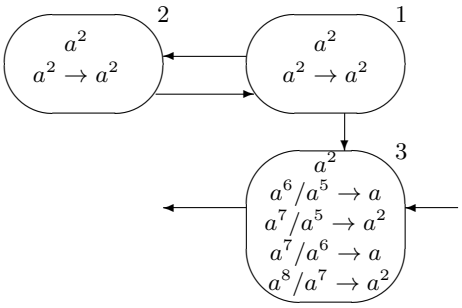


Fig. 4. SNP module with incorrect simulation of the DFA in Fig. 3.

$$\Pi'_D = (\{a\}, \sigma_1, syn, \{1\}, \{1\}),$$

where

- $\sigma_1 = (1, \{a^{k(2n+1)+i}/a^{k(2n+1)+i-j} \to a^j | \delta(q_i, b_k) = q_j\})$,
- $syn = \emptyset$.

We have Π_D containing only 1 neuron, which is both the input and output neuron. Again, $n, m \in \mathbb{N}$ are fixed numbers. Each state q_i is again represented as a^i spikes, for $1 \leq i \leq n$. Each symbol $b_k \in \Sigma$ is now represented as $a^{k(2n+1)}$ spikes. The operation of Π'_D is as follows: neuron 1 starts with a^1 spike, representing q_1 in D. Suppose that D is in some state q_i, receives input b_k, and transitions to q_j in the next step. We then have Π'_D combining $a^{k(2n+1)}$ spikes from Env with a^i spikes, so that a rule with regular expression $a^{k(2n+1)+i}$ is applied, producing a^j spikes to Env. After applying such rule, a^j spikes remain in σ_1, and a single transition of D is simulated.

Note that the construction for Π'_D does not involve nondeterminism, and hence the previous technical problem: Let D have at least two transitions, (1) $\delta(q_i, b_k) = q_j$, and (2) $\delta(q_{i'}, b_{k'}) = q_{j'}$. We again let $j \neq j', i = k'$, and $k = i'$. Note that being a DFA, we have $i \neq k$. Observe that $k(2n+1) + i \neq k'(2n+1) + i'$. Therefore, Π'_D is deterministic, and has two rules r_1 and r_2 correctly simulating (1) and (2), respectively. We now have the following result.

Theorem 3. *Any regular language L can be expressed as $L = L(\Pi'_D)$ for some 1-neuron SNP module Π'_D*

For a given DFST T, we construct an SNP module Π'_T simulating T as follows:

$$\Pi'_T = (\{a\}, \sigma_1, syn, \{1\}, \{1\}),$$

where

- $\sigma_1 = (1, \{a^{k(2n+1)+i+t}/a^{k(2n+1)+i+t-j} \to a^{n+s} | \delta'(q_i, b_k) = (q_j, c_s)\})$,
- $syn = \emptyset$.

We also have Π'_T as a 1-neuron SNP module similar to Π'_D. Again, $n, m, t \in \mathbb{N}$ are fixed numbers, and for each $1 \leq i \leq n, 1 \leq k \leq m$, and $1 \leq s \leq t$: each state $q_i \in Q$, each input symbol $b_k \in \Sigma$, and each output symbol $c_s \in \Delta$, is represented as $a^i, a^{k(2n+1)+t}$, and a^{n+s} spikes, respectively. The functioning of Π'_T is in parallel to Π'_D. Unlike Π_T, Π'_T is deterministic and correctly simulates T. We now have the next result.

Theorem 4. *Any finite transducer T can be simulated by some 1-neuron SNP module Π'_T.*

3.2 *k*-DFAO simulation and generating automatic sequences

Next, we modify the construction from Theorem 4 specifically for k-DFAOs by: (a) adding a second neuron σ_2 to handle the spikes from σ_1 until end of input is reached, and (b) using σ_2 to output a symbol once the end of input is reached. Also note that in k-DFAOs we have $t \leq n$, since each state must have exactly one output symbol associated with it. Observing k-DFAOs from Definition 3 and DFSTs from Definition 2, we find a subtle but interesting distinction as follows:

The output of the state after reading the last symbol in the input is the requirement from a k-DFAO, i.e. for every w over some Σ_k, the k-DFAO produces only one $c \in \Delta$ (recall the output function τ); In contrast, the output of DFSTs is a sequence of $Q \times \Delta$ (states and symbols), since $\delta''(q_i, b_k) = (q_j, c_s)$. Therefore, if we use the construction in Theorem 4 for DFST in order to simulate k-DFAOs, we must ignore the first $|w| - 1$ symbols in the output of the system in order to obtain the single symbol we require.

For a given k-DFAO $M = (Q, \Sigma, \Delta, \delta'', q_1, \tau)$, we have $1 \leq i, j \leq n$, $1 \leq s \leq t$, and $1 \leq k \leq m$. Construction of an SNP module Π_M simulating M, is as follows:

$$\Pi = (\{a\}, \sigma_1, \sigma_2, syn, \{1\}, \{2\}),$$

where

- $\sigma_1 = (1, R_1), \sigma_2 = (0, R_2)$,
- $R_1 = \{a^{k(2n+1)+i+t}/a^{k(2n+1)+i+t-j} \rightarrow a^{n+s} | \delta''(q_i, b_k) = q_j, \tau(q_j) = c_s\}$ $\cup \{a^{m(2n+1)+n+t+i} \rightarrow a^{m(2n+1)+n+t+i} | 1 \leq i \leq n\}$,
- $R_2 = \{a^{n+s} \rightarrow \lambda | \tau(q_i) = c_s\} \cup \{a^{m(2n+1)+n+t+i} \rightarrow a^{n+s} | \tau(q_i) = c_s\}$,
- $syn = \{(1, 2)\}$.

We have Π_M as a 2-neuron SNP module, and $n, m, t \in \mathbb{N}$ are fixed numbers. Each state $q_i \in Q$, each input symbol $b_k \in \Sigma$, and each output symbol $c_s \in \Delta$, is represented as a^i, $a^{k(2n+1)+t}$, and a^{n+s} spikes, respectively. In this case however, we add an end-of-input symbol \$ (represented as $a^{m(2n+1)+n+t}$ spikes) to the input string, i.e. if $w \in \Sigma^*$, the input for Π_M is $w\$$.

For any $b_k \in \Sigma$, σ_1 of Π_M functions in parallel to σ_1 of Π'_D and Π'_T, i.e. every transition $\delta''(q_i, b_k) = q_j$ is correctly simulated by σ_1. The difference however lies during the step when \$ enters σ_1, indicating the end of the input. Suppose during this step σ_1 has a^i spikes, then those spikes are combined with the

$a^{m(2n+1)+n+t}$ spikes from Env. Then, one of the n rules in σ_1 with regular expression $a^{m(2n+1)+n+t+i}$ is applied, sending $a^{m(2n+1)+n+t+i}$ spikes to σ_2.

The first function of σ_2 is to erase, using forgetting rules, all a^{n+s} spikes it receives from σ_1. Once σ_2 receives $a^{m(2n+1)+n+t+i}$ spikes from σ_1, this means that the end of the input has been reached. The second function of σ_2 is to produce a^{n+s} spikes exactly once, by using one rule of the form $a^{m(2n+1)+n+t+i} \to a^{n+s}$. The output function $\tau(\delta''(q_1, w\$))$ is therefore correctly simulated. We can then have the following result.

Theorem 5. *Any k-DFAO M can be simulated by some 2-neuron SNP module Π_M.*

Next, we establish the relationship of SNP modules and automatic sequences.

Theorem 6. *Let a sequence $\mathbf{a} = (a_n)_{n\geq 0}$ be k-automatic, then it can be generated by a 2-neuron SNP module Π.*

k-automatic sequences have several interesting robustness properties. One property is the capability to produce the same output sequence given that the input string is read in reverse, i.e. for some finite string $w = a_1 a_2 \dots a_n$, we have $w^R = a_n a_{n-1} \dots a_2 a_1$. It is known (e.g. [1]) that if $(a_n)_{n\geq 0}$ is a k-automatic sequence, then there exists a k-DFAO M such that $a_n = \tau(\delta''(q_0, w^R))$ for all $n \geq 0$, and all $w \in \Sigma_k^*$, where $[w]_k = n$. Since the construction of Theorem 5 simulates both δ'' and τ, we can include robustness properties as the following result shows.

Theorem 7. *Let $\mathbf{a} = (a_n)_{n\geq 0}$ be a k-automatic sequence. Then, there is some 2-neuron SNP module Π where $\Pi(w^R\$) = a_n$, $w \in \Sigma_k^*$, $[w]_k = n$, and $n \geq 0$.*

An illustration of the construction for Theorem 5 is given in example 3.

Example 3. (SNP module simulating the 2-DFAO generating the Thue-Morse sequence) The SNP module is given in Fig. 5, and we have $n = m = t = 2$. Based on the construction for Theorem 5, we associate symbols 0 and 1 with a^7 and a^{12} spikes, respectively. The end-of-input symbol \$, q_1, and q_2 are associated with a^{14}, a, and a^2 spikes, respectively (with a and a^2 appearing only inside σ_1).

The 2-DFAO in Fig. 1 has four transitions, and rules r_1 to r_4 simulate the four transitions. Rules r_5 and r_6 are only applied when \$ enters the system. Rules r_7 and r_8 are applied to "clean" the spikes from σ_1 while \$ is not yet encountered by the system. Rules r_8 and r_9 produce the correct output, simulating τ.

4 Final Remarks

In [3], strict inclusions for the types of languages characterized by SNP systems with extended rules having one, two, and three neurons were given. Then in [15], it was shown that there is no SNP transducer that can compute nonerasing and

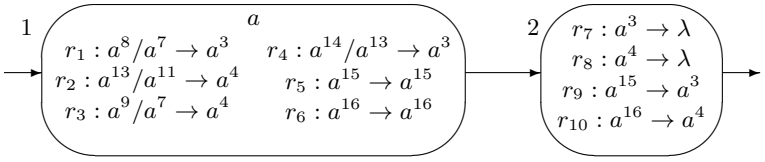


Fig. 5. SNP module simulating the 2-DFAO in Fig. 1.

nonlength preserving morphisms: for all $a \in \Sigma$, the former is a morphism h such that $h(a) \neq \lambda$, while the latter is a morphism h where $|h(a)| \geq 2$. It is known (e.g. in [1]) that the Thue-Morse morphism is given by $\mu(0) = 01$ and $\mu(1) = 10$. It is interesting to further investigate SNP modules with respect to other classes of sequences, morphisms, and finite transition systems. Another technical note is that in [15] a time step without a spike entering or leaving the system was considered as a symbol of the alphabet, while in [6] (and in this work) it was considered as λ.

We also leave as an open problem a more systematic analysis of input/output encoding size and system complexity: in the constructions for Theorems 3 to 4, SNP modules consist of only one neuron for each module, compared to three neurons in the constructions of [6]. However, the encoding used in our Theorems is more involved, i.e. with multiplication and addition of indices (instead of simply addition of indices in [6]). On the practical side, SNP modules might also be used for computing functions, as well as other tasks involving (streams of) input-output transformations. Practical applications might include image modification or recognition, sequence analyses, online algorithms, et al.

Some preliminary work on SNP modules and morphisms was given in [2]. From finite sequences, it is interesting to extend SNP modules to infinite sequences. In [4], extended SNP systems[2] were used as acceptors in relation to ω-languages. SNP modules could also be a way to "go beyond Turing" by way of *interactive computations*, as in interactive components or transducers given in [5]. While the syntax of SNP modules may prove sufficient for these "interactive tasks", or at least only minor modifications, a (major) change in the semantics is probably necessary.

Acknowledgements

Cabarle is supported by a scholarship from the DOST-ERDT of the Philippines. Adorna is funded by a DOST-ERDT grant and the Semirara Mining Corp. professorial chair of the College of Engineering, UP Diliman. M.J. Pérez-Jiménez acknowledges the support of the Project TIN2012-37434 of the "Ministerio de

[2] or ESNP systems, in short, are generalizations of SNP systems almost to the point of becoming tissue P systems. ESNP systems are thus generalizations also of (and not to be confused with) SNP systems with extended rules.

Economía y Competitividad" of Spain, co-financed by FEDER funds. Fruitful discussions with Miguel Ángel Martínez-del Amor are also acknowledged.

References

1. Allouche, J-P., Shallit, J.: Automatic Sequences: Theory, Applications, Generalizations. Cambridge University Press (2003)
2. Cabarle, F.G.C, Buño, K.C., Adorna, H.N.: Spiking Neural P Systems Generating the Thue-Morse Sequence. Asian Conference on Membrane Computing 2012 preproceedings, 15-18 Oct, Wuhan, China (2012)
3. Chen, H., Ionescu, M., Ishdorj, T-O., Păun, A., Păun, Gh., Pérez-Jiménez, M.J.: Spiking neural P systems with extended rules: universality and languages. Natural Computing, vol. 7, pp. 147-166 (2008)
4. Freund, R., Oswald, M.: Regular ω-languages defined by finite extended spiking neural P systems. Fundamenta Informaticae, vol. 81(1-2), pp. 65-73 (2008)
5. Goldin, D., Smolka, S., Wegner, P. (Eds.): Interactive Computation: The New Paradigm. Springer-Verlag (2006)
6. Ibarra, O., Peréz-Jiménez, M.J., Yokomori, T.: On spiking neural P systems. Natural Computing, vol. 9, pp. 475-491 (2010)
7. Ionescu, M., Păun, Gh., Yokomori, T.: Spiking Neural P Systems. Fundamenta Informaticae, vol. 71(2,3), pp. 279-308 (2006)
8. Leporati, A., Zandron, C., Ferretti, C., Mauri, G.: Solving Numerical NP-Complete Problems with Spiking Neural P Systems. Eleftherakis et al. (Eds.): WMC8 2007, LNCS 4860, pp. 336-352 (2007)
9. Pan, L., Păun, Gh., Pérez-Jiménez, M.J.: Spiking neural P systems with neuron division and budding. Science China Information Sciences. vol. 54(8) pp. 1596-1607 (2011)
10. Maass, W.: Computing with spikes, in: Special Issue on Foundations of Information Processing of TELEMATIK, vol. 8(1) pp. 32-36, (2002)
11. Maass, W., Bishop, C. (Eds.): Pulsed Neural Networks, MIT Press, Cambridge (1999)
12. Neary, T.: A Boundary between Universality and Non-universality in Extended Spiking Neural P Systems. LATA 2010, LNCS 6031, pp. 475-487 (2010)
13. Păun, Gh.: Membrane Computing: An Introduction. Springer-Verlag (2002)
14. Păun, A., Păun, Gh.: Small universal spiking neural P systems. BioSystems, vol. 90(1), pp. 48-60 (2007)
15. Păun, Gh., Pérez-Jiménez, M.J., Rozenberg, G.: Computing Morphisms by Spiking Neural P Systems. Int'l J. of Foundations of Computer Science. vol. 8(6) pp. 1371-1382 (2007)
16. Păun, Gh., Pérez-Jiménez, M.J.: Spiking Neural P Systems. Recent Results, Research Topics. A. Condon et al. (eds.), Algorithmic Bioprocesses, Springer-Verlag (2009)
17. Păun, Gh., Rozenberg, G., Salomaa, A. (Eds) The Oxford Handbook of Membrane Computing, OUP (2010)

Asynchronous Spiking Neural P Systems with Structural Plasticity*

Francis George C. Cabarle[1], Henry N. Adorna[1], Mario J. Pérez-Jiménez[2]

[1]Algorithms & Complexity Lab, Department of Computer Science
University of the Philippines Diliman
Diliman, 1101, Quezon City, Philippines;
[2]Department of Computer Science and AI
Universidad de Sevilla
Avda. Reina Mercedes s/n, 41012, Sevilla, Spain
fccabarle@up.edu.ph, hnadorna@dcs.upd.edu.ph, marper@us.es

Summary. Spiking neural P (in short, SNP) systems are computing devices inspired by biological spiking neurons. In this work we consider SNP systems with structural plasticity (in short, SNPSP systems) working in the asynchronous (in short, *asyn* mode). SNPSP systems represent a class of SNP systems that have dynamic synapses, i.e. neurons can use plasticity rules to create or remove synapses. We prove that for *asyn* mode, bounded SNPSP systems (where any neuron produces at most one spike each step) are not universal, while unbounded SNPSP systems with weighted synapses (a weight associated with each synapse allows a neuron to produce more than one spike each step) are universal. The latter systems are similar to SNP systems with extended rules in *asyn* mode (known to be universal) while the former are similar to SNP systems with standard rules only in *asyn* mode (conjectured not to be universal). Our results thus provide support to the conjecture of the still open problem.

Key words: Membrane computing, Spiking neural P systems, Structural plasticity, Asynchronous systems, Turing universality

1 Introduction

Spiking neural P systems (in short, SNP systems) are parallel, distributed, and nondeterministic devices introduced into the are of membrane computing in [7]. Neurons are often drawn as ovals, and they process only one type of object, the *spike* signal represented by a. Synapses between neurons are the arcs between ovals: neurons are then placed on the vertices of a directed graph. Since their introduction, several lines of investigations have been produced, e.g. (non)deterministic

* An improved version of this article will appear at the 14th Unconventional Computation and Natural Computation (2015), Auckland, New Zealand.

computing power in [7][14]; language generation in [4]; function computing devices in [11]; solving computationally hard problems in [9]. Many neuroscience inspirations have also been included for computing use, producing several variants (to which the previous investigation lines are also applied), e.g. use of weighted synapses [16], neuron division and budding [9], the use of astrocytes [10]. Furthermore, many restrictions have been applied to SNP systems (and variants), e.g. asynchronous SNP systems as in [6], [3], and [15], and sequential SNP systems as in [6].

In this work the variant we consider are SNP systems with structural plasticity, in short, SNPSP systems. SNPSP systems were first introduced in [1], then extended and improved in [2]. The biological motivation for SNPSP systems is structural plasticity, one form of neural plasticity, and distinct from the more common functional (Hebbian) plasticity. SNPSP systems represent a class of SNP systems using plasticity rules: synapses can be created or deleted so the synapse graph is dynamic. The restriction we apply to SNPSP systems is asynchronous operation: imposing synchronization on biological functions is sometimes "too much", i.e. not alway realistic. Hence, the asynchronous mode of operation is interesting to consider. Such restriction is also interesting mathematically, and we refer the readers again to [6], [3], and [15] for further details.

In this work we prove that (*i*) asynchronous bounded (i.e. there exists a bound on the number of stored spikes in any neuron) SNPSP systems are not universal, (*ii*) asynchronous weighted (i.e. a positive integer weight is associated with each synapse) SNPSP systems, even under a normal form (provided below), are universal. The open problem in [3] whether asynchronous bounded SNP systems with standard rules are universal is conjectured to be false. Also, asynchronous SNP systems with extended rules are known to be universal [5]. Our results provide some support to the conjecture, since neurons in SNPSP systems produce at most one spike each step (similar to standard rules) while synapses with weights function similar to extended rules (more than one spike can be produced each step). This work is organized as follows: Section 2 provides preliminaries for our results; syntax and semantics of SNPSP systems are given in Section 3; our (non)universality results are given in Section 4. Lastly, we provide final remarks and further directions in Section 5.

2 Preliminaries

It is assumed that the readers are familiar with the basics of membrane computing (a good introduction is [13] with recent results and information in the P systems webpage (`http://ppage.psystems.eu/`) and a recent handbook [14]) and formal language theory (available in many monographs). We only briefly mention notions and notations which will be useful throughout the paper.

We denote the set of positive integers as $\mathbb{N} = \{1, 2, \ldots\}$. Let V be an alphabet, V^* is the set of all finite strings over V with respect to concatenation and the

identity element λ (the empty string). The set of all non-empty strings over V is denoted as V^+ so $V^+ = V^* - \{\lambda\}$. If $V = \{a\}$, we simply write a^* and a^+ instead of $\{a\}^*$ and $\{a\}^+$. If a is a symbol in V, we write $a^0 = \lambda$ and we write the language generated by a regular expression E over V as $L(E)$.

In proving computational universality, we use the notion of register machines. A register machine is a construct $M = (m, I, l_0, l_h, R)$, where m is the number of registers, I is the set of instruction labels, l_0 is the start label, l_h is the halt label, and R is the set of instructions. Every label $l_i \in I$ uniquely labels only one instruction in R. Register machine instructions have the following forms:

- $l_i : (\texttt{ADD}(r), l_j, l_k)$, increase n by 1, then nondeterministically go to l_j or l_k;
- $l_i : (\texttt{SUB}(r), l_j, l_k)$, if $n \geq 1$, then subtract 1 from n and go to l_j, otherwise perform no operation on r and go to l_k;
- $l_h : \texttt{HALT}$, the halt instruction.

Given a register machine M, we say M computes or generates a number n as follows: M starts with all its registers empty. The register machine then applies its instructions starting with the instruction labeled l_0. Without loss of generality, we assume that l_0 labels an `ADD` instruction, and that the content of the output register is never decremented, only added to during computation, i.e. no `SUB` instruction is applied to it. If M reaches the halt instruction l_h, then the number n stored during this time in the first (also the output) register is said to be computed by M. We denote the set of all numbers computed by M as $N(M)$. It was proven that register machines compute all sets of numbers computed by a Turing machine, therefore characterizing NRE [8]. A strongly monotonic register machine is one restricted variant: it has only one register which is also the output register. The register initially stores zero, and can only be incremented by 1 at each step. Once the machine halts, the value stored in the register is said to be computed. It is known that strongly monotonic register machines characterize $SLIN$, the family of length sets of regular languages.

3 Spiking neural P systems with structural plasticity

In this section we define SNP systems with structural plasticity. Initial motivations and results for SNP systems are included in the seminal paper in [7]. A *spiking neural P system with structural plasticity* (SNPSP system) of degree $m \geq 1$ is a construct of the form $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, out)$, where:

- $O = \{a\}$ is the singleton alphabet (a is called spike);
- $\sigma_1, \ldots, \sigma_m$ are neurons of the form $(n_i, R_i), 1 \leq i \leq m$; $n_i \geq 0$ indicates the initial number of spikes in σ_i; R_i is a finite rule set of σ_i with two forms:
  1. Spiking rule: $E/a^c \rightarrow a$, where E is a regular expression over O, $c \geq 1$;
  2. Plasticity rule: $E/a^c \rightarrow \alpha k(i, N)$, where E is a regular expression over O, $c \geq 1$, $\alpha \in \{+, -, \pm, \mp\}$, $k \geq 1$, and $N \subseteq \{1, \ldots, m\} - \{i\}$;

- $syn \subseteq \{1,\ldots,m\} \times \{1,\ldots,m\}$, with $(i,i) \notin syn$ for $1 \leq i \leq m$ (synapses between neurons);
- $out \in \{1,\ldots,m\}$ indicate the output neuron.

Given neuron σ_i (we also say neuron i or simply σ_i) we denote the set of neuron labels with σ_i as their presynaptic (postsynaptic, resp.) neuron as $pres(i)$, i.e. $pres(i) = \{j|(i,j) \in syn\}$ (as $pos(i) = \{j|(j,i) \in syn\}$, resp.). Spiking rule semantics in SNPSP systems are similar with SNP systems in [7]. In this work we do not use forgetting rules (rules of the form $a^s \to \lambda$) or rules with delays of the form $E/a^c \to a; d$ for some $d \geq 1$. Spiking rules are applied as follows: If neuron σ_i contains b spikes and $a^b \in L(E)$, with $b \geq c$, then a rule $E/a^c \to a \in R_i$ can be applied. Applying such a rule means consuming c spikes from σ_i, thus only $b-c$ spikes remain in σ_i. Neuron i sends one spike to every neuron with label in $pres(i)$ at the same step as rule application. A nonzero delay d means that if σ_i spikes at step t, then neurons receive the spike at $t+d$. Spikes sent to σ_i from t to $t+d-1$ are lost (i.e. σ_i is closed), and σ_i can receive spikes (i.e. σ_i is open) and apply a rule again at $t+d$ and $t+d+1$, respectively. If a rule $E/a^c \to a$ has $L(E) = \{a^c\}$, we simply write this as $a^c \to a$.

Plasticity rules are applied as follows. If at step t we have that σ_i has $b \geq c$ spikes and $a^b \in L(E)$, a rule $E/a^c \to \alpha k(i,N) \in R_i$ can be applied. The set N is a collection of neurons to which σ_i can connect to or disconnect from using the applied plasticity rule. The rule application consumes c spikes and performs one of the following, depending on α:

- If $\alpha := +$ and $N - pres(i) = \emptyset$, or if $\alpha := -$ and $pres(i) = \emptyset$, then there is nothing more to do, i.e. c spikes are consumed but no synapses are created or removed. Notice that with these semantics, a plasticity rule functions similar to a forgetting rule, i.e. the former can be used to consume spikes without producing any spike.
- for $\alpha := +$, if $|N - pres(i)| \leq k$, deterministically create a synapse to every σ_l, $l \in N_j - pres(i)$. If however $|N - pres(i)| > k$, nondeterministically select k neurons in $N - pres(i)$, and create one synapse to each selected neuron.
- for $\alpha := -$, if $|pres(i)| \leq k$, deterministically delete all synapses in $pres(i)$. If however $|pres(i)| > k$, nondeterministically select k neurons in $pres(i)$, and delete each synapse to the selected neurons.

If $\alpha \in \{\pm, \mp\}$: create (respectively, delete) synapses at step t and then delete (respectively, create) synapses at step $t+1$. Only the priority of application of synapse creation or deletion is changed, but the application is similar to $\alpha \in \{+,-\}$. Neuron i is always open from t until $t+1$, but σ_i can only apply another rule at time $t+2$.

An important note is that for σ_i applying a rule with $\alpha \in \{+, \pm, \mp\}$, creating a synapse always involves an *embedded* sending of one spike when σ_i connects to a neuron. This single spike is sent at the time the synapse creation is applied, i.e. whenever σ_i *attaches* to σ_j using a synapse during synapse creation, we have σ_i immediately transferring one spike to σ_j.

Let t be a step during a computation: we say a σ_i is *activated* at step t if there is at least one $r \in R_i$ that can be applied; σ_i is *simple* if $|R_i| = 1$, with a nice biological and computing interpretation, i.e. some neurons do not need to be complex, but merely act as spike repositories or relays. We have the following nondeterminism levels: *rule-level*, if at least one neuron has at least two rules with regular expressions E_1 and E_2 such that $E_1 \neq E_2$ and $L(E_1) \cap L(E_2) \neq \emptyset$; *synapse-level*, if initially Π has at least one σ_i with a plasticity rule where $k < |N - pres(i)|$; *neuron-level*, if at least one activated neuron with rule r can choose to apply its rule r or not (i.e. asynchronous).

By default SNP and SNPSP systems are locally sequential (at most one rule is applied per neuron) but globally parallel (all activated neurons must apply a rule). The application of rules in neurons are usually synchronized, i.e. a global clock is assumed. However, in the asynchronous ($asyn$, in short) mode we release this synchronization so that neuron-level nondeterminism is implied. A configuration of an SNPSP system is based on (`a`) distribution of spikes in neurons, and (`b`) neuron connections based on syn. For some step t, we can represent: (`a`) as $\langle s_1, \ldots, s_m \rangle$ where $s_i, 1 \leq i \leq m$, is the number of spikes contained in σ_i; for (`b`) we can derive $pres(i)$ and $pos(i)$ from syn, for a given σ_i. The initial configuration therefore is represented as $\langle n_1, \ldots, n_m \rangle$, with the possibility of a disconnected graph, or $syn = \emptyset$. A computation is defined as a sequence of configuration transitions, from an initial configuration, and following rule application semantics. A computation halts if the system reaches a halting configuration, i.e. no rules can be applied and all neurons are open.

A result of a computation can be defined in several ways in SNP systems literature. For SNP systems in $asyn$ mode however, and as in [3] [5] [15], the output is obtained by counting the total spikes sent out by σ_{out} to the environment (in short, Env) upon reaching a halting configuration. We refer to Π as generator, if Π computes in this asynchronous manner. Π can also work as an acceptor but this is not given in this work.

For our universality results, the following simplifying features are used in our systems as the normal form: (i) plasticity rules can only be found in purely plastic neurons (i.e. neurons with plasticity rules only), (ii) neurons with standard rules are simple, and (iii) we do not use forgetting rules or rules with delays. We denote the family of sets computed by asynchronous SNPSP systems (under the mentioned normal form) as generators as $N_{tot}SNPSP^{asyn}$: subscript tot indicates the total number of spikes sent to Env as the result; Other parameters are as follows: $+syn_k$ ($-syn_j$, respectively) where at most k (j, resp.) synapses are created (deleted, resp.) each step; $nd_\beta, \beta \in \{syn, rule, neur\}$ indicate additional levels of nondeterminism source; $rule_m$ indicates at most m rules (either standard or plasticity) per neuron; Since our results for k and j for $+syn_k$ and $-syn_j$ are equal, we write them instead in the compressed form $\pm syn_k$, where $\pm$ in this sense is not the same as when $\alpha := \pm$. A bound p on the number of spikes stored in any neuron of the system is denoted as $bound_p$. We omit nd_{neur} from writing since it is implied in $asyn$ mode.

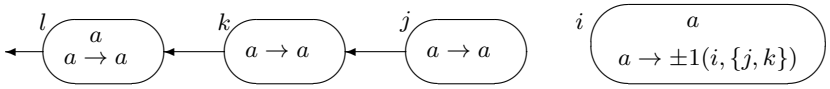


Fig. 1. An SNPSP system Π_{ej}.

To illustrate the notions and semantics in SNPSP systems, we take as an example the SNPSP system Π_{ej} of degree 4 in Fig. 1, and describe its computations. The initial configuration is as follows: spike distribution is $\langle 1, 0, 0, 1\rangle$ for the neuron order σ_i, σ_j, σ_k, σ_l, respectively; $syn = \{(j, k), (k, l)\}$; output neuron is σ_l, indicated by the outgoing synapse to Env.

Given the initial configuration, σ_i and σ_l can become activated. Due to *asyn* mode however, they can decide to apply their rules at a later step. If σ_l applies its rule before it receives a spike from σ_i, then it will spike to Env twice so that $N_{tot}(\Pi_{ej}) = \{2\}$. Since $k = 1 < |\{j, k\}|$ and $pres(i) = \emptyset$, σ_i nondeterministically selects whether to create synapse (i, j) or (i, k); if (i, j) ((i, k), resp.) is created; a spike is sent from σ_i to σ_j (σ_k, resp.) due to the embedded sending of a spike during synapse creation. Let this be step t. If (i, j) is created then $syn' := syn \cup \{(i, j)\}$, otherwise $syn'' := syn \cup \{(i, k)\}$. At $t + 1$, σ_i deletes the created synapse at t (since $\alpha := \pm$), and we have syn again. Note that if σ_l does not apply its rule and collects two spikes (one spike from σ_i), the computation is aborted or blocked, i.e. no output is produced since $a^2 \notin L(a)$.

4 Main results

In this section we use at most two nondeterminism sources: nd_{neur} (in *asyn* mode), and nd_{syn}. Recall that in *asyn* mode, if σ_i is activated at step t so that an $r \in R_i$ can be applied, σ_i can choose to apply r or not. If σ_i did not choose to apply r, σ_i can continue to receive spikes so that for some $t' > t$, it is possible that: r can never be applied again, or some $r' \in R_i, r' \neq r$, is applied.

For the next result, each neuron can store only a bounded number of spikes (see for example [3][6][7] and references therein). In [6], it is known that bounded SNP systems with extended rules in *asyn* mode characterize $SLIN$, but it is open whether such result holds for systems with standard rules only. In [3], a negative answer was conjectured for the following open problem: are asynchronous SNP systems with standard rules universal? First, we prove that bounded SNPSP systems in *asyn* mode characterize $SLIN$, hence they are not universal.

Lemma 1 $N_{tot}SNPSP^{asyn}(bound_p, nd_{syn}) \subseteq SLIN, p \geq 1$.

Proof. Taking any asynchronous SNPSP system Π with a given bound p on the number of spikes stored in any neuron, we observe that the number of possible configurations is finite: Π has a constant number of neurons, and that the number of spikes stored in each neuron are bounded. We then construct a right-linear

grammar G, such that Π generates the length set of the regular language $L(G)$. Let us denote by $\mathcal{C}$ the set of all possible configurations of Π, with C_0 being the initial configuration. The right-linear grammar $G = (\mathcal{C}, \{a\}, C_0, P)$, where the production rules in P are as follows:

(1) $C \to C'$, for $C, C' \in \mathcal{C}$ if Π has a transition $C \Rightarrow C'$ in which the output neuron does not spike;
(2) $C \to aC'$, for $C, C' \in \mathcal{C}$ if Π has a transition $C \Rightarrow C'$ in which the output neuron spikes;
(3) $C \to \lambda$, for any $C \in \mathcal{C}$ in which Π halts.

Due to the construction of G, Π generates the length set of $L(G)$, hence the set is semilinear. □

Lemma 2 $SLIN \subseteq N_{tot}SNPSP^{asyn}(bound_p, nd_{syn}), p \geq 1$.

The proof is based on the following observation: A set Q is semilinear if and only if Q is generated by a strongly monotonic register machine M. It suffices to construct an SNPSP system Π with restrictions given in the theorem statement, such that Π simulates M. Recall that M has precisely register 1 only (it is also the output register) and addition instructions of the form $l_i : (\texttt{ADD}(1), l_j, l_k)$. The ADD module for Π is given in Fig. 2. Next, we describe the computations in Π.

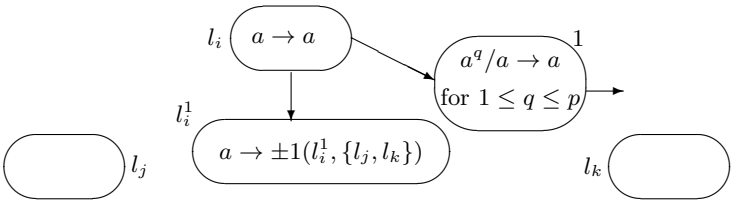


Fig. 2. Module ADD simulating $l_i : (\texttt{ADD}(1) : l_j, l_k)$ in the proof of Lemma 2.

Once ADD instruction l_i of M is applied, σ_{l_i} is activated and it sends one spike each to σ_1 and $\sigma_{l_i^1}$. At this point we have two possible cases due to *asyn* mode, i.e. either σ_1 spikes to Env before $\sigma_{l_i^1}$ spikes, or after. If σ_1 spikes before $\sigma_{l_i^1}$, then the number of spikes in Env is immediately incremented by 1. After some time, the computation will proceed if $\sigma_{l_i^1}$ applies its only (plasticity) rule. Once $\sigma_{l_i^1}$ applies its rule, either σ_{l_j} or σ_{l_k} becomes nondeterministically activated.

However, if σ_1 spikes after $\sigma_{l_i^1}$ spikes, then the number of spikes in Env is not immediately incremented by 1 since σ_1 does not consume a spike and fire to Env. The next instruction, either l_j or l_k, is then simulated by Π. Furthermore, due to *asyn* mode, the following "worst case" computation is possible: σ_{l_h} becomes activated (corresponding to l_h in M being applied, thus halting M) before σ_1 spikes. In this computation, M has halted and has applied an m number of ADD instructions since the application of l_i. Without loss of generality we can have the

arbitrary bound $p > m$, for some positive integer p. We then have the output neuron σ_1 storing m spikes. Since the rules in σ_1 are of the form $a^q/a \to a$, $1 \leq q \leq p$, σ_1 consumes one spike at each step it decides to apply a rule, starting with rule $a^m/a \to a$, until rule $a \to a$. Thus, Π will only halt once σ_1 has emptied all spikes it stores, sending m spikes to Env in the process.

The FIN module is not necessary, and we add σ_{l_h} without any rule (or maintain $pres(l_h) = \emptyset$). Once M halts by reaching instruction l_h, a spike in Π is sent to neuron l_h. Π is clearly bounded: every neuron in Π can only store at most p spikes, at any step. We then have Π correctly simulating the strongly monotonic register machine M. This completes the proof. □

From Lemma 1 and 2, we can have the next result.

Theorem 1 $SLIN = N_{tot}SNPSP^{asyn}(bound_p, nd_{syn}), p \geq 1$.

Next, in order to achieve universality, we add an additional ingredient to asynchronous SNPSP systems: weighted synapses. The ingredient of weighted synapses has already been introduced in SNP systems literature, and we refer the reader to [16] (and references therein) for computing and biological motivations. In particular, if σ_i applies a rule $E/a^c \to a^p$, and the weighted synapse (i, j, r) exists (i.e. the weight of synapse (i, j) is r) then σ_j receives $p \times r$ spikes.

It seems natural to consider weighted synapses for asynchronous SNPSP systems: since asynchronous SNPSP systems are not universal, we look for other ways to improve their power. SNPSP systems with weighted synapses (in short, WSNPSP systems) are defined in a similar way as SNPSP systems, except for the plasticity rules and the synapse set. Plasticity rules in σ_i are now of the form

$$E/a^c \to \alpha k(i, N, r),$$

where $r \geq 1$, and E, c, α, k, N are as previously defined. Every synapse created by σ_i using a plasticity rule with weight r receives the weight r. Instead of one spike sent from σ_i to a σ_j during synapse creation, $j \in N$, r spikes are sent to σ_j. The synapse set is now of the form

$$syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\} \times \mathbb{N}.$$

We note that SNPSP systems are special cases of SNPSP systems with weighted synapses where $r = 1$, and when $r = 1$ we omit it from writing. In weighted SNP systems with standard rules, the weights can allow neurons to produce more than one spike each step, similar to having extended rules. In this way, our next result parallels the result that asynchronous SNP systems with extended rules are universal in [5]. However, our next result uses nd_{syn} with $asyn$ mode, while in [5] their systems use nd_{rule} with $asyn$ mode. We also add the additional parameter l in our universality result, where the synapse weight in the system is at most l. Our universality result also makes use of the normal form given in Section 3.

Theorem 2 $N_{tot}WSNPSP^{asyn}(rule_m, \pm syn_k, weight_l, nd_{syn}) = NRE, m \geq 9, k \geq 1, l \geq 3$.

Proof. We construct an asynchronous SNPSP system with weighted synapses Π, with restrictions given in the theorem statement, to simulate a register machine M. The general description of the simulation is as follows: each register r of M corresponds to σ_r in Π. If register r stores the value n, σ_r stores $2n$ spikes. Simulating instruction $l_i : (\texttt{OP}(r) : l_j, l_k)$ of M in Π corresponds to σ_{l_i} becoming activated. After σ_{l_i} is activated, the operation OP is performed on σ_r, and σ_{l_j} or σ_{l_k} becomes activated. We make use of modules in Π to perform addition, subtraction, and halting of the computation.

Module ADD: The module is shown in Fig. 3. At some step t, σ_{l_i} sends a spike to $\sigma_{l_i^1}$. At some $t' > t$, $\sigma_{l_i^1}$ sends a spike: the spike sent to σ_r is multiplied by two, while 1 spike is received by $\sigma_{l_i^2}$. For now we omit further details for σ_r, since it is never activated with an even number of spikes.

At some $t'' > t'$, $\sigma_{l_i^2}$ nondeterministically creates (then deletes) either (l_i^2, l_j) or (l_i^2, l_k). The chosen synapse then allows either σ_{l_j} or σ_{l_k} to become activated. The ADD module thus increments the contents of σ_r by 2, simulating the increment by 1 of register r. Next, only one among σ_{l_j} or σ_{l_k} becomes nondeterministically activated. The addition operation is correctly simulated.

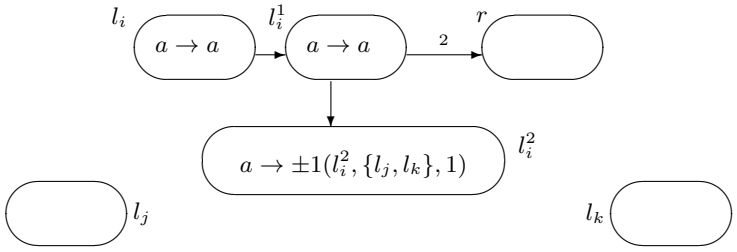


Fig. 3. Module ADD simulating $l_i : (\texttt{ADD}(r) : l_j, l_k)$ in the proof of Theorem 2.

Module SUB: The module is shown in Fig. 4. Let $|S_r|$ be the number of instructions with form $l_i : (\texttt{SUB}(r), l_j, l_k)$, and $1 \leq s \leq |S_r|$. $|S_r|$ is the number of SUB instructions operating on register r, and we explain in a moment why we use a size of a set for this number. Clearly, when no SUB operation is performed on r, then $|S_r| = 0$, as in the case of register 1. At some step t, σ_{l_i} spikes, sending 1 spike to σ_r, and $4|S_r| - s$ spikes to $\sigma_{l_i^1}$ (the weight of synapse (l_i, l_i^1)).

$\sigma_{l_i^1}$ has rules of the form $a^p \to -1(l_i^1, \{r\}, 1)$, for $3|S_r| \leq p < 8|S_r|$. When one of these rules is applied, it performs similar to a forgetting rule: p spikes are consumed and deletes a nonexisting synapse (l_i^1, r). Since $\sigma_{l_i^1}$ received $4|S_r| - s$ spikes from σ_{l_i}, and $3|S_r| \leq 4|S_r| - s < 8|S_r|$, then one of these rules can be applied. If $\sigma_{l_i^1}$ applies one of these rules at $t' > t$, no spike remains. Otherwise, the $4|S_r| - s$ spikes can combine with the spikes from σ_r at a later step.

In the case where register r stores $n = 0$ (respectively, $n \geq 1$), then instruction l_k (respectively, l_j) is applied next. This case corresponds to σ_r applying the

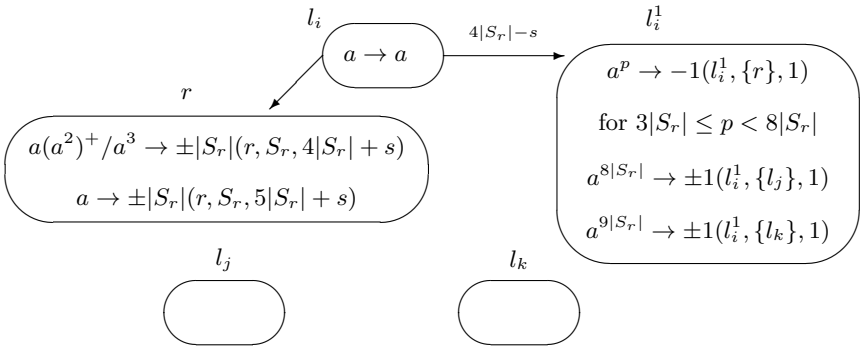


Fig. 4. Module SUB simulating $l_i : (\texttt{SUB}(r) : l_j, l_k)$ in the proof of Theorem 2.

rule with $E = a$ (respectively, $E = a(a^2)^+$), which at some later step allows σ_{l_k} (respectively, σ_{l_j}) to be activated.

For the moment let us simply define $S_r = \{l_i^1\}$. For case $n = 0$ (respectively, $n \geq 1$), σ_r stores 0 spikes (respectively, at least 2 spikes), so that at some $t'' > t$ the synapse $(r, l_i^1, 5|S_r| + s)$ (respectively, $(r, l_i^1, 4|S_r| + s)$) is created and then deleted. $\sigma_{l_i^1}$ then receives $5|S_r| + s$ spikes (respectively, $4|S_r| + s$ spikes) from σ_r. Note that we can have $t'' \geq t'$ or $t'' \leq t'$, due to *asyn* mode, where t' is again the step that $\sigma_{l_i^1}$ applies a rule. If $\sigma_{l_i^1}$ previously removed all of its spikes using its rules with $E = a^p$, then it again removes all spikes from σ_r because $3|S_r| \leq x < 8|S_r|$, where $x \in \{4|S_r| + s, 5|S_r| + s\}$. At this point, no further rules can be applied, and the computation aborts, i.e. no output is produced. If however $\sigma_{l_i^1}$ did not remove its spikes previously, then it collects a total of either $8|S_r|$ or $9|S_r|$ spikes. Either σ_{l_j} or σ_{l_k} is then activated by $\sigma_{l_i^1}$ at a step after t''.

To remove the possibility of "wrong" simulations when at least two SUB instructions operate on register r, we give the general definition of S_r: $S_r = \{l_v^1 | l_v$ is a SUB instruction on register $r\}$. In the SUB module, a rule application in σ_r creates (and then deletes) an $|S_r|$ number of synapses: one synapse from σ_r to all neurons with label $l_v^1 \in S_r$. Again, each neuron with label l_v^1 can receive either $4|S_r| + s$, or $5|S_r| + s$ spikes from σ_r, and $4|S_r| - s$ spikes from σ_{l_v}.

Let l_i be the SUB instruction that is currently being simulated in Π. In order for the correct computation to continue, only $\sigma_{l_i^1}$ must not apply a rule with $E = a^p$, i.e. it must not remove any spikes from σ_r or σ_{l_i}. The remaining $|S_r| - 1$ neurons of the form l_v^1 must apply their rules with $E = a^p$ and remove the spikes from σ_r. Due to *asyn* mode, the $|S_r| - 1$ neurons can choose not to remove the spikes from σ_r: these neurons can then receive further spikes from σ_r in future steps, in particular they receive either $4|S_r| + s'$ or $5|S_r| + s'$ spikes, for $1 \leq s' \leq S_r$; these neurons then accumulate a number of spikes greater than $8|S_r|$ (hence, no rule with $E = a^p$ can be applied), but not equal to $8|S_r|$ or $9|S_r|$ (hence, no plasticity rule can be applied). Similarly, if these spikes are not removed, and spikes from $\sigma_{l_{v'}}$ are received, $v \neq v'$ and $l_{v'} \in S_r$, no rule can again be applied: if $l_{v'}$ is the s'th SUB instruction operating on register r, then $s \neq s'$ and $\sigma_{l_{v'}}$ accumulates a number

of spikes greater than $8|S_r|$ (the synapse weight of $(l_{v'}, l^1_{v'})$ is $4|S_r| - s'$), but not equal to $8|S_r|$ or $9|S_r|$. No computation can continue if the $|S_r|-1$ neurons do not remove their spikes from σ_r, so computation aborts and no output is produced. This means that only the computations in Π that are allowed to continue are the computations that correctly simulate a `SUB` instruction in M.

The SUB module correctly simulates a `SUB` instruction: instruction l_j is simulated only if r stores a positive value (after decrementing by 1 the value of r), otherwise instruction l_k is simulated (the value of r is not decremented).

Module FIN: The module FIN for halting the computation of Π is shown in Fig. 5. The operation of the module is clear: once M reaches instruction l_h and halts, σ_{l_h} becomes activated. Neuron l_h sends a spike to σ_1, the neuron corresponding to register 1 of M. Once the number of spikes in σ_1 become odd (of the form $2n+1$, where n is the value stored in register 1), σ_1 keeps applying its only rule: at every step, 2 spikes are consumed, and 1 spike is sent to Env. In this way, the number n is computed since σ_1 will send precisely n spikes to Env.

The ADD module has nd_{syn}: initially it has $pres(l_i^2) = \emptyset$, and its $k = 1 < |N|$. We also observe the parameter values: m is at least 9 by setting $|S_r| = 1$, then adding the two additional rules in $\sigma_{l_i^1}$; k is clearly at least 1; lastly, the synapse weight l is at least 3 by again setting $|S_r| = 1$. This completes the proof. □

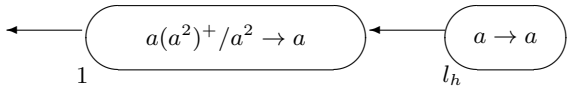


Fig. 5. Module FIN in the proof of Theorem 2.

5 Conclusions and final remarks

In [5] it is known that asynchronous SNP systems with extended rules are universal, while the conjecture is that asynchronous SNP systems with standard rules are not [3]. In Theorem 1, we showed that asynchronous bounded SNPSP systems are not universal where, similar to standard rules, each neuron can only produce at most one spike each step. In Theorem 2, asynchronous WSNPSP systems are shown to be universal. In WSNPSP systems, the synapse weights perform a function similar to extended rules in the sense that a neuron can produce more than one spike each step. Our results thus provide support to the conjecture about the nonuniversality of asynchronous SNP systems with standard rules. It is also interesting to realize the computing power of asynchronous unbounded (in spikes) SNPSP systems.

It can be argued that when $\alpha \in \{\pm, \mp\}$, the synapse creation (resp., deletion) immediately followed by a synapse deletion (resp., creation) is another form of synchronization. Can asynchronous WSNPSP systems maintain their computing power, if we further restrict them by removing such semantic? Another interesting

question is as follows: in the ADD module in Theorem 2, we have nd_{syn}. Can we still maintain universality if we remove this level, so that nd_{neur} in *asyn* mode is the only source of nondeterminism? In [5] for example, the modules used *asyn* mode and nd_{rule}, while in [15], only *asyn* mode was used (but with the use of a new ingredient called local synchronization).

In Theorem 2, the construction is based on the value $|S_r|$. Can we have a uniform construction while maintaining universality? i.e. can we construct a Π such that $N(\Pi) = NRE$, but is independent on the number of `SUB` instructions of M? Then perhaps parameters m and l in Theorem 2 can be reduced.

Acknowledgments

Cabarle is supported by a scholarship from the DOST-ERDT of the Philippines. Adorna is funded by a DOST-ERDT grant and the Semirara Mining Corp. Professorial Chair of the College of Engineering, UP Diliman. M.J. Pérez-Jiménez acknowledges the support of the Project TIN2012-37434 of the "Ministerio de Economía y Competitividad" of Spain, co-financed by FEDER funds.

References

1. Cabarle, F.G.C, Adorna, H., Ibo, N.: Spiking neural P systems with structural plasticity. Pre-proc. of 2nd Asian Conference on Membrane Computing, Chengdu, China, pp. 13 - 26, 4-7 November (2013)
2. Cabarle, F.G.C., Adorna, H.N., Pérez-Jiménez, M.J., Song, T.: Spiking neural P systems with structural plasticity (to appear). Neural Computing and Applications doi:10.1007/s00521-015-1857-4 (2015)
3. Cavaliere, M., Egecioglu, O., Woodworth, S., Ionescu, I., Păun, G.: Asynchronous spiking neural P systems: Decidability and undecidability. DNA 2008, LNCS vol. 4848, pp. 246 - 255 (2008)
4. Chen, H., Ionescu, M., Ishdorj, T.-I., Păun, A., Păun, G., Pérez-Jiménez, M.J.: Spiking neural P systems with extended rules: universality and languages. Natural Computing, vol. 7, pp. 147 - 166 (2008)
5. Cavaliere, M., Ibarra, O., Păun, G., Egecioglu, O., Ionescu, M., Woodworth, S.: Asynchronous spiking neural P systems. Theor. Com. Sci. vol. 410, pp. 2352 - 2364 (2009)
6. Ibarra, O.H., Woodworth, S.: Spiking neural P systems: some characterizations. FCT 2007, LNCS vol. 4639, pp. 23 - 37 (2007)
7. Ionescu, M., Păun, Gh., Yokomori, T.: Spiking Neural P Systems. Fundamenta Informaticae, vol. 71(2,3), pp. 279-308 (2006)
8. Minsky, M.: Computation: Finite and infinite machines. Englewood Cliffs, NJ: Prentice Hall (1967)
9. Pan, L., Păun, Gh., Pérez-Jiménez, M.J.: Spiking neural P systems with neuron division and budding. Science China Information Sciences. vol. 54(8) pp. 1596 - 1607 (2011)

10. Pan, L., Wang, J., Hoogeboom, J.H.: Spiking Neural P Systems with Astrocytes. Neural Computation vol. 24, pp. 805 - 825 (2012)
11. Păun, A., Păun, G.: Small universal spiking neural P systems. Biosystems, vol. 90, pp. 48 - 60 (2007)
12. Păun, G.: Computing with membranes. J. of Computer and System Science, vol. 61(1), pp. 108 - 143 (1999)
13. Păun, Gh.: Membrane Computing: An Introduction. Springer (2002)
14. Păun, G., Rozenberg, G., Salomaa, A., Eds.: The Oxford Handbook of Membrane Computing. Oxford Univ. Press. (2009)
15. Song, T., Pan, L., Păun, G.: Asynchronous spiking neural P systems with local synchronization. Information Sciences, vol. 219(10), pp. 197 - 207 (2013)
16. Wang, J., Hoogeboom, H.J., Pan, L., Păun, G., Pérez-Jiménez, M.J.: Spiking Neural P Systems with Weights. Neural Computation, vol. 22(10), pp. 2615 - 2646 (2010)

Automaton-like P Colonies

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

[1] Institute of Computer Science
and
Research Institute of the IT4Innovations Centre of Excellence,
Silesian University in Opava, Czech Republic
{lucie.ciencialova,ludek.cienciala}@fpf.slu.cz
[2] Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary

Summary. In this paper we study P colonies where the environment is given as a string. These variants, called automaton-like P systems or
APCol systems, behave like automata: during functioning, the agents change their own states and process the symbols of the string. We develop the concept of APCol systems by introducing the notion of their generating working mode. We then compare the power of APCol systems working in the generating mode and that of register machines and context-free matrix grammars with and without appearance checking.

Key words: String processing; P Colonies; computational power

1 Introduction

P colonies are formal models of a computing device combining properties of membrane systems and distributed systems of formal grammars called colonies [16].

In the basic model, the cells or agents are represented by a finite collection of objects and rules for processing these objects. The agents are restricted in their capabilities, i.e., only a limited number of objects, say, k objects, are allowed to be inside any cell during the functioning of the system. These objects represent the current state (contents) of the agents. The rules of the cells are either of the form $a \to b$, specifying that an internal object a is transformed into an internal object b, or of the form $c \leftrightarrow d$, specifying that an internal object c is exchanged by an object d in the environment. After applying these rules in parallel, the state of the agent will consist of objects b, d. Each agent is associated with a set of programs composed of such rules.

The agents of a P colony perform a computation by synchronously applying their programs to the objects representing the state of the agents and objects in the environment. These systems have been extensively investigated during the years; for example, it was shown that they are computationally complete computing

devices even with very restricted size parameters and with other (syntactic or functioning) restrictions [1, 3, 5, 6, 7, 8, 11, 12].

According to the the basic model, the impact of the environment on the behaviour of the P colony is indirect. To describe the situation when the behaviour of the components of the P colony is influenced by direct impulses coming from the environment step-by-step, the model was augmented with a string put on an input tape to be processed by the P colony [4]. These strings correspond to the impulse sequence coming from the environment. In addition to their rewriting rules and the rules for communicating with the environment, the agents have so-called tape rules which are used for reading the next symbol on the input tape. The model, called a P colony automaton or a PCol automaton, combines properties of standard finite automata and standard P colonies. It was shown that these variants of P colonies are able to describe the class of recursively enumerable languages, taking various working mode into account.

In [2] one step further was made in combining properties of P colonies and automata. While in the case of PCol automata the behaviour of the system is influenced both by the string to be processed and the environment consisting of multisets of symbols, in the case of automaton-like P colonies or APCol systems the environment is given as a string. The interaction between the agents in the P colony and the environment is realized by exchanging symbols between the objects of the agents and the environment (communication rules), and the states of the agents may change both via communication and evolution; the latter one is an application of a rewriting rule to an object. The distinguished symbol, e (in the previous models the environmental symbol) has a special role: whenever it is introduced in the string by communication, the corresponding input symbol is erased.

The computation in APCol systems starts with an input string, representing the environment, and with each agent having only symbols e in its state. Every computational step means a maximally parallel action of the active agents: an agent is active if it is able to perform at least one of its programs, and the joint action of the agents is maximally parallel if no more active agent can be added to the synchronously acting agents. The computation ends if the input string is reduced to the empty word, there are no more applicable programs in the system, and meantime at least one of the agents is in so-called final state.

In this paper, after recalling the model and its accepting working mode, we introduce its generating working mode. The result of computation depends on the mode in which the APCol system works.

In the case of accepting mode, a computation is called accepting if and only if at least one agent is in final state and the string obtained after the computation is ε, the empty word.

When the APCol system works in the generating mode, then a computation is called successful if only if it is halting and at least one agent is in final state. A string w is generated by the APCol system if starting with the empty string

in the environment, after finishing the computation the obtained string is w, the computation is halting and at least one agent is in final state.

After introducing the notion of the generating working mode, we compared the power of APCol systems working in the generating mode and that of register machines and context-free matrix grammars with and without appearance checking.

2 Definitions

Throughout the paper the reader is assumed to be familiar with the basics of formal language theory and membrane computing. For further details we refer to [14] and [20].

For an alphabet Σ, the set of all words over Σ (including the empty word, ε), is denoted by Σ^*. We denote the length of a word $w \in \Sigma^*$ by $|w|$ and the number of occurrences of the symbol $a \in \Sigma$ in w by $|w|_a$. For a language $L \subseteq \Sigma^*$, the set $length(L) = \{|w| \mid w \in L\}$ is called the length set of L. For a family of languages FL, the family of length sets of languages in FL is denoted by NFL.

A multiset of objects M is a pair $M = (V, f)$, where V is an arbitrary (not necessarily finite) set of objects and f is a mapping $f : V \to N$; f assigns to each object in V its multiplicity in M. The set of all multisets with the set of objects V is denoted by V°. The set V' is called the support of M and denoted by $supp(M)$. The cardinality of M, denoted by $|M|$, is defined by $|M| = \sum_{a \in V} f(a)$. Any multiset of objects M with the set of objects $V' = \{a_1, \ldots a_n\}$ can be represented as a string w over alphabet V' with $|w|_{a_i} = f(a_i);\ 1 \leq i \leq n$. Obviously, all words obtained from w by permuting the letters can also represent the same multiset M, and ε represents the empty multiset.

2.1 Register machines and matrix grammars

Definition 1. [19] *A register machine is the construct $M = (m, H, l_0, l_h, P)$ where:*

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

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

$l_1 : (ADD(r), l_2, l_3)$	Add 1 to the content of the register r and proceed to the instruction (labelled with) l_2 or l_3.
$l_1 : (SUB(r), l_2, l_3)$	If the register r stores a value different from zero, then subtract 1 from its content and go to instruction l_2, otherwise proceed to instruction l_3.
$l_h : HALT$	Halt the machine. The final label l_h is only assigned to this instruction.

Without loss of generality, we may assume that in each *ADD*-instruction $l_1 : (ADD(r), l_2, l_3)$ and in each *SUB*-instruction $l_1 : (SUB(r), l_2, l_3)$ the labels l_1, l_2, l_3 are pairwise different.

The register machine M computes a set $N(M)$ of numbers in the following way: it starts with all registers empty (hence storing the number zero) with the instruction labelled by l_0 and it proceeds to apply the instructions as indicated by the labels (and made possible by the contents of registers). If it reaches the halt instruction, then the number stored at that time in the register 1 is said to be computed by M and hence it is introduced in $N(M)$. (Because of the non-determinism in choosing the continuation of the computation in the case of *ADD*-instructions, $N(M)$ can be an infinite set.) It is known (see e.g.[19]) that in this way we compute all Turing computable sets.

Moreover, we call a register machine partially blind [13], if we interpret a subtract instruction in the following way: $l_1 : (SUB(r); l_2; l_3)$ - if in register r there is value different from zero, then subtract one from its contents and go to instruction l_2 or to instruction l_3; if in register r there is stored zero when attempting to decrement register r, then the program ends without yielding a result.

When the partially blind register machine reaches the final state, the result obtained in the first register is only taken into account if the remaining registers store value zero. The family of sets of non-negative integers generated by partially blind register machines is denoted by NRM_{pb}. Partially blind register machines accept a proper subset of *NRE*, the family of recursively enumerable sets of numbers.

Definition 2. *A context-free matrix grammar is a construct*

$$G = (N, T, S, M, F), \text{ where}$$

- N *and* T *are sets of non-terminal and terminal symbols, respectively, with* $N \cap T = \emptyset$,
- $S \in N$ *is the start symbol,*
- M *is a finite set of matrices,* $M = \{m_i \mid 1 \leq i \leq n\}$, *where matrices* m_i *are sequences of the form* $m_i = (m_{i,1}, \ldots, m_{i,n_i}), n_i \geq 1,\ 1 \leq i \leq n$, *and* $m_{i,j},\ 1 \leq j \leq n_i,\ 1 \leq i \leq n$, *are context-free rules over* $(N \cup T)$,
- F *is a subset of all productions occurring in the elements of* M, *i.e.* $F \in \{m_{i_j} \mid 1 \leq i \leq n, 1 \leq j \leq n_i\}$.

We say that $x \in (N \cup T)^+$ *directly derives* $y \in (N \cup T)^*$ *in the appearance checking mode* by application of $m_{i,j} = A \to w \in m_i$ – denoted by $x \Rightarrow^{ac} y$, – if one of the following conditions hold:

$$x = x_1 A x_2 \text{ and } y = x_1 w x_2 \quad \text{or} \quad A \text{ does not appear in } x, m_{i,j} \in F \text{ and } x = y.$$

For $m_i = (m_{i,1}, \ldots, m_{i,n_i})$ and $v, w \in (N \cup T)^*$ we define $v \Rightarrow_{m_i} w$ if and only if there are $w_0, w_1, \ldots, w_{n_i} \in (N \cup T)^*$ such that

$$v = w_0 \Rightarrow^{ac}_{m_{i,1}} w_1 \Rightarrow^{ac}_{m_{i,2}} w_2 \Rightarrow^{ac}_{m_{i,3}} \ \ldots \ \Rightarrow^{ac}_{m_{i,n_i}} w_{n_i} = w$$

The language generated by G is

$$L(G) = \{w \in T^* \mid S \Rightarrow_{m_{i_1}} w_1 \cdots \Rightarrow_{m_{i_k}} w_k,\ w_k = w,\\ w_j \in (N \cup T)^*,\ m_{i_j} \in M \text{ for } 1 \leq j \leq k,\ k \geq 1,\ 1 \leq i \leq n\}$$

The family of languages generated by matrix grammars with appearance checking is denoted by MAT^{λ}_{ac}. The superscript λ indicates that erasing rules (λ-rules) are allowed.

We say that M is a matrix grammar without appearance checking if and only if $F = \emptyset$. The family of languages generated by matrix grammars without appearance checking is denoted by MAT^{λ}.

The following results are known about matrix languages:

- $CF \subset MAT \subseteq\ MAT^{\lambda}\ \subset RE$
- $MAT \subset MAT_{ac} \subset CS$
- $MAT^{\lambda}\ \subset\ MAT^{\lambda}_{ac}\ =\ CS$, where CF, CS, RE are the context-free, context-sensitive, and recursively enumerable language classes, respectively
- $NRM_{pb} = NMAT^{\lambda}$, where $NMAT^{\lambda}$ is class of the length sets associated with matrix languages without appearance checking (in [11]).

Further details about matrix grammars can be found in [9].

2.2 Automaton-like P colony

In the following we recall the concept of an automaton-like P colony (an APCol system, for short) where the environment of the agents is given in the form of a string [2].

As in the case of standard P colonies, agents of APCol systems contain objects, each being an element of a finite alphabet. With every agent, a set of programs is associated. There are two types of rules in the programs. The first one, called an evolution rule, is of the form $a\ \ \rightarrow b$. It means that object a inside of the agent is rewritten (evolved) to the object b. The second type of rules, called a communication rule, is in the form $c \leftrightarrow d$. When this rule is performed, the object c inside the agent and a symbol d in the string are exchanged, so, we can say that the agent rewrites symbol d to symbol c in the input string. If $c = e$, then the agent erases d from the input string and if $d = e$, symbol c is inserted into the string.

An automaton-like P colony works successfully, if it is able to reduce the given string to ε, i.e., to enter a configuration where at least one agent is in accepting state and the processed string is the empty word.

Definition 3. *[2] An automaton-like P colony (an APCol system, for short) is a construct*

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

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

During the work of the APCol system, the agents perform programs. Since both rules in a program can be communication rules, an agent can work with two objects in the string in one step of the computation. In the case of program $\langle a \leftrightarrow b; c \leftrightarrow d\rangle$, a sub-string bd of the input string is replaced by string ac. If the program is of the form $\langle c \leftrightarrow d; a \leftrightarrow b\rangle$, then a sub-string db of the input string is replaced by string ca. That is, the agent can act only in one place in a computation step and the change of the string depends both on the order of the rules in the program and on the interacting objects. In particular, we have the following types of programs with two communication rules:

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

The program is said to be *restricted* if it is formed from one rewriting and one communication rule. The APCol system is restricted if all the programs the agents have are restricted.

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

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

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

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

3 Accepting and generating mode of computation

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

The situation is different when the APCol system works in the generating mode. A computation is called successful if only if it is halting and at least one agent is in final state. The string w_F is generated by Π iff there exists computation starting in an initial configuration $(\varepsilon; \omega_1, \ldots, \omega_n)$ and the computation ends by halting in the configuration $(w_F; w_1, \ldots, w_n)$, where at least one of $w_i \in F_i$ for $1 \leq i \leq n$.

We denote by $APCol_{acc}R(n)$ (or $APCol_{acc}(n)$) the family of languages accepted by APCol system having at most n agents with restricted programs only (or without this restriction). Similarly we denote by $APCol_{gen}R(n)$ the family of languages generated by APCol systems having at most n agents with restricted programs only.

APCol system Π can generate or accept a set of numbers $|L(\Pi)|$.

By $NAPCol_xR(n), x \in \{acc, gen\}$, is denoted the family of sets of natural numbers accepted or generated by APCol systems with at most n agents.

In [2] the authors proved that the family of languages accepted by jumping finite automata (introduced in [18]) is properly included in the family of languages accepted by APCol systems with one agent, and it is proved that any recursively enumerable language can be obtained as a projection of a language accepted by an automaton-like P colony with two agents.

Theorem 1. *[2] The family of languages accepted by automaton-like P colonies with one agent properly includes the family of languages accepted by jumping finite automata.*

Theorem 2. *[2] Any recursively enumerable language can be obtained as a projection of a language accepted by an automaton-like P colony with two agents.*

4 The power of restricted generating APCol systems

In this section we compare the computational power of automaton-like P colonies working in generating mode with one and two agents and that of register machine and matrix grammars with erasing rules. We start with the comparison of APCol systems and register machines.

Theorem 3. $NAPCol_{gen}R(2) = NRE$

Proof. Let us consider a register machine M with m registers. We construct an APCol system $\Pi = (O, e, A_1, A_2)$ simulating the computations of register machine M. To help the easier understanding of the simulation, we provide the components together with some explanations of their role. Let

- $O = \{G\} \cup \{l_i, l_i^I, l_i^{II}, l_i^{III}, l_i^{IV}, l_i^V, l_i^{VI}, \underline{l_i}, \underline{\underline{l_i}}, L_i, L_i', L_i'', F_i \mid l_i \in H\} \cup$
  $\cup \{r \mid 1 \leq r \leq m\}$,
- $A_1 = (ee, P_1, \{eG\})$
- $A_2 = (ee, P_2, \{ee\})$

At the beginning of the computation the first agent generates object l_0 (the label of starting instruction of M). Then it starts to simulate instruction labelled by l_0 and it generates the label of the next instruction. The number stored at the register r corresponds to the number of symbols r placed on the input string. The set of programs is as follows:

(1) For initializing the simulation there is one program in P_1:

P_1

$1 : \langle e \to l_0; e \leftrightarrow e \rangle$

The initial configuration of Π is $(\varepsilon; ee, ee)$. After the first step of computation (only the program 1 is applicable) the system enters configuration $(\varepsilon; l_0 e, ee)$.

(2) For every ADD-instruction $l_1 : (ADD(r), l_2, l_3)$ we add to P_1 the next programs:

P_1

$2 : \langle e \to r; l_1 \leftrightarrow e \rangle$, $3 : \langle e \to a; r \leftrightarrow l_1 \rangle$,
$4 : \langle l_1 \to l_2; a \leftrightarrow e \rangle$, $5 : \langle l_1 \to l_3; a \leftrightarrow e \rangle$

When there is object l_1 inside the agent, it generates one copy of r, puts it to the environment and generates the label of the next instruction (it non-deterministically chooses one of the last two programs 4 and 5)

	configuration of Π			labels of applicable programs	
	A_1	A_2	string	P_1	P_2
1.	$l_1 e$	ee	r^x	2	–
2.	re	ee	$l_1 r^x$	3	–
3.	al_1	ee	r^{x+1}	**4** *or* 5	–
4.	$l_2 e$	ee	$r^{x+1}a$		

(3) For every SUB-instruction $l_1 : (SUB(r), l_2, l_3)$, the next programs are added to sets P_1 and P_2:

P_1 :		P_2 :
$6 : \langle l_1 \to l_1^I; e \leftrightarrow e \rangle$	$12 : \langle l_1^{VI} \to \underline{l_2}; e \leftrightarrow L_1'' \rangle$	$19 : \langle e \to L_1; e \leftrightarrow l_1^I \rangle$
$7 : \langle e \to l_1^{II}; l_1^I \leftrightarrow e \rangle$	$13 : \langle L_1'' \to l_2; \underline{l_2} \leftrightarrow e \rangle$	$20 : \langle l_1^I \to L_1'; L_1 \leftrightarrow l_1^{II} \rangle$
$8 : \langle e \to l_1^{III}; l_1^{II} \leftrightarrow e \rangle$	$14 : \langle l_1^{VI} \to \underline{l_3}; e \leftrightarrow L_1 \rangle$	$21 : \langle l_1^{II} \to L_1''; L_1' \leftrightarrow r \rangle$
$9 : \langle l_1^{III} \to l_1^{IV}; e \leftrightarrow e \rangle$	$15 : \langle L_1 \to F_3; \underline{l_3} \leftrightarrow e \rangle$	$22 : \langle r \to e; L_1'' \leftrightarrow L_1 \rangle$
$10 : \langle l_1^{IV} \to l_1^V; e \leftrightarrow e \rangle$	$16 : \left\langle e \to \underline{\underline{l_3}}; F_3 \leftrightarrow \underline{l_3} \right\rangle$	$23 : \langle L_1 \to e; e \leftrightarrow e \rangle$
$11 : \langle l_1^V \to l_1^{VI}; e \leftrightarrow e \rangle$	$17 : \left\langle \underline{l_3} \to F_3'; \underline{\underline{l_3}} \leftrightarrow e \right\rangle$	$24 : \langle l_1^{II} \to e; L_1' \leftrightarrow F_3 \rangle$
	$18 : \langle F_3' \to l_3; e \leftrightarrow e \rangle$	$25 : \langle F_3 \to e; e \leftrightarrow e \rangle$

At the first phase of the simulation of the SUB instruction the first agent generates object l_1', which is consumed by the second agent. The agent A_2 generates symbol L_1 and tries to consume one copy of symbol r. If there is any r, the agent sends to the environment object L_1'' and consumes L_1. After this step the first agent consumes L_1'' or L_1 and rewrites it to l_2 or l_3.

Instruction $l_1 : (SUB(r), l_2, l_3)$ is simulated by the following sequence of steps.

If the register r stores non-zero value:

	configuration of Π			labels of applicable programs	
	A_1	A_2	string	P_1	P_2
1.	$l_1 e$	ee	r^x	6	–
2.	$l_1^I e$	ee	r^x	7	–
3.	$l_1^{II} e$	ee	$l_1^I r^x$	8	19
4.	$l_1^{III} e$	$L_1 l_1^I$	$l_1^{II} r^x$	9	20
5.	$l_1^{IV} e$	$L_1' l_1^{II}$	$L_1 r^x$	10	21
6.	$l_1^V e$	$L_1'' r$	$L_1 L_1' r^{x-1}$	11	22
7.	$l_1^{VI} e$	$e L_1$	$L_1'' r^{x-1}$	12	23
8.	$\underline{l_2} L_1''$	ee	r^{x-1}	14	–
9.	$l_2 e$	ee	$r^{x-1} \underline{l_2}$		

If the register r stores value zero :

	configuration of Π			labels of applicable programs	
	A_1	A_2	string	P_1	P_2
1.	$l_1 e$	ee		6	–
2.	$l_1^I e$	ee		7	–
3.	$l_1^{II} e$	ee	l_1^I	8	19
4.	$l_1^{III} e$	$L_1 l_1^I$	l_1^{II}	9	20
5.	$l_1^{IV} e$	$L_1' l_1^{II}$	L_1	10	–
6.	$l_1^V e$	$L_1' l_1^{II}$	L_1	11	–
7.	$l_1^{VI} e$	$L_1' l_1^{II}$	L_1	13	–
8.	$\underline{l_3} L_1$	$L_1' l_1^{II}$		15	–
9.	$F_3 e$	$L_1' l_1^{II}$	$\underline{l_3}$	16	–
10.	$\underline{\underline{l_3}}\ \underline{l_3}$	$L_1' l_1^{II}$	F_3	17	24
11.	$F_3' e$	$F_3 e$	$\underline{l_3} L_1'$	18	25
12.	$l_3 e$	ee	$\underline{\underline{l_3}} L_1'$		

(4) For halting instruction l_h there are programs belonging to the sets P_1 and P_2. After that agent A_1 generates the object l_h, it writes the symbol G to the tape. After consuming it, the second agent erases all the symbols from the tape except these ones which correspond to the first register of the register machine.

P_1 :	P_2 :
$26 : \langle e \to G; l_h \leftrightarrow e \rangle$ $27 : \langle e \to e; e \leftrightarrow l_h \rangle$	$28 : \langle e \to e; l_h \leftrightarrow X \rangle$
	$29 : \langle X \to e; e \leftrightarrow l_h \rangle$

$X \in \{a\} \cup \{L'_i, \underline{l_i}, \underline{\underline{l_i}} \mid 0 \leq i \leq p, \text{ where } |H| = p\} \cup \{r \mid 1 < r \leq m\}$

The APCol system Π starts computation in the initial configuration with empty tape. It starts the simulation of register machine M with instruction labelled by l_0 and it proceeds the simulation according to the instructions of the register machine. After M reaches the halting instruction, then agent A_2 in the APCol system Π erases from the tape all the symbols except symbols 1 and then APCol systems halts. So the length of the word placed on the tape in the last configuration corresponds to the number stored in the first register of M at the end of its computation.

We proved that the family of length sets of languages generated by restricted APCol systems with two agents equals to *NRE*. If the APCol system is formed from only one agent, there are some limitation for generated languages. In the following we show the limits of the computational power of restricted APCol systems with only one agent.

Theorem 4. $NRM_{PB} \subseteq NAPCol_{gen}R(1)$

Proof. Let us consider a partially blind register machine M with m registers. We construct an APCol system $\Pi = (O, e, A)$ simulating the computations of register machine M with:

- $O = \{G\} \cup \{l_i, l_i^I, l_i^{II}, l_i^{III}, l_i^{IV}, l_i^V, l_i^{VI}, \underline{l_i}, \underline{\underline{l_i}}, L_i, L'_i, L''_i, F_i \mid l_i \in H\} \cup$
  $\cup \{r \mid 1 \leq r \leq m\}$,
- $A = (ee, P, \{eG\})$

The functioning of the constructed APCol system is very similar to the one from the proof of previous theorem. At the beginning of the computation, the agent generates the object l_0 (the label of starting instruction of M). Then it starts to simulate instruction labelled by l_0 and generates the label of the next instruction. The number stored at register r corresponds to the number of symbols r placed on the input string. The set of programs is given as follows:

(1) For initializing the simulation there is one program in P_1:

P :
$1 : \langle e \to l_0; e \leftrightarrow e \rangle$

The initial configuration of Π is $(\varepsilon; ee)$. After the first step of computation (only the program 1 is applicable) the system enters configuration $(\varepsilon; l_0 e)$.

(2) For every *ADD*-instruction $l_1 : (ADD(r), l_2, l_3)$ we add to P the next programs:

P	
$2 : \langle e \to r; l_1 \leftrightarrow e \rangle$,	$3 : \langle e \to a; r \leftrightarrow l_1 \rangle$,
$4 : \langle l_1 \to l_2; a \leftrightarrow e \rangle$,	$5 : \langle l_1 \to l_3; a \leftrightarrow e \rangle$

When there is object l_1 inside the agent, it generates one copy of r, puts it to the environment and generates the label of the next instruction (it non-deterministically chooses one of two programs 4 and 5)

	configuration of Π		labels of applicable programs
	A	string	P
1.	$l_1 e$	r^x	2
2.	re	$l_1 r^x$	3
3.	al_1	r^{x+1}	**4** *or* 5
4.	$l_2 e$	$r^{x+1}a$	

(3) For every *SUB*-instruction $l_1 : (SUB(r), l_2, l_3)$, the next programs are added to set P:

P :

$6 : \langle l_1 \to l_1^I; e \leftrightarrow r \rangle$ $7 : \langle r \to l_2; l_1^I \leftrightarrow e \rangle$ $8 : \langle r \to l_3; l_1^I \leftrightarrow e \rangle$

The agent generates object l_1' even if there is at least one object r in the environment. Then it rewrites r to l_2 or l_3. If there is no r in the environment, the computation halts and the agent is in non-final state.

(4) For halting instruction l_h there are programs belonging to the set P:

P :

$9 : \langle l_h \to G; e \leftrightarrow e \rangle$ $10 : \langle e \to e; G \leftrightarrow X \rangle$ $11 : \langle X \to e; e \leftrightarrow G \rangle$
$X \in \{a\} \cup \{l_i^I \mid 0 \leq i \leq p, \text{ where } |H| = p\} \cup \{r \mid 1 < r \leq m\}$

After the object l_h appears inside agent A, it generates the symbol G. Using G, the agent erases all the symbols from the tape except those ones which correspond to the first register of the register machine.

The APCol system Π starts computation in the initial configuration with empty tape. It starts the simulation of the partially blind register machine M with instruction labelled by l_0 and it proceeds the simulation according to the instructions of register machine. After M reaches the halting instruction, the agent A in the APCol system Π erases from the tape all the symbols except symbols 1 and then the APCol system halts. So the length of the word placed on the tape in the last configuration corresponds to the number stored in the first register of M at the end of its computation.

In the following we will examine the language generating power of restricted APCoL systems.

Theorem 5. $APCol_{gen}R(1) \subseteq MAT^{\lambda}$

Proof. Let $\Pi = (O, e, A)$ be a restricted APCol system with one agent. We construct a matrix grammar $G = (N, T, S, M)$ simulating Π as follows:

The symbol on the tape $a \neq e$ is represented by $\boxed{a} \in N$ and at the end of simulation it can be rewritten to $a \in T$. The content of the agent is represented

by a non-terminal symbol $\boxed{AB} \in N$, where $a, b \in O$ are objects placed inside the agent. As in the previous cases, we provide only the necessary details.

The first applied matrix is $\left(S \to C\ \boxed{EE}\right)$, representing the initial content of the agent.

For every program of type $\langle a \to b; c \leftrightarrow d\rangle\,,\ c, d \neq e$ there is a matrix in M:

$$\left(C \to C, \boxed{AC} \to \boxed{BD}, \boxed{d} \to \boxed{c}\right)$$

For every program of type $\langle a \to b; e \leftrightarrow d\rangle\,,\ d \neq e$ there is a matrix in M:

$$\left(C \to C, \boxed{AC} \to \boxed{BD}, \boxed{d} \to \varepsilon\right)$$

For every program of type $\langle a \to b; e \leftrightarrow e\rangle$ there is a matrix in M:

$$\left(C \to C, \boxed{AE} \to \boxed{BE}\right)$$

For every program of the type $\langle a \to b; c \leftrightarrow e\rangle\,,\ c \neq e$ there is a matrix in M:

$$\left(C \to C, \boxed{AC} \to \boxed{BE}\,\textcircled{c}\right)$$

and a set of matrices generating $\boxed{c}$ somewhere in the string and deleting $\textcircled{c}$:

$$\left(C \to C, \boxed{x} \to \boxed{x}\,\boxed{c}, \textcircled{c} \to \varepsilon\right),$$
$$\left(C \to C, \boxed{x} \to \boxed{c}\,\boxed{x}, \textcircled{c} \to \varepsilon\right),$$

for all $\boxed{x}$ such that $x \in T$.

When the APCol system reaches the halting configuration, the matrix grammar generates the corresponding string. The string is formed from only non-terminals. The matrix grammar has to rewrite the rammed terminal symbols to terminals and to delete non-terminal representing the content of the agent and the non-terminal C. The halting configuration can be presented by a string $AB \cdot w$, where $|w|_a = 1$ for all $a \in T$ such that a is present in this halting configuration and AB is content of the agent such that $ab \in F$. The set of such a representations is finite.

For each representation $AB \cdot a_1 a_2 \ldots a_p,\ p \leq |T|$, we add the following matrices to the matrix grammar:

$$\left(C \to [\ \boxed{AB} a_1 a_2 \ldots a_p]\right)$$
$$\left([\ \boxed{AB} a_1 a_2 \ldots a_q] \to [\ \boxed{AB} a_1 a_2 \ldots a_q], \boxed{a_q} \to a_q\right),$$
$$\left([\ \boxed{AB} a_1 a_2 \ldots a_q] \to [\ \boxed{AB} a_1 a_2 \ldots a_{q-1}]\right), 1 < q \leq p,$$
$$\left([\ \boxed{AB} a_1] \to [\ \boxed{AB} a_1], \boxed{a_1} \to a_1\right),\ \left([\ \boxed{AB} a_1] \to [\ \boxed{AB}\ ]\right)$$
$$\left([\ \boxed{AB}\ ] \to \varepsilon, \boxed{AB} \to \varepsilon\right)$$

In this way all non-terminal symbols are rewritten to the corresponding terminals and the non-terminal symbol corresponding to the contents of the agent is deleted.

If the restricted APCol system Π generates a string ω, then the matrix grammar is able to generate it, too. If the APCol system halts and the agent is not in final state, then the matrix grammar cannot generate a string consisting only of terminals.

The last theorem is devoted to the relationship of MAT^{λ}_{ac} to restricted APCol systems with two agents.

Theorem 6. $APCol_{gen}R(2) \subseteq MAT^{\lambda}_{ac}$

Proof. Let $\Pi = (O, e, A_1, A_2)$ be the restricted APCol system with two agents. We construct a matrix grammar $G = (N, T, S, M)$ simulating Π as follows:

The symbol on the tape $a \neq e$ is represented by $\boxed{a} \in N$ and at the end of the simulation it is rewritten to $a \in T$. The contents of the agent A_1 is represented by a non-terminal symbol $\boxed{AB} \in N$, where $a, b \in O$ are the objects placed inside the first agent. The contents of the agent A_2 is represented by a non-terminal symbol $(AB) \in N$, where $a, b \in O$ are the objects placed inside the second agent.

We add label p_i from the set of labels P to every program associated with both agents. Let $P \cap (N \cup T) = \emptyset$. We add a set of new non-terminals $\{P_i \mid p_i \in P\}$ to the set N . To control the derivation, we add non-terminals S, C, C_P and $\#$.

The first applied matrix is $\big(S \to C\boxed{EE}(EE)\big)$, representing the initial states of the agents.

For every possible pair of states of agents, we add the following types of matrices to M. Let ab be a state of the agent A_1. In this state there are three applicable types of programs: $\langle a \to c; b \leftrightarrow d\rangle$ – (A), $\langle a \to c; b \leftrightarrow e\rangle$ – (B). The first type includes the case where $b = e$ and $\langle a \to c; e \leftrightarrow d\rangle$ – (A').

We divide the execution of such a program into three phases. The first phase is to choose two programs to apply. The corresponding matrices (forming the set $M_1 \subset M$) are of the form

$$\big(C \to C, \boxed{AB} \to \boxed{ABv}, (XY) \to (XYz)\big)\,, \tag{1}$$

where v and z depend on the type of the used programs. If the program is of the type $\langle a \to c; b \leftrightarrow d\rangle$ or $\langle a \to c; e \leftrightarrow d\rangle$, v (or z) is d and it means that d will be erased in following derivation steps. If the program is of the type $\langle a \to c; b \leftrightarrow e\rangle$, v (or z) is $\sqcup$ and it means that there is nothing to erase from string.

Because of maximal parallelism in the computation, an agent can be in a state when there is no program to use. For this situation, we construct another set of matrices. We choose from the set M_1 all matrices corresponding to the application of a program of the type $\langle a \to c; b \leftrightarrow d\rangle$ by the "sleeping agent". We need not to include matrices for the programs of type $\langle a \to c; b \leftrightarrow e\rangle$ because these programs in state ab are always applicable. To the selected matrices of the type (1), we add the following matrices:

$$\big(C \to C_P, \boxed{AB} \to \boxed{ABN}P_i, (XY) \to (XYz)\big)\,, \tag{2}$$

for the first agent with no applicable program and

$$\left(C \to C_P, \boxed{AB} \to \boxed{ABv}, \textcircled{XY} \to \textcircled{XYN} P_j\right), \tag{3}$$

for the second agent with no applicable program and $p_i \in P$, resp. $p_j \in P$ is the label of program corresponding to triplet $\boxed{ABv}$ resp. to $\textcircled{XYz}$.

To check whether in the given state the agent has no applicable program, we perform the following construction. Let P_i' be the set of all applicable programs of the type (A) in the state ab of the agent. Then matrices

$$(C_P \to C, P_i \to \varepsilon, d_1 \to \#, d_2 \to \#, \ldots, d_k \to \#), \tag{4}$$

are for checking the inactivity of the agent. $d_1, \ldots, d_k \in N, k = |P_i'|$ are non-terminals corresponding to the objects that the agent needs to consume from the string applying any program from P_i'. All the rules of the type $d_l \to \#, 1 \leq l \leq k$ are in the set F. If $\#$ appears in the string, then the derivation cannot end with a terminal word.

The last phase of the simulation of execution of programs in the APCol system Π corresponds to the change of the state of the agents and to the changes changes in the string. We add to M the following matrices: For the pair of programs $(\langle a \to c; b \leftrightarrow d\rangle, \langle x \to u; y \leftrightarrow z\rangle)$

$$\left(C \to C, \boxed{ABd} \to \boxed{CD}, \boxed{d} \to \boxed{b}, \textcircled{XYz} \to \textcircled{UZ}, \boxed{z} \to \boxed{y}\right) . \tag{5}$$

For the pair of programs $(\langle a \to c; e \leftrightarrow d\rangle, \langle x \to u; y \leftrightarrow z\rangle)$

$$\left(C \to C, \boxed{ABd} \to \boxed{CD}, \boxed{d} \to \varepsilon, \textcircled{XYz} \to \textcircled{UZ}, \boxed{z} \to \boxed{y}\right) \tag{6}$$

For the pair of programs $(\langle a \to c; b \leftrightarrow e\rangle, \langle x \to u; y \leftrightarrow z\rangle)$

$$\left(C \to C, \boxed{AB\sqcup} \to \boxed{CD}\textcircled{b}, \textcircled{XYz} \to \textcircled{UZ}, \boxed{z} \to \boxed{y}\right) \tag{7}$$

For the remaining program combinations we add another six types of matrices.

We also add the set of matrices for generating $\boxed{c}$ somewhere in the string and for deleting $\textcircled{c}$:

$$\left(C \to C, \boxed{x} \to \boxed{x}\,\boxed{c}, \textcircled{c} \to \varepsilon\right),$$
$$\left(C \to C, \boxed{x} \to \boxed{c}\,\boxed{x}, \textcircled{c} \to \varepsilon\right), \text{ for all } q, \boxed{x} \text{ such that } x \in T.$$

When the APCol system reaches the halting configuration, the matrix grammar generates the corresponding string. The string is formed from non-terminals only. The matrix grammar has to rewrite the rammed terminal symbols to terminals and to delete the non-terminals representing the contents of the agents and non-terminal C. The halting configuration can be represented by a string $AB \cdot XY \cdot w$, where $|w|_a = 1$ for all $a \in T$ such that a is present in this halting configuration and AB, XY are the contents of the agents such that $ab \in F_1$ and $xy \in F_2$. The set of such a representations is finite.

For each representation $AB \cdot XY \cdot a_1a_2 \dots a_p,\ p \leq |T|$, we add the following matrices to the matrix grammar:

$$\begin{gathered}
\left(C \to [\ \boxed{AB}\,(XY)\,a_1a_2 \dots a_p]\right) \\
\left([\ \boxed{AB}\,(XY)\,a_1a_2 \dots a_q] \to [\ \boxed{AB}\,(XY)\,a_1a_2 \dots a_q], \boxed{a_q} \to a_q\right), \\
\left([\ \boxed{AB}\,(XY)\,a_1a_2 \dots a_q] \to [\ \boxed{AB}\,(XY)\,a_1a_2 \dots a_{q-1}]\right), 1 < q \leq p \\
\left([\ \boxed{AB}\,(XY)\,a_1] \to [\ \boxed{AB}\,(XY)\,a_1], \boxed{a_1} \to a_1\right),\ \left([\ \boxed{AB}\,(XY)\,a_1] \to [\ \boxed{AB}\,(XY)\ ]\right) \\
\left([\ \boxed{AB}\,(XY)\ ] \to \varepsilon, \boxed{AB} \to \varepsilon, (XY) \to \varepsilon\right)
\end{gathered}$$

In this way all non-terminal symbols are rewritten to the corresponding terminals and the non-terminals corresponding to the contents of the agents are deleted.

If the restricted APCol system Π generates the string ω, then the matrix grammar can generate it, too. If the APCol system halts and the agent is not in final state, then the matrix grammar cannot generate a string consisting of only terminals.

5 Conclusions

We developed the concept of automaton-like P colonies (APCol systems) - variants of P colonies that work on a string. We introduced the generating mode of computation of these systems and compared the generative and computational power of automaton-like P colonies and the generative power of context-free matrix grammars with and without appearance checking and the computational power of variants of register machines. The results of this paper can be summarized as follows:

- $NRM_{PB} \subseteq NAPCol_{gen}R(1)$
- $APCol_{gen}R(1) \subseteq MAT^{\lambda}$
- $NAPCol_{gen}R(2) = NRE$
- $APCol_{gen}R(2) \subseteq MAT^{\lambda}_{ac}$

Remark 1. This work was partially supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/-02.0070), by SGS/24/2013 and by project OPVK no. CZ.1.07/2.2.00/28.0014.

References

1. Ciencialová, L., Cienciala, L.: Variation on the theme: P colonies. In: Kolář, D., Meduna, A. (eds.) Proc. 1st Intern. Workshop on Formal Models, pp. 27–34. Ostrava (2006)

2. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E.: P Colonies Processing Strings. Fundam. Inform. 134(1-2), 51–65 (2014)
3. Ciencialová, L., Csuhaj-Varjú, E., Kelemenová, A., Vaszil, Gy.: Variants of P colonies with very simple cell structure. International Journal of Computers, Communication and Control 4(3), 224–233 (2009)
4. Ciencala, L., Ciencialová, L., Csuhaj-Varjú, E., Vaszil, Gy.: PCol Automata: Recognizing strings with P colonies. In: Martinez-del-Amor, M. A. et al. (eds.) Proc. BWMC 2010, pp. 65–76. Fénix Editora, Sevilla (2010)
5. Cienciala, L., Ciencialová, L., Kelemenová, A.: Homogeneous P colonies. Computing and Informatics 27, 481–496 (2008)
6. Cienciala, L., Ciencialová, L., Kelemenová, A.: On the number of agents in P colonies. In: Eleftherakis,G. et. al (eds.) Membrane Computing, 8th International Workshop, WMC 2007, Thessaloniki, Greece, June 25-28, 2007 Revised Selected and Invited Papers. LNCS, vol. 4860, pp. 193–208. Springer (2007)
7. Csuhaj-Varjú, E., Kelemen, J., Kelemenová, A., Păun, Gh., Vaszil, Gy.: Computing with cells in environment: P colonies. Journal of Multi-Valued Logic and Soft Computing 12, 201–215 (2006)
8. Csuhaj-Varjú, E.,Margenstern, M., Vaszil, Gy.: P colonies with a bounded number of cells and programs. In: Hoogeboom, H-J. et. al (eds.) Membrane Computing, 7th International Workshop, WMC 2006, Leiden, The Netherlands, July 17-21, 2006, Revised, Selected, and Invited Papers. LNCS, vol. 4361, pp. 352–366. Springer(2006)
9. Dassow, J., Păun, Gh.: Regulated Rewriting in Formal Language Theory. EATCS Monographs in Theoretical Computer Science 18. Springer-Verlag Berlin (1989)
10. Fischer, P. C.: Turing machines with restricted memory access. Information and Control 9, 364–379 (1966)
11. Freund, R., Oswald, M.: P colonies working in the maximally parallel and in the sequential mode. In: Ciobanu, G., Păun, Gh. (eds.) Pre-Proc. 1st Intern. Workshop on Theory and Application of P Systems, pp. 49–56. Timisoara, Romania (2005)
12. Freund, R., Oswald, M.: P colonies and prescribed teams. International Journal of Computer Mathematics 83, 569–592 (2006)
13. Greibach, S. A.: Remarks on blind and partially blind one-way multicounter machines. Theoretical Computer Science 7(1), 311–324 (1978)
14. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading, Mass. (1979)
15. Kelemen, J., Kelemenová, A.: A grammar-theoretic treatment of multi-agent systems. Cybernetics and Systems 23, 621–633 (1992)
16. Kelemen, J., Kelemenová, A., Păun, Gh.: Preview of P colonies: A biochemically inspired computing model. In: Bedau, M. et al. (eds.) Workshop and Tutorial Proceedings. Ninth International Conference on the Simulation and Synthesis of Living Systems (Alife IX), pp. 82–86. Boston Mass. (2004)
17. Kelemenová, A.: P Colonies. Chapter 23.1, In: Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) The Oxford Handbook of Membrane Computing, pp. 584–593. Oxford University Press (2010)
18. Meduna, A., Zemek, P.: Jumping Finite Automata. Int. J. Found. Comput. Sci. 23, 1555–1578 (2012)
19. Minsky, M.: Computation – Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, NJ (1967)
20. Păun, Gh., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press (2010)

Solving SAT with Antimatter in Membrane Computing

Daniel Díaz-Pernil[1], Artiom Alhazov[2], Rudolf Freund[3],
Miguel A. Gutiérrez-Naranjo[4]

[1] Research Group on Computational Topology and Applied Mathematics
Department of Applied Mathematics - University of Sevilla, 41012, Spain
E-mail: sbdani@us.es
[2] Institute of Mathematics and Computer Science, Academy of Sciences of Moldova
Academiei 5, Chişinău, MD-2028, Republic of Moldova
E-mail: artiom@math.md
[3] Faculty of Informatics, Vienna University of Technology,
Favoritenstr. 9, 1040 Vienna, Austria
E-mail: rudi@emcc.at
[4] Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence, University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: magutier@us.es

Summary. The set of **NP**-complete problems is split into *weakly* and *strongly* **NP**-complete ones. The difference consists in the influence of the encoding scheme of the input. In the case of weakly **NP**-complete problems, the intractability depends on the encoding scheme, whereas in the case of strongly **NP**-complete problems the problem is intractable even if all data are encoded in a unary way. The reference for *strongly* **NP**-complete problems is the Satisfiability Problem (the **SAT** problem). In this paper, we provide a uniform family of P systems with active membranes which solves **SAT** – without polarizations, without dissolution, with division for elementary membranes and with matter/antimatter annihilation. To the best of our knowledge, it is the first solution to a *strongly* **NP**-complete problem in this P system model.

1 Introduction

In [7], a solution of the Subset Sum problem in the polynomial complexity class of recognizer P systems with active membranes without polarizations, without dissolution and with division for elementary membranes endowed with antimatter and matter/antimatter annihilation rules was provided. In this way, antimatter was shown to be a frontier of tractability in Membrane Computing, since the P systems class without antimatter and matter/antimatter annihilation rules is exactly the complexity class P (see [10]).

The Subset Sum problem belongs to the so-called *weakly* **NP**-complete problems, since its intractability strongly depends on the fact that extremely large input numbers are allowed [8]. The reason for this *weakness* is based on the encoding scheme of the input, since every integer in the input denoting a weight w_i should be encoded by a string of length only $O(\log w_i)$.

On the other hand, *strongly* **NP**-complete problems are those which remain **NP**-complete even if the data are encoded in a unary way. The best-known one of these problems is the satisfiability problem (**SAT** for short). **SAT** was the first problem shown to be **NP**-complete, as proved by Stephen Cook at the University of Toronto in 1971 [5], and it has been widely used in Membrane Computing to prove the ability of a P system model to solve **NP**-problems (e.g. [9, 11, 12, 14, 16, 17]).

In this paper, we provide a solution to the **SAT** problem in the polynomial complexity class of recognizer P systems with active membranes without polarizations, without dissolution and with division for elementary membranes endowed with antimatter and matter/antimatter annihilation rules. To the best of our knowledge, this is the first time that a *strongly* **NP**-complete problem is solved in this P system model. The details of the implementation can provide new tools for a better understanding of the problem of searching new frontiers of tractability in Membrane Computing.

The paper is organized as follows. In Section 2, we present a general discussion about the relationship of model ingredients used in different solutions for solving computationally difficult problems by P systems with active membranes, and the emerging computational power. In Section 3, we recall the P systems model used in this paper. The main novelty is the use of antimatter and matter/antimatter annihilation rules as well as their semantics. In Section 4, some basics on recognizer P systems are recalled, and in Section 5 our solution for the **SAT** problem is provided. The paper finishes with some conclusions and hints for future work.

2 Computation Theory Remarks

A configuration consists of symbols (which, in the general sense, may include instances of objects, instances of membranes, or any other entities bearing information). A computation consists of transformations of symbols. Clearly, the computations without cooperation of symbols are quite limited in power (e.g., it is known that $E0L$-behavior with standard halting yields $PsREG$, and accepting P systems are considerably more degenerate).

In this sense, interaction of symbols is a fundamental part of Membrane Computing, or of Theoretical Computer Science in general. Various ways of interaction of symbols have been studied in membrane computing. For the models with active membranes, the most commonly studied ways are various rules changing polarizations (or even sometimes labels), and membrane dissolution rules. One object may engage such a rule, which would affect the *context* (polarization or label) of other objects in the same membrane, thus affecting the behavior of the latter, e.g., in

case of dissolution, such objects find themselves in the parent membrane, which usually has a different label.

In the literature on P systems with active membranes, normally only the rules with at most one object on the left side were studied. Since recently, the model with matter/antimatter annihilation rules, e.g. see [1] and [2], attracted the attention of researchers. Clearly, it provides a form of *direct* object-object interaction, albeit in a rather restricted way (i.e., by erasing a pair of objects that are in a bijective relation). Although it is known that non-cooperative P systems with antimatter are already universal, studying their efficiency turned out to be an interesting line of research. So how does matter/antimatter annihilation compare to other ways of organizing interaction of objects?

First, all known solutions of **NP**-complete (or more difficult) problems in membrane computing rely on the possibility of P systems to obtain *exponential space* in polynomial time (note that object replication alone does not count as building exponential space, since an exponential number can be written, e.g. in binary, in polynomial space). Such possibility is provided by either of membrane division rules, membrane separation rules, membrane creation rules (or string replication rules, but string-objects lie outside of the scope of the current paper); in tissue P systems, one could apply similar approach to cells instead of membranes.

Note that in case of cell-like P systems, membrane creation alone (unlike the other types of rules mentioned above) makes it also possible to construct a hierarchy of membranes, let us refer to it as *structured workspace*, which is used to solve **PSPACE**-complete problems. The structured workspace can be alternatively created by elementary membrane division plus non-elementary membrane division (plus membrane dissolution if we have no polarizations).

Besides creating workspace, to solve **NP**-complete problems, we need to be able to effectively use that workspace, by making objects interact. For instance, it is known that, even with membrane division, without polarizations and without dissolution only problems in **P** may be solved. However, already with two polarizations (the smallest non-degenerate value) P systems can solve **NP**-complete problems. What can be done without polarizations?

One solution is to use the power of switching the context by membrane dissolution. Coupled with non-elementary division, a suitable membrane structure can be constructed so that the needed interactions can be performed solving **NP**-complete or even **PSPACE**-complete problems, [4]. It is not difficult to realize that elementary and non-elementary division rules can be replaced by membrane creation rules, or elementary division rules can be replaced by separation rules.

Finally, an alternative way of interaction of objects considered in this paper following [7] is matter/antimatter annihilation. What are the strengths and the weaknesses of these three possibilities (the weaker is an ingredient, the stronger is the result, while sometimes a weaker ingredient does not let us do what a stronger one can)?

The power of matter/antimatter annihilation makes it possible to carry out multiple simultaneous interactions (for example, the checking phase is constant-

time instead of linear with respect to the number of clauses), and it is a direct object-object interaction.

The power of polarizations is the possibility of mass action (not critical for studying computational efficiency within **PSPACE** as all multiplicities are bounded with respect to the problem size) by changing context.

The power of non-elementary division lets us build structured workspace (probably necessary for **PSPACE** if membrane creation is not used instead of membrane division, unless $\mathbf{P}^{\mathbf{PP}}$=**PSPACE**), see [13], and change non-local context (e.g., the label of the parent membrane).

The power of dissolution provides mass action (not critical for studying computational efficiency within **PSPACE** as all multiplicities are bounded with respect to the problem size) by changing context.

3 The P System Model

In this paper, we use the common rules of evolution, communication and division of elementary membranes which are usual in P systems with active membranes. The main novelty in the model is the use of antimatter and matter/antimatter annihilation rules. The concept of antimatter was introduced in the framework of Membrane Computing as a control tool for the flow of spikes in spiking neural P systems [15, 18, 22, 23]. In this context, when one spike and one anti-spike appear in the same neuron, the annihilation occurs and both, spike and anti-spike, disappear. Antimatter and matter/antimatter annihilation rules later were adapted to other contexts in Membrane Computing, and currently this an active research area [1, 2, 7].

Inspired by physics, we consider the annihilation of two objects a and b from the alphabet Γ in a membrane with label h, with the annihilation rule for a and b written as $[\ ab \to \lambda\ ]_h$. The *meaning* of the rule follows the idea of annihilation: If a and b occur simultaneously in the same membrane, then both are consumed (disappear) and nothing is produced (denoted by the empty string λ). The object b is called the *antiparticle* of a and it is usually written $\overline{a}$ instead of b.

With respect to the semantics, let us recall that this rule must be applied as many times as possible in each membrane, according to the maximal parallelism. Following the intuition from physics, if a and $\overline{a}$ occur simultaneously in the same membrane h and the annihilation rule $[\ a\overline{a} \to \lambda\ ]_h$ is defined, then it has to be applied, regardless any other option. In this sense, any annihilation rule has priority over all rules of the other types of rules (see [7]).

A P system with active membranes without polarizations, without dissolution and with division of elementary membranes and with annihilation rules is a cell-like P system with rules of the following kinds (following [3], we use subscript 0 for the rule type to represent a restriction that such rule does not depend on polarization and is now allowed to change it; if all rules have this subscript, this is equivalent to saying that the P system is without polarizations):

(a_0) $[\ a \rightarrow u\ ]_h$ for $h \in H$, $a \in \Gamma$, $u \in \Gamma^*$. This is an object evolution rule, associated with a membrane labeled by h: an object $a \in \Gamma$ belonging to that membrane evolves to a string $u \in \Gamma^*$.

(b_0) $a[\ \ ]_h \rightarrow [\ b\ ]_h$ for $h \in H$, $a, b \in \Gamma$. An object from the region immediately outside a membrane labeled by h is taken into this membrane, possibly being transformed into another object.

(c_0) $[\ a\ ]_h \rightarrow b[\ \ ]_h$ for $h \in H$, $a, b \in \Gamma$. An object is sent out from a membrane labeled by h to the region immediately outside, possibly being transformed into another object.

(e_0) $[\ a\ ]_h \rightarrow [\ b\ ]_h[\ c\ ]_h$ for $h \in H$, $a, b, c \in \Gamma$. An elementary membrane can be divided into two membranes with the same label, possibly transforming one original object into a different one in each of the new membranes.

(g_0) $[\ a\overline{a} \rightarrow \lambda\ ]_h$ for $h \in H$, $a, \overline{a} \in O$. This is an annihilation rule, associated with a membrane labeled by h: the pair of objects $a, \overline{a} \in O$ belonging simultaneously to this membrane disappears.

Let us remark that dissolution rules - type (d_0) - and rules for non-elementary division - type (f_0) - are not considered in this model.

These rules are applied according to the following principles (with the special restrictions for annihilation rules specified above):

- All the rules are applied in parallel and in a maximal manner. In one step, one object of a membrane can be used by at most one rule (chosen in a non–deterministic way), and each membrane can be the subject of *at most one* rule of types (b_0), (c_0) and (e_0).
- If at the same time a membrane labeled with h is divided by a rule of type (e_0) triggered by some object a and there are other objects in this membrane to which rules of type (a_0) or (g_0) can be applied, then we suppose that first the rules of type (g_0) and only then those of type (a_0) are used, before finally the division is executed. Of course, this process in total takes only one step.
- The rules associated with membranes labeled by h are used for all copies of membranes with label h.

4 Recognizer P Systems

Recognizer P systems are a well-known model of P systems which are basic for the study of complexity aspects in Membrane Computing. Next, we briefly recall some basic ideas related to them. For a detailed description, for example, see [19, 20]. In recognizer P systems all computations halt; there are two distinguished objects traditionally called `yes` and `no` (used to signal the result of the computation), and exactly one of these objects is sent out to the environment (only) in the last computation step.

Let us recall that a decision problem X is a pair (I_X, θ_X) where I_X is a language over a finite alphabet (the elements are called *instances*) and θ_X is a predicate

(a total Boolean function) over I_X. Let $X = (I_X, \theta_X)$ be a decision problem. A *polynomial encoding* of X is a pair (cod, s) of polynomial time computable functions over I_X such that for each instance $w \in I_X$, $s(w)$ is a natural number representing the *size* of the instance and $cod(w)$ is a multiset representing an encoding of the instance. Polynomial encodings are stable under polynomial time reductions.

Let $\mathcal{R}$ be a class of recognizer P systems with input membrane. A decision problem $X = (I_X, \theta_X)$ is solvable in a uniform way and polynomial time by a family $\mathbf{\Pi} = (\Pi(n))_{n\in\mathbf{N}}$ of P systems from $\mathcal{R}$ – we denote this by $X \in \mathbf{PMC}_{\mathcal{R}}$ – if the family $\mathbf{\Pi}$ is polynomially uniform by Turing machines, i.e., there exists a polynomial encoding (cod, s) from I_X to $\mathbf{\Pi}$ such that the family $\mathbf{\Pi}$ is polynomially bounded with regard to (X, cod, s); this means that there exists a polynomial function p such that for each $u \in I_X$ every computation of $\Pi(s(u))$ with input $cod(u)$ is halting and, moreover, it performs at most $p(|u|)$ steps; the family $\mathbf{\Pi}$ is sound and complete with regard to (X, cod, s).

5 Solving SAT

Propositional Satisfiability is the problem of determining, for a formula of the propositional calculus, if there is an assignment of truth values to its variables for which that formula evaluates to true. By **SAT** we mean the problem of propositional satisfiability for formulas in conjunctive normal form (CNF). In this section we describe a family of P systems which solves it. As usual, we will address the resolution via a brute force algorithm, which consists of the following stages (some of the ideas for the design are taken from [6] and [21]):

- *Generation and Evaluation Stage:* All possible assignments associated with the formula are created and evaluated (in this paper we have subdivided this group into *Generation* and *Input processing* groups of rules, which take place in parallel).
- *Checking Stage:* In each membrane we check whether or not the formula evaluates to true for the assignment associated with it.
- *Output Stage:* The systems sends out the correct answer to the environment.

Let us consider the pair function $\langle\ ,\ \rangle$ defined by $\langle n, m\rangle = ((n + m)(n + m + 1)/2) + n$. This function is polynomial-time computable (it is primitive recursive and bijective from $\mathbb{N}^2$ onto $\mathbb{N}$). For any given formula in CNF, $\varphi = C_1 \wedge \cdots \wedge C_m$, with m clauses and n variables $Var(\varphi) = \{x_1, \ldots, x_n\}$ we construct a P system $\Pi(\langle n, m\rangle)$ solving it, where the multiset encoding of the problem to be the input of $\Pi(\langle n, m\rangle)$ (for the sake of simplicity, in the following we will omit m and n) is

$$cod(\varphi) = \{x_{i,j} :\ x_j \in C_i\} \cup \{y_{i,j} :\ \ \neg x_j \in C_i\}.$$

For solving SAT by a uniform family of deterministic recognizer P systems with active membranes, without polarizations, without non-elementary membrane

division and without dissolution, yet with matter/antimatter annihilation rules, we now construct the members of this family as follows:

$$\begin{aligned}
&\Pi = (O, \Sigma, H = \{0,1\}, \mu = [\ [\ ]_2\ ]_1, w_1, w_2, R, i_{in} = 2), \text{ where}\\
&\Sigma = \{x_{i,j}, y_{i,j} \mid 1 \le i \le m,\ 1 \le j \le n\},\\
&O = \{d, t, f, F, \overline{F}, T, \overline{no}_{n+5}, \overline{F}_{n+5}, \overline{yes}_{n+6}, yes_{n+6}, no_{n+6}, yes, no\}\\
&\quad \cup \{x_{i,j}, y_{i,j} \mid 1 \le i \le m,\ -1 \le j \le n\} \cup \{\overline{x}_{i,-1}, \overline{y}_{i,-1} \mid 1 \le i \le m\}\\
&\quad \cup \{c_i, \overline{c}_i \mid 1 \le i \le m\} \cup \{e_j \mid 1 \le j \le n+3\}\\
&\quad \cup \{yes_j, no_j, F_j \mid 1 \le j \le n+5\},\\
&w_1 = no_0\ yes_0\ F_0,\ w_2 = d^n\ e_1,
\end{aligned}$$

and the rules of set R are given below, presented in groups Generation, Input processing, Checking and Output, together with explanations how the rules in the groups work.

Generation

G1. $[\ d\ ]_2 \to [\ t\ ]_2 [\ f\ ]_2$;
G2. $[\ t \to \overline{y}_{1,-1} \cdots \overline{y}_{m,-1}\ ]_2$;
G3. $[\ f \to \overline{x}_{1,-1} \cdots \overline{x}_{m,-1}\ ]_2$;
G4. $[\ \overline{x}_{i,-1} \to \lambda\ ]_2$, $1 \le i \le m$;
G5. $[\ \overline{y}_{i,-1} \to \lambda\ ]_2$, $1 \le i \le m$.

In each step j, $1 \le j \le n$, every elementary membrane is divided, one child membrane corresponding with assigning *true* to variable j and the other one with assigning *false* to it. One step later, proper objects are produced to annihilate the input objects associated to variable j: in the *true* case, we introduce the antimatter object for the negated variable, i.e., it will annihilate the corresponding negated variable, and in the *false* case, we introduce the antimatter object for the variable itself, i.e., it will annihilate the corresponding variable. Remaining barred (antimatter) objects not having been annihilated with the input objects, are erased in the next step.

Input processing

I1. $[\ x_{i,j} \to x_{i,j-1}\ ]_2$, $1 \le i \le m$, $0 \le j \le n$;
I2. $[\ y_{i,j} \to y_{i,j-1}\ ]_2$, $1 \le i \le m$, $0 \le j \le n$;
I3. $[\ x_{i,-1}\ \overline{x}_{i,-1} \to \lambda\ ]_2$, $1 \le i \le m$;
I4. $[\ y_{i,-1}\ \overline{y}_{i,-1} \to \lambda\ ]_2$, $1 \le i \le m$;
I5. $[\ x_{i,-1} \to c_i\ ]_2$, $1 \le i \le m$;
I6. $[\ y_{i,-1} \to c_i\ ]_2$, $1 \le i \le m$.

Input objects associated with variable j decrement their second subscript during $j+1$ steps to -1. The variables not representing the desired truth value are eliminated by the corresponding antimatter object generated by the rules G2 and G3, whereas any of the input variables not annihilated then, shows that the associated clause i is satisfied, which situation is represented by the introduction of the object c_i.

Checking

C1. $[\ e_j \to e_{j+1}\ ]_2$, $1 \leq j \leq n+1$;
C2. $[\ e_{n+2} \to \overline{c}_1 \cdots \overline{c}_m e_{n+3}\ ]_2$;
C3. $[\ c_i\ \overline{c}_i \to \lambda\ ]_2$, $1 \leq i \leq m$;
C4. $[\ \overline{c}_i \to F\ ]_2$, $1 \leq i \leq m$;
C5. $[\ e_{n+3} \to \overline{F}\ ]_2$;
C6. $[\ F\ \overline{F} \to \lambda\ ]_2$, $1 \leq i \leq m$;
C7. $[\ \overline{F}\ ]_2 \to [\ \ ]_2 T$.

It took $n+2$ steps to produce objects c_i for every satisfied clause, possibly multiple times. Starting from object e_1, we have obtained the object e_{n+2} until then; from this object e_{n+2}, at step $n+2$ one anti-object is produced for each clause. Any of these clause anti-objects that is not annihilated, is transformed into F, showing that the chosen variable assignment did not satisfy the corresponding clause. It remains to notice that object T is sent to the skin (at step $n+4$) if and only if an object $\overline{F}$ did not get annihilated, i.e., no clause failed to be satisfied.

Output

O1. $[\ yes_j \to yes_{j+1}\ ]_1$, $1 \leq j \leq n+5$;
O2. $[\ no_j \to no_{j+1}\ ]_1$, $1 \leq j \leq n+5$;
O3. $[\ F_j \to F_{j+1}\ ]_1$, $1 \leq j \leq n+4$;
O4. $[\ T \to \overline{no}_{n+5} \overline{F}_{n+5}\ ]_1$;
O5. $[\ no_{n+5}\ \overline{no}_{n+5} \to \lambda\ ]_1$;
O6. $[\ no_{n+6}\ ]_1 \to [\ \ ]_1 no$;
O7. $[\ F_{n+5}\ \overline{F}_{n+5} \to \lambda\ ]_1$;
O8. $[\ F_{n+5} \to \overline{yes}_{n+6}\ ]_1$;
O9. $[\ yes_{n+6}\ \overline{yes}_{n+6} \to \lambda\ ]_1$;
O10. $[\ yes_{n+6}\ ]_1 \to [\ \ ]_1 yes$.

If no object T has been sent to the skin, then the initial *no*-object can count up to $n+6$ and then sends out the negative answer *no*, while the initial F-object counts up to $n+5$, generates the antimatter object for the *yes*-object at stage $n+6$ and annihilates with the corresponding *yes*-object at stage $n+6$. On the other hand, if (at least one) object T arrives in the skin, then the *no*-object is annihilated at stage $n+5$ before it would be sent out in the next step, and the F-object is annihilated before it could annihilate with the *yes*-object, so that the positive answer *yes* can be sent out in step $n+6$.

Finally, we notice that the solution is uniform, deterministic, and uses only rules of types (a_0), (c_0), (e_0) as well as matter/antimatter annihilation rules. The result is produced in $n+6$ steps.

6 Conclusions

Although the ability of the model for solving NP problems was proved in [7], to the best of our knowledge, this is the first solution to a strongly **NP** problem by using

annihilation rules in Membrane Computing. Let us remark the important role of the definition for recognizer P systems we have used in this paper. This definition is quite restrictive, since only one object *yes* or *no* is sent to the environment in any computation. In the literature one can find other definitions of recognizer P systems and therefore other definitions of what it means *to solve* a problem in the framework of Membrane Computing. The study of the complexity classes in Membrane Computing deserves a deep revision under these new definitions.

Acknowledgements

M.A. Gutiérrez-Naranjo acknowledges the support of the project TIN2012-37434 of the Ministerio de Economía y Competitividad of Spain.

References

1. Artiom Alhazov, Bogdan Aman, and Rudolf Freund. P systems with anti-matter. In Marian Gheorghe, Grzegorz Rozenberg, Arto Salomaa, Petr Sosík, and Claudio Zandron, editors, *Membrane Computing - 15th International Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers*, volume 8961 of *Lecture Notes in Computer Science*, pages 66–85, Springer, 2014.
2. Artiom Alhazov, Bogdan Aman, Rudolf Freund, and Gheorghe Păun. Matter and anti-matter in membrane systems. In *DCFS 2014*, volume 8614 of *Lecture Notes in Computer Science*, pages 65–76, Springer, 2014.
3. Artiom Alhazov, Linqiang Pan, and Gheorghe Păun. Trading polarizations for labels in P systems with active membranes. *Acta Informatica*, 41(2-3): 111–144, 2004.
4. Artiom Alhazov and Mario J. Pérez-Jiménez. Uniform solution of QSAT using polarizationless active membranes. In *MCU 2007*, volume 4664 of *Lecture Notes in Computer Science*, pages 122–133, Springer, 2007.
5. Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, ACM, New York, NY, USA, 1971.
6. Andrés Cordón-Franco, Miguel A. Gutiérrez-Naranjo, Mario J. Pérez-Jiménez, and Fernando Sancho-Caparrini. A Prolog simulator for deterministic P systems with active membranes. *New Generation Computing*, 22(4): 349–363, 2004.
7. Daniel Díaz-Pernil, Francisco Peña-Cantillana, Artiom Alhazov, Rudolf Freund, and Miguel A. Gutiérrez-Naranjo. Antimatter as a frontier of tractability in membrane computing. *Fundamenta Informaticae*, 134: 83–96, 2014.
8. Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.
9. Zsolt Gazdag and Gábor Kolonits. A new approach for solving SAT by P systems with active membranes. In Erzsébet Csuhaj-Varjú, Marian Gheorghe, Grzegorz Rozenberg, Arto Salomaa, and György Vaszil, editors, *International Conference on Membrane Computing*, volume 7762 of *Lecture Notes in Computer Science*, pages 195–207, Springer, 2012.

10. Miguel A. Gutiérrez-Naranjo, Mario J. Pérez-Jiménez, Agustin Riscos-Núñez, and Francisco José Romero-Campero. On the power of dissolution in P systems with active membranes. In Rudolf Freund, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Workshop on Membrane Computing*, volume 3850 of *Lecture Notes in Computer Science*, pages 224–240, Springer, 2005.
11. Miguel A. Gutiérrez-Naranjo, Mario J. Pérez-Jiménez, and Francisco José Romero-Campero. A uniform solution to SAT using membrane creation. *Theoretical Computer Science*, 371(1-2): 54–61, 2007.
12. Tseren-Onolt Ishdorj and Alberto Leporati. Uniform solutions to SAT and 3-SAT by spiking neural P systems with pre-computed resources. *Natural Computing*, 7(4): 519–534, 2008.
13. Alberto Leporati, Luca Manzoni, Giancarlo Mauri, Antonio E. Porreca, Claudio Zandron. Simulating elementary active membranes - with an application to the P conjecture. In Marian Gheorghe, Grzegorz Rozenberg, Arto Salomaa, Petr Sosík, Claudio Zandron, editors, *Conference on Membrane Computing*, volume 8961 of *Lecture Notes in Computer Science*, pages 284–299, Springer, 2014.
14. Alberto Leporati, Giancarlo Mauri, Claudio Zandron, Gheorghe Păun, and Mario J. Pérez-Jiménez. Uniform solutions to SAT and subset sum by spiking neural P systems. *Natural Computing*, 8(4): 681–702, 2009.
15. Venkata Padmavati Metta, Kamala Krithivasan, and Deepak Garg. Computability of spiking neural P systems with anti-spikes. *New Mathematics and Natural Computation (NMNC)*, 08(03): 283–295, 2012.
16. Adam Obtulowicz. Deterministic P-systems for solving SAT-problem. *Romanian Journal of Information Science and Technology*, 4(1-2): 195–201, 2001.
17. Linqiang Pan and Artiom Alhazov. Solving HPP and SAT by P systems with active membranes and separation rules. *Acta Informatica*, 43(2): 131–145, 2006.
18. Linqiang Pan and Gheorghe Păun. Spiking neural P systems with anti-spikes. *International Journal of Computers, Communications & Control*, IV(3): 273–282, September 2009.
19. Mario J. Pérez-Jiménez. An approach to computational complexity in membrane computing. In Giancarlo Mauri, Gheorghe Păun, Mario J. Pérez-Jiménez, Grzegorz Rozenberg, and Arto Salomaa, editors, *Workshop on Membrane Computing*, volume 3365 of *Lecture Notes in Computer Science*, pages 85–109, Springer, 2004.
20. Mario J. Pérez-Jiménez, Agustin Riscos-Núñez, Álvaro Romero-Jiménez, and Damien Woods. Complexity - membrane division, membrane creation. In Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *The Oxford Handbook of Membrane Computing*, pages 302 – 336. Oxford University Press, Oxford, England, 2010.
21. Mario J. Pérez-Jiménez, Álvaro Romero-Jiménez, and Fernando Sancho-Caparrini. Complexity classes in models of cellular computing with membranes. *Natural Computing*, 2(3): 265–285, 2003.
22. Tao Song, Yun Jiang, Xiaolong Shi, and Xiangxiang Zeng. Small universal spiking neural P systems with anti-spikes. *Journal of Computational and Theoretical Nanoscience*, 10(4): 999–1006, 2013.
23. Gangjun Tan, Tao Song, Zhihua Chen, and Xiangxiang Zeng. Spiking neural P systems with anti-spikes and without annihilating priority working in a 'flip-flop' way. *International Journal of Computing Science and Mathematics*, 4(2): 152–162, July 2013.

On The Semantics of Annihilation Rules in Membrane Computing

Daniel Díaz-Pernil[1], Rudolf Freund[2],
Miguel A. Gutiérrez-Naranjo[3], Alberto Leporati[4]

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

[2]Faculty of Informatics
Vienna University of Technology, Austria
rudi@emcc.at

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

[4]Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano-Bicocca, Italy
alberto.leporati@unimib.it

Summary. It is well known that polarizationless recognizer P systems with active membranes, without dissolution, with division of elementary and non-elementary membranes, with antimatter and matter/antimatter annihilation rules can solve all problems in **NP** when the annihilation rules have (weak) priority over all the other rules. Until now, it was an open problem whether these systems can still solve all **NP** problems if the priority of the matter/antimatter annihilation rules is removed.

In this paper we provide a negative answer to this question: we prove that the class of problems solvable by this model of P systems without priority of the matter/antimatter annihilation rules is exactly **P**. To the best of our knowledge, this is the first paper in the literature of P systems where the semantics of applying the rules constitutes a frontier of tractability.

1 Introduction

The concept of antimatter was first introduced in the framework of membrane computing as a control tool for the flow of spikes in spiking neural P systems [6, 5, 10, 11]). In this context, when one spike and one anti-spike appear in the same neuron, the annihilation occurs and both, spike and anti-spike, disappear. The concept of antimatter and matter/antimatter annihilation rules later was

adapted to other contexts in membrane computing, and currently it is an active research area [1, 2, 3].

In [3], the authors show that antimatter and matter/antimatter annihilation rules are a frontier of tractability. The starting point is a well-known result in the complexity theory of membrane computing: the decision problems which can be solved by polarizationless recognizer P systems with active membranes, without dissolution and with division of elementary and non-elementary membranes (denoted by $\mathcal{AM}^0_{-d,+ne}$) are exactly those in the complexity class **P** (see [4], Th. 2). The main result in [3] is that systems from $\mathcal{AM}^0_{-d,+ne}$ endowed with antimatter and matter/antimatter annihilation rules (denoted by $\mathcal{AM}^0_{-d,+ne,+ant}$) can solve all problems in **NP** and, hence, annihilation rules constitute a frontier of tractability.

In this paper, we revisit the question of determining the computational complexity of the problems which can be solved by P systems with the matter/antimatter annihilation rules not having priority over all the other rules. As previously pointed out (see [3, 9]), the solution presented in [3] to an **NP**-complete problem, namely Subset Sum, uses this weak priority of the annihilation rules, and until now it has been an open problem if the model $\mathcal{AM}^0_{-d,+ne,+ant}$ is still capable to solve **NP**-complete problems *without this priority*. In this paper we show that the answer to this open question is negative. We prove that the complexity class of decision problems solvable by $\mathcal{AM}^0_{-d,+ne,+ant}$ systems is exactly equal to **P** if the priority relation is removed from the semantics for the annihilation rules.

In this way, we propose a new kind of frontier of tractability. Up to now, these frontiers were based on *syntactic ingredients* of the P systems, that is, the type of rules and not the *way* in which such rules are applied. In this paper, the frontier of tractability is based on the *semantics* of the P system, i.e., on the way the rules are applied.

The paper is organised as follows. First, we recall some concepts about recognizer P systems, antimatter, matter/antimatter annihilation rules and the model $\mathcal{AM}^0_{-d,+ne,+ant}$. Next, we prove our main result of computational complexity. The paper ends with some final considerations.

2 Recognizer P Systems

First of all, we recall the main notions related to recognizer P systems and computational complexity in membrane computing. For a detailed description see, for example, [7, 8].

The main *syntactic* ingredients of a cell–like P system are the *membrane structure*, the *multisets*, and the *evolution rules*. A *membrane structure* consists of several membranes arranged hierarchically inside a main membrane, called the *skin*. Each membrane identifies a region inside the system. When a membrane has no membrane inside, it is called *elementary*. The objects are instances of symbols from a finite alphabet, and *multisets of objects* are placed in the regions determined by

the membrane structure. The objects can evolve according to given *evolution rules*, associated with the regions.

The *semantics* of cell–like P systems is defined through a non-deterministic and synchronous model. A *configuration* of a cell-like P system consists of a membrane structure and a sequence of multisets of objects, each associated with one region of the structure. At the beginning of the computation, the system is in the *initial configuration*, which possibly comprises an *input multiset*. In each time step the system transforms its current configuration into another configuration by applying the evolution rules to the objects placed inside the regions of the system, in a non-deterministic and maximally parallel manner (the precise semantics will be described later). In this way, we get *transitions* from one configuration of the system to the next one. A *computation* of the system is a (finite or infinite) sequence of configurations such that each configuration –except the initial one– is obtained from the previous one by a transition. A computation which reaches a configuration where no more rules can be applied to the existing objects and membranes, is called a *halting computation*. The result of a halting computation is usually defined by the multiset associated with a specific output membrane (or the environment) in the final configuration.

In this paper we deal with *recognizer* P systems, where all computations halt and exactly one of the distinguished objects *yes* and *no* is sent to the environment, and only in the last step of any computation, in order to signal acceptance or rejection, respectively. All recognizer P systems considered in this paper are *confluent*, meaning that if computations start from the same initial configuration then either all are accepting or all are rejecting.

Recognizer P systems can thus be used to recognize formal languages (equivalently, solve decision problems). Let us recall that a decision problem X is a pair (I_X, θ_X) where I_X is a language over a finite alphabet and θ_X is a predicate (a total Boolean function) over I_X. The elements of I_X are called *instances* of the problem, and those for which predicate θ_X is true (respectively false) are called *positive* (respectively *negative*) instances. A *polynomial encoding* of a decision problem X is a pair (cod, s) of functions over I_X, computable in polynomial time by a deterministic Turing machine, such that for each instance $u \in I_X$, $s(u)$ is a natural number representing the *size* of the instance and $cod(u)$ is a multiset representing an encoding of the instance. Polynomial encodings are stable under polynomial time reductions.

2.1 The Class $\mathcal{AM}^{\mathbf{0}}_{-d,+ne}$

A P system with active membranes without polarizations, without dissolution and with division of elementary and non-elementary membranes is a P system with Γ as the alphabet of symbols, with H as the finite set of labels for membranes, and where the rules are of the following forms:

(a_0) $[\, a \to u \,]_h$ for $h \in H$, $a \in \Gamma$, $u \in \Gamma^*$. This is an *object evolution rule*, associated with the membrane labelled with h. When the rule is applied, an object $a \in \Gamma$

inside that membrane is rewritten into the multiset $u \in \Gamma^*$. (Note that here and in the rest of the paper, we write $u \in \Gamma^*$ to indicate both the multiset u of objects from the alphabet Γ and one of the possible strings which represent it.)

(b_0) $a\,[\ ]_h \to [\,b\,]_h$ for $h \in H$, $a, b \in \Gamma$ *(send-in rules)*. An object from the region immediately outside a membrane labelled with h is sent into this membrane, possibly transformed into another object.

(c_0) $[\,a\,]_h \to b\,[\ ]_h$ for $h \in H$, $a, b \in \Gamma$ *(send-out rules)*. An object is sent out from the membrane labelled with h to the region immediately outside, possibly transformed into another object.

(d_0) $[\,a\,]_h \to [\,b\,]_h\,[\,c\,]_h$ for $h \in H$, $a, b, c \in \Gamma$ *(division rules for elementary membranes)*. An elementary membrane can be divided into two membranes with the same label; object a in the original membrane is rewritten to b (respectively to c) in the first (respectively second) generated membrane.

(e_0) $[\,[\,]_{h_1}[\,]_{h_2}\,]_{h_0} \to [\,[\,]_{h_1}\,]_{h_0}\ [\,[\,]_{h_2}\,]_{h_0}$, for $h_0, h_1, h_2 \in H$ *(division rules for non-elementary membranes)*. If the membrane with label h_0 contains other membranes than those with labels h_1, h_2, then such membranes and their contents are duplicated and placed in both new copies of the membrane h_0; all membranes and objects placed inside membranes h_1, h_2, as well as the objects from membrane h_0 placed outside membranes h_1 and h_2, are reproduced in the new copies of membrane h_0.

These rules are applied according to the following principles:

- All the rules are applied in parallel and in a maximal manner. In one step, one object of a membrane can be used by at most one rule (chosen in a non–deterministic way), and each membrane can be the subject of *at most one* rule of types (b_0), (c_0), (d_0), and (e_0).
- If at the same time a membrane labelled with h is divided by a rule of type (d_0) or (e_0) and there are objects in this membrane which evolve by means of rules of type (a_0), then we suppose that first the evolution rules of type (a_0) are used, and then the division is produced. Of course, this process takes only one step.
- The rules associated with membranes labelled with h are used for all copies of this membrane with label h.

The class of all polarizationless recognizer P systems with active membranes, without dissolution and with division of elementary and non-elementary membranes is denoted by $\mathcal{AM}^0_{-d,+ne}$.

2.2 Polynomial Complexity Classes in Recognizer P Systems

Let $\mathcal{R}$ be a class of recognizer P systems. A decision problem $X = (I_X, \theta_X)$ is solvable in a *semi-uniform* way and in polynomial time by a family of recognizer P systems $\mathbf{\Pi} = \{\Pi(w)\}_{w \in I_X}$ of type $\mathcal{R}$, denoted by $X \in \mathbf{PMC}^{\star}_{\mathcal{R}}$, if $\mathbf{\Pi}$ is *polynomially uniform* by Turing machines, that is, there exists a deterministic Turing machine

working in polynomial time which constructs the system $\Pi(w)$ from the instance $w \in I_X$, and $\mathbf{\Pi}$ is *polynomially bounded*, that is, there exists a polynomial function $p(n)$ such that for each $w \in I_X$, all computations of $\Pi(w)$ halt in at most $p(|w|)$ steps. It is said that $\mathbf{\Pi}$ is *sound* with regard to X if for each instance of the problem $w \in I_X$, if there exists an accepting computation of $\Pi(w)$ then $\theta_X(w)$ is true, and $\mathbf{\Pi}$ is *complete* with regard to X if for each instance of the problem $w \in I_X$, if $\theta_X(w)$ is true then every computation of $\Pi(w)$ is an accepting computation.

Let $\mathcal{R}$ be a class of recognizer P systems with a distinguished input membrane, and let $\mathbf{\Pi} = \{\Pi(n)\}_{n\in\mathbf{N}}$ be a family of recognizer P systems of type $\mathcal{R}$. A decision problem $X = (I_X, \theta_X)$ is solvable in a *uniform way* and polynomial time by $\mathbf{\Pi}$, denoted by $X \in \mathbf{PMC}_{\mathcal{R}}$, if $\mathbf{\Pi}$ is *polynomially uniform* by Turing machines, i.e., there exists a polynomial encoding[1] (cod, s) such that the family $\mathbf{\Pi}$ is *polynomially bounded* with regard to (X, cod, s); that is, there exists a polynomial function $p(n)$ such that for each $u \in I_X$, every computation of $\Pi(s(u))$ with input $cod(u)$ – denoted by $\Pi(s(u)) + cod(u)$, for short– is halting and, moreover, it performs at most $p(|u|)$ steps, and the family $\mathbf{\Pi}$ is *sound* and *complete* with regard to (X, cod, s). It is easy to see that the classes $\mathbf{PMC}^{\star}_{\mathcal{R}}$ and $\mathbf{PMC}_{\mathcal{R}}$ are closed under polynomial-time reduction and complement. Moreover, since uniformity can be considered to be a special case of semi-uniformity, the inclusion $\mathbf{PMC}_{\mathcal{R}} \subseteq \mathbf{PMC}^{\star}_{\mathcal{R}}$ holds.

According to these formal definitions, in [4] it is proved that the complexity class of decision problems solved by uniform or semi-uniform families of polarizationless recognizer P systems with active membranes, without dissolution and with division of elementary and non-elementary membranes, is exactly $\mathbf{P}$. With the standard notation, $\mathbf{P} = \mathbf{PMC}_{\mathcal{AM}^0_{-d,+ne}} = \mathbf{PMC}^{\star}_{\mathcal{AM}^0_{-d,+ne}}$.

2.3 Antimatter

Antimatter and matter/antimatter annihilation rules have been introduced in the framework of cell-like P systems in [2]. Given two objects a and b from the alphabet Γ in a membrane labeled by h, an annihilation rule of a and b is written as $[ab \to \lambda]_h$. The *meaning* of the rule follows the physical idea of annihilation: If a and b occur simultaneously in the same region with label h, then both are consumed (disappear) and nothing is produced (denoted by the empty string λ). Let us remark that both objects a and b are ordinary elements from Γ and they can trigger any other rule of type (a_0) to (d_0) described above, not only annihilation rules. Nonetheless, in order to improve the readability, if b annihilates the object a then b will be called the *antiparticle* of a and we will write $\overline{a}$ instead of b.

[1] See [7, 8] for the details. Informally, given an instance $u \in I_X$, $s(u)$ is a natural number which identifies a P system $\Pi(s(u))$ in the family. When fed with the multiset $cod(u)$ as input, this P system computes the value of predicate $\theta_X(u)$. In uniform families of P systems, the structure and definition of $\Pi(s(u))$ is the same for all instances $u \in I_X$ having the same size $s(u)$.

With respect to the *semantics*, let us recall that the rule $[a\overline{a} \to \lambda]_h$, provided that annihilation rules have priority over all other rules, must be applied as many times as possible in every membrane labeled by h, according to the maximal parallelism, i.e., if m copies of a and n copies of $\overline{a}$ occur simultaneously in a membrane of label h, with $m \geq n$ (respectively $m \leq n$), then the rule is applied n times (respectively m times), n (respectively m) copies of a and $\overline{a}$ are consumed and $m - n$ copies of a (respectively $n - m$ copies of $\overline{a}$) are not affected by this rule.

The key point in the use of the semantics of the annihilation rules in this paper is related to the priority of this type of rules with respect to the other types. In [3], according to the non-determinism, if an object a can trigger more than one rule of types (a_0) to (d_0), then one rule among the applicable ones is non-deterministically chosen. Nonetheless, if a and $\overline{a}$ occur simultaneously in the same membrane h and the annihilation rule $[a\overline{a} \to \lambda]_h$ is defined, then it is applied, regardless of other options. In this sense, any annihilation rule had priority over the other types of rules.

In this paper, we consider the case that the annihilation does not have priority over the other rules. If an object a can trigger more than one rule, then one rule among the applicable ones is non-deterministically chosen regardless of its type (obviously, for annihilation rules object $\overline{a}$ has also to occur in the same region).

Formally, a polarizationless P system with active membranes, without dissolution, with division of elementary and non-elementary membranes and with annihilation rules is a construct of the form $\Pi = (\Gamma, H, \mu, w_1, \ldots, w_m, R)$, where:

1. $m \geq 1$ is the initial degree of the system;
2. Γ is the alphabet of *objects*;
3. H is a finite set of *labels* for membranes;
4. μ is a *membrane structure* consisting of m membranes labelled in a one-to-one way with elements of H;
5. $w_1, \ldots, w_m$ are strings over Γ, describing the *multisets of objects* placed in the m regions of μ;
6. R is a finite set of *rules* of the types (a_0) to (e_0) described in Section 2.1, and the following type of rules:

   (f_0) $[\, a\overline{a} \to \lambda \,]_h$ for $h \in H$, $a, \overline{a} \in \Gamma$ *(annihilation rules)*. The pair of objects $a, \overline{a} \in \Gamma$ occurring simultaneously inside membrane h disappears.

As stated above, in this paper rules of type (f_0) have no priority over the other types of rules. If at the same time a membrane labelled with h is divided by a rule of type (d_0) or (e_0) and there are objects in this membrane which are chosen to be annihilated by means of rules of type (f_0), then we assume that first the annihilation is performed and then the division is produced. Of course, this process takes only one step.

By following the standard notation, in [3] the authors denote the class of polarizationless recognizer P systems with active membranes without dissolution, with division of elementary and non-elementary membranes, and with antimatter and matter/antimatter annihilation rules by $\mathcal{AM}^0_{-d,+ne,+ant}$. They do not include

any symbol in the name to specify the priority, because they assume it as being part of the model definition. In this paper, we will consider a class of P systems which uses the same model of P systems $\mathcal{AM}^0_{-d,+ne,+ant}$, but without priority for the application of the annihilation rules; in order to stress this difference, we will denote this class of P systems by $\mathcal{AM}^0_{-d,+ne,+ant\_NoPri}$.

3 Removing Priority for Annihilation Rules

The main contribution of this paper is the proof of the following claim.

Theorem 1. $\mathbf{PMC}_{\mathcal{AM}^0_{-d,+ne,+ant\_NoPri}} = \mathbf{P}$

Proof. It is well known (e.g., see [4]) that $\mathbf{PMC}_{\mathcal{AM}^0_{-d,+ne}} = \mathbf{PMC}^{\star}_{\mathcal{AM}^0_{-d,+ne}} = \mathbf{P}$. On the other hand, the following inclusion obviously holds:

$$\mathbf{PMC}_{\mathcal{AM}^0_{-d,+ne}} \subseteq \mathbf{PMC}_{\mathcal{AM}^0_{-d,+ne,+ant\_NoPri}},$$

therefore $\mathbf{P} \subseteq \mathbf{PMC}_{\mathcal{AM}^0_{-d,+ne,+ant\_NoPri}}$. Thus it only remains to prove that also the converse inclusion holds:

$$\mathbf{PMC}_{\mathcal{AM}^0_{-d,+ne,+ant\_NoPri}} \subseteq \mathbf{P}. \tag{1}$$

Since $\mathbf{PMC}^{\star}_{\mathcal{AM}^0_{-d,+ne}} = \mathbf{P}$, in order to prove (1) it suffices to prove that $\mathbf{PMC}_{\mathcal{AM}^0_{-d,+ne,+ant\_NoPri}} \subseteq \mathbf{PMC}^{\star}_{\mathcal{AM}^0_{-d,+ne}}$.

Hence, let $X \in \mathbf{PMC}_{\mathcal{AM}^0_{-d,+ne,+ant\_NoPri}}$ be a decision problem. By definition, there exist a polynomial encoding (cod, s) and a family of P systems $\{\Pi(i)\}_{i\in\mathbb{N}}$ in $\mathcal{AM}^0_{-d,+ne,+ant\_NoPri}$ such that for each instance u of the problem X:

- all computations of $\Pi(s(u)) + cod(u)$ halt;
- in all computations, the system sends out either one copy of the object *yes* or one copy of the object *no* (but not both), and only in the last step of computation.

Let us first provide an informal idea of the proof. Given an instance $u \in I_X$, we know that all computations of $\Pi(s(u)) + cod(u)$ halt, and that they all answer *yes* or all answer *no*. Let $\mathcal{C} = \{C_0, \ldots, C_n\}$ be one of these halting computations, and let us assume that the answer is *yes* (the other case is analogous). Then there exists an object a_1 and a rule $r_1 \equiv [\, a_1 \,]_{skin} \to yes\, [\,]_{skin}$ which has been applied in the last step of the computation. There are two possibilities: either object a_1 is in the skin membrane since the beginning of the computation, or there exists a rule r_2 which must have produced it inside or moved it into the skin membrane. Rule r_2 is triggered by the occurrence of an object a_2 in a membrane with label h_2. Obviously, r_2 cannot be an annihilation rule, since no object is produced by such rules, then rule r_2 must belong to types (a_0) to (d_0). Going back with the

reasoning, either a_2 appears in the membrane with label h_2 since the beginning of the computation, or it is produced or moved there by the application of a rule r_3, and so on.

Finally we have a chain

$$(yes, env) \xleftarrow{r_1} (a_1, skin) \xleftarrow{r_2} (a_2, h_2) \xleftarrow{r_3} \cdots \xleftarrow{r_k} (a_k, h_k)$$

where $k \leq n$ and a_k appears in a membrane with label h_k in the initial configuration (possibly as part of the input multiset). The key idea here is two-folded. On the one hand, annihilation rules do not produce any object; the objects that trigger an annihilation rule disappear and nothing is produced. On the other hand, for any halting configuration there *must* exist a finite sequence of rules $(r_k, r_{k-1}, \ldots, r_2, r_1)$ where r_k is triggered by an object from the initial configuration, r_1 produces yes and each r_i produces an object that triggers r_{i-1}. Therefore, none of rules $r_1, \ldots, r_k$ is an annihilation rule.

To formally prove the result we have to check that the amount of resources for finding the sequence of rules is polynomially bounded. With this aim, we will start by considering the dependency graph associated with $\Pi(s(u))$, but considering only evolution, communication and division rules[2] (i.e., only rules which can produce new occurrences of objects). Namely, if R is the set of rules associated with $\Pi(s(u))$, we will consider the corresponding directed graph $G = (V, E)$ defined as follows, where the function $f : H \to H$ returns the label of the parent membrane:

$V = VL \cup VR$,

$$\begin{aligned} VL = \{(a,h) \in \Gamma \times H :\ & \exists u \in \Gamma^* \ ([a \to u]_h \in R) \vee \\ & \exists b \in \Gamma \ ([a]_h \to [\ ]_h b \in R) \vee \\ & \exists b \in \Gamma \ \exists h' \in H \ (h = f(h') \wedge a[\ ]_{h'} \to [b]_{h'} \in R) \vee \\ & \exists b, c \in \Gamma \ ([a]_h \to [b]_h [c]_h \in R)\}, \end{aligned}$$

$$\begin{aligned} VR = \{(b,h) \in \Gamma \times H :\ & \exists a \in \Gamma \ \exists u \in \Gamma^* \ ([a \to u]_h \in R \wedge b \in u) \vee \\ & \exists a \in \Gamma \ \exists h' \in H \ (h = f(h') \wedge [a]_{h'} \to [\ ]_{h'} b \in R) \vee \\ & \exists a \in \Gamma \ (a[\ ]_h \to [b]_h \in R) \vee \\ & \exists a, c \in \Gamma \ ([a]_h \to [b]_h [c]_h \in R)\}, \end{aligned}$$

$$\begin{aligned} E = \{((a,h),(b,h')) : \ & \exists u \in \Gamma^* \ ([a \to u]_h \in R \wedge b \in u \wedge h = h') \vee \\ & ([a]_h \to [\ ]_h b \in R \ \wedge \ h' = f(h)) \vee \\ & (a[\ ]_{h'} \to [b]_{h'} \in R \ \wedge h \ = f(h')) \vee \\ & \exists c \in \Gamma \ ([a]_h \to [b]_h [c]_h \in R \wedge h = h')\}. \end{aligned}$$

Such a dependency graph can be constructed by a Turing machine working in polynomial time with respect to the instance size. Finally, let us consider the set

[2] See [4] for the details about polynomial resources.

$$\Delta_\Pi = \{(a,h) \in \Gamma \times H : \text{there exists a path (within the dependency graph) from } (a,h) \text{ to } (yes, env)\}.$$

It has also been proved that there exists a Turing machine that constructs Δ_Π in polynomial time; the proof uses the *Reachability Problem* in order to prove the polynomially bounded construction.

From this construction we directly obtain that the set of rules used in the chain

$$(yes, env) \xleftarrow{r_1} (a_1, skin) \xleftarrow{r_2} (a_2, h_2) \xleftarrow{r_3} \cdots \xleftarrow{r_k} (a_k, h_k)$$

described above can be found in polynomial time.

Finally, for the instance $u \in I_X$, let us consider the P system $\Pi(u')$ with only one membrane with label s and only one object (a_k, h_k) in the initial configuration. The set of rules is

- $[(a_i, h_i) \to (a_{i-1}, h_{i-1})]_s$ for each $i \in \{3, \ldots, k-1\}$
- $[(a_2, h_2) \to (a_1, skin)]_s$
- $[(a_1, skin)]_s \to yes\,[\,]_s$

The system $\Pi(u')$ can be built in polynomial time by a deterministic Turing machine. A direct inspection of the rules shows that $\Pi(u') \in \mathcal{AM}^0_{-d,+ne}$. The behavior of the system is deterministic, and it computes the correct answer for the instance $u \in I_X$, sending out the object *yes* to the environment in the last step of computation.

We finally observe that a similar construction can be carried out for the answer *no*. Hence, we conclude that $X \in \mathbf{PMC}^\star_{\mathcal{AM}^0_{-d,+ne}} = \mathbf{P}$. □

Remark 1. Let us finally explain the idea how to even get a uniform family of recognizer P systems from the family constructed in the preceding proof by making some preprocessing: For any input of length n, we include all possible input symbols in the dependency graph. If there is a path from some symbol to *yes* and from another symbol to *no*, then by the definition of confluence, an input containing both of these symbols simultaneously cannot be a valid input. So, once we get an input of length n, we first check if it has symbols deriving *yes* and symbols deriving *no*. This certainly is possible within polynomial time.

4 Conclusions

We have proved that by removing priority in polarizationless recognizer P systems with antimatter and annihilation rules, without dissolution, and with division of elementary and non-elementary membranes, we obtain a new characterization of the standard complexity class **P**. Since it was previously known that the same model of P systems can solve the **NP**-complete problem Subset Sum when the priority of annihilation rules is used [3], we have shown that this priority plays an important role in the computational power of these P systems.

Indeed, the most interesting aspect of our result is the fact that if the rules of these P systems are applied in different ways, a different computational power is obtained. We have thus proved that the semantics of a model can be a useful tool for studying problems of tractability. To the best of our knowledge, this is the first time where it is proved that two models of P systems syntactically identical correspond to two (presumably) different complexity classes simply because they use different semantics.

This opens a new research area in the study of tractability in membrane computing. Not only new ingredients or new models must be studied in order to find new frontiers: classical results can also be revisited in order to explore the consequences of considering alternative semantics.

Acknowledgements

Miguel A. Gutiérrez-Naranjo acknowledges the support of the project TIN2012-37434 of the Ministerio de Economía y Competitividad of Spain. The authors are very grateful to Artiom Alhazov for carefully reading the paper and for also pointing out Remark 1.

References

1. Alhazov, A., Aman, B., Freund, R.: P systems with anti-matter. In: Gheorghe, M., Rozenberg, G., Salomaa, A., Sosík, P., Zandron, C. (eds.) Membrane Computing - 15th International Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8961, pp. 66–85. Springer (2014)
2. Alhazov, A., Aman, B., Freund, R., Păun, Gh.: Matter and anti-matter in membrane systems. In: DCFS 2014. Lecture Notes in Computer Science, vol. 8614, pp. 65–76. Springer (2014)
3. Díaz-Pernil, D., Peña-Cantillana, F., Alhazov, A., Freund, R., Gutiérrez-Naranjo, M.A.: Antimatter as a frontier of tractability in membrane computing. Fundamenta Informaticae 134, 83–96 (2014)
4. Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J., Riscos-Núñez, A., Romero-Campero, F.J.: On the power of dissolution in P systems with active membranes. In: Freund, R., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) Workshop on Membrane Computing. Lecture Notes in Computer Science, vol. 3850, pp. 224–240. Springer, Berlin Heidelberg (2005)
5. Metta, V.P., Krithivasan, K., Garg, D.: Computability of spiking neural P systems with anti-spikes. New Mathematics and Natural Computation (NMNC) 08(03), 283–295 (2012)
6. Pan, L., Păun, Gh.: Spiking neural P systems with anti-spikes. International Journal of Computers, Communications & Control IV(3), 273–282 (2009)
7. Pérez-Jiménez, M.J.: An approach to computational complexity in membrane computing. In: Mauri, G., Păun, Gh., Pérez-Jiménez, M.J., Rozenberg, G., Salomaa, A.

(eds.) Workshop on Membrane Computing. Lecture Notes in Computer Science, vol. 3365, pp. 85–109. Springer (2004)
8. Pérez-Jiménez, M.J., Riscos-Núñez, A., Romero-Jiménez, A., Woods, D.: Complexity - membrane division, membrane creation. In: Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) The Oxford Handbook of Membrane Computing, pp. 302 – 336. Oxford University Press, Oxford, England (2010)
9. Păun, Gh.: Some quick research topics., In these proceedings.
10. Song, T., Jiang, Y., Shi, X., Zeng, X.: Small universal spiking neural P systems with anti-spikes. Journal of Computational and Theoretical Nanoscience 10(4), 999–1006 (2013)
11. Tan, G., Song, T., Chen, Z., Zeng, X.: Spiking neural P systems with anti-spikes and without annihilating priority working in a 'flip-flop' way. International Journal of Computing Science and Mathematics 4(2), 152–162 (2013)

How to Go Beyond Turing with P Automata: Time Travels, Regular Observer ω-Languages, and Partial Adult Halting

Rudolf Freund[1], Sergiu Ivanov[2], and Ludwig Staiger[3]

[1] Technische Universität Wien, Austria
Email: rudi@emcc.at

[2] Université Paris Est, France
Email: sergiu.ivanov@u-pec.fr

[3] Martin-Luther-Universität Halle-Wittenberg, Germany
Email: staiger@informatik.uni-halle.de

Summary. In this paper we investigate several variants of P automata having infinite runs on finite inputs. By imposing specific conditions on the infinite evolution of the systems, it is easy to find ways for going beyond Turing if we are watching the behavior of the systems on infinite runs. As specific variants we introduce a new halting variant for P automata which we call *partial adult halting* with the meaning that a specific predefined part of the P automaton does not change any more from some moment on during the infinite run. In a more general way, we can assign ω-languages as observer languages to the infinite runs of a P automaton. Specific variants of regular ω-languages then, for example, characterize the red-green P automata.

1 Introduction

Various possibilities how one can "go beyond Turing" are discussed in [11], for example, the definitions and results for red-green Turing machines can be found there. In [2] the notion of red-green automata for register machines with input strings given on an input tape (often also called *counter automata*) was introduced and the concept of *red-green P automata* for several specific models of membrane systems was explained. Via red-green counter automata, the results for acceptance and recognizability of finite strings by red-green Turing machines were carried over to red-green P automata. The basic idea of red-green automata is to distinguish between two different sets of states (red and green states) and to consider infinite runs of the automaton on finite input objects (strings, multisets); allowed to change between red and green states more than once, red-green automata can recognize more than the recursively enumerable sets (of strings, multisets), i.e., in that way we can "go beyond Turing". In the area of P systems, first attempts to do that can

be found in [4] and [18]. Computations with infinite words by P automata were investigated in [9].

In this paper, we also consider infinite runs of P automata, but in a more general way take into account the existence/non-existence of a recursive feature of the current sequence of configurations. In that way, we obtain infinite sequences over $\{0,1\}$ which we call "observer languages" where 1 indicates that the specific feature is fulfilled by the current configuration and 0 indicates that this specific feature is not fulfilled. The recognizing runs of red-green automata then correspond with ω-regular languages over $\{0,1\}$ of a specific form ending with 1^ω as observer languages. A very special observer language is $\{0,1\}^* \{1\}^\omega$ which corresponds with a very special acceptance condition for P automata which we call "partial adult halting". This special acceptance variant for P automata with infinite runs on finite multisets is motivated by an observation we make for the evolution of time lines described by P systems – at some moment, a specific part of the evolving time lines, for example, the part describing time 0, shall not change any more.

2 Definitions

We assume the reader to be familiar with the underlying notions and concepts from formal language theory, e.g., see [17], as well as from the area of P systems, e.g., see [13, 14, 15]; we also refer the reader to [25] for actual news.

2.1 Prerequisites

The set of integers is denoted by $\mathbb{Z}$, and the set of non-negative integers by $\mathbb{N}$. Given an alphabet V, a finite non-empty set of abstract symbols, the free monoid generated by V under the operation of concatenation is denoted by V^*. The elements of V^* are called strings, the empty string is denoted by λ, and $V^* \setminus \{\lambda\}$ 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 $|x| = \sum_{a_i \in V} |x|_{a_i}$. A (finite) multiset over a (finite) alphabet $V = \{a_1, \ldots, a_n\}$ is a mapping $f : V \to \mathbb{N}$ and can be represented by $\left\langle a_1^{f(a_1)}, \ldots, a_n^{f(a_n)} \right\rangle$ or by any string x for which $(|x|_{a_1}, \ldots, |x|_{a_n}) = (f(a_1), \ldots, f(a_n))$. The families of regular and recursively enumerable string languages are denoted by REG and RE, respectively.

2.2 Register Machines

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

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

A *configuration* of a register machine is described by the contents (i.e., by the number stored in the register) of each register and by the current label, which indicates the next instruction to be executed. Computations start by executing the instruction l_0 of P, and terminate with reaching the HALT-instruction l_h.

In order to deal with strings, this basic model of register machines can be extended by instructions for reading from an input tape and writing to an output tape containing strings over an input alphabet T_{in} and an output alphabet T_{out}, respectively:

- $l_1 : (read(a), l_2)$, with $l_1 \in B \setminus \{l_h\}$, $l_2 \in B$, $a \in T_{in}$.
  Reads the symbol a from the input tape and jumps to instruction l_2.
- $l_1 : (write(a), l_2)$, with $l_1 \in B \setminus \{l_h\}$, $l_2 \in B$, $a \in T_{out}$.
  Writes the symbol a on the output tape and jumps to instruction l_2.

Such a register machine working on strings often is also called a *counter automaton*, and we write $M = (m, B, l_0, l_h, P, T_{in}, T_{out})$. If no output is written, we omit T_{out}.

As is well known (e.g., see [12]), for any recursively enumerable set of natural numbers there exists a register machine with (at most) three registers accepting the numbers in this set. Counter automata, i.e., register machines with an input tape, with two registers can simulate the computations of Turing machines and thus characterize RE. All these results are obtained with deterministic register machines, where the ADD-instructions are of the form $l_1 : (ADD(r), l_2)$, with $l_1 \in B \setminus \{l_h\}$, $l_2 \in B$, $1 \leq j \leq m$.

2.3 The Arithmetical Hierarchy

The Arithmetical Hierarchy (e.g., see [3]) is usually developed with the universal ($\forall$) and existential ($\exists$) quantifiers restricted to the integers. Levels in the Arithmetical Hierarchy are labeled as Σ_n if they can be defined by expressions beginning with a sequence of n alternating quantifiers starting with $\exists$; levels are labeled as Π_n if they can be defined by such expressions of n alternating quantifiers that start with $\forall$. Σ_0 and Π_0 are defined as having no quantifiers and are equivalent. Σ_1 and Π_1 only have the single quantifier $\exists$ and $\forall$, respectively. We only need to consider alternating pairs of the quantifiers $\forall$ and $\exists$ because two quantifiers of the same type occurring together are equivalent to a single quantifier.

3 Time Travel P Systems

In the most general case, we can think of P systems as devices manipulating multisets in a hierarchical membrane structure. The membranes can have labels and polarizations both eventually changing with the application of rules. Membranes may be divided, generated or deleted. Together with the division or the generation of a new membrane the whole contents of another membrane may be copied. For a general framework of P systems we refer to [7].

Usually, configurations in P systems (and other systems like Turing machines) evolve step by step through time, see Figure 1.

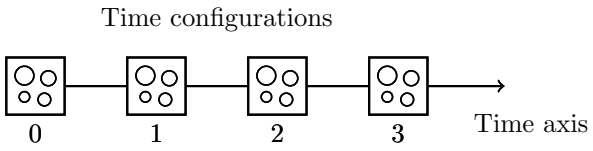


Fig. 1. Standard time line evolution.

Without time travel option, we need only consider the evolution of the system on one time axis from time n to time $n+1$. The situation becomes more difficult if we follow the idea of parallel worlds (*time axes*), which means that we have another time dimension, described by the vertical evolution in Figure 2, i.e., the time configurations at time n may be altered depending on future evolutions.

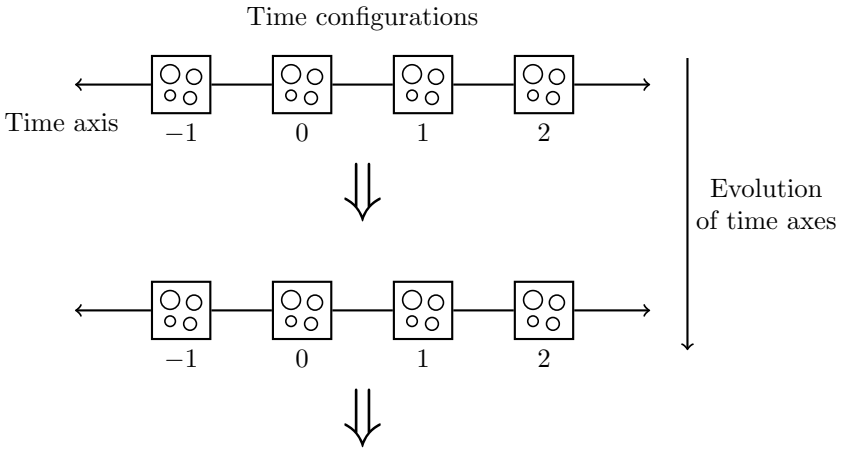


Fig. 2. Time lines evolution.

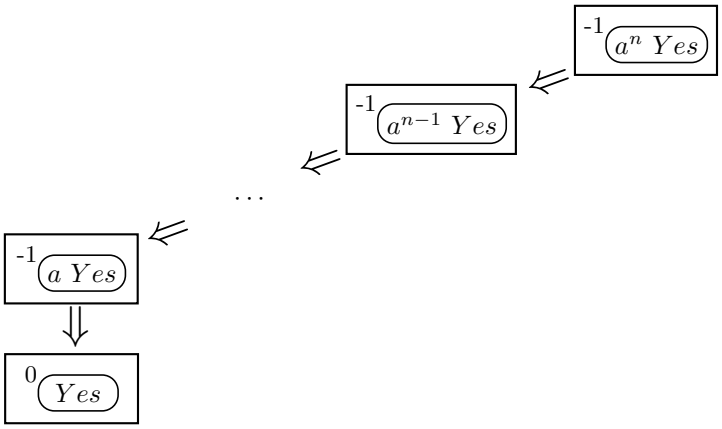


Fig. 3. Sending back an answer from time n to time 0.

For example, we can consider membrane systems with polarizations assigned to the membranes. The usual polarization of the whole time configuration in the normal case is $+1$, indicating that the evolution of the membrane(s) goes from time configuration n to time configuration $n+1$. Now assume we allow polarization -1 indicating that the corresponding membrane evolves from time configuration n to time configuration $n-1$. Having kept trace of the number of computation steps, e.g., by the multiplicity of a specific object a, we are able to send back information – like the answer *yes* to a question we have posed at time 0 which then is sent back to time configuration 0, i.e., to the time we have posed the question. In that way, on a specific time line we can have answers to questions in zero time, see Figure 3.

During its travel through the time back, the time capsule with polarization -1 can be assumed not to be affected by the other membranes in the intermediate time configurations. Obviously, this restriction can be alleviated for even more complex systems.

Putting a new skin membrane around all the current time configurations of one time axis, we again obtain a conventional evolution model, yet now with a vertical time evolution as depicted in Figure 4. The only assumption we have to do for making this variant possible is that at the beginning only a finite number of time configurations exists (in fact, we usually will start with the time configuration at time 0).

3.1 Partial Adult Halting

Going back to the time travel model of Figure 2 the question that arises is what kind of results we may obtain and how. For example, given a specific input in time configuration 0, we may request that from some moment on this time configuration becomes stable, i.e., it is not changed any more (by time capsules arriving there).

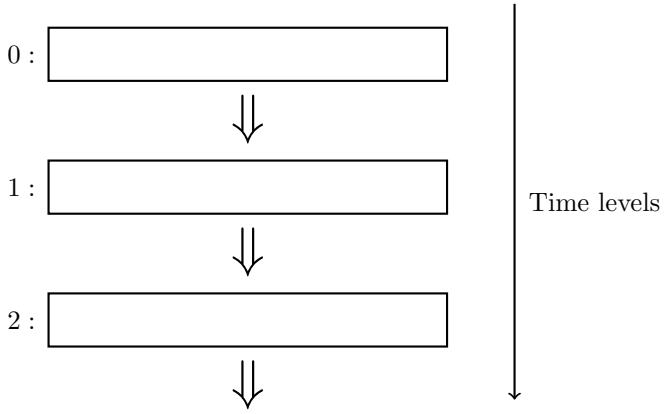


Fig. 4. Conventional Evolution Model.

So the specific feature an external observer would see is that the time configuration at time 0 is not changing any more starting from some specific time line at level tl_0 on, i.e., for all time levels $tl \geq tl_0$ the *time configuration at time* 0 *stays stable*.

With respect to the situation described in Figure 4 this means that one specific part (one membrane and all its contents) does not change any more.

In that way we obtain a new variant of a halting condition in P systems which we call *partial adult halting*:

adult halting:

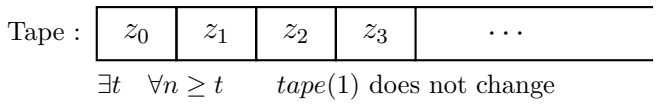
means that the configuration does not change any more

partial:

we only look at some part of the configuration

3.2 Partial Adult Halting for Turing Machines

The idea of partial adult halting can also be applied to Turing machines:



On tape cell 1 we want to obtain an "answer" whether the given input word is accepted – 1 – or not – 0. We first put 0 there, and if the computation ends saying "accept" we go back to tape cell 1 and write 1 there. Hence, with looking to infinity in that way we obtain a "decider" for recursively enumerable languages.

Generation of Complements of Recursively Enumerable Languages

Another example based on a similar idea as described above shows how to generate the complement of an arbitrary recursively enumerable language L.

In this case, we use the model of a generating Turing machine with output tape, and a string is said to be generated by the Turing machine M if from some moment of the computation the output tape is not changed anymore.

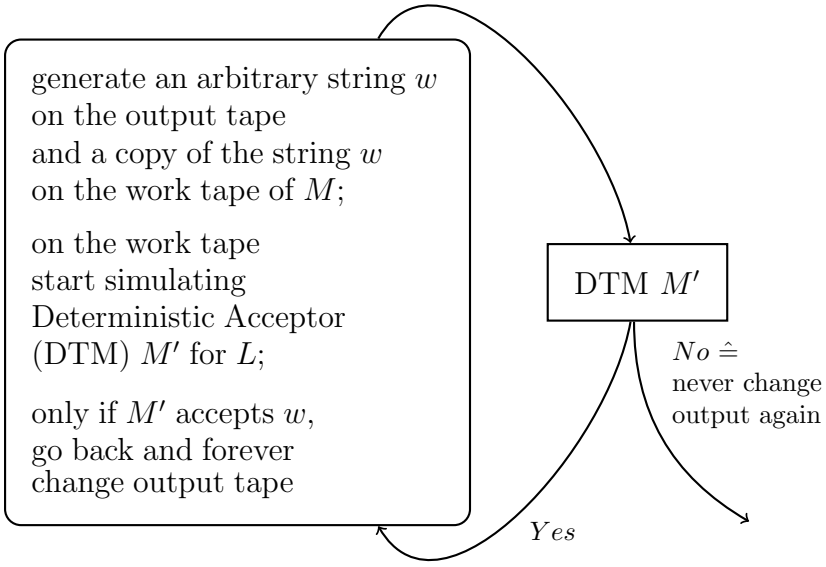


Fig. 5. Generation of the complements of a recursively enumerable language L.

4 Variants of P Automata

In this section, we shortly describe some variants of P automata.

4.1 The Basic Model of P Automata with Antiport Rules

The basic model of P automata as introduced in [6] and in a similar way in [8] is based on antiport rules, i.e., on rules of the form u/v, which means that the multiset u goes out through the membrane and v comes in instead.

A *P automaton (with antiport rules)* is a construct

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

- O is the alphabet of *objects*;
- $T \subset O$ is the alphabet of *terminal objects*;
- μ is the hierarchical membrane structure, with the membranes uniquely labeled by the numbers from 1 to m;
- $w_i \in (O \setminus T)^*$, $1 \leq i \leq m$, is the *initial multiset* in membrane i;
- R_i, $1 \leq i \leq m$, is a finite set of *antiport rules* assigned to membrane i.

Given a multiset of terminal symbols in the skin membrane 1, it is usually accepted by Π via a halting computation.

Now consider the situation of partial adult halting for a P automaton

$$\Pi = (O, T, [_1[_2\ ]_2]_1, q_0, n, R_1, R_2)$$

which – with the input multiset in addition given in the skin membrane – simulates, in a deterministic way, a register machine defining a recursively enumerable set L of multisets (see [12]), by the rules in R_1. If the computation stops in the final state q_h, i.e., the multiset is accepted, we add the rules q_h/y and n/n in R_1. R_2 only contains the rule n/y. In case the multiset is accepted, n in the second membrane is replaced by y, while the rule n/n in R_1 guarantees an infinite computation. In case the input multiset is not accepted, the register machine already guarantees an infinite computation by the simulating P automaton, too. Hence, as in the case of the Turing machine with partial adult halting we get a "decider" for L, with the result from some moment on to be found in membrane 2.

4.2 P Automata with Anti-Matter

In *P automata with anti-matter*, for each object a we may have its anti-matter object a^-. If an object a meets its anti-matter object a^-, then these two objects annihilate each other, which corresponds to the application of the cooperative erasing rule $aa^- \to \lambda$. In the following, we shall only consider the variant where these annihilation rules have weak priority over all other rules, which allows for a deterministic simulation of deterministic register machines, see [1].

A *P automaton with anti-matter* is a construct

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

- O is the alphabet of *objects*;

- $T \subset O$ is the alphabet of *terminal objects*;
- μ is the hierarchical membrane structure, with the membranes uniquely labeled by the numbers from 1 to m;
- $w_i \in (O \setminus T)^*$, $1 \leq i \leq m$, is the *initial multiset* in membrane i;
- R_i, $1 \leq i \leq m$, is a finite set of

  non-cooperative rules: are rules of the form $u \to v$ where $u \in O$ and $v \in (O \times \{here, in, out\})^*$;

  matter/anti-matter annihilation rules: are cooperative rules of the form $aa^- \to \lambda$, i.e., the matter object a and its anti-matter object a^- annihilate each other, and these annihilation rules have weak priority over all other rules.

With the target indications $\{here, in, out\}$ we can leave an object in the current membrane ($here$), whereas with $\{in\}$ we send it into an inner membrane and with $\{out\}$ we send it into the surrounding membrane region.

In a similar way as in the preceding subsection we may consider the situation of partial adult halting for a P automaton

$$\Pi = (O, T, [_1[_2\ ]_2]_1, q_0, n, R_1, R_2)$$

where following the proof from [1] the register machine actions are simulated in the skin membrane; if the input multiset is accepted, by using the rules $q_h \to (f, here)(n^-, in), f \to f$, we obtain an infinite computation with the contents of membrane 2 being empty indicating the acceptance, as by the annihilation rule $nn^- \to \lambda$ the original object n is annihilated.

5 Red-Green Automata

In general, a red-green automaton M is an automaton whose set of internal states Q is partitioned into two subsets, Q_{red} and Q_{green}, and M operates without halting. Q_{red} is called the set of "red states", Q_{green} the set of "green states". Moreover, we shall assume M to be deterministic, i.e., for each configuration there exists exactly one transition to the next one.

5.1 Red-Green Turing Machines

Red-green Turing machines, see [11], can be seen as a type of ω-Turing machines on finite inputs with a recognition criterion based on some property of the set(s) of states visited (in)finitely often, in the tradition of ω-automata (see [9]), i.e., we call an infinite run of the Turing machine M on input w *recognizing* if and only if

- no red state is visited infinitely often and
- some green states (one or more) are visited infinitely often.

A set of strings $L \subset \Sigma^*$ is said to be *accepted* by M if and only if the following two conditions are satisfied:

(a) $L = \{w \mid w \text{ is recognized by } M\}$.
(b) For every string $w \notin L$, the computation of M on input w eventually stabilizes in red; in this case w is said to be *rejected*.

The phrase "mind change" is used in the sense of changing the color, i.e., changing from red to green or vice versa.

The following results were established in [11]:

Theorem 1. *A set of strings L is recognized by a red-green Turing machine with one mind change if and only if $L \in \Sigma_1$, i.e., if L is recursively enumerable.*

Theorem 2. *(Computational power of red-green Turing machines)*

(a) Red-green Turing machines recognize exactly the Σ_2-sets of the Arithmetical Hierarchy.
(b) Red-green Turing machines accept exactly those sets which simultaneously are Σ_2- and Π_2-sets of the Arithmetical Hierarchy.

5.2 Red–Green Register Machines

In [2], similar results as for red-green Turing machines were shown for red-green counter automata and register machines, respectively.

As it is well-known folklore, e.g., see [12], the computations of a Turing machine can be simulated by a counter automaton with (only two) counters; in this paper, we will rather speak of a register machine with (two) registers and with string input. As for red-green Turing machines, we can also color the "states", i.e., the labels, of a register machine $M = (m, B, l_0, l_h, P, T_{in})$ by the two colors red and green, i.e., partition its set of labels B into two disjoint sets B_{red} (red "states") and B_{green} (green "states"), and we then write $RM = (m, B, B_{red}, B_{green}, l_0, P, T_{in})$, as we can omit the halting label l_h.

The following two lemmas were proved in [2]; the step from red-green Turing machines to red-green register machines is important for the succeeding sections, as usually register machines are simulated when proving a model of P systems to be computationally complete. Therefore, in the following we always have in mind this specific relation between red-green Turing machines and red-green register machines when investigating the infinite behavior of specific models of P automata, as we will only have to argue how red-green register machines can be simulated.

Lemma 1. *The computations of a red-green Turing machine TM can be simulated by a red-green register machine RM with two registers and with string input in such a way that during the simulation of a transition of TM leading from a state p with color c to a state p' with color c' the simulating register machine uses instructions with labels ("states") of color c and only in the last step of the simulation changes to a label ("state") of color c'.*

Lemma 2. *The computations of a red-green register machine RM with an arbitrary number of registers and with string input can be simulated by a red-green Turing machine TM in such a way that during the simulation of a computation step of RM leading from an instruction with label ("state") p with color c to an instruction with label ("state") p' with color c' the simulating Turing machine stays in states of color c and only in the last step of the simulation changes to a state of color c'.*

As an immediate consequence, the preceding two lemmas yield the characterization of Σ_2 and Π_2 by red-green register machines as Theorem 2 does for red-green Turing machines, see [2]:

Theorem 3. *(Computational power of red-green register machines)*

(i) A set of strings L is recognized by a red-green register machine with one mind change if and only if $L \in \Sigma_1$, i.e., if L is recursively enumerable.
(ii) Red-green register machines recognize exactly the Σ_2-sets of the Arithmetical Hierarchy.
(iii) Red-green register machines accept exactly those sets which simultaneously are Σ_2- and Π_2-sets of the Arithmetical Hierarchy.

5.3 Red-Green P Automata

As it was shown in [2], P automata with antiport rules and with anti-matter can simulate the infinite computations of any red-green register machine, even with a clearly specified finite set of "states" having the same color as the corresponding labels ("states") of the instructions of the red-green register machine.

Hence, as a consequence, similar results as for red-green Turing machines also hold for red-green P automata with antiport rules and with anti-matter. From the results shown in [2] we therefore infer:

Theorem 4. *(Computational power of red-green P automata)*

(i) A set of multisets L is recognized by a red-green P automaton (with antiport rules, with anti-matter) with one mind change if and only if L is recursively enumerable.
(ii) Red-green P automata (with antiport rules, with anti-matter) recognize exactly the Σ_2-sets.
(iii) Red-green P automata (with antiport rules, with anti-matter) accept exactly those sets which simultaneously are Σ_2- and Π_2-sets of the Arithmetical Hierarchy.

6 Observer Languages

An observer language for infinite computations is an ω-language over $\{0,1\}$ where 1 indicates that a specific feature of the current configuration in the infinite computation sequence is fulfilled and 0 indicates that this specific feature of the current configuration is not fulfilled.

6.1 Expressing Partial Adult Halting as Observer Language

If we define the specific feature to be that no rule is applicable in the specified "observed" membrane, then acceptance by partial adult halting can be described by the (regular) ω-language $\{0,1\}^*\{1\}^\omega$.

6.2 Expressing Recognition by Red-Green P Automata Using Observer Languages

As observer languages for infinite computations in red-green P automata we again use ω-languages over $\{0,1\}$ where now 1 indicates that we will have to apply a green multiset of rules to the current configuration in the infinite computation sequence and 0 indicates that we will have to apply a red multiset of rules to the current configuration.

So for recognizing a language from RE we use the the ω-language $\{0\}^+\{1\}^\omega$, for a language from *co-RE* we use the the ω-language $\{0\}\{1\}^\omega$.

The corresponding regular ω-languages for the recognition by red-green automata (Turing machines, P automata) with multiple mind-changes are described as follows:

exactly $2k+1$ mind-changes, $k \geq 0$: $\{0\}^+\big(\{1\}^+\{0\}^+\big)^k\{1\}^\omega$

at most $2k+1$ mind-changes, $k \geq 0$: $\bigcup_{i=0}^{k}\{0\}^+\big(\{1\}^+\{0\}^+\big)^i\{1\}^\omega$

The upper bound for languages recognized by red-green P automata (with antiport rules, with anti-matter) with k mind-changes for some $k \geq 0$ is Σ_2, see [2].

These results will be refined in the next section.

7 Recognition Using Regular Observer Languages

In this section we investigate which languages are recognized by red-green P automata using observer languages defined by finite automata. This class of ω-languages defined by finite automata is well-understood and has widely been investigated (see [16, 21, 23, 24]). We follow the line of [20] where for Turing machines infinite computations accepting finite words were investigated in detail (see also [5]). In this paper a word w was accepted by a Turing machine when the sequence

$(s_i)_{i\in\mathbb{N}}$ of states the machine runs through during its accepting process fulfills certain simple conditions known from the acceptance of ω-languages. This can be seen as w to be accepted if the observed state sequence $(s_i)_{i\in\mathbb{N}}$ belongs to a certain (regular) observer language. We have to point out that usually the notion *acceptance* is used here instead of the notion *recognition* as used by van Leeuwen and Wiedermann for the red-green Turing machines.

7.1 Observer Languages of the form $W \cdot \{1\}^\omega$ with $W \in REG$

The observer languages in Section 6 all were of the form $W\cdot\{1\}^\omega$ where $W \subseteq \{0,1\}^*$ is a regular language. In this section we investigate which languages can be accepted by red-green P automata using observer languages of this form. Here we follow the line of the papers [20] and [11] where the influence of regular observer languages on the acceptance and recognition, respectively, behavior of Turing machines was investigated.

To this end we use the following theorem which follows from a general classification of regular ω-languages (see [19, 22] and also the survey [21]).

Theorem 5. *If $F \subseteq \{0,1\}^\omega$ is a regular ω-language, then*

1. *F is in the Boolean closure of Σ_2, and*
2. *if $F \in \Sigma_2 \cap \Pi_2$, then F is in the Boolean closure of Σ_1.*

Since every regular $F \subseteq \{0,1\}^*\cdot\{1\}^\omega$ as a countable set is in Σ_2, we immediately obtain the following.

Corollary 1. *If $W \subseteq \{0,1\}^*$ is a regular language then $W \cdot \{1\}^\omega$ satisfies one of the following conditions:*

1. *$W \cdot \{1\}^\omega \in \Sigma_2 \setminus \Pi_2$, or*
2. *$W \cdot \{1\}^\omega$ is a Boolean combination of ω-languages in Σ_1.*

Remark 1. In the second case we can obtain an even sharper result:

$$W \cdot \{1\}^\omega = \bigcup\nolimits_{i=0}^{k} (W_i \cdot \{0,1\}^\omega \setminus V_i \cdot \{0,1\}^\omega)$$

for suitable $k \in \mathbb{N}$ and *regular* languages $W_i, V_i \subseteq \{0,1\}^*$, $0 \le i \le k$. In particular, this is true for the ω-languages representing a bounded number of mind-changes from Subsection 6.2:

$$\bigcup\nolimits_{i=0}^{k} \{0\}^+ \big(\{1\}^+\{0\}^+\big)^i \{1\}^\omega =$$

$$\bigcup\nolimits_{i=0}^{k} \Big(\{0\}^+\big(\{1\}^+\{0\}^+\big)^i\{1\} \cdot \{0,1\}^\omega \setminus \{0\}^+\big(\{1\}^+\{0\}^+\big)^i\{1\}^+\{0\} \cdot \{0,1\}^\omega\Big)$$

From Corollary 1 we immediately infer:

Theorem 6. *Let L be recognized by a red-green P automaton (with antiport rules, with anti-matter) using an observer language $W \cdot \{1\}^\omega$ where $W \subseteq \{0,1\}^*$ is regular.*

1. *Then $L \in \Sigma_2$.*
2. *If $W \cdot \{1\}^\omega = \bigcup_{i=0}^{k}(F_i \setminus E_i)$ is a Boolean combination of ω-languages $F_i, E_i \in \Sigma_1$, $0 \leq i \leq k$, then $L = \bigcup_{i=0}^{k}(K_i \setminus L_i)$ where $K_i, L_i \in RE$, $0 \leq i \leq k$.*

The converse of Theorem 6 is also true. In particular, it shows that we can restrict ourselves to the observer languages of Subsections 6.1 and 6.2.

Theorem 7. *Let $L \in \Sigma_2$.*

1. *Then L is recognized by a red-green P automaton Π using the observer language $\{0,1\}^* \cdot \{1\}^\omega$, i.e., L is accepted by Π by partial adult halting.*
2. *Let $L = \bigcup_{i=0}^{k}(K_i \setminus L_i)$ where $K_i, L_i \in RE$, $0 \leq i \leq k$. Then there exists a red-green P automaton which recognizes L using an observer language with a bounded number of mind-changes.*

7.2 Regular Observer Languages

Admitting all regular ω-languages as observer languages extends the range of recognizable languages. In view of Theorem 5 we obtain a result extending what was shown in Theorem 6.

Theorem 8. *Let L be recognized by a red-green P automaton using an observer language $F \subseteq \{0,1\}^\omega$. Then*

1. *if F is a Boolean combination of ω-languages $F_i, E_i \in \Sigma_2$, $0 \leq i \leq k$, then $L = \bigcup_{i=0}^{k}(K_i \setminus L_i)$ where $K_i, L_i \in \Sigma_2$, $0 \leq i \leq k$,*
2. *if $F \in \Sigma_2$, then $L \in \Sigma_2$,*
3. *if $F \in \Pi_2$, then $L \in \Pi_2$, and*
4. *if F is regular and $F \in \Sigma_2 \cap \Pi_2$, then $L = \bigcup_{i=0}^{k}(K_i \setminus L_i)$ where $K_i, L_i \in RE$, $0 \leq i \leq k$.*

The converse of Theorem 8 is also true:

Theorem 9. *Let L be a Boolean combination of languages in Σ_2. Then L is recognized by a red-green P automaton using a regular observer language $F \subseteq \{0,1\}^\omega$.*

8 Conclusion

In this paper we have investigated the computational power of P automata working with infinite runs on finite input multisets. With regular observer languages $W \cdot \{1\}^\omega$, $W \in REG$, we obtain the Σ_2-sets, the same as with red-green P automata. Moreover, the Σ_2-sets are already obtained by the special observer language $\{0,1\}^* \cdot \{1\}^\omega$, which corresponds to the special acceptance condition of *partial adult halting*.

References

1. A. Alhazov, B. Aman, R. Freund: P Systems with Anti-Matter. In: [10], 66–85.
2. B. Aman, E. Csuhaj-Varjú, R. Freund: Red-Green P Automata. In: [10], 139–157.
3. P. Budnik: *What Is and What Will Be.* Mountain Math Software, 2006.
4. C.S. Calude, Gh. Păun: Bio-steps Beyond Turing. *Biosystems* **77** (2004), 175–194.
5. C.S. Calude, L. Staiger: A note on accelerated Turing machines. *Math. Structures Comput. Sci.* **20** (6) (2010), 1011–1017.
6. E. Csuhaj-Varjú, Gy. Vaszil: P Automata or Purely Communicating Accepting P Systems. In: Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron (Eds.): *Membrane Computing, International Workshop, WMC-CdeA 2002, Curtea de Argeş, Romania, August 19–23, 2002, Revised Papers.* Lecture Notes in Computer Science **2597**, Springer, 2003, 219–233.
7. R. Freund, I. Pérez-Hurtado, A. Riscos-Núñez , S. Verlan: A formalization of membrane systems with dynamically evolving structures. *International Journal of Computer Mathematics* **90** (4) (2013), 801-815.
8. R. Freund, M. Oswald: A Short Note on Analysing P Systems. *Bulletin of the EATCS* **78**, 2002, 231–236.
9. R. Freund, M. Oswald, L. Staiger: ω-P Automata with Communication Rules. *Workshop on Membrane Computing, 2003*, Lecture Notes in Computer Science **2933**, Springer, 2004, 203–217.
10. M. Gheorghe, G. Rozenberg, A. Salomaa, P. Sosík, C. Zandron: 15th International Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers. Lecture Notes in Computer Science **8961**, Springer, 2014.
11. J. van Leeuwen, J. Wiedermann: Computation as an Unbounded Process. *Theoretical Computer Science* **429** (2012), 202–212.
12. M. L. Minsky: *Computation: Finite and Infinite Machines.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 1967.
13. Gh. Păun: Computing with Membranes. *Journal of Computer and System Sciences* **61** (1) (2000), 108–143 (and Turku Center for Computer Science-TUCS Report 208, November 1998, `www.tucs.fi`).
14. Gh. Păun: *Membrane Computing. An Introduction.* Springer, 2002.
15. Gh. Păun, G. Rozenberg, A. Salomaa (Eds.): *The Oxford Handbook of Membrane Computing.* Oxford University Press, 2010.
16. D. Perrin and J.-É. Pin. *Infinite Words*, vol. 141 of *Pure and Applied Mathematics.* Elsevier, Amsterdam, 2004.
17. G. Rozenberg, A. Salomaa (Eds.): *Handbook of Formal Languages,* 3 volumes. Springer, 1997.

18. P. Sosík, O. Valík: On Evolutionary Lineages of Membrane Systems. In: R. Freund, Gh. Păun, G. Rozenberg, A. Salomaa (Eds.): *6th International Workshop, WMC 2005, Vienna, Austria, July 18-21, 2005, Revised Selected and Invited Papers*. Lecture Notes in Computer Science **3850**, Springer, 2006, 67–78.
19. L. Staiger: Finite-state ω-languages. *J. Comput. System Sci.* **27** (3) (1983), 434–448.
20. L. Staiger: ω-computations on Turing machines and the accepted languages. In: L. Lovász, E. Szemerédi (Eds.): *Theory of Algorithms*, Coll. Math. Soc. Janos Bolyai No.44, North Holland, Amsterdam, 1986, 393–403.
21. L. Staiger: ω-languages. In: [17], vol. 3, 339–387.
22. L. Staiger, K. Wagner: Automatentheoretische und automatenfreie Charakterisierungen topologischer Klassen regulärer Folgenmengen. *Elektron. Informationsverarb. Kybernetik* **10** (7) (1974), 379–392.
23. W. Thomas: Automata on infinite objects. In: J. van Leeuwen (Ed.): *Handbook of Theoretical Computer Science*, vol. B, pages 133–192. North Holland, Amsterdam, 1990.
24. K. Wagner: On ω-regular sets. *Inform. and Control*, 43 (2) (1979), 123–177.
25. The P Systems Website: `http://ppage.psystems.eu`.

A Characterization of PSPACE with Antimatter and Membrane Creation

Zsolt Gazdag[1], Miguel A. Gutiérrez–Naranjo[2]

[1]Department of Algorithms and their Applications
Faculty of Informatics
Eötvös Loránd University, Hungary
gazdagzs@inf.elte.hu

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

Summary. The use of negative information provides a new tool for exploring the limits of P systems as computational devices. In this paper we prove that the combination of antimatter and annihilation rules (based on the annihilation of physical particles and antiparticles) and membrane creation (based on autopoiesis) provides a P system model able to solve **PSPACE**-complete problems. Namely, we provide a uniform family of P system in such P system model which solves the satisfiability problem for quantified Boolean formulas (**QSAT**). In the second part of the paper, we prove that all the decision problems which can be solved with this P system model belong to the complexity class **PSPACE**, so this P system model characterises **PSPACE**.

1 Introduction

The use of *negative information* provides a new challenge in the development of theoretical aspects in Membrane Computing (see [20]). Such *negative information* can be considered by extending the definition of a multiset f on a set X from $f : X \to \mathbb{N}$ to $f : X \to \mathbb{Z}$ (i.e., admitting negative multiplicity of the elements of the multiset [4, 13]) or even considering negative time and the possibility of travelling in time [7].

One of the most extended uses of negative information in Membrane Computing is considering *anti-spikes* in the framework of Spiking Neural P systems. In such model when one *spike* and one *anti-spike* appear in the same neuron, the annihilation occurs and both, spike and anti-spike, disappear [15, 17, 22, 24]. The use of antimatter, as an extension of the concept of anti-spikes, is being explored in other P system models [1, 2, 5].

Recently, it has been proved that antimatter and annihilation rules are a frontier of tractability in Membrane Computing [5]. The starting point for the study was a well-known result in the complexity theory of Membrane Computing: P systems with active membranes without polarizations, without dissolution and with division of elementary and non-elementary membranes (denoted by $\mathcal{AM}^0_{-d,+ne}$) can solve exactly problems in the complexity class **P** (see [8], Th. 2). The main result in [5] is that $\mathcal{AM}^0_{-d,+ne}$ endowed with antimatter and annihilation rules (denoted by $\mathcal{AM}^0_{-d,+ne,+ant}$) can solve **NP**-complete problems.

In a certain sense, such results show that if the number of membranes of the P system can be increased by membrane division, then endowing the model with dissolution or annihilation rules, then the model is capable to solve **NP**-complete problems.

Similar results hold in the case of P systems with membrane creation. In [9] it is shown that these P systems when dissolution rules are allowed can solve **PSPACE**-complete problems (i.e, they can solve all the decision problems which can be solved by Turing machines, deterministic or non-deterministic, in polynomial space). In this paper, we show that using annihilation rules instead of dissolution rules, P systems with membrane creation are not only able to solve **NP**-complete problems, but **PSPACE**-complete problems too. By taking [23] as starting point, in the second part of the paper, we prove that all the decision problems which can be solved with this P system model belong to the complexity class **PSPACE**, so this P system model characterises **PSPACE**.

The paper is organized as follows. In the next section, the notion of P systems with membrane creation and annihilation rules is introduced. Then recognizer P systems are briefly described. In Section 4 we show that the well known QSAT problem (i.e., the problem of deciding if a fully quantified Boolean formula is true or not) can be solved in linear time by P systems with membrane creation, with annihilation rules and without dissolution rules. In Section 5, we prove that **PSPACE** is an upper bound for the set of decision problems which can be solved with this model. Finally, some conclusions are given in the last section.

2 The P System Model

The basis of the model is two types of rules which are not so common on complexity studies in Membrane Computing. The first type, rules of membrane creation, is based on the biological process of autopoiesis [14]. It creates a membrane from a single object in a similar way to the creation of a vesicle in a cell by a metabolite. This type of rule was first considered in [12, 16] and it has been proved that P systems with membrane creation and *dissolution rules* can solve **NP**-complete problems (see [10, 11]) or even **PSPACE**-complete problems (see [9]).

The idea of using antimatter as a generalization of the anti-spikes used in Spiking Neural P Systems was firstly proposed in [21]. Based on the physical inspiration of particles and antiparticles, if an object a and its opposite one $\overline{a}$

appears simultaneously in the same membrane, they are annihilated by application of the corresponding rule $a\overline{a} \to \lambda$. As pointed above, several authors have started to explore the possibilities of using antimatter in Membrane Computing [1, 2, 5].

Formally, a *P system with membrane creation and annihilation rules* is a construct of the form $\Pi = (O, H, \mu, w_1, \ldots, w_m, R)$, where:

1. $m \geq 1$ is the initial degree of the system; O is the alphabet of *objects* and H is a finite set of *labels* for membranes;
2. μ is a *membrane structure* consisting of m membranes labelled (not necessarily in a one-to-one manner) with elements of H and $w_1, \ldots, w_m$ are strings over O, describing the *multisets of objects* placed in the m regions of μ;
3. R is a finite set of *rules*, of the following forms:
   (a) $[a \to v]_h$ where $h \in H$, $a \in O$, and v is a string over O describing a multiset of objects. These are *object evolution rules* associated with membranes and depending only on the label of the membrane.
   (b) $a[\,]_h \to [b]_h$ where $h \in H$, $a, b \in O$. These are *send-in communication rules*. An object is introduced in the membrane possibly modified.
   (c) $[a]_h \to [\,]_h\, b$ where $h \in H$, $a, b \in O$. These are *send-out communication rules*. An object is sent out of the membrane possibly modified.
   (d) $[a \to [v]_{h_2}]_{h_1}$ where $h_1, h_2 \in H$, $a \in O$, and v is a string over O describing a multiset of objects. These are *creation rules*. In reaction with an object, a new membrane is created. This new membrane is placed inside of the membrane of the object which triggers the rule and has associated an initial multiset and a label.
   (e) $[\,a\overline{a} \to \lambda\,]_h$ for $h \in H$, $a, \overline{a} \in O$. This is an annihilation rule, associated with a membrane labelled by h: the pair of objects $a, \overline{a} \in O$ belonging simultaneously to this membrane disappears.

Rules are applied according to the following principles:

- Rules of type (a) - (d) are applied in parallel and in a maximal manner. In one step, one object of a membrane can be used by only one rule (chosen in a non–deterministic way), but any object which can evolve by one rule of any form, must evolve.
- If an object can trigger two or more rules, one of such rules is non-deterministically chosen, except for annihilation rules (type (e)). Any annihilation rule has priority over all rules of the other types of rules. This fact has a clear physical inspiration. If a particle and its antiparticle meet, they *do* disappear and no other option is possible. This semantics was also used in [5].
- All the elements which are not involved in any of the operations to be applied remain unchanged.
- The rules associated with the label h are used for all membranes with this label, irrespective of whether or not the membrane is an initial one or it was obtained by creation.
- Several rules can be applied to different objects in the same membrane simultaneously.

Following the standard notations, the class of these P systems is denoted by $\mathcal{AM}^0_{-d,+mc,+antPri}$, where $-d$ indicates that dissolution rules are not used, $+mc$ indicates the use of membrane creation and we add $+antPri$ to denote the use of antimatter and annihilation rules with priority.

3 Recognizer P Systems

We recall the main notions related to recognizer P systems and complexity in Membrane Computing. For a detailed description see, e.g., [18, 19].

A decision problem X is a pair (I_X, θ_X) such that I_X is a language over a finite alphabet (whose elements are called *instances*) and θ_X is a total Boolean function over I_X. A *P system with input* is a tuple (Π, Σ, i_Π), where Π is a P system, with working alphabet Γ, with p membranes labelled by $1, \ldots, p$, and initial multisets $\mathcal{M}_1, \ldots, \mathcal{M}_p$ associated with them; Σ is an (input) alphabet strictly contained in Γ; the initial multisets are over $\Gamma - \Sigma$; and i_Π is the label of a distinguished (input) membrane. Let (Π, Σ, i_Π) be a P system with input, Γ be the working alphabet of Π, μ its membrane structure, and $\mathcal{M}_1, \ldots, \mathcal{M}_p$ the initial multisets of Π. Let m be a multiset over Σ. The *initial configuration of* (Π, Σ, i_Π) *with input* m is $(\mu, \mathcal{M}_1, \ldots, \mathcal{M}_{i_\Pi} \cup m, \ldots, \mathcal{M}_p)$. In the case of P systems with input and *with external output*, the above concepts are introduced in a similar way.

A *recognizer P system* is a P system with input and with external output such that:

- The working alphabet contains two distinguished elements *yes*, *no*.
- All its computations halt.
- If $\mathcal{C}$ is a computation of Π, then either the object *yes* or the object *no* (but not both) must have been released into the environment, and only in the last step of the computation. We say that $\mathcal{C}$ is an accepting computation (respectively, rejecting computation) if the object *yes* (respectively, *no*) appears in the external environment associated to the corresponding halting configuration of $\mathcal{C}$.

A decision problem X can be solved in a polynomially uniform way by a family $\mathbf{\Pi} = \{\Pi(n)\}_{n \in \mathbb{N}}$ of P systems of type $\mathcal{F}$ if the following holds:

- There is a deterministic Turing machine M such that, for every $n \in \mathbb{N}$, starting M with the unary representation of n on its input tape, it constructs the P system $\Pi(n)$ in polynomial time in n.
- There is a deterministic Turing machine N that started with an instance $I \in I_X$ with size n on its input tape, it computes a multiset w_I (called the *encoding of* I) over the input alphabet of $\Pi(n)$ in polynomial time in n.
- For every instance $I \in I_X$ with size n, starting $\Pi(n)$ with w_I in its input membrane, every computation of $\Pi(n)$ halts and sends out to the environment *yes* if and only if I is a positive instance of X.

We denote by $\mathbf{PMC}_{\mathcal{F}}$ the set of problems decidable in polynomial time using a polynomially uniform family of P systems of type $\mathcal{F}$.

4 Solving QSAT

In this section, we show that QSAT can be solved in linear time by a polynomially uniform family of recognizer P systems of type $\mathcal{AM}^0_{-d,+mc,+antPri}$.

The QSAT problem is the following one. Given a Boolean formula in conjunctive normal form over the propositional variables $\{x_1, \ldots, x_n\}$. Then the fully (existentially) quantified Boolean formula associated to φ is $\varphi^* = \exists x_1 \forall x_2 \ldots Q_n x_n \varphi$, (where Q_n is $\exists$ if n is odd, and it is $\forall$, otherwise). Now, the task is to decide if φ^* is *true*, i.e., to decide if there exists a truth assignment I of the variables $\{x_i \mid 1 \leq i \leq n, i \text{ is odd}\}$ such that each extension I^* of I to the variables $\{x_i \mid 1 \leq i \leq n, i \text{ is even}\}$ satisfies φ.

Next, we construct a recognizer P system of type $\mathcal{AM}^0_{-d,+mc,+antPri}$ to solve QSAT. The construction is a variant of the one occurring in [9] where it is shown that QSAT can be solved in linear time using a family of P systems with membrane creation using dissolution rules. The main difference between the construction in [9] and the one in this paper is that instead of dissolution rules we use annihilation rules to control the computations.

Similarly as in [9], the work of our P systems can be divided into three stages:

- Generation and evaluation stage: Using membrane creation we construct a binary complete tree where the leaves encode all possible truth assignments associated with the formula. The values of the formula corresponding to these truth assignments are obtained in the corresponding leaves. Moreover, the nodes at even (resp. odd) levels from the root are codified by OR gates (respectively, AND gates).
- Checking stage: In this stage the membrane structure corresponds to a Boolean circuit with gates AND and OR. We compute the values of the gates starting with the truth values computed at the leaves towards the root of the circuit which is the output gate.
- Output stage: The system sends out to the environment the answer of the system computed in the previous stages.

The evaluation stage will be the same as in [9], since there no dissolution rules are applied. In the other two stages we will use annihilation rules instead of using membrane dissolution.

Let $\varphi = C_1 \wedge \cdots \wedge C_m$ be a Boolean formula in conjunctive normal form over n variables. Then φ can be encoded as a multiset over the alphabet $\{x_{i,j}, y_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq n\}$, where $x_{i,j}$ (resp. $y_{i,j}$) represents the fact that x_j (resp. $\neg x_j$) occurs in C_i (notice that since barred objects usually denote antimatters, we cannot use $\bar{x}_{i,j}$ to represent negated variables). Let us denote the above encoding of φ by $cod(\varphi)$. Let us moreover choose an appropriate pairing function $\langle\, , \rangle$ from

$\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$. We construct a P system $\Pi(\langle n,m\rangle)$ processing the fully quantified formula φ^* associated with φ, when $cod(\varphi)$ is supplied in its input membrane. The family presented here is:

$$\Pi = \{(\Pi(\langle n,m\rangle), \Sigma(\langle n,m\rangle), i(\langle n,m\rangle)) \mid (n,m) \in \mathbb{N}^2\},$$

where the input alphabet is $\Sigma(\langle n,m\rangle) = \{x_{i,j}, y_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq n\}$, the input membrane is $i(\langle n,m\rangle) =< t, \vee >$, and the P system $\Pi(\langle n,m\rangle) = (\Gamma(\langle n,m\rangle), H(\langle n,m\rangle), \mu, w_s, w_{<t,\vee>}, R(\langle n,m\rangle))$ is defined as follows:

- Working alphabet:

$$\begin{aligned}\Gamma(\langle n,m\rangle) = \Sigma(\langle n,m\rangle) & \\ \cup \{z_{j,c} \mid j \in \{0,\dots n\}, c \in \{\wedge,\vee\}\} & \\ \cup \{z_{j,c,l} \mid j \in \{0,\dots,n-1\}, c \in \{\wedge,\vee\}, l \in \{t,f\}\} & \\ \cup \{x_{i,j}, y_{i,j}, x_{i,j,l}, y_{i,j,l} \mid j \in \{1,\dots,n\}, i \in \{1,\dots,m\}, l \in \{t,f\}\} & \\ \cup \{r_i, \overline{r}_i, r_{i,t}, r_{i,f} \mid i \in \{1,\dots,m\}\} & \\ \cup \{p_i, q_i, s_i, t_i, u_i, v_i \mid i \in \{1,2,3\}\} \cup \{\overline{q}_2, \overline{p}_3, \overline{t}_2, \overline{s}_3, \overline{u}_2, \overline{v}_3\} & \\ \cup \{a_i \mid i \in \{0,\dots,n-1\}\} \cup \{b_{i,l} \mid i \in \{1,\dots,n-1\}, l \in \{t,f\}\} & \\ \cup \{c_{i,j} \mid i \in \{1,\dots,n-1\}, j \in \{1,\dots,5(n-i+1)\}\} & \\ \cup \{yes, no, yes_\vee, no_\vee, yes_\wedge, no_\wedge, \overline{yes}_\wedge, \overline{no}_\vee\}. &\end{aligned}$$

- The set of labels: $H(\langle n,m\rangle) = \{< l,c >|\ l \in \{t,f\}, c \in \{\wedge,\vee\}\} \cup \{s\}$.
- Initial membrane structure: $\mu = [\,[\,]_{<t,\vee>}\,]_s$.
- Initial multiset: $w_s = \emptyset$, $w_{<t,\vee>} = \{a_0\, z_{0,\wedge,t}\, z_{0,\wedge,f}\}$.
- Input membrane: $[\,]_{<t,\vee>}$.
- The set of evolution rules, $R(\langle n,m\rangle)$, consists of the following rules (recall that λ denotes the empty string and if c is $\wedge$ then $\overline{c}$ is $\vee$ and if c is $\vee$ then $\overline{c}$ is $\wedge$):

1. $\left.\begin{array}{l}[z_{j,c} \to z_{j,c,t}\, z_{j,c,f}]_{<l,\overline{c}>} \\ [z_{j,c,l} \to [z_{j+1,\overline{c}}]_{<l,c>}]_{<l',\overline{c}>}\end{array}\right\}$ for $\begin{array}{l} l,l' \in \{t,f\}, \quad c \in \{\vee,\wedge\}, \\ j \in \{0,\dots,n-1\}.\end{array}$

With these rules the P system creates a nested membrane structure with 2^n innermost cells each of which corresponding to a truth assignment of the variables of the input formula. At the first step, the object $z_{j,c}$ evolves to two objects, one for the assignment *true* (the object $z_{j,c,t}$), and a second one for the assignment *false* (the object $z_{j,c,f}$). In the second step these objects create two membranes. The new membrane with t in its label represents the assignment $x_{j+1} = true$; on the other hand, the new membrane with f in its label represents the assignment $x_{j+1} =$*false*.

2. $\left.\begin{array}{l}[x_{i,j} \to x_{i,j,t}\, x_{i,j,f}]_{<l,c>} \\ [y_{i,j} \to y_{i,j,t}\, y_{i,j,f}]_{<l,c>} \\ [r_i \to r_{i,t}\, r_{i,f}]_{<l,c>}\end{array}\right\}$ for $\begin{array}{ll} l \in \{t,f\} & i \in \{1,\dots,m\}, \\ c \in \{\vee,\wedge\} & j \in \{1,\dots,n\}.\end{array}$

These rules duplicate the objects representing the formula. One copy corresponds

to the case when the variable is assigned $true$, the other copy corresponds to the case when it is assigned $false$. The objects r_i are also duplicated ($r_{i,t}$, $r_{i,f}$) in order to keep track of those clauses that evaluate true on the previous assignments to the variables.

3. $\left.\begin{array}{l} x_{i,1,t}[\,]_{<t,c>} \to [r_i]_{<t,c>} \\ y_{i,1,t}[\,]_{<t,c>} \to [\lambda]_{<t,c>} \\ x_{i,1,f}[\,]_{<f,c>} \to [\lambda]_{<f,c>} \\ y_{i,1,f}[\,]_{<f,c>} \to [r_i]_{<f,c>} \end{array}\right\}$ for $\begin{array}{l} i \in \{1,\ldots,m\}, \\ c \in \{\vee,\wedge\}. \end{array}$

Using these rules the P system can evaluate which clauses are $true$ under the possible ($true$ or $false$) truth assignments of the corresponding variable.

4. $\left.\begin{array}{l} x_{i,j,l}[\,]_{<l,c>} \to [x_{i,j-1}]_{<l,c>} \\ y_{i,j,l}[\,]_{<l,c>} \to [y_{i,j-1}]_{<l,c>} \\ r_{i,l}[\,]_{<l,c>} \to [r_i]_{<l,c>} \end{array}\right\}$ for $\begin{array}{l} l \in \{t,f\}, \quad i \in \{1,\ldots,m\}, \\ c \in \{\vee,\wedge\}, \quad j \in \{2,\ldots,n\}. \end{array}$

In order to analyse the next variable the second subscript of the objects $x_{i,j,l}$ and $y_{i,j,l}$ are decreased when they are sent into the corresponding membrane labelled with l. Moreover, following the last rule, the objects $r_{i,l}$ get into the new membranes to keep track of the clauses that evaluate true on the previous truth assignments.

5. $[z_{n,c} \to \overline{r}_1 \ldots \overline{r}_m\, p_1\, q_1]_{<l,\overline{c}>}$ for $l \in \{t,f\}$ and $c \in \{\vee,\wedge\}$.

After the evaluation stage, these rules introduce antimatters $\overline{z}_i$, $i \in \{1,\ldots,m\}$, in the inner membranes. These antimatters will be used to check if there is a clause that is not satisfied by the corresponding truth assignment.

6. $\left.\begin{array}{l} [r_i\, \overline{r}_i \to \lambda]_{<l,c>} \\ [\overline{r}_i \to \overline{q}_2]_{<l,c>} \end{array}\right\}$ for $\begin{array}{l} l \in \{t,f\}, \quad i \in \{1,\ldots,m\}, \\ c \in \{\vee,\wedge\} \end{array}$

If an antimatter is not annihilated by the first rule, i.e., there is a clause that is not satisfied by the corresponding truth assignment, then this antimatter introduces the antimatter $\overline{q}_2$.

7. $\left.\begin{array}{l} [q_1 \to q_2]_{<l,c>} \\ [p_1 \to p_2]_{<l,c>} \\ [q_2\overline{q}_2 \to \lambda]_{<l,c>} \\ [p_3\overline{p}_3 \to \lambda]_{<l,c>} \\ [p_2 \to p_3]_{<l,c>} \\ [p_3 \to no]_{<l,c>} \\ [q_2 \to q_3\overline{p}_3]_{<l,c>} \\ [q_3 \to yes]_{<l,c>} \end{array}\right\}$ for $\begin{array}{l} l \in \{t,f\} \\ c \in \{\vee,\wedge\} \end{array}$

These rules introduce yes in an innermost cell with label $< l, c >$ if and only if the antimatter $\overline{q}_2$ is not present in this cell. On the other hand, if $\overline{q}_2$ is in

the cell, then object *no* is introduced. Since $\overline{q}_2$ is introduced if and only if the corresponding truth assignment does not satisfy all the clauses of the formula, the appearance of *yes* or *no* in this cell indicates correctly whether the corresponding truth assignment satisfies the formula or not.

8. $\left.\begin{array}{l}[yes]_{<l,c>} \rightarrow yes_{\overline{c}}\,[\,]_{<l,c>} \\ [no]_{<l,c>} \rightarrow no_{\overline{c}}\,[\,]_{<l,c>}\end{array}\right\}$ for $\begin{array}{l} l \in \{t, f\} \\ c \in \{\vee, \wedge\}\end{array}$

These rules with the rules in groups **9** and **10** below will be used to check whether an appropriate combination of truth assignments according to the quantifiers $\exists$ and $\forall$ are founded or not.

9. $\left.\begin{array}{l}[yes_\wedge\,\overline{yes}_\wedge \rightarrow \lambda]_{<l,\wedge>} \\ [t_1 \rightarrow t_2]_{<l,\wedge>} \\ [s_1 \rightarrow s_2]_{<l,\wedge>} \\ [t_2 \rightarrow t_3\,\overline{s}_3]_{<l,\wedge>} \\ [s_2 \rightarrow s_3]_{<l,\wedge>} \\ [t_3]_{<l,\wedge>} \rightarrow yes_\vee\,[\,]_{<l,\wedge>} \\ [\overline{yes}_\wedge \rightarrow \overline{t}_2]_{<l,\wedge>} \\ [s_3]_{<l,\wedge>} \rightarrow no_\vee\,[\,]_{<l,\wedge>} \\ [t_2\,\overline{t_2} \rightarrow \lambda]_{<l,\wedge>} \\ [s_3\,\overline{s_3} \rightarrow \lambda]_{<l,\wedge>}\end{array}\right\}$ for $l \in \{t, f\}$

10. $\left.\begin{array}{l}[no_\vee\,\overline{no}_\vee \rightarrow \lambda]_{<l,\vee>} \\ [u_1 \rightarrow u_2]_{<l,\vee>} \\ [v_1 \rightarrow v_2]_{<l,\vee>} \\ [u_2 \rightarrow u_3\,\overline{v}_3]_{<l,\vee>} \\ [v_2 \rightarrow v_3]_{<l,\vee>} \\ [u_3]_{<l,\vee>} \rightarrow no_\wedge\,[\,]_{<l,\vee>} \\ [\overline{no}_\vee \rightarrow \overline{u}_2]_{<l,\vee>} \\ [v_3]_{<l,\vee>} \rightarrow yes_\wedge\,[\,]_{<l,\vee>} \\ [u_2\,\overline{u_2} \rightarrow \lambda]_{<l,\vee>} \\ [v_3\,\overline{v_3} \rightarrow \lambda]_{<l,\vee>}\end{array}\right\}$ for $l \in \{t, f\}$

11. $\left.\begin{array}{l}[a_i \rightarrow b_{i+1,t}\,b_{i+1,f}\,c_{i+1,1}]_{<l,c>} \\ b_{i+1,l}\,[\,]_{<l,c>} \rightarrow [a_{i+1}]_{<l,c>}\end{array}\right\}$ for $\begin{array}{l} l \in \{t, f\}, \quad i \in \{0, \ldots, n-2\}, \\ c \in \{\vee, \wedge\}\end{array}$

These rules with the rules in groups **12**-**14** below will be used to introduce the multisets $s_1 t_1 \overline{yes}^2_\wedge$ and $u_1 v_1 \overline{no}^2_\vee$ in the appropriate membranes. These multisets will be used then by rules in groups **9** and **10**, respectively. Since these multisets will be needed at different levels in the membrane structure in different time steps, we need to employ a counter $c_{i,j}$ for the appropriate timing (see also the

groups below).

12. $[a_{n-1} \to c_{n-1,1}]_{<l,c>}$ } for $l \in \{t, f\}, \quad c \in \{\vee, \wedge\}$

13. $[c_{i,j} \to c_{i,j+1}]_{<l,c>}$ } for $\begin{array}{l} l \in \{t, f\}, \quad i \in \{1, \ldots, n\}, \\ c \in \{\vee, \wedge\}, \quad j \in \{1, \ldots, 5n - 5i + 4\} \end{array}$

14. $\left.\begin{array}{l} [c_{n-k,5k+5} \to s_1\, t_1\, \overline{yes}^2_{\wedge}]_{<l,\wedge>} \\ [c_{n-k,5k+5} \to u_1\, v_1\, \overline{no}^2_{\vee}]_{<l,\vee>} \end{array}\right\}$ for $\begin{array}{l} l \in \{t, f\} \\ k \in \{0, \ldots, n-1\} \end{array}$

15. $[yes_{\wedge}]_s \to yes\,[\,]_s$
$[no_{\wedge}]_s \to no\,[\,]_s$.

These rules are used to send out the computed answer to the environment.

4.1 A Short Overview of the Computation

The initial configuration only has two membranes, the skin and an elementary membrane with label $< t, \vee >$. Labels have two types of information. On the one hand, the first symbol can be t or f, (*true* of *false*) and the second symbol can be $\wedge$ or $\vee$ to denote if the corresponding variable is universally or existentially quantified. Membrane creation rules are applied in parallel in order to obtain a binary tree like structure of membranes enclosed in the skin. In the $2n$-th step of the computation, 2^n elementary membranes are created. One for each possible truth assignment of the variables. The key set of rules for the evaluation of the variables is the set **3**. According to this set of rules, a symbol r_j is produced for each variable such that its truth value makes true the clause C_j.

Each of the 2^n elementary membranes in the configuration after $2n$ steps can be seen as one of the possible truth assignments for the variables and the set of different r_j objects inside represent the set of clauses satisfied by the corresponding truth assignment. In order to check if all the clauses are satisfied, a set with all the antiparticle $\overline{r}_j$ objects is generated in each elementary membrane. If all of these $\overline{r}_j$ objects are annihilated, it means that in this elementary membrane there were all the objects r_j (maybe with multiple copies). This means that the truth assignment associated with the elementary membrane satisfies all the clauses. Otherwise, if any $\overline{r}_j$ is not consumed after the annihilation process, then we conclude that the corresponding assignment does not satisfy the corresponding clause.

A set of technical rules produce an object *yes* or *no* inside each elementary membrane. The target of most of these rules is to control that only one object *yes* or *no* is generated, regardless the possible combination of multiple copies of r_j in the membrane.

Once the objects *yes* and *no* are generated in the elementary membranes, they are sent up in the tree-like membrane structure. When two of these objects arrive to an intermediate membrane, a new object *yes* or *no* is sent up, according to the label of the membrane. Such label encodes the type of quantification (universal or existential) of the corresponding variable. This stage is controlled by rules from the sets **9** and **10**.

Finally, an object *yes* or *no* arrives to the skin and it is sent out to the environment.

Consequently, the family $\mathbf{\Pi}$ solves in linear time the QSAT problem. Since QSAT is a **PSPACE**-complete problem, we have the following result:

Theorem 1. $\mathbf{PSPACE} \subseteq \mathbf{PMC}_{\mathcal{AM}^0_{-d,+mc,+antPri}}$.

5 PSPACE upper bound

In this section we show that $PMC^0_{\mathcal{AM}_{-d,+mc,+antPri}} \subseteq \mathbf{PSPACE}$. The proof is similar to the corresponding one in [23] where it is shown that $PMC_{\mathcal{AM}_{+d,+ne}} \subseteq \mathbf{PSPACE}$ (i.e., polynomially uniform families of P systems with active membranes, with polarizations, with dissolution and nonelementary membrane division rules can solve only problems in **PSPACE**). Nevertheless, there are substantial differences between the two proofs due to the different behaviour of these systems. In [23] it is observed that the multiset content and the polarization (so called, the *state*) of a membrane M after n steps of a P system Π can be obtained by recursively calculating the states of M, its parent, and its children after $n-1$ steps. To achieve that always the same computations are calculated by the recursive calls, a weak determinism on the rules of Π was introduced in [23] (notice that since Π is a recognizer P system, it is confluent and thus it is enough to simulate only one of its computations). Moreover, to distinguish between membranes having same labels, unique indexes were associated to the membranes of a configuration. The index of a new membrane in a configuration is derived from the index of the corresponding membrane in the previous configuration.

In our proof, on the one hand, we do not have to deal with the polarizations of the membranes. On the other hand, we should employ an indexing technique that is different to that occurring in [23] due to the reason that in P systems with membrane creation new membranes are created from objects and not from membranes. The rest of this section is devoted to the proof of the following theorem:

Theorem 2. $\mathbf{PMC}_{\mathcal{AM}^0_{-d,+mc,+antPri}} \subseteq \boldsymbol{PSPACE}$.

We give an algorithm A with the following properties. Let $\mathbf{\Pi} = \{\Pi(n)\}_{n\in\mathbb{N}}$ be a polynomially uniform family of recognizer P systems of type $\mathcal{AM}_{-d,+mc,+antPri}$. Then, for every $n \in \mathbb{N}$ and input multiset m of $\Pi(n)$, A decides using polynomial space in n if $\Pi(n)$ produces *yes* started on input m.

Assume that $\Pi(n) = (\Gamma, H, \mu, W, h_i, R)$. Since $\Pi(n)$ is a recognizer P system, all of its computations yield the same answer. Thus, it is enough to simulate one particular computation of $\Pi(n)$. To this end, we introduce the following weak priorities on the rules other than annihilation rules in R (clearly, by definition, annihilation rules have priority over the rest of the rules). We assume that evolution rules have the highest priority, followed by send-out communication, send-in communication, and membrane creation rules. Similar type of rules have priority

over each other as follows. Assume we have two rules r_1 and r_2 of the same type. Then r_1 has priority over r_2 if and only if one of the following conditions holds:

- $r_1 = [a \to \alpha]_i$, $r_2 = [a \to \beta]_i$ and $\alpha < \beta$ (where $<$ is the usual lexicographical order on words),
- $r_1 = a[\ ]_i \to [b]_i$, $r_2 = a[\ ]_j \to [c]_j$ and $(i < j$ or $(i = j$ and $b < c))$,
- $r_1 = [a]_i \to b[\ ]_i$, $r_2 = [a]_i \to c[\ ]_i$ and $b < c$,
- $r_1 = [a \to [\alpha]_j]_i$, $r_2 = [a \to [\beta]_k]_i$ and $(j < k$ or $(j = k$ and $\alpha < \beta))$.

One can see that even with the above priorities, $\Pi(n)$ can have different computations on the same input. Indeed, assume, for example, that $\Pi(n)$ has a configuration which contains a membrane structure $[[\ ]_2\ [\ ]_2]_1$ with an object a in membrane 1. Assume also that $\Pi(n)$ has the rule $r = a[\ ]_2 \to [b]_2$. Then when $\Pi(n)$ applies r, it nondeterministically chooses a membrane with label 1 and sends a into this membrane. It also can bee seen that there is no such nondeterminism concerning the other types of rules. As we will see later, using unique indexes of the membranes having the same labels, we can avoid of this nondeterminism during the simulation.

Next we define these unique indexes. First of all, we assume that different membranes have different labels in the initial configuration. Assume now that $C = C_1, \ldots, C_l$ is a computation of $\Pi(n)$. Let $i \in \{1, \ldots, l\}$ and M be a membrane in C_i. Let $d(M, C_i)$ denote the depth of M in the membrane structure in C_i. More precisely, if M is the skin, then $d(M, C_i) = 1$; if M is a child of a membrane M', then $d(M, C_i) := d(M', C_i) + 1$. Let moreover $d(C_i) := max\{d(M, C_i) \mid M$ is a membrane in $C_i\}$. We inductively define a function f_C that assigns to every membrane M in C_i an index from $((H \cup \mathbb{N})^{i+1})^{d(M,C_i)}$ (i.e., the index of M will be a $d(M, C_i)$-tuple of words with length $i+1$ containing letters from $H \cup \mathbb{N}$). The indexes of the membranes in C_1 are inductively defined as follows. For the skin membrane M with label s, let $F_C(M) := (s1)$. Now let M be a membrane in C_1 and assume that $F_C(M) = (w_1, \ldots, w_{d(M,C_1)})$. If M' is a child membrane of M with label h, then $F_C(M') := (h1, w_1, \ldots, w_{d(M,C_1)})$. An example of this indexing in the initial configuration can be seen on Fig. 1, where these indexes are written in the lower-right corner of the membranes. Now assume that f_C already assigns

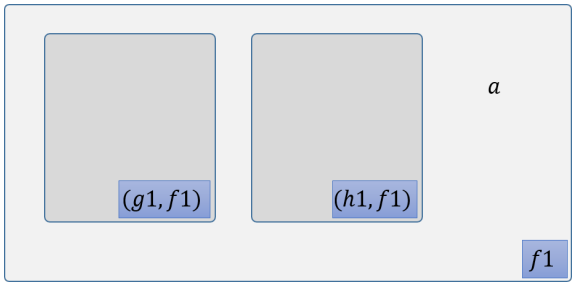


Fig. 1.

an index to every membrane in C_i ($i < l$). Let M be a membrane in C_i and assume that $f_C(M) = (w_1, \ldots, w_{d(M,C_i)})$. If M' is the membrane in C_{i+1} that corresponds to M, then let $f_C(M') = (w_1 1, \ldots, w_{d(M,C_i)} 1)$ (notice that since dissolution and membrane duplication rules are not allowed, every membrane in C_i has a corresponding membrane in C_{i+1}). Finally, let $h \in H$ and assume that $a_1, \ldots, a_k$ are those objects in M (ordered lexicographically) that create membranes with label h in the step from C_i to C_{i+1}. For every $j \in \{1, \ldots, k\}$, let M_j be that membrane which is created from a_j. Then $f_C(M_j) := (ha_j^i j, w_1 1, \ldots, w_{d(M,C_i)} 1)$. An example of this indexing can be seen in Fig. 2, where at the first step a enters to membrane with index $(h1, f1)$ and evolves to b. Then, during the second step, b creates the membrane with index $(gbb1, h111, f111)$. Notice that from this index we can

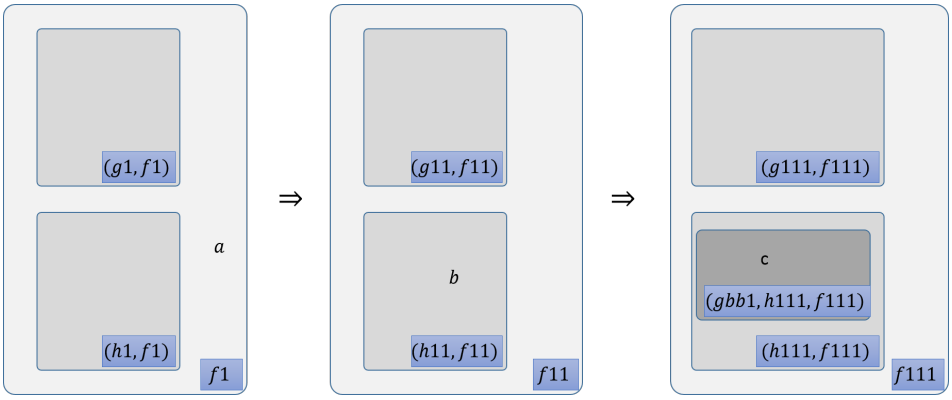


Fig. 2.

decode the following information. The label of the membrane is g, its parent has label h and index $(h111, f111)$, and the membrane was created in the second step of the computation from an object b. In general, the above defined indexes have the following properties:

- For a given initial configuration C_1 and a computation $C = C_1, \ldots, C_l$, the possible indexes of the membranes in C can be effectively enumerated (notice that the maximal number of objects in a membrane can be calculated from the number of objects in C_1 and the number of computation steps);
- For a membrane M with index $(h_1 i_{1,1} \ldots i_{1,j}, \ldots, h_k i_{k,1} \ldots i_{k,j})$,
  - if $k > 1$, then the index of the parent membrane of M is $(h_2 i_{2,1} \ldots i_{2,j}, \ldots, h_k i_{k,1} \ldots i_{k,j})$, and
  - the possible indexes of the children of M can be effectively enumerated;
- For a membrane M with index $(h_1 i_{1,1} \ldots i_{1,j}, \ldots, h_k i_{k,1} \ldots i_{k,j})$ such that $j > 1$, either
  - $i_{1,1} = \ldots = i_{1,j} = 1$ and M occurs already in the initial configuration, or

– $i_{1,1} = \ldots = i_{1,j-1} = a$, for some $a \in \Gamma$, and M is created from a in the $(j-1)$th step of the computation.

Let $C = C_1, \ldots, C_l$ be a computation of $\Pi(n)$ and $j \in \{1, \ldots, l\}$. We introduce an order on the indexes of membranes occurring in C_j and satisfying that $d(M, C_j) = d(M', C_j)$. Assume that M and M' are membranes with these properties and $f_C(M) = (h_1 i_{1,1} \ldots i_{1,j}, \ldots, h_k i_{k,1} \ldots i_{k,j})$, and $f_C(M') = (h'_1 i'_{1,1} \ldots i'_{1,j}, \ldots, h'_k i'_{k,1} \ldots i'_{k,j})$, where $k = d(M, C_j)$. Then $(h_1 i_{1,1} \ldots i_{1,j}, \ldots, h_k i_{k,1} \ldots i_{k,j}) \leq (h'_1 i'_{1,1} \ldots i'_{1,j}, \ldots, h'_k i'_{k,1} \ldots i'_{k,j})$ if and only if $h_1 i_{1,1} \ldots i_{1,j} \leq h'_1 i'_{1,1} \ldots i'_{1,j}$, where $\leq$ is the usual lexicographical order on words assuming that, for every object $a \in \Gamma$ and number $n \in \mathbb{N}$, $a < n$.

Let $C = C_1, \ldots, C_l$ be a halting computation of $\Pi(n)$ such that, for every $i \in \{1, \ldots, l\}$, C_i has the following property. Assume that there is a membrane M with label h in C_i and there are more than one membranes with label g in M. Assume also that there is a rule $r = a[\ ]_g \to [b]_g$ in R. Then C_{i+1} is that configuration of $\Pi(n)$ where the objects a in M are sent by the rule r to that membrane with label g which has a smaller index by the above defined order on the indexes. We will simulate this particular computation C by recursively calculating the multiset content of membranes in C. This is done using a function called CONTENT. CONTENT gets as parameters an index of a membrane M and a number j and returns with the multiset content of M in C_j (i.e., the content of M after $j-1$ computation steps). The basic strategy of the computation, roughly, is the following. First we try to compute the content of M and the content of its parent M' in C_{j-1}. If M' does not exist in C_{j-1}, then M also does not exist and we can return *nil* showing that the content of M in C_j is undefined. If only M does not exist in C_{j-1}, we check whether it was created in the step from C_{j-1} to C_j. If no, then we return *nil*, otherwise we return the newly created content of M. If both M and M' exist in C_{j-1}, then we calculate the content of M in C_j using the contents of M and M' in C_{j-1} and by calculating the contents of the children of M in C_{j-1}.

For the better readability, in the algorithms defined below we will refer to the annihilation (resp. evolution, send-out communication, send-in communication, and membrane creation) rules as *ann* (resp. *evo*, *in_com*, *out_com*, and *cre*).

1. **function** CONTENT($(h_1 i_{1,1} \ldots i_{1,j}, \ldots, h_k i_{k,1} \ldots i_{k,j}), j$)
   // We calculate the multiset content of a membrane M with index $(h_1 i_{1,1} \ldots i_{1,j}, \ldots, h_k i_{k,1} \ldots i_{k,j})$ in C_j;
2. **if** $j = 1$ **then**
3. **if** $i_{1,1} = 1, \ldots, i_{k,1} = 1$ AND there is a membrane structure $\mu = [[[]_{h_1} \ldots]_{h_{k-1}}]_{h_k}$ in C_1 **then**
4. **return** the multiset content of the inner membrane in μ
5. **else return** *nil*
6. **end if**;
7. **exit**

end if;
// If $j = 1$ and the index corresponds to a membrane in C_1, then return the content of this membrane, and return *nil*, otherwise;
$S \leftarrow$ CONTENT$((h_1 i_{1,1} \dots i_{1,j-1}, \dots, h_k i_{k,1} \dots i_{k,j-1}), j-1)$;
// If $j > 1$, then we recursively calculate the content of M in C_{j-1};
$S_p \leftarrow \varnothing$; $S' \leftarrow \varnothing$; $S_c \leftarrow \varnothing$; $X' \leftarrow \varnothing$;
if h_1 is not the label of the skin membrane **then**
  $S_p \leftarrow$ CONTENT$((h_2 i_{2,1} \dots i_{2,j-1}, \dots, h_k i_{k,1} \dots i_{k,j-1}), j-1)$;
  // If M is not the skin, then we calculate the content of the parent M' of M in C_{j-1};
  if $S_p = nil$ **then return** *nil*; **exit**
  // If the parent M' of M does not exist in C_{j-1}, then M cannot exist in C_j;
  else
    TRYRULES(h_2, ann, S_p, X', X');
    TRYRULES(h_2, evo, S_p, X', X');
    TRYRULES$(h_2, out_com, S_p, X', X')$;
    // We remove from S_p those objects that do not contribute to the content of M in C_j by applying rules *ann*, *evo*, and *out_com* to the content of M' in C_{j-1};
    for all possible index $(h'_1 i'_{1,1} \dots i'_{1,j-1}, \dots, h_k i_{k,1} \dots i_{k,j-1})$ such that $(h'_1 i'_{1,1} \dots i'_{1,j-1}, \dots, h_k i_{k,1} \dots i_{k,j-1}) < (h_1 i_{1,1} \dots i_{1,j-1}, \dots, h_k i_{k,1} \dots i_{k,j-1})$
      $S_c \leftarrow$ CONTENT$((h'_1 i'_{1,1} \dots i'_{1,j-1}, \dots, h_k i_{k,1} \dots i_{k,j-1}), j-1)$;
      if $S_c \neq nil$ **then**
        TRYRULES$(h'_1, in_com, X', X', S_p)$
      end if
    end for
  // We remove those objects from the content of M' that are sent to child membranes other than M;
  end if
end if
if $S \neq nil$ **then**
  TRYRULES(h_1, ann, S, X', X');
  TRYRULES(h_1, evo, S, S', X');
  TRYRULES$(h_1, out_com, S, X', X')$;
  //We apply rules *ann*, *evo*, and *out_com* to the content S of M in C_{j-1};
  TRYRULES$(h_1, in_com, S_p, S', X')$
  // We send objects from the parent M' to the children M by applying *in_com* rules;
  COMMWITHCHILDREN$((h_1 i_{1,1} \dots i_{1,j-1}, \dots, h_k i_{k,1} \dots i_{k,j-1}), j-1, S, S')$;
  //We calculate the interactions between M and its children in C_{j-1};
  $S' \leftarrow S \cup S'$;
  // We calculate the content of M in C_j; here S contains those objects that were not involved by any rule; S' contains the results of the applicable rules;
  return S'; **exit**
end if

35. **if** $S = nil$ **then**
36. CommWithChildren$((h_2 i_{2,1} \dots i_{2,j-1}, \dots, h_k i_{k,1} \dots i_{k,j-1}), j-1, S_p, X')$;
    // If M does not exists in C_{j-1}, then we examine if it can be created in M' in the step from C_{j-1} to C_j; first we remove those objects from the content of M' that are sent to its children during the step form C_{j-1} to C_j;
37. **if** $a_1 \dots a_t \subseteq S_p$ $(a_1 \leq \dots \leq a_t)$ such that $i_{1,1} = \dots = i_{1,j-1} = a_t$ AND $i_{1,j} = t$ AND $[a_t \rightarrow [\, v \,]_{h_1}]_{h_2} \in R$ **then**
38. $S' \leftarrow v$;
    // If $a_1, \dots, a_t$ occur in S_p and M can be created in M' by the rule $[a_t \rightarrow [\, v \,]_{h_1}]_{h_2}$, then the content of M in C_j is v;
39. **return** S'
40. **else**
41. **return** nil
42. **end if**
43. **end if**

Next we define the procedure TryRules which have five parameters. The first one is a label of the membrane, the next one is a type of rules, and the last three parameters are those sets of objects that are involved by the application of the corresponding type of rules.

1. **procedure** TryRules$(g, type, X, Y, Z)$
2. **case** *type* **of**
3. *ann*: **for each** rule $[\, a\overline{a} \rightarrow \lambda \,]_g$ **do**
4. remove every pair $a, \overline{a}$ from X
5. **end for**
6. *evo*: **for each** rule $[\, a \rightarrow \alpha \,]_g$ **do**
7. remove every occurrence of a from X;
8. add to Y the same number of multiset represented by α
9. **end for**
10. *in_com*: **for each** rule $a[\ ]_g \rightarrow [\, b \,]_g$ **do**
11. remove every occurrence of a from Z;
12. add to Y the same number of objects b
13. **end for**
14. *out_com*: **for each** rule $[\, a \,]_g \rightarrow b[\ ]_g$ **do**
15. remove every occurrence of a from X;
16. add to Z the same number of objects b
17. **end for**
18. *cre*: **for each** rule $[\, a \rightarrow [\ ]_h \,]_g$ **do**
19. remove every occurrence of a from X
20. **end for**
21. **end case**

Now, we define the procedure CommWithChildren which calculates the communications between a membrane and its children. This procedure has four parameters. The second parameter is a number j which determines which step of

the computation is considered. The first parameter is an index of a membrane in C_j. The last two parameters are those sets of objects that are involved by the communications between this membrane and its children in the step from C_j to C_{j+1}.

1. **procedure** CommWithChildren$((h_1 i_{1,1} \dots i_{1,j}, \dots, h_k i_{k,1} \dots i_{k,j}), j, X, Y)$
2. **for each** $g l_{1,1} \dots l_{1,j}$, where $g \in H, l_{1,1}, \dots, l_{1,j} \in H \cup \mathbb{N}$ **do**
3. $S_c \leftarrow Content((g l_{1,1} \dots l_{1,j}, h_1 i_{1,1} \dots i_{1,j}, \dots, h_k i_{k,1} \dots i_{k,j}), j)$;
4. **if** $S_c \neq nil$ **then**
5. $Y' \leftarrow \varnothing$;
6. TryRules(g, ann, S_c, Y', Y');
7. TryRules(g, evo, S_c, Y', Y');
8. TryRules(g, out_com, S_c, Y', Y);
9. TryRules(g, in_com, Y', Y', X);
   // We apply rules of type *ann* and *evo* to keep the computation deterministic; membrane creation rules are skipped as they do not contribute to the content of the parent membrane stored in X; in-communication rules involve only the content of the parent membrane;
10. **end if**
11. **end for**

Finally, we present the procedure A to decide if $\Pi(n)$ sends out to the environment *yes* on a given input multiset m. We assume without loss of generality that those rules that send out to the environment *yes* (resp. *no*) have the form $[\, yes \,]_s \rightarrow yes$ (resp. $[\, yes \,]_s \rightarrow yes$), where s is the label of the skin membrane.

1. **procedure** A$(\Pi(n))$
   // $\Pi(n) = (\Gamma, H, \mu, W, h_i, R)$ is a recognizer P systems of type $\mathcal{AM}_{-d,+mc,+antPri}$ with input multiset m
2. $s \leftarrow$ the label of the skin in μ;
3. $S \leftarrow \varnothing$;
4. **for each** $j = 1, 2, \dots$ **do**
5. $S \leftarrow$ Content$((s i_1 \dots i_j), j)$ where $i_1 = 1, \dots, i_j = 1$;
6. **if** $yes \in S$ and there is a rule $[\, yes \,]_s \rightarrow yes$ **then output:** *yes*; **exit**
7. **end if**
8. **if** $no \in S$ and there is a rule $[\, no \,]_s \rightarrow no$ **then output:** *no*; **exit**
9. **end if**
10. **end for**

First we show that A halts on $\Pi(n)$ and m. Since Content recursively calls itself with a decreasing second parameter and Content with second parameter 1 exits after finite steps, we can conclude that Content always exits after finite steps. Moreover, since $\Pi(n)$ is a recognizer P system, it halts in l steps, for an appropriate number l. Thus the multiset content of the skin of Π should contain *yes* or *no* after at most $l - 1$ steps. Therefore, $l + 1$ is the highest number that occurs as a second parameter in the calls of Content in A. This means that A stops after a finite number of steps.

Next we discuss the space complexity of A. Let $\mathbf{\Pi} = \{\Pi(n)\}_{n\in\mathbb{N}}$ be a polynomially uniform family of P systems of type $\mathcal{AM}_{-d,+mc,+antPri}$. By definition, there is a polynomial $p(n)$ such that the size of the initial configuration of $\Pi(n)$ containing an encoding of a formula in its input membrane is upper bounded by $p(n)$. Moreover, there is a polynomial $t(n)$ such that the running time of $\Pi(n)$ is upper bounded by $t(n)$.

Let $C = C_1, \ldots, C_l$ be a halting computation of $\Pi(n)$, for some $l \leq t(n)$, and M be a membrane in C_i ($i \in \{1, \ldots, l\}$). Then the index $w = f_C(M)$ contains at most $k + i - 1$ components, where $k = d(C_1)$. Clearly, k is upper bounded by $p(n)$. Moreover, every component of w is a word with length at most $t(n) + 2$. It follows then that w contains at most $(p(n)+t(n)-1)\cdot(t(n)+2)$ letters. Clearly, for every $i \in \{1, \ldots, l\}$, the size of C_i is at most $p(n)^{O(t(n))}$ (the size of a configuration is the sum of the number of objects and membranes in the configuration). Thus, every letter in w that is contained in $\mathbb{N}$ is at most $p(n)^{k\cdot t(n)}$, for some appropriate constant k. Therefore, storing a letter of a word in w needs at most $\log(p(n)^{k\cdot t(n)}) = O(nt(n))$ bits (notice that the first letters of the words in w are labels and the number of different labels is bounded by $p(n)$). This implies that the index w can be stored using at most $O((p(n) + t(n)) \cdot t(n) \cdot nt(n))$ bits, i.e., the number of necessary bits is polynomial in n. Therefore, on every level of the recursion in the function Content, the number of bits that is used to store the parameters is upper bounded by an appropriate polynomial. Moreover, the depth of the recursion in Content is bounded by the poynomial $t(n)$. It follows, that the space complexity of A is bounded by a polinomial too.

6 Conclusions and Future Work

In this paper, we have proved that the family of P systems with membrane creation and annihilation rules characterizes the complexity class **PSPACE**. In [5] it has been proved that P systems with active membranes without polarizations, without dissolution and with division of elementary and non-elementary membranes endowed with antimatter and annihilation rules can solve **NP**-complete problems. It is an interesting research topic to explore the exact computational power of these systems. It seems that these systems can only solve problems in **PSPACE**. On the other hand, solving a **PSPACE**-complete problem with these systems seems to be a challenging task.

Acknowledgements

MAGN acknowledges the support of the project TIN2012-37434 of the Ministerio de Economía y Competitividad of Spain.

References

1. Alhazov, A., Aman, B., Freund, R.: P systems with anti-matter. In: Gheorghe, M., Rozenberg, G., Salomaa, A., Sosík, P., Zandron, C. (eds.) Membrane Computing - 15th International Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8961, pp. 66–85. Springer (2014)
2. Alhazov, A., Aman, B., Freund, R., Păun, Gh.: Matter and anti-matter in membrane systems. In: DCFS 2014. Lecture Notes in Computer Science, vol. 8614, pp. 65–76. Springer (2014)
3. Berman, L.: The complexity of logical theories. Theoretical Computer Science 11, 71–77 (1980)
4. Blizard, W.D.: Negative membership. Notre Dame Journal of Formal Logic 31(3), 346–368 (1990)
5. Díaz-Pernil, D., Peña-Cantillana, F., Alhazov, A., Freund, R., Gutiérrez-Naranjo, M.A.: Antimatter as a frontier of tractability in membrane computing. Fundamenta Informaticae 134, 83–96 (2014)
6. Freund, R., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.): Membrane Computing, 6th International Workshop, WMC 2005, Vienna, Austria, July 18-21, 2005, Revised Selected and Invited Papers, Lecture Notes in Computer Science, vol. 3850. Springer, Berlin Heidelberg (2006)
7. Gott, J.: Time Travel in Einstein's Universe: The Physical Possibilities of Travel Through Time. Houghton Mifflin (2001)
8. Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J., Riscos-Núñez, A., Romero-Campero, F.J.: On the power of dissolution in P systems with active membranes. In: Freund et al. [6], pp. 224–240
9. Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J., Romero-Campero, F.J.: A linear solution for QSAT with membrane creation. In: Freund et al. [6], pp. 241–252
10. Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J., Romero-Campero, F.J.: A linear solution of subset sum problem by using membrane creation. In: Mira, J., Álvarez, J.R. (eds.) IWINAC (1). Lecture Notes in Computer Science, vol. 3561, pp. 258–267. Springer, Berlin Heidelberg (2005)
11. Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J., Romero-Campero, F.J.: A uniform solution to SAT using membrane creation. Theoretical Computer Science 371(1-2), 54–61 (2007)
12. Ito, M., Martín-Vide, C., Păun, G.: A characterization of parikh sets of ET0L languages in terms of P systems. In: Ito, M., Păun, G., Yu, S. (eds.) Words, Semigroups, and Transductions - Festschrift in Honor of Gabriel Thierrin. pp. 239–253. World Scientific (2001)
13. Loeb, D.: Sets with a negative number of elements. Advances in Mathematics 91, 64–74 (1992)
14. Luisi, P.: The chemical implementation of autopoiesis. In: Fleischaker, G., Colonna, S., Luisi, P. (eds.) Self-Production of Supramolecular Structures, NATO ASI Series, vol. 446, pp. 179–197. Springer Netherlands (1994)
15. Metta, V.P., Krithivasan, K., Garg, D.: Computability of spiking nueral P systems with anti-spikes. New Mathematics and Natural Computation (NMNC) 08(03), 283–295 (2012)
16. Mutyam, M., Krithivasan, K.: P systems with membrane creation: Universality and efficiency. In: Margenstern, M., Rogozhin, Y. (eds.) MCU. Lecture Notes in Computer Science, vol. 2055, pp. 276–287. Springer (2001)

17. Pan, L., Păun, Gh.: Spiking neural P systems with anti-spikes. International Journal of Computers, Communications & Control IV(3), 273–282 (September 2009)
18. Pérez-Jiménez, M.J.: An approach to computational complexity in membrane computing. In: Mauri, G., Păun, Gh., Pérez-Jiménez, M.J., Rozenberg, G., Salomaa, A. (eds.) Workshop on Membrane Computing. Lecture Notes in Computer Science, vol. 3365, pp. 85–109. Springer (2004)
19. Pérez-Jiménez, M.J., Riscos-Núñez, A., Romero-Jiménez, A., Woods, D.: Complexity - membrane division, membrane creation. In: Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) The Oxford Handbook of Membrane Computing, pp. 302 – 336. Oxford University Press, Oxford, England (2010)
20. Păun, Gh.: Some quick research topics., In these proceedings.
21. Păun, G.: Four (somewhat nonstandard) research topics. In: Maccías-Ramos, L.F., del Amor, M.A.M., Păun, G., Riscos-Núñez, A., Valencia-Cabrera, L. (eds.) Twelfth Brainstorming Week on Membrane Computing. pp. 305–309. Fénix Editora, Sevilla, Spain (2014)
22. Song, T., Jiang, Y., Shi, X., Zeng, X.: Small universal spiking neural P systems with anti-spikes. Journal of Computational and Theoretical Nanoscience 10(4), 999–1006 (2013)
23. Sosík, P., Rodríguez-Patón, A.: Membrane computing and complexity theory: A characterization of PSPACE. J. Comput. Syst. Sci. 73(1), 137–152 (2007)
24. Tan, G., Song, T., Chen, Z., Zeng, X.: Spiking neural P systems with anti-spikes and without annihilating priority working in a 'flip-flop' way. International Journal of Computing Science and Mathematics 4(2), 152–162 (Jul 2013)

kPWorkbench: A Software Framework for Kernel P Systems

Marian Gheorghe[1], Florentin Ipate[2], Laurentiu Mierla[2], and Savas Konur[1]

[1] School of Electrical Engineering and Computer Science, University of Bradford
Bradford BD7 1DP, UK
{m.gheorghe, s.konur}@bradford.ac.uk
[2] Department of Computer Science, University of Bucharest
Str. Academiei nr. 14, 010014, Bucharest, Romania
florentin.ipate@ifsoft.ro, laurentiu.mierla@gmail.com

Summary. *P* systems are the computational models introduced in the context of membrane computing, a computational paradigm within the more general area of unconventional computing. *Kernel P (kP)* systems are defined to unify the specification of different variants of P systems, motivated by challenging theoretical aspects and the need to model different problems. In this paper, we present kPWorkbench, a software framework developed to support kP systems. kPWorkbench integrates several *simulation* and *verification* tools and methods, and provides a software suit for the modelling and analysis of membrane systems.

1 Introduction

Membrane computing is a computational paradigm, within the more general area of unconventional computing [24], inspired by the structure and behaviour of the eukaryotic cell. The formal models introduced in this context are called membrane systems or P systems. After their introduction [22], membrane systems have been widely investigated for computational properties and complexity aspects, but also as a model for various applications [23]. The introduction of different variants of P systems has been motivated by challenging theoretical aspects, but also by the need to model different problems. An account of the theoretical developments is presented in [23], a set of general applications can be found in [6], whereas specific applications in systems and synthetic biology are provided in [11] and some of the future challenges are presented in [14]. More recently, applications in optimisations and graphics [16] and synchronisation of distributed systems [9] have been developed.

Several variants of P systems have been introduced and studied to model and analyse different problems, e.g., systems and synthetic biology [11], synchronisation of distributed systems [9], optimisations and graphics [16]. While the introduction of new variants allowed modelling different sets of problems, the ad-hoc addition

of new features has caused an abundance of P system variants, with a lack of a coherent integrating view, and well-defined framework would allow us to analyse, verify and validate the system behaviour.

We introduced *kernel P systems (kP systems)* [15] as an attempt to target these issues and create more general membrane computing models, integrating the most used concepts from P systems. A revised version of the model and the specification language can be found in [12] and its usage to specify the 3-colouring problem and a comparison to another solution provided in a similar context [8], is described in [13]. The kP systems have been also used to specify and analyse, through formal verification, synthetic biology systems [21, 20].

We have previously studied the theoretical aspects [15] and the verification and simulation techniques developed for kP systems [10, 3, 2]. In this paper, we present kPWorkbench (available and can be downloaded from its website `http://www.kpworkbench.org`), a software framework developed to support the analysis of kP systems. kPWorkbench integrates several *simulation* and *verification* tools and methods. The framework also facilitates verification by incorporating a property language based on *natural language* statements, which makes the property specification a very easy task. These features make kPWorkbench the only available tool supporting the non-probabilistic analysis of membrane systems through simulation and verification. The usability and novelty of our approach have been illustrated by some case studies [21, 20] chosen from synthetic biology (a new and emerging branch of biology that aspires to the engineering of new biological systems).

The paper is organised as follows: in Section 2 are introduced the key concepts and definitions related to kP systems; the kPWorkbench is discussed in Section 3; in Section 4 are summarised some kP systems applications; Section 5 illustrates through some examples the use of the kPWorkbench platform and final conclusions are provided in Section 6.

2 Kernel P Systems

A kP system is made of compartments placed in a graph-like structure. A compartment C_i has a type $t_i = (R_i, \sigma_i)$, $t_i \in T$, where T represents the set of all types, describing the associated set of rules R_i and the execution strategy that the compartment may follow. Note that, unlike traditional P system models, in kP systems each compartment may have its own rule application strategy. The following definitions are largely from [15].

Definition 1. *A* kernel P (kP) system *of degree n is a tuple*

$$k\Pi = (A, \mu, C_1, \ldots, C_n, i_0),$$

where A is a finite set of elements called objects*; μ defines the* membrane structure*, which is a graph, (V, E), where V are vertices indicating components, and E edges; $C_i = (t_i, w_i)$, $1 \leq i \leq n$, is a* compartment *of the system consisting of*

a compartment type from T and an initial multiset*, w_i over A; i_0 is the* output compartment *where the result is obtained.*

Each rule r may have a **guard** g denoted as $r\ \{g\}$. The rule r is applicable to a multiset w when its left hand side is contained into w and g holds for w. The guards are constructed using multisets over A and relational and Boolean operators. For example, rule $r : ac \to c\ \{\geq a^3 \wedge \geq b^2 \vee \neg > c\}$ can be applied iff the current multiset, w, includes the left hand side of r, i.e., ac and the guard holds for w - it has at least 3 a's and 2 b's or no more than a c. A formal definition may be found in [15].

Definition 2. *A rule associated with a compartment type l_i can have one of the following types:*

(a) **rewriting and communication** *rule: $x \to y\ \{g\}$,*
where $x \in A^+$ and y has the form $y = (a_1, t_1) \dots (a_h, t_h)$, $h \geq 0$, $a_j \in A$ and t_j indicates a compartment type from T – see Definition 1 – with instance compartments linked to the current compartment; t_j might indicate the type of the current compartment, i.e., t_{l_i} – in this case it is ignored; if a link does not exist (the two compartments are not in E) then the rule is not applied; if a target, t_j, refers to a compartment type that has more than one instance connected to l_i, then one of them will be non-deterministically chosen;

(b) **structure changing rules***; the following types are considered:*

(b1) **membrane division** *rule: $[x]_{t_{l_i}} \to [y_1]_{t_{i_1}} \dots [y_p]_{t_{i_p}}\ \{g\}$,*
where $x \in A^+$ and y_j has the form $y_j = (a_{j,1}, t_{j,1}) \dots (a_{j,h_j}, t_{j,h_j})$ like in rewriting and communication rules; the compartment l_i will be replaced by p compartments; the j-th compartment, instantiated from the compartment type t_{i_j} contains the same objects as l_i, but x, which will be replaced by y_j; all the links of l_i are inherited by each of the newly created compartments;

(b2) **membrane dissolution** *rule: $[x]_{t_{l_i}} \to \lambda\ \{g\}$;*
the compartment l_i and its entire contents is destroyed together with its links. This contrasts with the classical dissolution semantics where the inner multiset is passed to the parent *membrane - in a tree-like membrane structure;*

(b3) **link creation** *rule: $[x]_{t_{l_i}}; []_{t_{l_j}} \to [y]_{t_{l_i}} - []_{t_{l_j}}\ \{g\}$;*
the current compartment is linked to a compartment of type t_{l_j} and x is transformed into y; if more than one instance of the compartment type t_{l_j} exists then one of them will be non-deterministically picked up; g is a guard that refers to the compartment instantiated from the compartment type t_{l_i};

(b4) **link destruction** *rule: $[x]_{t_{l_i}} - []_{t_{l_j}} \to [y]_{t_{l_i}}; []_{t_{l_j}}\ \{g\}$;*
is the opposite of link creation and means that the compartments are disconnected.

Each compartment can be regarded as an instance of a particular *compartment type* and is therefore subject to its associated rules. One of the main distinctive features of kP systems is the execution strategy which is now statutory to types

rather than unitary across the system. Thus, each membrane applies its type specific instruction set, as coordinated by the associated execution strategy.

An execution strategy can be defined as a sequence $\sigma = \sigma_1 \& \sigma_2 \& \ldots \& \sigma_n$, where σ_i denotes an atomic component of the form:

- ϵ, an analogue to the generic *skip* instruction; ϵ is generally used to denote an *empty* execution strategy;
- r, a rule from the set R_t (the set of rules associated with type t). If r is applicable, then it is executed, advancing towards the next rule in the succession; otherwise, the compartment terminates the execution thread for this particular computational step and thus, no further rule will be applied;
- $(r_1, \ldots, r_n)$, with $r_i \in R_t, 1 \leq i \leq n$ symbolizes a non-deterministic choice within a set of rules. One and only one applicable rule will be executed if such a rule exists, otherwise the atom is simply skipped. In other words the non-deterministic choice block is always applicable;
- $(r_1, \ldots, r_n)^*$, with $r_i \in R_t, 1 \leq i \leq n$ indicates the arbitrary execution of a set of rules in R_t. The group can execute zero or more times, arbitrarily but also depending on the applicability of the constituent rules;
- $(r_1, \ldots, r_n)^\top, r_i \in R_t, 1 \leq i \leq n$ represents the maximally parallel execution of a set of rules. If no rules are applicable, then execution proceeds to the subsequent atom in the chain.

The execution strategy itself is a notable asset in defining more complex behaviour at the compartment level. For instance, weak priorities can be easily expressed as sequences of maximally parallel execution blocks: $(r_1)^\top \& (r_2)^\top \& \ldots \& (r_3)^\top$ or non-deterministic choice groups if single execution is required. Together with composite guards, they provide an unprecedented modelling fluency and plasticity for membrane systems. Whether such macro-like concepts and structures are preferred over traditional modelling with simple but numerous compartments in complex arrangements is a debatable aspect.

The kP system models are described in a machine readable language, called *kP–Lingua* [10]. Below, we illustrate the kP systems concepts on an example, which is slightly adjusted from [10, 2].

Example 1. A type definition in kP–Lingua.

```
type C1 {
    choice {
        > 2b : 2b -> b, a(C2) .
         b ->  2b .
    }
  }
  type C2 {
    choice {
      a -> a, {b, 2c}(C1) .
    }
 }
m1 {2x, b} (C1) - m2 {x} (C2) .
```

Above, `C1, C2` denote two compartment types, which are instantiated as `m1, m2`, respectively. `m1` starts with the initial multiset `2x, b` and `m2` starts with `x`. The rules of `C1` are chosen non-deterministically, only one at a time – this is achieved by the use of the key word `choice`. The first rule is fired only when its guard becomes true; in other words, only when the current multiset has at least three `b`'s. This rule also sends an `a` to the instance of `C2` that is linked. In the type `C2`, there is only one rule to be fired, which happens only when there is an `a` in the compartment `C1`.

3 kPWorkbench

kPWorkbench is an integrated software suit developed to provide a tool support for kP systems. kPWorkbench employs a set of tools and methods, allowing one to model membrane systems and to analyse them through *simulation* and *verification*. In the following, we briefly discuss some features of the software framework.

3.1 Features

Modeling.

kPWorkbench accepts kP system models specified in an intuitive modelling language, kP–Lingua. kP systems accumulate the most important aspects of various P system variants, so kP–Lingua provides a generic language to model various membrane systems. kPWorkbench features a graphical model editor, permitting to create new model files and editing existing files.

The grammar of the kP–Lingua language is written in ANTLR (ANother Tool for Language Recognition) [1], automatically generating the necessary syntactic and semantic analysers. ANTLR also constructs the data structures that represent the corresponding *abstract syntax tree* (AST) together with a traversing functionality.

Simulation.

kPWorkbench offers two different approaches to simulate kP systems. In both approaches, a kP–Lingua model is provided as an input, and the execution traces of the model are returned as an output. These traces permit exploring the dynamics of the system and observing how the system evolves over time.

In the first approach, we have developed a custom simulation tool [3], which recreates the system dynamics as a set of simulation runs. The tool translates a kP–Lingua specification into an internal data structure, which permits representing compartments, containing multisets of objects and rules, and their connections with other compartments.

In the second approach, we have integrated the Flame simulator [7], a general purpose large scale agent based simulation environment. Flame is based on the X-machine formalism [17], a type of extended finite state machine whose transitions

Prop. Pattern	Lang. Construct	LTL formula	CTL formula
Next	next p	X p	EX p
Existence	eventually p	F p	EF p
Absence	never p	$\neg$(F p)	$\neg$(EF p)
Universality	always p	G p	AG p
Recurrence	infinitely-often p	G F p	AG EF p
Steady-State	steady-state p	F G p	AF AG p
Until	p until q	p U q	A (p U q)
Response	p followed-by q	G ($p \rightarrow$ F q)	AG ($p \rightarrow$ EF q)
Precedence	p preceded-by q	$\neg(\neg p$ U $(\neg p \wedge q))$	$\neg$(E $(\neg p$ U $(\neg p \wedge q)))$

Table 1: Some property patterns defined in kP–Queries and the LTL and CTL translations. Note that LTL implicitly quantifies *universally* over paths (i.e. "necessity"). To complement this semantics, in CTL we translate some formulas by assuming quantification over *some* paths (i.e. "possibility").

are labelled by processing functions that operate on a (possibly infinite) set called memory, that models the system data. FLAME has been successfully used in various applications, ranging from biology to macroeconomics.

In order to simulate kernel P system models using the FLAME framework, an automated model translation has been implemented for converting the kP–Lingua specification into communicating X-machines [17]. One of the main advantages of this approach is the high scalability degree and efficiency for simulating large scale models.

Verification.

Although there have been some efforts to apply formal verification, in particular model checking, methods and methodologies for various P systems (e.g., [19, 4]), utilising a comprehensive, integrated and automated verification approach is a very challenging task in the context of membrane computing. For example, it is very difficult to transform some complex features, e.g. membrane division, dissolution and link creation/destruction, into suitable abstractions in model checking tools.

We have successfully addressed these issues, and developed a verification environment [10, 2] for kPWORKBENCH, integrating some state of the art model checking tools, e.g. the SPIN [18] and NUSMV [5]. The translations from a kP–Lingua representation to the corresponding SPIN and NUSMV inputs (i.e. PROMELA and SMV, respectively) are automatically performed.

In order to facilitate the property specification task, kPWORKBENCH features a property language, *kP–Queries*, based on *natural language* statements. The language also provides a list of property patterns (templates), generated from most commonly used queries (see Table 1). The property language permits specifying the target logic (i.e. LTL and CTL) for different properties without placing a requirement on a specific model checker. In this way, we can use the same set of properties in various verification experiments.

3.2 System architecture

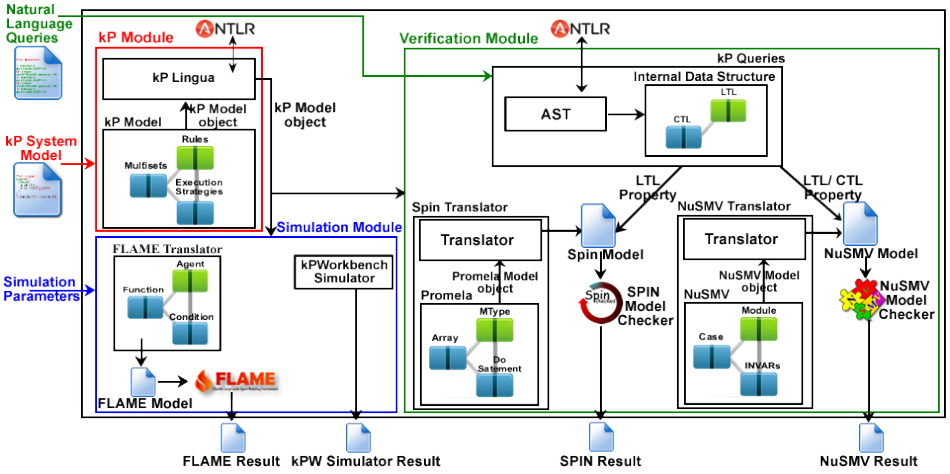


Fig. 1: The overview architecture of kPWorkbench framework

Figure 1 depicts an overview of the kPWorkbench system architecture, which consists of three modules:

1. The **kernel P (kP)** module takes a kP system model specified in kP–Lingua, which can be created or edited using a dedicated model editor, as input. The *kP–Lingua* module parses the input file and validates its syntax via ANTLR (which generates the necessary syntactic and semantic analysers). The *kP–Model* module accommodates the corresponding data structures of the input model, comprising compartment types, execution strategies, rules, multiset of objects and connections between compartments. The kP–Lingua module instantiates a kP–Model object and maps the AST generated by ANTLR to that object. This object is used as Data Transfer Object (DTO) between different modules of the framework. This separation helps developers to easily add new components to the framework.

2. The **Simulation** module consists of two components, kPWorkbench Simulator and Flame Translator. Both require the kP–Model object and simulator parameters, e.g. number of steps, as input. The kPWorkbench Simulator component is a custom simulator, which processes the multisets of objects of the input model with respect to its execution strategies and rules. The Flame Translator transforms the kP–Model object into a Flame Model object that aggregates agent, function, input, condition and output classes. It assigns each compartment to an agent, and the rules and the multiset of objects are stored as agent data. It creates a specific function for each type of execution strategy. In addition it creates C functions that represent the system behaviour (they are executed by Flame

when the agent makes a transition from one state to another). The FLAME Translator uses the ANTLR template group feature to produce the FLAME simulator specifications from the FLAME Model object.

3. The **Verification** module contains three components: the SPIN and NUSMV translators and the *kP–Queries* module:

The SPIN Translator has two main components: *Translator* and *Promela* (SPIN's specification language). The Promela component aggregates the Promela language specifications: *MType*, *Array*, *Do statement*, *If statement*, *Init*, etc. The Translator maps the kP–Model object to a *Promela* object using the following procedure [10]: (i) A compartment type is translated into a data type definition with the multiset of objects and links to other compartments, and also with temporary storage variables that provide the parallelism of P systems. (ii) Multiset of objects is assigned to an integer array where an index denotes the object and its value represents the multiplicity of the object. (iii) The set of rules are organised according to the execution strategies mapped by a *Proctype* definition – a Promela process. (iv) Maximal parallelism and arbitrary execution strategies are mapped to the *Do* statement, and choice execution strategy is mapped to *If* statement.

After the mapping process, the *Translator* component translates the Promela object to the corresponding Promela model, used by the SPIN model checker. However, this translation is not simple and straightforward, especially the structure changing rules, and arbitrary and maximal parallelism execution strategies complicate the translation process. More details about the translation from kP System model to the SPIN model checker specification can be found in [10].

Similarly, the NUSMV Translator translates the kP–Model object to the corresponding NUSMV representation (NUSMV's specification language). The translator has two main components: *Translator* and NUSMV. The NUSMV component consists of subcomponents representing the NUSMV language objects, such as *module*, *variables*, *INVARs*, *Case Statements*, *Conditions*, and *logical connectives*. The *Translator* maps the kP–Model object to the NUSMV object as follows: (i) Each compartment is translated into a module. (ii) The content of compartments is translated into variables. (iii) The initial multisets of the compartment are assigned into module parameters. (iv) Rules and guards are translated into the case statements. (v) The behaviour of execution strategies and the parallelism of P systems are achieved by introducing custom variables.

After the mapping process, the Translator component generates the NUSMV model from the NUSMV object, which is then provided as input to the NUSMV model checker. During the mapping process, we have overcome a few challenging domain specific restrictions. For example, unlike Promela, NUSMV has restrictions on defining arrays, and only allows accessing a value of array by a symbolic constant index; but it does not allow assigning a value by a symbolic constant. Therefore, instead of using arrays, we created a variable for each multiset of objects. Also, in Promela, we can non-deterministically pick a true statement among branches when there are more than one true statements; whereas, in NUSMV the selections are only deterministic. It always chooses the first true statement from a list of

conditions. We overcome that issue by introducing an *INVAR* declaration whenever a non-determinism behaviour is required.

The *kP–Queries* module receives a property, natural language based statements, as input. The user can build properties from the property language editor. The editor interacts with the kP–Lingua model, and permits accessing the native model elements, which simplifies the property building process. The kP–Queries' domain language has its own grammar, which is independent from and much simpler than the target model checking languages. The DSL (domain specific language) of the property language is written in ANTLR, receiving the EBNF grammar as input and generates the corresponding syntactic and semantic analysers as well as the corresponding AST. In order to simplify the traversal of the AST, we adapt a strategy, which maps the AST to a better structured internal data representation. To traverse between the elements of the internal data structure (a tree-like hierarchy), we follow the *Visitor design pattern*. Namely, the internal data nodes are treated as visitable entities, which are able to accept visitors and request to visit them. Each visitor has a specific functionality for visiting every single node. The visitor design pattern approach enables the kP–Queries module to translate every node of the internal presentation of property into the target model checker's corresponding property specification language.

4 Applications

Although membrane computing is mainly inspired from biology, its application to biological systems has been very limited due to the lack of a coherent and well-defined framework that allows us to analyse, verify and validate these systems. The methods and methodologies we have developed in [15, 10, 3, 2] to tackle these issues have filled an important gap in this respect. kPWorkbench, implementing these methodologies and algorithms, now provides a fully automated tool support, facilitating the modelling and analysis of biological systems through simulation and verification.

The usability and novelty of our approach has already been illustrated in some well-known case studies, chosen from systems and synthetic biology. In [21], we showed how our approach utilises the non-deterministic analysis of two biological systems, the quorum sensing in *P. aeruginosas* (a bacterial pathogen) and the synthetic pulse generator. Namely, we used our approach to observe various phenomena in genetic regulatory networks, e.g. various interactions between molecular species and various dependencies between molecules. Likewise, in [20], we showed how our approach can be used to formally analyse unconventional programs, e.g. some genetic Boolean gates.

We believe that our methods and techniques, and hence the kPWorkbench platform, provide significant contributions to the membrane & unconventional computing communities.

Prop.	Pattern	(i) Informal, (ii) Formal, (iii) Spin (LTL) Representations
1	Universality	**(i)** *No more than one termination signal will be generated*
		(ii) always m.t <= 1
		(iii) `ltl prop { [] (c[0].x[t_] <= 1 || state != step_complete) }`
2	Absence	**(i)** *The system will never generate 15 as a square number*
		(ii) never m.s = 15
		(iii) `ltl prop { !(<> (c[0].x[s_] == 15 && state == step_complete)) }`
3	Steady-state	**(i)** *In the long run, the system will converge to a state in which, if the termination signal is generated, no more a objects will be available*
		(ii) steady-state (m.a = 0 implies m.t = 1)
		(iii) `ltl prop { <> ([] ((c[0].x[a_] == 0 -> c[0].x[t_] == 1) ||`
		`state != step_complete) && state != step_complete) }`

Prop.	Pattern	(i) Informal, (ii) Formal, (iii) NuSMV (CTL) Representations
4	Existence	**(i)** *The system will eventually consume all a objects, on some runs*
		(ii) eventually m.a = 0
		(iii) `SPEC EF m.a = 0`
5	Existence	**(i)** *On some runs the system will eventually halt*
		(ii) eventually m.t = 1
		(iii) `SPEC EF m.t = 1`
6	Universality	**(i)** *No more than one termination signal will be generated*
		(ii) always m.t <= 1
		(iii) `SPEC AG m.t <= 1`
7	Absence	**(i)** *The system will never generate 15 as a square number*
		(ii) never m.s = 15
		(iii) `SPEC !(EF m.s = 15)`
8	Precedence	**(i)** *The consumption of all a objects will always be preceded by a halting signal*
		(ii) m.a = 0 **preceded-by** m.t = 1
		(iii) `SPEC !(E [!(m.a = 0) U (!(m.a = 0) & m.t = 1)])`
9	Response	**(i)** *By starting the computation with at least one a object, on some runs the system will eventually consume all of them*
		(ii) m.a >0 followed-by m.a = 0
		(iii) `SPEC AG (m.a > 0 -> EF m.a = 0)`
10	Response	**(i)** *A halting signal will always be followed by the consumption of all a objects*
		(ii) m.t = 1 followed-by m.a = 0
		(iii) `SPEC AG (m.t = 1 -> EF m.a = 0)`

Table 2: List of properties derived from the property language and their representations in different formats.

5 Examples

5.1 Generating square numbers

We present below a kernel P systems model that generates square numbers (starting with 1) each step. The multiplicity of object "s" is equal to the square number produced each step.

```
type main {
max {
= t: a -> {} .
< t: a -> a, 2b, s .
< t: a -> a, s, t .
< t: b -> b, s .
```

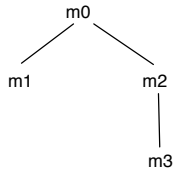


Fig. 2: The structure.

```
    }
}

m {a} (main) .
```

An execution trace for this model can be visualised as follows:

```
a
a 2b s
a 4b 4s
a 6b 9s
...
```

kPWorkbench automatically converts the kP-Lingua model into the corresponding input languages of the Spin, and NuSMV model checkers. In order to verify that the problem works as desired, we have constructed a set of properties specified in kP-Queries, listed in Table 2. The applied pattern types are given in the second column of the table. For each property we provide the following information; **(i)** informal description of each kP-Query, **(ii)** the formal kP-Query, **(iii)** the translated form of the kP-Query into the LTL specifications written in Spin modelling language, and CTL specifications written in the NuSMV language. The results of all queries are positive, as expected.

5.2 Broadcasting with acknowledgement

In this case study, we consider broadcasting with acknowledgement in ad-hoc networks. Each level of nodes in the hierarchy has associated a unique type with communication rules to neighbouring (lower and upper) levels. This is the only way we can simulate signalling with kP systems such that we do not hard-wire the target membranes in communication rules, i.e. assume we do not know how many child-nodes are connected to each parent as long as we group them by the same type; evidently, this only applies to tree structures. The kP Systems model written in kP–Lingua is given as follows:

```
type L0 {
max {
a -> b, a (L1), a (L2) .
    }
}
```

```
type L1 {
max {
a, c -> c (L0) .
    }
}

type L2 {
max {
a -> b, a (L3) .
b, c -> c (L0) .
    }
}

type L3 {
max {
a, c -> c (L2) .
    }
}

m0 {a} (L0) .

m1 {c} (L1) - m0 .
m2 {} (L2) - m0.
m3 {c} (L3) - m2 .
```

In order to verify that the model works as desired, we have verified some properties, presented in Table 3. The results are positive, except Properties 1 and 5, as expected. These results confirm the desired system behaviour.

6 Conclusion

We have presented the kPWorkbench toolset developed to support kernel P systems. kPWorkbench integrates several simulation and verification tools and methods and permits modelling and analysis of membrane systems. It also features a property language based on natural language statements to facilitate property specification. These features make kPWorkbench the only available integrated toolset permitting non-deterministic analysis (through simulation and verification) of membrane systems.

We are planning to work on more case studies from different fields, e.g., systems & synthetic biology, engineering and economics.

Acknowledgements. The work of FI and LM was supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI (project number: PN-II-ID-PCE-2011-3-0688). MG and SK acknowledge the support provided for synthetic biology research by EPSRC ROADBLOCK (project number: EP/I031812/1).

Prop.	Pattern	(i) Informal, (ii) Formal, (iii) Spin (LTL) Representations
3	Existence	**(i)** *The terminal nodes will receive the broadcast message at the same time*
		(ii) eventually (m1.a >0 and m3.a >0)
		(iii) `ltl prop { <> ((c[0].x[a_] > 0 && c[0].x[a_] > 0) && state == step_complete) }`
3	Absence	**(i)** *The root node will never receive an acknowledgement without sending a broadcast*
		(ii) never m0.a >0 and m0.c >0
		(iii) `ltl prop { !(<> ((c[0].x[a_] > 0 && c[0].x[c_] > 0) && state == step_complete)) }`
3	Response	**(i)** *The node m2 will always receive broadcast message before its child node (m3)*
		(ii) m2.a = 1 **followed-by** m3.a = 1
		(iii) `ltl prop { [] ((c[0].x[a_] == 1 -> <> (c[0].x[a_] == 1 && state == step_complete)) || state != step_complete) }`

Prop.	Pattern	(i) Informal, (ii) Formal, (iii) NuSMV (CTL) Representations
4	Existence	**(i)** *The node m1 will eventually receive the broadcast message*
		(ii) eventually m1.a >0
		(iii) `SPEC EF m1.a > 0`
5	Existence	**(i)** *The terminal nodes will receive the broadcast message at the same time*
		(ii) eventually m1.a >0 and m3.a >0
		(iii) `SPEC EF (m1.a > 0 & m3.a > 0)`
6	Absence	**(i)** *The root node will never receive an acknowledgement without sending a broadcast*
		(ii) never m0.a >0 and m0.c >0
		(iii) `SPEC !(EF (m0.a > 0 & m0.c > 0))`
7	Response	**(i)** *The node m2 will always receive the broadcast message before its child node (m3)*
		(ii) m2.a = 1 **followed-by** m3.a = 1
		(iii) `SPEC AG (m2.a = 1 -> EF m3.a = 1)`
9	Steady-state	**(i)** *In the long run, the system will converge to a state in which the root node will have been received the acknowledgement from all the terminal nodes and no more broadcasts will occur*
		(ii) steady-state (m0.c = 2 implies m0.a = 0)
		(iii) `SPEC AF (AG (m0.c = 2 -> m0.a = 0))`
9	Steady-state	**(i)** *In the long run, the system will converge to a state in which the root node will have been received the acknowledgement from all the terminal nodes and no more acknowledgements will occur*
		(ii) steady-state (m0.c = 2 implies (m1.c = 0 and m3.c = 0))
		(iii) `SPEC AF (AG (m0.c = 2 -> (m1.c = 0 & m3.c = 0)))`

Table 3: List of properties derived from the property language and their representations in different formats.

References

1. ANTLR website, url: `http://www.antlr.org`
2. Bakir, M.E., Ipate, F., Konur, S., Mierlă, L., Niculescu, I.: Extended simulation and verification platform for kernel P systems. In: 15th International Conference on Membrane Computing. LNCS, vol. 8961, pp. 158–168. Springer (2014)
3. Bakir, M.E., Konur, S., Gheorghe, M., Niculescu, I., Ipate, F.: High performance simulations of kernel P systems. In: Proceedings of the 2014 IEEE 16th International Conference on High Performance Computing and Communication. pp. 409–412. HPCC '14, Paris, France (2014)
4. Blakes, J., Twycross, J., Konur, S., Romero-Campero, F., Krasnogor, N., Gheorghe, M.: Infobiotics workbench: A P systems based tool for systems and synthetic biology. In: [11], pp. 1–41

5. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV version 2: An open source tool for symbolic model checking. In: Proc. International Conference on Computer-Aided Verification (CAV 2002). LNCS, vol. 2404, pp. 359–364. Springer, Copenhagen, Denmark (2002)
6. Ciobanu, G., Pérez-Jiménez, M.J., Păun, G. (eds.): Applications of Membrane Computing. Springer (2006)
7. Coakley, S., Gheorghe, M., Holcombe, M., Chin, S., Worth, D., Greenough, C.: Exploitation of high performance computing in the FLAME agent-based simulation framework. In: Proceedings of the IEEE 14th International Conference on High Performance Computing and Communication. pp. 538–545. HPCC '12, Liverpool, UK (2012)
8. Díaz-Pernil, D., Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J.: A uniform family of tissue P systems with cell division solving 3-COL in a linear time. Theoretical Computer Science 404, 76–87 (2008)
9. Dinneen, M.J., Yun-Bum, K., Nicolescu, R.: Faster synchronization in P systems. Natural Computing 11(4), 637–651 (2012)
10. Dragomir, C., Ipate, F., Konur, S., Lefticaru, R., Mierlă, L.: Model checking kernel P systems. In: 14th International Conference on Membrane Computing. LNCS, vol. 8340, pp. 151–172. Springer (2013)
11. Frisco, P., Gheorghe, M., Pérez-Jiménez, M.J. (eds.): Applications of Membrane Computing in Systems and Synthetic Biology. Springer (2014)
12. Gheorghe, M., Ipate, F., Dragomir, C., Mierlă, L., Valencia-Cabrera, L., García-Quismondo, M., Pérez-Jiménez, M.J.: Kernel P systems - version 1. In: 11th Brainstorming Week on Membrane Computing, pp. 97–124. Fénix Editora (2013)
13. Gheorghe, M., Ipate, F., Lefticaru, R., Pérez-Jiménez, M.J., Ţurcanu, A., Valencia-Cabrera, L., García-Quismondo, M., Mierlă, L.: 3-Col problem modelling using simple kernel P systems. Int. Journal of Computer Mathematics 90(4), 816–830 (2012)
14. Gheorghe, M., Păun, G., Pérez-Jiménez, M.J., Rozenberg, G.: Research frontiers of membrane computing: Open problems and research topics. International Journal of Foundations of Computer Scence 24, 547–624 (2013)
15. Gheorghe, M., Ipate, F., Dragomir, C.: Kernel P systems. In: 10th Brainstorming Week on Membrane Computing, pp. 153–170. Fénix Editora (2012)
16. Gimel'farb, G.L., Nicolescu, R., Ragavan, S.: P system implementation of dynamic programming stereo. Journal of Mathematical Imaging and Vision 47(1–2), 13–26 (2013)
17. Holcombe, M.: X-machines as a basis for dynamic system specification. Softw. Eng. J. 3(2), 69–76 (1988)
18. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Soft. Eng. 23(5), 275–295 (1997)
19. Ipate, F., Lefticaru, R., Tudose, C.: Formal verification of P systems using Spin. International Journal of Foundations of Computer Science 22(1), 133–142 (2011)
20. Konur, S., Gheorghe, M., Dragomir, C., Ipate, F., Krasnogor, N.: Conventional verification for unconventional computing: a genetic XOR gate example. Fundamenta Informaticae 134(1-2), 97–110 (2014)
21. Konur, S., Gheorghe, M., Dragomir, C., Mierlă, L., Ipate, F., Krasnogor, N.: Qualitative and quantitative analysis of systems and synthetic biology constructs using P systems. ACS Synthetic Biology 4(1), 83–92 (2015)
22. Păun, G.: Computing with membranes. Journal of Computer and System Sciences 61(1), 108–143 (2000)

23. Păun, G., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
24. Rozenberg, G., Bäck, T., Kok, J.N. (eds.): Handbook of Natural Computing. Springer (2012)

The Pole Balancing Problem with Enzymatic Numerical P Systems

Domingo Llorente–Rivera[1], Miguel A. Gutiérrez–Naranjo[2]

[1] Department of Computer Science and Artificial Intelligence
University of Seville, Seville, Spain
domingo.llorente.rivera@gmail.com

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

Summary. Pole balancing is a control benchmark widely used in engineering. It involves a pole affixed to a cart via a joint which allows movement along a single axis. In this problem, the movement of the cart is restricted to the horizontal axis by a track and the pole is free to move about the horizontal axis of the pivot. The system is extremely unstable and, the cart must be in constant movement in order to preserve the equilibrium and avoid the fall of the pendulum.

In this paper, we study the pole balancing problem in the framework of Enzymatic Numerical P Systems and provide some clues for using them in more complex systems.

1 Introduction

Numerical P systems (NPS for short) were introduced in [7] with the aim of adding ideas from economic and business processes to the framework of Membrane Computing. They represent a break with respect to the previous P system models since they introduce the concept of *variable* and *real numbers* in the framework of Membrane Computing. In the general framework of Membrane Computing (called *symbolic* P systems, in order to stress the differences with *numerical* P systems), membranes can be seen as encapsulations of the Euclidean space where multisets of objects are placed. The computation in such devices is performed by the application of rules which send objects from one to other membrane (maybe modified) or modify the membrane structure (see [8]). In NPS, membranes do not contain multisets of objects. They contain variables with associated numerical values. These numerical values can be integer, rational or real numbers. Instead of using rules inspired in biochemical reactions, the computation of these new devices is performed by *programs* consisting of two parts: a *production function* and a *repartition protocol*. Production functions are real-valued functions of type $F : \mathbb{R}^k \to \mathbb{R}$ which take

the k variables which appear in the membrane where the program is defined and computes a real value. The computed number is then distributed among different variables according to the *repartition protocol*.

In spite of its undoubted potential as computational devices, in the literature there are very few papers devoted to this model (see, e.g., [1, 2, 3, 4, 5, 9, 10, 11] and references therein). Most of them devoted to *enzymatic* numerical P system (ENPS), a model introduced in [3] where enzymatic-like variables are introduced in the NPS in order to avoid the non-determinism in the choice of a program in a membrane.

Although the original inspiration of numerical P system was the economic processes, the main field of the applications has been control problems. These problems are on the basis of many industrial processes and the design of software controllers for more and more sophisticated devices is nowadays a challenge for researchers. The household thermostat is a classic example of control problem: provided the changing temperature outside, the thermostat must maintain the temperature inside home close to a desired level. This implies react to the changes in an unpredictable real-world providing an appropriate response in a short interval of time.

Beyond simple examples, the design of controllers for many real world is an extremely complex task. If we extend the thermostat example to a more general climate control system, a linear controller will not be able to regulate the temperature adequately.

Usually, the control system is a software program that takes the right decision for the input. For this input-output interaction, the software receives an input from the sensor and takes a decision as output. It is crucial for the final solution to obtain a real-time response in less than 10 milliseconds. For this reason, the control software must be as small as possible in order to obtain a quick response.

In this paper we go on with the study of NPS as devices for control problem (see, e.g. [2, 4]). As pointed out by Gh. Păun in [6], controlling drones can be a good application for this model and it can be an extension of the use of NPS for 2D travelling robots found in the literature. *Drone* is the popular name for an unmanned aerial vehicle which can be seen as a mobile 3D robot. From a technical point of view, the main difference between the control of 2D travelling robots and drones is the stability. The drone must keep the horizontal position as much as possible regardless the air conditions. This implies the effective real-time control of the different engines according to the changes in the environment. The control of drones is nowadays a research field for the industry and it is a really hard task. In a certain sense, the stability problem of a drone can be seen as the generalization of a well-known problem in control, the *pole-balancing* problem.

The pole-balancing problem is a feedback control system with the desired behavior of balancing a pole (an inverted pendulum) that is connected to a motor driven cart by a ball-baring pivot (see Fig. 1). In this problem, the movement of the cart is restricted to the horizontal axis by a track, and the pole is free to move about the horizontal axis of the pivot. The system is extremely unstable and the

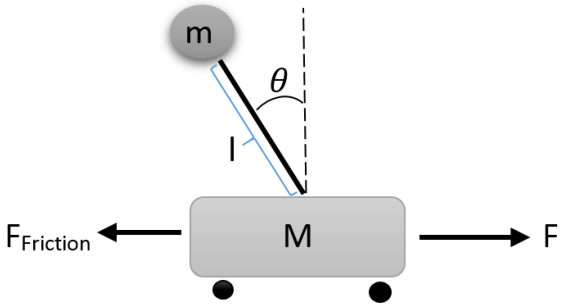


Fig. 1. Pole Balancing problem

cart must be in constant movement in order to preserve the equilibrium and avoid the fall of the pendulum. In a more general situation (a drone, by example) the movement of the device must be controlled in three degrees of freedom, but it is essentially the same problem, so the pole-balancing problem can be seen as a first approach.

In this paper, we provide a theoretical study of the pole-balancing problem in the framework of the ENPS and provide some ideas for further uses of ENPS in control problems. The paper is organized as follows: Firstly, a brief introduction to ENPS and to the Pole Balancing Problem is given. Next we provide some hints about how the problem can be dealt with ENPS and finally some conclusions and future work lines are presented.

2 Enzymatic Numerical P Systems

Next, we briefly recall the definition of enzymatic numerical P systems, More details can be found in [3]. An enzymatic numerical P system is formally expressed by:

$$\Pi = (m, H, \mu, (Var_1, Pr_1, Var_1(0)), \ldots, (Var_m, Pr_m, Var_m(0)))$$

where:

- m is the number of membranes used in the system (degree of Π) ($m \geq 1$);
- H is an alphabet that contains m symbols (the labels of the membranes);
- μ is a tree-like membrane structure;
- Var_i is a set of variables from membrane i, and the initial values for these variables are $Var_i(0)$, $i \in \{1, \ldots, m\}$;
- Pr_i is the set of programs from membrane i, $i \in \{1, \ldots, m\}$. Programs process variables and have one of the following forms:
  (a) Non-enzymatic form

$$Pr_{j,i} = F_{j,i}(x_{1,i}, \ldots, x_{k_i,i}) \rightarrow c_{j,1}|v_1 + \cdots + c_{j,n_i}|v_{n_i}$$

(b) Enzymatic form

$$Pr_{j,i} = F_{j,i}(x_{1,i}, \ldots, x_{k_i,i})(e_j \rightarrow) \rightarrow c_{j,1}|v_1 + \cdots + c_{j,n_i}|v_{n_i}$$

where $e_j \in Var_i$ is an enzyme-like variable which controls the activation of the rule.

Rules have two components, a *production function* and a *repartition protocol*. The l-th program of the membrane i has the following form:

$$Pr_{l,i} = (F_{l,i}, c_{l,1}|v_1 + \cdots + c_{l,n_i}|v_{n_i})$$

where $F_{l,i} : \mathbb{R}^{card(Var_i)} \rightarrow \mathbb{R}$ is a real-valued function such that computes a real number from the values of the variables in Var_i; $c_{l,1}, \ldots, c_{l,n_i}$ are natural numbers and $v_1, \ldots, v_{n_1}$ are the variables of the membrane i together with the variables from the immediately upper membrane, and those from the immediately lower membranes. If the corresponding c_i is 0, the expression $0|v_i$ is omitted.

If $card(Pri) = 1$ for $i \in \{1, \ldots, m\}$, then there is one production function per each membrane and the system is deterministic. In case of multiple programs per membrane, one rule is non-deterministically selected.

A universal clock is considered and, at each time t, all the variables have associated a value. The computation is performed by computing the new value of the variables. Such computation is performed in the following way. A rule is *active* if it is in the non enzymatic form or if the associated enzyme has a greater value than one of the variables involved in the production function. In parallel, in each membrane an active program is chosen and its production function is used in order to calculate a *production* from the value of the local variables. Once calculated, the repartition protocol is used in order to compute the proportion of such value that it is send to each variable. The coefficients $c_1 \ldots c_n$ in the repartition protocol $c_1|v_1 + \cdots + c_n|v_n$ specify the proportion of production distributed to each variable $v_1 \ldots v_n$. Namely, such protocol sends to the variable v_i the value

$$q_i = \frac{production \times c_i}{\sum_{i=j}^{n} c_j}$$

The new value of the variable is the addition of the contribution of each applied program. In each membrane of the system one uses one program at the time, and this happens in parallel in all membranes.

A variable x is called *productive* if it does appear in a production function, and then is *consumed* and reset to zero, otherwise the initial value is added to the received contributions. The values of the variables at next time step are computed by using repartition protocols, and so, portions distributed to variables are added to form the new value.

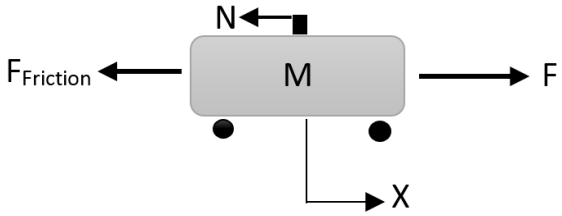


Fig. 2. Cart of the Pole Balancing

3 The Pole Balancing Problem

Pole balancing is an control benchmark historically used in engineering. It involves a pole affixed to a cart via a joint which allows movement along a single axis. The cart is able to move along a track of fixed length.

A trial typically begins with the pole off-center by a certain number of degrees. The goal is to keep the pole from falling over by moving the cart in either direction, without falling off either edge of the track. The controller receives as input information about the system at each time step, such as the positions of the poles, their respective velocities, the position and velocity of the cart, etc. An even more difficult extension of this problem involves a cart which can move in a three dimensional space via three or more engines. In such situation the target is not keeping a pole in a vertical position but keeping the cart as horizontal as possible. In this paper we do not consider such generalization and focus on the simple pole balancing problem.

The pole balancing problem can be analysed as the conjunction of two models: focusing on the cart (see Fig. 2) and focusing on the bar (see Fig. 3). Obviously, the applied force over one of these models results in the modification of the state of the other model. In the first model (Fig. 2) several parameters must be considered: F, force for controlling the system; $F_{Friction}$, force of the friction of the cart in its movement on the railway; M, mass of the cart; N, force of the pole over the cart. The second model focus on the bar of the pole balancing (Fig. 3), where θ is the angle of the bar with respect to the vertical, l is the length of the bar and m is the mass of the ball placed on the top of the bar. For the control of the pole balancing, the control software (the NPS in our study) has to know the current state of the pole, (x, θ) and ($\dot{x}$, $\ddot{x}$, $\dot{\theta}$, $\ddot{\theta}$), where x represents the *position* of the cart, and $\dot{x}$, $\ddot{x}$ the *speed* and *acceleration* respectively. The angle θ represents the angle of the bar with respect to the vertical position and $\dot{\theta}$, $\ddot{\theta}$ the *angular* speed and acceleration (resp).

The equations that define this system are:

$$F = M\ddot{x} + b\dot{x} + N \tag{1}$$

$$N = m\ddot{x} + ml\ddot{\theta}\cos\theta - ml\dot{\theta}^2\sin\theta \tag{2}$$

The system of control is represented by the equation (3), which is the result of adding the equations (1) and (2), where F is the output for the system of control and the force that the controller has to apply to the system, and b is the friction of the cart.

$$F = (M+m)\ddot{x} + b\dot{x} + ml\ddot{\theta}\cos\theta - ml\dot{\theta}^2\sin\theta \tag{3}$$

For computing $\cos\theta$ and $\sin\theta$ using ENPS, we use the same idea proposed in [5] where the functions are approximated by using their analytic expressions as infinite sums shown in equations (4) and (5). There approaches will be calculated in the designed ENPS by the membranes *Cosine* and *Sine*, respectively.

$$cos(x) = \sum_{0}^{\infty}(-1)^n \frac{x^{2n}}{(2n)!} \tag{4}$$

$$sin(x) = \sum_{0}^{\infty}(-1)^n \frac{x^{(2n+1)}}{(2n+1)!} \tag{5}$$

The analytic expression of the *cosine* can be written as

$$cos(x) = \sum_{n=0}^{\infty} ac_n$$

where $ac_0 = 1$ and ac_n is recursively obtained as follows:

$$ac_{n+1} = (-ac_n) \times \frac{\theta^2}{(2n)(2n-1)}$$

Analogously, the analytic expression of the *sine* can be written as

$$sin(x) = \sum_{0}^{\infty} as_n$$

where $as_0 = 1$ and as_n is computed as

$$as_{n+1} = (-as_n) \times \frac{\theta^2}{(2n)(2n+1)}$$

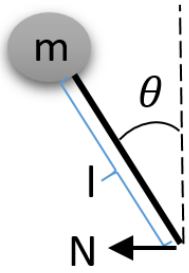


Fig. 3. Bar of the Pole Balancing

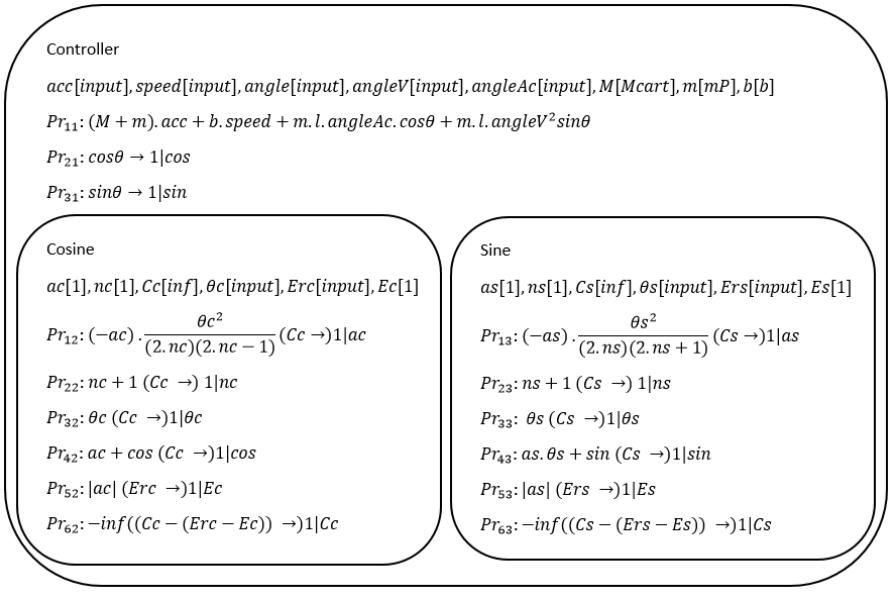


Fig. 4. ENPS membrane applied system for control pole balancing

4 ENPS Applied to the Pole Balancing Problem

In this section, we report a work-in-progress on the design of an ENPS as a software solution for the control of the pole balancing problem. To this aim, the different forces that affect the system are examined and the interaction among them are computed as a flow of information between the variables of the ENPS. The basic schema is chown in Fig. 3.

The membrane system shown in Fig. 4 is proposed as a preliminar solution for the pole balancing problem, using three membranes: the first membrane *Controller* calculates the necessary force in order to keep the vertical position; the membranes *Cosine* and *Sine* calculate the cos and sin functions for the angle θ. The ENPS can be considered as a software module which receives as input the data $\dot{x}$, $\ddot{x}$, θ, $\dot{\theta}$, $\ddot{\theta}$ and outputs the force F for controlling the system.

The control of the pole balancing is calculated by the rule Pr_{11} which encodes the Equation 3. This rule needs the constants: M, the cart mass; m, the mass of the ball; and l, the length of the bar. It takes as input the state of the system, encoded in the variables: *acc*, acceleration of the cart ($\ddot{x}$); *speed*, velocity of the cart ($\dot{x}$); *angleSpeed* ($\dot{\theta}$) and *angleAcc*, ($\ddot{\theta}$) angle speed and acceleration. In order to approximate $\cos\theta$ and $\sin\theta$ from θ, the *Controller* membrane uses the rules Pr_{21} and Pr_{31}. The *cosine* and *sine* are computed recursively by the rules Pr_{12} for the *cosine* and Pr_{13} for the *sine*, until the current errors, Ec for the *cosine* and Es for the *sine*, are less than Erc and Ers respectively as it is proposed in

[12].Finally, the system returns the control related to equation 3 with the $\cos\theta$ and $\sin\theta$ calculated previously.

Membranes *Cosine* and *Sine* approximate the *cos* and *sin* functions by using the analytic expressions from Eq. (4) and (5). These membranes return *cos* by the rule Pr_{12} and *sin* by the rule Pr_{13}, where the system adds the result for each one in *cos* and *sin*. The membranes stop when the current error is less than the errors provided as parameters, Erc and Ers. The system stop is controlled by rule Pr_{62} for the cosine and Pr_{63} for the sine as the current error is lower than the parameter Erc for the cosine membrane and Ers for the sine membrane.

The following trace shows how the system from Fig. 4 should work:

- Membrane Controller:
  - The input of the system $\dot{x}_0$, $\ddot{x}_0$, θ_0, $\dot{\theta}_0$, $\ddot{\theta}_0$ are the values of the corresponding variables in the initial configuration. We also consider two variables *cosApp* and *sinApp* where the approximated values of the *cos* and *sin* functions will be stored.
  - Production Function:
    - $F_1 = (M+m)\ddot{x} + b\dot{x} + ml\ddot{\theta}\cos\theta - ml\dot{\theta}^2\sin\theta$;
    - $F_2 = \cos\theta$;
    - $F_3 = \sin\theta$;
- Membrane Cosine:
  - Variables: ac has an initial value of 1, nc has an initial value of 1, Cc has an initial value of ∞, Ec has an initial value of 1;
  - Production function:
    - $F_4 = (-ac) \times \frac{\theta^2}{(2nc)(2nc-1)}$;
    - $F_5 = nc + 1$;
    - $F_6 = \theta$;
    - $F_7 = ac + cos$;
    - $F_8 = |ac|$;
    - $F_9 = -\infty$;
  - Reparation protocol: ac receives 1 ($C_{21} = 1$), nc receives 1 ($C_{22} = 1$), cos receives 1 ($C_{23} = 1$), Cc receives 1 ($C_{24} = 1$), Ec receives 1 ($C_{25} = 1$);
- Membrane Sine:
  - Variables: as has an initial value of 1, ns has an initial value of 1, Cs has an initial value of ∞, Es has an initial value of 1;
  - Production function:
    - $F_{10} = (-as) \times \frac{\theta^2}{(2ns)(2ns+1)}$;
    - $F_{11} = ns + 1$;
    - $F_{12} = \theta$;
    - $F_{13} = as * \theta$;
    - $F_{14} = |as|$;
    - $F_{15} = -\infty$;
  - Reparation protocol: as receives 1 ($C_{31} = 1$), ns receives 1 ($C_{32} = 1$), sin receives 1 ($C_{33} = 1$), Cs receive ∞ ($C_{34} = 1$), Es receive 1 ($C_{35} = 1$);

- Step 1
  - Membrane Cosine:
    - $ac_{21} = 1$, $nc_{22} = 1$,$cos_{23} = 0$, $\theta = 1$, $Cc = \infty$, $Ec = 1$, $Erc = 0.0001$;
    - Compute productions function's value:
      - $F_4 = (-ac_{21}) \times \frac{\theta^2}{(2nc_{22})(2nc_{22}-1)} \Rightarrow F_4 = -\frac{1}{2}$;
      - $F_5 = nc_{22} + 1 \Rightarrow F_5 = 2$;
      - $F_6 = \theta \Rightarrow F_6 = 1$;
      - $F_7 = ac_{21} + cos_{23} \Rightarrow F_7 = 1$;
      - $F_8 = ac_{21} \Rightarrow F_8 = 1$;
      - F_9 is not executed, because $Cc - (Erc - Ec) = \infty - (0.0001 - 1) = \infty + 1$ is not bigger than Cc;
    - Compute 'unitary portion':
      - $q_4 = F_4/C_{21} \Rightarrow ac_{21} = -\frac{1}{2}$;
      - $q_5 = F_5/C_{22} \Rightarrow nc_{22} = 2$;
      - $q_6 = F_6/\theta \Rightarrow \Theta = 1$;
      - $q_7 = F_7/C_{23} \Rightarrow cos_{23} = 1$;
      - $q_8 = F_8/C_{25} \Rightarrow Ec = 1$;
  - Membrane Sine:
    - $as_{31} = 1$, $ns_{32} = 1$,$sin_{33} = 0$, $\theta = 1$, $Cs = \infty$, $Es = 1$, $Ers = 0.0001$;
    - Compute productions function's value:
      - $F_{10} = (-as_{31}) \times \frac{\theta^2}{(2ns_{32})(2ns_{32}+1)} \Rightarrow F_8 = -\frac{1}{6}$;
      - $F_{11} = ns_{32} + 1 \Rightarrow F_9 = 2$;
      - $F_{12} = \theta \Rightarrow F_{10} = 1$;
      - $F_{13} = as_{31} + sin_{33} \Rightarrow F_{11} = 1$;
      - $F_{14} = |as_{31}| \Rightarrow F_{14} = 1$;
      - F_{15} is not executed, because $Cs - (Ers - Es) = \infty - (0.0001 - 1) = \infty + 1$ is not bigger than Cs;
    - Compute 'unitary portion':
      - $q_{10} = F_{10}/C_{31} \Rightarrow as_{31} = -\frac{1}{6}$;
      - $q_{11} = F_{11}/C_{32} \Rightarrow ns_{32} = 2$;
      - $q_{12} = F_{12}/\theta \Rightarrow \Theta = 1$;
      - $q_{13} = F_{13}/C_{33} \Rightarrow sin_{33} = 1$;
      - $q_{14} = F_{14}/C_{35} \Rightarrow Es = 1$;
- Step 2:
  - Membrane Cosine:
    - $ac_{21} = -\frac{1}{2}$, $nc_{22} = 2$,$cos_{23} = 1$, $\theta = 1$, $Cc = \infty$, $Ec = 1$, $Erc = 0.0001$;
    - Compute productions function's value:
      - $F_4 = (-ac_{21}) \times \frac{\theta^2}{(2nc_{22})(2nc_{22}-1)} \Rightarrow F_4 = \frac{1}{24}$;
      - $F_5 = nc_{22} + 1 \Rightarrow F_5 = 3$;
      - $F_6 = \theta \Rightarrow F_6 = 1$;
      - $F_7 = ac_{21} + cos_{23} \Rightarrow F_7 = -\frac{1}{2}$;
      - $F_8 = ac_{21} \Rightarrow F_8 = \frac{1}{2}$;
      - F_9 is not executed, because $Cc - (Erc - Ec) = \infty - (0.0001 - 1) = \infty + 1$ is not bigger than Cc;

  - Compute 'unitary portion':
    - $q_4 = F_4/C_{23} \Rightarrow ac_{23} = \frac{1}{24}$;
    - $q_5 = F_5/C_{22} \Rightarrow nc_{22} = 3$;
    - $q_6 = F_6/\theta \Rightarrow \Theta = 1$;
    - $q_7 = F_7/C_{23} \Rightarrow cos_{23} = \frac{1}{2}$;
    - $q_8 = F_8/C_{25} \Rightarrow Ec = \frac{1}{2}$;
- Membrane Sine:
  - $as_{31} = -\frac{1}{6}$, $ns_{32} = 2$,$sin_{33} = 1$, $\theta = 1$, $Cs = \infty$, $Es = 1$, $Ers = 0.0001$;
  - Compute productions function's value:
    - $F_{10} = (-as_{31}) \times \frac{\theta^2}{(2ns_{32})(2ns_{32}+1)} \Rightarrow F_8 = \frac{1}{120}$;
    - $F_{11} = ns_{32} + 1 \Rightarrow F_9 = 3$;
    - $F_{12} = \theta \Rightarrow F_{10} = 1$;
    - $F_{13} = as_{31} + ns_{33} \Rightarrow F_{11} = 1 - \frac{1}{6} = \frac{5}{6}$;
    - $F_{14} = |as_{31}| \Rightarrow F_{14} = \frac{1}{6}$;
    - F_{15} is not executed, because $Cs - (Ers - Es) = \infty - (0.0001 - 1) = \infty + 1$ is not bigger than Cs;
  - Compute 'unitary portion':
    - $q_{10} = F_{10}/C_{33} \Rightarrow as_{33} = \frac{1}{120}$;
    - $q_{11} = F_{11}/C_{32} \Rightarrow ns_{32} = 3$;
    - $q_{12} = F_{12}/\theta \Rightarrow \Theta = 1$;
    - $q_{13} = F_{13}/C_{33} \Rightarrow sin_{33} = \frac{5}{6}$;
    - $q_{14} = F_{14}/C_{35} \Rightarrow Es = \frac{1}{6}$;

- Step N-1:
  - Using the same reason for the membranes *Cosine* and *Sine*, both membranes are executed until error is less than Erc, $Ec < Erc$, for the Cosine and Ers, $Es < Ers$, for the Sine. Then the execution stops.
  - Membrane Controller:
    - $\cos\theta = 1$, $\sin\theta = 1$, $F[0]$;
    - Compute productions function's value:
      - $F_1 = (M+m)\ddot{x} + b\dot{x} + ml\ddot{\theta}\cos\theta - ml\dot{\theta}^2\sin\theta \Rightarrow F_1 = (M+m)\ddot{x} + b\dot{x} + ml\ddot{\theta} - ml\dot{\theta}^2$;
      - $F_2 = \cos\theta \Rightarrow F_2 = \cos$;
      - $F_3 = \sin\theta \Rightarrow F_3 = \sin$;
    - Compute 'unitary portion':
      - $q_1 = F_1/(C_{11} + C_{12}) \Rightarrow F_{13} = (M+m)\ddot{x} + b\dot{x} + ml\ddot{\theta} - ml\dot{\theta}^2$;
      - $q_2 = F_2/C_{11} \Rightarrow \cos\theta = \cos$;
      - $q_3 = F_3/C_{12} \Rightarrow \sin\Theta = \sin$;
- Step N:
  - Membrane Controller:
    - $\cos\theta = \cos$, $\sin\theta = \sin$, $F_{13} = (M+m)\ddot{x} + b\dot{x} + ml\ddot{\theta} - ml\dot{\theta}^2$;
    - Compute productions function's value:
      - $F_1 = (M+m)\ddot{x} + b\dot{x} + ml\ddot{\theta}\cos\theta - ml\dot{\theta}^2\sin\theta \Rightarrow F_1 = (M+m)\ddot{x} + b\dot{x} + ml\ddot{\theta}\cos - ml\dot{\theta}^2\sin$;
      - $F_2 = \cos\theta \Rightarrow F_2 = \cos$;

- $F_3 = \sin\theta \Rightarrow F_3 = \text{sin}$;
- Compute 'unitary portion':
  - $q_1 = F_1/(C_{11} + C_{12}) \Rightarrow F_{13} = (M+m)\ddot{x} + b\dot{x} + ml\ddot{\theta}\cos - ml\dot{\theta}^2\,\text{sin}$;
  - $q_2 = F_2/C_{11} \Rightarrow \cos\theta = \text{cos}$;
  - $q_3 = F_3/C_{12} \Rightarrow \sin\Theta = \text{sin}$;

5 Conclusions and Future Work

In this paper, we study the use of the ENPS model in a control benchmark widely used in engineering and report our work-in-progress on the design of an efficient system able to control real-life pole balancing devices. Such design can be seen of a first approach to more complex control systems. One of the most important features of such control systems is the simplicity since they must provide an answer as soon as possible in order to effectively solve real-time problems. In this first approach, the solution is based on the mathematical approach known as *PID*, Proportional Integral Derivative, but other approaches are possible.

After completing the design, the immediate future work is to prove the designed NPS by integrating a NPS simulator as SNUPS [1] with a physics simulation environment as *Webots*. The experimental results will provide useful feedback in order to improve our design to make competitive with other control software.

A future second stage will be to generalize the design to 3D vehicles and check the design with the appropriate drone flight simulator.

Acknowledgements

MAGN acknowledges the support of the project TIN2012-37434 of the Ministerio de Economía y Competitividad of Spain.

References

1. Buiu, C., Arsene, O., Cipu, C., Patrascu, M.: A software tool for modeling and simulation of numerical P systems. Biosystems 103(3), 442–447 (2011)
2. Buiu, C., Vasile, C., Arsene, O.: Development of membrane controllers for mobile robots. Information Sciences 187, 33–51 (2012)
3. Pavel, A., Arsene, O., Buiu, C.: Enzymatic numerical P systems - a new class of membrane computing systems. In: BIC-TA. pp. 1331–1336. IEEE (2010)
4. Pavel, A.B., Buiu, C.: Using enzymatic numerical P systems for modeling mobile robot controllers. Natural Computing 11(3), 387–393 (2012)
5. Pavel, A.B., Vasile, C.I., Dumitrache, I.: Robot Localization Implemented with Enzymatic Numerical P Systems. In: Prescott, T.J., Lepora, N.F., Mura, A., Verschure, P.F.M.J. (eds.) Living Machines. Lecture Notes in Computer Science, vol. 7375, pp. 204–215. Springer (2012)

6. Păun, Gh.: Some quick research topics., In these proceedings.
7. Păun, Gh., Păun, R.A.: Membrane computing and economics: Numerical P systems. Fundamenta Informaticae 73(1-2), 213–227 (2006)
8. Păun, Gh., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, Oxford, England (2010)
9. Vasile, C.I., Pavel, A.B., Dumitrache, I., Kelemen, J.: Implementing obstacle avoidance and follower behaviors on koala robots using numerical P systems. In: García-Quismondo, M., Macías-Ramos, L.F., Păun, Gh., Valencia-Cabrera, L. (eds.) Tenth Brainstorming Week on Membrane Computing. vol. II, pp. 215–227. Fénix Editora, Sevilla, Spain (2012)
10. Vasile, C.I., Pavel, A.B., Dumitrache, I., Păun, Gh.: On the power of enzymatic numerical P systems. Acta Informatica 49(6), 395–412 (2012)
11. Vasile, C.I., Pavel, A.B., Dumitrache, I.: Universality of enzymatic numerical p systems. International Journal of Computer Mathematics 90(4), 869–879 (2013)
12. Ana Brânduçsa Pavel, Cristian Ioan Vasile and Ioan Dumitrach: Robot Localization Implemented with Enzymatic Numerical P Systems. Living Machines 2012, 204-215, 2012

Monodirectional P Systems*

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

Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano-Bicocca
Viale Sarca 336/14, 20126 Milano, Italy
{leporati,luca.manzoni,mauri,porreca,zandron}@disco.unimib.it

Summary. We investigate the influence that the flow of information in membrane systems has on their computational complexity. In particular, we analyse the behaviour of P systems with active membranes where communication only happens from a membrane towards its parent, and never in the opposite direction. We prove that these "monodirectional P systems" are, when working in polynomial time and under standard complexity-theoretic assumptions, much less powerful than unrestricted ones: indeed, they characterise classes of problems defined by polynomial-time Turing machines with **NP** oracles, rather than the whole class **PSPACE** of problems solvable in polynomial space.

1 Introduction

P systems with active membranes working in polynomial time are known to be able to solve all **PSPACE**-complete problems [1]; this exploits membrane structures of polynomial depth and a bidirectional flow of information (in terms of moving objects or changing charges), both from a parent membrane to its children, and the in opposite direction.

When restricting the depth of the membrane structures of a family of P systems to a constant amount, it is still possible to solve problems in the counting hierarchy **CH**, defined in terms of polynomial-time Turing machines with oracles for counting problems [4]. In the proof of this result, it has been noticed that send-in communication rules of the form $a\,[\,]_h^\alpha \to [b]_h^\beta$ allow us to check whether the amount of objects located in a membrane exceeds a (possibly exponential) threshold in polynomial time.

It is then natural to ask whether that feature is actually necessary in order to obtain the power of counting in polynomial time. In this paper we prove (under the standard complexity-theoretic assumption that $\mathbf{P}^{\mathbf{NP}} \neq \mathbf{P}^{\#\mathbf{P}}$) that this is actually

* This work was partially supported by Università degli Studi di Milano-Bicocca, FA 2013: "Complessità computazionale in modelli di calcolo bioispirati: Sistemi a membrane e sistemi di reazioni".

the case: P systems with monodirectional communication, where the information flows only towards the outermost membrane, are limited to $\mathbf{P}^{\mathbf{NP}}$, the class of problems efficiently solved by Turing machines with **NP** oracles. This happens even when allowing polynomially deep membrane structures, a weak form of non-elementary membrane division, or dissolution (which, in this case, turns out to be as powerful as weak non-elementary division). The $\mathbf{P}^{\mathbf{NP}}$ upper bound is actually reached when dissolution or weak non-elementary division are allowed; if neither is available, then the computation power decreases to $\mathbf{P}_{\parallel}^{\mathbf{NP}}$, where the queries must all be fixed in advance, rather than asked adaptively. Chapter 17 of Papadimitriou's book [7] provides more details on complexity classes defined in terms of Turing machines with **NP** oracles.

For an introduction to P systems with active membranes ($\mathcal{AM}$), we refer the reader to the original paper by Gh. Păun [8], supplemented by the definitions of complexity classes $\mathbf{PMC}_{\mathcal{AM}}$ (resp., $\mathbf{PMC}^{\star}_{\mathcal{AM}}$) of problems solved by uniform (resp., semi-uniform) families of confluent P systems in polynomial time [5]. Define $\mathcal{M} = \mathcal{AM}(-\mathrm{i}, -\mathrm{n}, +\mathrm{wn})$ to be the class of *monodirectional P systems with active membranes*, without send-in rules; we also remove the usual ("strong") non-elementary division rules, of the form

$$\left[\left[\,\right]^{+}_{h_1} \cdots \left[\,\right]^{+}_{h_m} \left[\,\right]^{-}_{h_{m+1}} \cdots \left[\,\right]^{-}_{h_n} \right]^{\alpha}_{h} \rightarrow \left[\left[\,\right]^{\delta}_{h_1} \cdots \left[\,\right]^{\delta}_{h_m} \right]^{\beta}_{h} \left[\left[\,\right]^{\zeta}_{h_{m+1}} \cdots \left[\,\right]^{\zeta}_{h_n} \right]^{\gamma}_{h}$$

since they also provide a way for membrane h to share information with its children by changing their charge. We replace these rules by "weak" non-elementary division rules [11] of the form $[a]^{\alpha}_{h} \rightarrow [b]^{\beta}_{h}\, [c]^{\gamma}_{h}$, which allow the creation of complex membrane structures (such as complete binary trees) without exchanging information with the children membranes.

Let $\mathcal{M}(-\mathrm{d})$, $\mathcal{M}(-\mathrm{wn})$, and $\mathcal{M}(-\mathrm{d}, -\mathrm{wn})$ denote monodirectional P systems without dissolution, without weak non-elementary division, and without both kinds of rules, respectively. For each class $\mathcal{D}$ of P systems, let $\mathbf{PMC}_{\mathcal{D}}$ and $\mathbf{PMC}^{\star}_{\mathcal{D}}$ be the classes of problems solvable by uniform and semi-uniform families of P systems of class $\mathcal{D}$. Then, the main results of this paper can be summarised as follows:

- The whole class $\mathbf{PMC}^{[\star]}_{\mathcal{M}}$, as well as $\mathbf{PMC}^{[\star]}_{\mathcal{M}(-\mathrm{d})}$ and $\mathbf{PMC}^{[\star]}_{\mathcal{M}(-\mathrm{wn})}$, are equivalent to $\mathbf{P}^{\mathbf{NP}}$. Here $[\star]$ denotes optional semi-uniformity.
- The class $\mathbf{PMC}^{[\star]}_{\mathcal{M}(-\mathrm{d},-\mathrm{wn})}$ is equivalent to $\mathbf{P}_{\parallel}^{\mathbf{NP}}$.

The rest of the paper is structured as follows: in Section 2 we prove some basic limitations of monodirectional P systems; in Section 3 we exploit these results to prove upper bounds to the complexity classes for monodirectional P systems; in Section 4 we provide the corresponding lower bounds by simulating Turing machines with **NP** oracles; in Section 5 some results of the preceding sections are improved; finally, in Section 6 we present some open problems and directions for future research.

2 Properties of monodirectional P systems

We begin by proving some properties of monodirectional P systems that show how the lack of inbound communication substantially restricts the range of behaviours exhibited during the computations.

Definition 1. *Let Π be a P system, and let $\mathcal{C}$ and $\mathcal{D}$ be configurations of Π. We say that $\mathcal{C}$ is a* restriction *of $\mathcal{D}$, in symbols $\mathcal{C} \sqsubseteq \mathcal{D}$, if the membrane structures of the two configurations are identical (i.e., they have the same shape, labelling, and charges) and each multiset of objects of $\mathcal{C}$ is a submultiset of that located in the corresponding region of $\mathcal{D}$.*

The following proposition shows that, while a recogniser P system working in time t might create exponentially many objects per region during its computation, only a polynomial amount (with respect to t) of them in each region does actually play a useful role if the system is monodirectional: indeed, the final result of the computation can be identified by just keeping track of a number of objects per region equal to the number of steps yet to be carried out.

Lemma 1. *Let Π be a monodirectional recogniser P system, and let $\boldsymbol{\mathcal{C}} = (\mathcal{C}_0, \ldots, \mathcal{C}_t)$, with $t \geq 1$, be a halting computation of Π. Then, there exists a sequence of configurations $(\mathcal{D}_0, \ldots, \mathcal{D}_t)$ such that*

- *(i) we have $\mathcal{D}_i \sqsubseteq \mathcal{C}_i$ for $0 \leq i \leq t$, and each multiset of $\mathcal{D}_i$ has at most $t - i$ objects;*
- *(ii) for all $i < t$ there exists a configuration $\mathcal{E}_{i+1}$ such that $\mathcal{E}_{i+1}$ is reachable in one step from $\mathcal{D}_i$ ($\mathcal{D}_i \to \mathcal{E}_{i+1}$ for brevity) and $\mathcal{D}_{i+1} \sqsubseteq \mathcal{E}_{i+1}$;*
- *(iii) a send-out rule of the form $[a]_h^\alpha \to [\,]_h^\beta$ yes (resp., $[a]_h^\alpha \to [\,]_h^\beta$ no) is applied to the outermost membrane during the transition step $\mathcal{D}_{t-1} \to \mathcal{E}_t$ if and only if $\boldsymbol{\mathcal{C}}$ is an accepting (resp., rejecting) computation.*

Proof. By induction on t. If $t = 1$, then the environment of $\mathcal{C}_1$ contains *yes* or *no*, which have been sent out during the computation step $\mathcal{C}_0 \to \mathcal{C}_1$ by a rule $[a]_h^\alpha \to [\,]_h^\beta$ *yes* or $[a]_h^\alpha \to [\,]_h^\beta$ *no*. Let $\mathcal{D}_0 \sqsubseteq \mathcal{C}_0$ be obtained by keeping only the objects on the left-hand side of send-out, dissolution, and division rules applied during $\mathcal{C}_0 \to \mathcal{C}_1$ (we call these rules "blocking", since at most one of them can be applied inside each membrane at each step). At most one object per region is kept, given the lack of send-in rules. Let $\mathcal{D}_1 \sqsubseteq \mathcal{C}_1$ be obtained by deleting all objects. Then:

- *(i)* we have $\mathcal{D}_0 \sqsubseteq \mathcal{C}_0$ and $\mathcal{D}_1 \sqsubseteq \mathcal{C}_1$ by construction, and all multisets of $\mathcal{D}_0$ and $\mathcal{D}_1$ have at most 1 and exactly 0 objects, respectively;
- *(ii)* let the transition $\mathcal{D}_0 \to \mathcal{E}_1$ be computed by applying all blocking rules applied during the step $\mathcal{C}_0 \to \mathcal{C}_1$, which are all enabled by construction; then $\mathcal{E}_1 \sqsubseteq \mathcal{C}_1$ and, since $\mathcal{D}_1 \sqsubseteq \mathcal{C}_1$ and $\mathcal{D}_1$ contains no objects, necessarily $\mathcal{D}_1 \sqsubseteq \mathcal{E}_1$;
- *(iii)* the computation $\boldsymbol{\mathcal{C}}$ is accepting if and only if the rule $[a]_h^\alpha \to [\,]_h^\beta$ *yes* is applied from $\mathcal{C}_0$, and the latter is equivalent by construction to that rule being applicable from $\mathcal{D}_0$ (the reasoning is similar if $\boldsymbol{\mathcal{C}}$ is rejecting).

This proves the base case. Now let $\boldsymbol{\mathcal{C}} = (\mathcal{C}_{-1}, \mathcal{C}_0, \ldots, \mathcal{C}_t)$ be a halting computation of length $t+1$. The sub-computation $(\mathcal{C}_0, \ldots, \mathcal{C}_t)$ is also halting, and by induction hypothesis there exists a sequence of configurations $(\mathcal{D}_0, \ldots, \mathcal{D}_t)$ satisfying (i)–(iii). Construct the configuration $\mathcal{D}_{-1}$ as follows: first of all, keep all objects from $\mathcal{C}_{-1}$ that appear on the left-hand side of blocking rules applied during the computation step $\mathcal{C}_{-1} \to \mathcal{C}_0$; this requires at most one object per region, and guarantees that the membrane structure's shape and charges can be updated correctly (i.e., the same as $\mathcal{C}_0$ and $\mathcal{D}_0$).

We must also ensure that all objects of $\mathcal{D}_0$ can be generated from $\mathcal{D}_{-1}$ during the transition $\mathcal{D}_{-1} \to \mathcal{E}_0$. Once the blocking rules to be applied have been chosen, any object a located inside a membrane of $\mathcal{D}_0$ can be traced back to a single object in $\mathcal{D}_{-1}$. Either a appears on the right-hand side of one of those blocking rules, or it appears on the right-hand side of an object evolution rule applied in the step $\mathcal{C}_{-1} \to \mathcal{C}_0$, or it does not appear explicitly in any rule applied in that step; in the latter case, it is either carried on unchanged from $\mathcal{D}_{-1}$ (possibly from another region, if membrane dissolution occurred), or is created by duplicating the content of a membrane by applying a division rule (triggered by a different object). As a consequence, at most t objects per region of $\mathcal{D}_{-1}$, possibly in conjunction with a single object per region involved in blocking rules, suffice in order to generate the t objects per region of $\mathcal{D}_0$. As a consequence,

- (i) we have $\mathcal{D}_{-1} \sqsubseteq \mathcal{C}_{-1}$ by construction, and $\mathcal{D}_{-1}$ contains at most $t+1$ objects per region;
- (ii) by applying all blocking rules and as many evolution rules as possible from the computation step $\mathcal{C}_{-1} \to \mathcal{C}_0$ in $\mathcal{D}_{-1}$, we obtain a configuration $\mathcal{E}_0$ with the same membrane structure as $\mathcal{D}_0$ and, as mentioned above, containing all objects from $\mathcal{D}_0$ (and possibly other objects generated by evolution rules).

Since (iii) holds by induction hypothesis, this completes the proof. □

Notice that this lemma does not give us an efficient algorithm for choosing *which* objects are important for each step of the computation; it only proves that a small (i.e., polynomial-sized) multiset per region exists. However, it is easy to find such an algorithm by slightly relaxing the conditions: instead of limiting the cardinality of the multisets to $t - i$, we limit the number of occurrences *of each symbol* to that value, and simply delete the occurrences in excess separately for each symbol. This gives us the larger cardinality bound $|\Gamma| \times (t - i)$ per region, which is polynomial whenever the number of computation steps of the system is, and still allows us to simulate the overall behaviour of the P system.

Lemma 1 fails for P systems with send-in rules because some configurations where each multiset is small nonetheless require a previous configuration with a region containing exponentially many objects. This is the case, for instance, for P systems solving counting problems, where the number of assignments satisfying a Boolean formula is checked against a threshold by means of send-in rules [4]. Those assignments are represented in the P system by a potentially exponential number of objects located in the same region, which are sent into exponentially many children

membranes in parallel (i.e., at most one object enters each child membrane), and cannot always be reduced to a polynomial amount without changing the accepting behaviour of the P system.

Another property of monodirectional P systems is the existence of computations where membranes having the same labels always have children (and, recursively, all the descendents) with the same configuration. This property will be useful when simulating confluent recogniser monodirectional P systems in Section 3.

Lemma 2. *Let Π be a monodirectional P system. Then there exists a computation $\boldsymbol{\mathcal{C}} = (\mathcal{C}_0, \ldots, \mathcal{C}_t)$ of Π where, in each configuration $\mathcal{C}_i$, the following holds: any two subconfigurations[2] of $\mathcal{C}_i$ having membranes with the same label as roots are identical, except possibly for the multiset and charge of the root membranes themselves.*

Proof. By induction on i. The statement trivially holds for the initial configuration of Π, since the membranes are injectively labelled.

When a division rule is applied to a membrane h, two subconfigurations with root h are created; this is the only way to generate multiple membranes sharing the same label. The two resulting subconfigurations may only differ with respect to the contents and charges of the root membranes, since the internal membranes have evolved before the division of h occurs (recall that the rules are applied, from a logical standpoint, in a bottom-up way [8]).

On the other hand, if two subconfigurations with identically labelled root membranes already exist in a configuration $\mathcal{C}_i$, then we can assume that the property holds by induction hypothesis. We can then nondeterministically choose which rules to apply in the subconfiguration having the first membrane as root, excluding the root itself; since the other subconfiguration is identical (except possibly for the root), the same multiset of rules can also be applied to it, thus preserving the property in the next configuration of the system. □

While Lemma 2 somehow "compresses" each level of the configuration of monodirectional P systems, it does not, however, reduce the number of distinct membranes per level to a polynomial number. Indeed, the standard membrane computing technique of generating all (exponentially many) possible assignments to a set of variables does not require send-in rules [10], and can be carried out in parallel on all levels of the membrane structure.

Lemma 2 also fails for P systems with send-in rules. The reason is that two identical subconfigurations can be made different by having a single object located immediately outside, and nondeterministically sending it into one of the root membranes of the two subtrees; the evolution of the two branches of the system might then diverge completely.

[2] We define a subconfiguration of $\mathcal{C}_i$ as a subtree (a root node together with all its descendents) of the membrane structure of $\mathcal{C}_i$, including labels, multisets, and charges of the membranes.

3 Simulation of monodirectional P systems

It is a well-known result in membrane computing that P systems with active membranes can be simulated in polynomial time by deterministic Turing machines if no membrane division rules are allowed [10]. More specifically, the portion of the system that is not subject to membrane division can be simulated deterministically with a polynomial slowdown, while the output of the dividing membranes can be obtained by querying an appropriate oracle. It was recently proved that, for standard (bidirectional) P systems where only elementary membranes can divide, an oracle for a **#P** function is necessary and sufficient [5].

In what follows we prove that an **NP**-oracle is sufficient for the simulation of monodirectional P systems. In particular, the oracle will solve the following problem.

Lemma 3. *Given the initial configuration of an elementary membrane with label h of a monodirectional P system, an object type $a \in \Gamma$, and two integers $k, t \in \mathbb{N}$* in unary notation*, it is* **NP***-complete to decide whether the set of membranes with label h existing at time t emits (via send-out or dissolution rules) at least k copies of object a at that time step.*

Proof. The problem is **NP**-hard, since one can simulate an arbitrary polynomial-time, nondeterministic Turing machine M by using a single membrane with elementary division (without using send-in rules) and obtain the same result as M by checking if the resulting membranes send out at least one ($k = 1$) "acceptance object" at a specific time step [4].

Conversely, the problem can be solved by a nondeterministic, polynomial-time Turing machine M as follows. Simulate t computation steps of the membrane explicitly, by keeping track of its charge and multiset, as in any standard simulation [10]. If the membrane divides, then M keeps track of all the resulting membranes, *until the number exceeds k*. If that happens, then k copies of the membrane are chosen nondeterministically among those being simulated (which are at most $2k$ after any simulated step, if all membranes divide), and the remaining ones are discarded. Since there is no incoming communication, any instance of the membrane can be simulated correctly, as its behaviour does not depend on the behaviour of its siblings. If one of the simulated membranes dissolves before t steps, one of the k "slots" is released and can be reused in case of a further membrane division.

After having simulated t steps as described, the machine M accepts if and only if at least k copies of a are emitted (sent out, or released by dissolution) in the last step by the membranes being simulated. At most k membranes need to be simulated in order to check whether at least k copies of the object are emitted and, by exploiting nondeterminism, we are guaranteed that the correct subset of membranes is chosen by at least one computation of M. Since k and t are polynomial with respect to the size of the input, the result follows. □

The values of t and k are given in unary since, otherwise, the number of steps or the number of membranes to simulate could be exponential with respect to the size of the input, and the problem would not be solvable in polynomial time.

As a consequence of Lemma 3, monodirectional P systems without non-elementary division can be simulated in polynomial time with access to an **NP** oracle.

Theorem 1. $\mathbf{PMC}^{\star}_{\mathcal{M}(-\mathrm{wn})} \subseteq \mathbf{P}^{\mathbf{NP}}$.

Proof. The rules applied to non-elementary membranes can be simulated directly in deterministic polynomial time by a Turing machine M [5]; this includes the outermost membrane, which ultimately sends out the result object. In order to update the configurations of the non-elementary membranes correctly, the objects emitted from elementary membranes (which potentially divide) have to be added to their multisets.

Suppose the P systems of the family being simulated work in polynomial time $p(n)$. By Lemma 1, the final result of the computation can be correctly determined by keeping track of at most $p(n)$ copies of each object per region. Hence, we can update the configurations by using an oracle for the problem of Lemma 3. At time step t, we make multiple queries for each label h of an elementary membrane and for each object type $a \in \Gamma$: by performing a binary search on k over the range $[0, p(n)]$, we can find the exact number of copies of a emitted by membranes with label h at time t, or discover that this number is at least $p(n)$ (and, in that case, we only add $p(n)$ objects to the multiset). This completes the proof. □

Monodirectional P systems without non-elementary division become weaker if dissolution is also disallowed: now a membrane cannot *become* elementary during the computation, and thus the evolution of each dividing membrane is always independent of the rest of the system. This allows us to perform all queries in parallel, rather than sequentially (in an adaptive way).

Theorem 2. $\mathbf{PMC}^{\star}_{\mathcal{M}(-\mathrm{d},-\mathrm{wn})} \subseteq \mathbf{P}^{\mathbf{NP}}_{\parallel}$.

Proof. If dissolution rules are not allowed, being elementary is a static property of the membranes, i.e., a membrane is elementary for the whole computation if and only if it is elementary in the initial configuration. By observing that each query is completely independent of the others (i.e., each query involves a different membrane, time step and object) and also independent of the configurations of the non-dividing membranes (due to the lack of send-in rules), we can perform them in parallel even before starting to simulate the P system. This proves the inclusion in $\mathbf{P}^{\mathbf{NP}}_{\parallel}$. □

Now consider monodirectional P systems with non-elementary membrane division. For this kind of systems, the behaviour of a dividing membrane is, of course, dependent on the behaviour of its children and, recursively, of all its descendants.

In order to simulate the behaviour of the children by using oracles, we define a more general query problem, where we assume that the behaviour of the descendents of the membrane mentioned in the query has already been established.

First of all, notice that the lack of send-in rules allows us to extend the notion of transition step $\mathcal{C} \to \mathcal{D}$ between configurations to labelled subforests[3] $\mathcal{E}$ of $\mathcal{C}$ and $\mathcal{F}$ of $\mathcal{D}$ as $\mathcal{E} \to \mathcal{F}$; the only differences from the standard definition are that $\mathcal{E}$ is not necessarily a single tree, and that its outermost membranes may divide and dissolve.

Definition 2. *Let Π be a monodirectional P system, let $\mathcal{C}$ be a configuration of Π, and let $h \in \Lambda$ be a membrane label. A subforest $\mathcal{S}$ of $\mathcal{C}$ is called a* label-subforest induced by h, *or* h-subforest *for brevity, if one of the following conditions hold:*

- *$\mathcal{C}$ is the initial configuration of Π, and $\mathcal{S}$ consists of a single tree rooted in the (unique) membrane h,*
- *$\mathcal{C}$ is a possible configuration of Π at time $t+1$ with $\mathcal{C}' \to \mathcal{C}$, and there exists an h-subforest $\mathcal{S}'$ in $\mathcal{C}'$ such that $\mathcal{S}' \to \mathcal{S}$.*

The notion of h-subforest can be viewed as a generalisation of the equivalence classes of membranes in P systems without charges defined by Murphy and Woods [6].

Lemma 4. *Let Π be a monodirectional P system. Then there exists a computation of Π where, at each time step and for each membrane label $h \in \Lambda$, all h-subforests are identical.*

Proof. Multiple h-subforests can only be created by division of an ancestor of h; but then, by Lemma 2, there exists a computation of Π where the resulting h-subforests are identical. □

Example 1. Figure 1 shows the evolution of the membrane structure of a monodirectional P system and its label-subforests. The label-subforests in the initial configuration $\mathcal{C}_0$ coincide with all downward-closed subtrees. In the computation step $\mathcal{C}_0 \to \mathcal{C}_1$ both h_2 and h_3 divide; the division of the latter causes the duplication of the h_3- and h_4-subforests (and, indirectly, of the h_5-subforest); the division of an ancestor membrane is the only way to have more than one label-subforest. By Lemma 4, we can always assume that multiple label-subforests induced by the same label are identical. In the computation step $\mathcal{C}_1 \to \mathcal{C}_2$, the rightmost membrane having label h_2 and both instances of h_4 dissolve. Notice that this does *not* cause the disappearance of the two h_4-subforests: in the general case, the membranes h_4 might contain label-subforests induced by different labels, and we still need to refer to them as a single entity (the h_4-subforest), without the need to describe the internal structure, even when h_4 ceases to exist.

As can be observed from Figure 1, a subforest can be identified as an h-subforest by checking whether it can be generated from the downward-closed subtree rooted in h in the initial configuration.

[3] We define a subforest F' of a forest F to be any subgraph such that, whenever F' includes a vertex v, it also includes all the descendents of v.

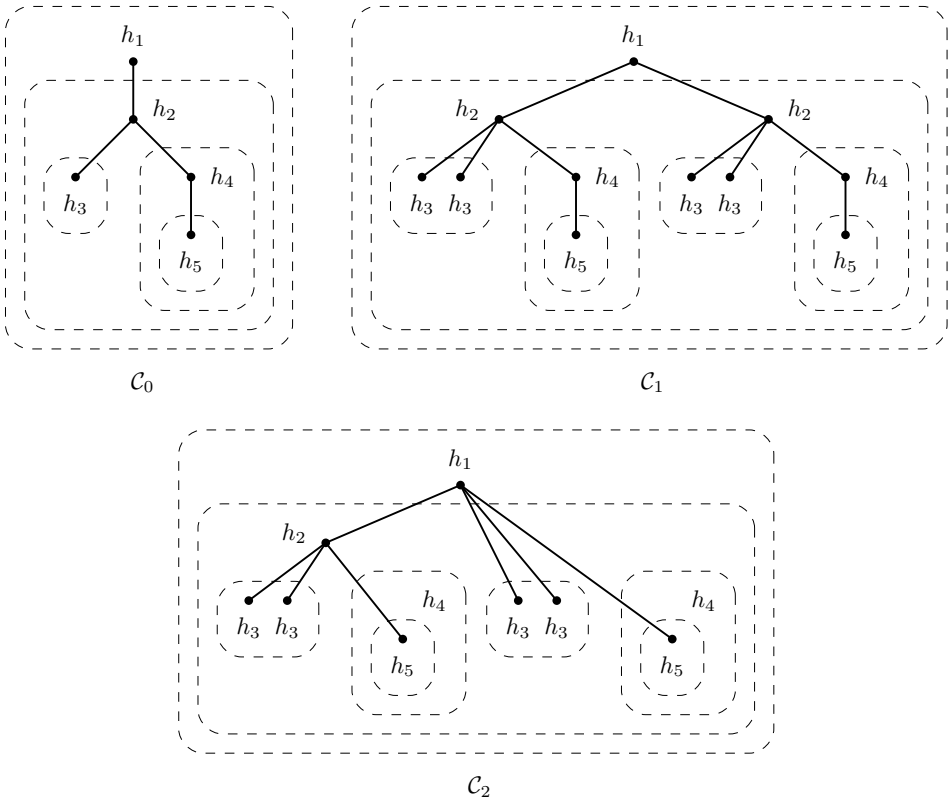


Fig. 1. Evolution of a membrane structure and its label-subforests, which are enclosed by dashed rectangles.

A computation that ensures that all h-subforests are identical for all $h \in \Lambda$ can be obtained by imposing a total ordering (a priority) on the set of rules of the P system, and applying inside each membrane the rules with higher priority whenever possible. In the following, we assume that a priority order (e.g., the lexicographic order) has been fixed; there is no loss of generality in doing that, since we only focus on *confluent* P systems in this paper. We define the multiset of objects emitted by a label-subforest as the union of the multisets emitted by its outermost membranes.

Lemma 5. *Given the initial configuration of a membrane with label h of a monodirectional P system, an object $a \in \Gamma$, two integers $k, t \in \mathbb{N}$* in unary notation*, and a table T of the objects emitted during computation steps $1, \ldots, t$ by the label-subforests immediately contained in h, it is* **NP***-complete to decide whether each h-subforest emits at least k copies of object a at time t.*

Proof. The problem is **NP**-hard, since the set of elementary membranes with label h of Lemma 3 is an example of h-subforest; that problem is thus a special case (limited to label-subforests of height 0) of the current one.

To prove membership in **NP** we also use an algorithm similar to the proof of Lemma 3: simulate up to k instances of membrane h, nondeterministically choosing which ones to keep when a membrane division occurs. However, besides simulating the rules directly involving the membranes with label h, we need to update their configuration by adding, at each computation step, the objects emitted by the label-subforests they contain. This is trivial, since the required data is supplied as the input table T. Here we exploit Lemma 4, and simulate a computation where all label-subforests contained in multiple instances of h are identical, and always emit the same objects.

The other main difference from the proof of Lemma 3 is that we do not release one of the k slots when one instance of membrane h dissolves, since its children may still emit objects, and those count in determining the output of the h-subforest. Rather, if an instance of h currently being simulated dissolved during steps $1, \ldots, t$, then we add the outputs at time t of the label-subforests immediately contained in h to the result of the computation; those outputs are obtained from table T.

The statement of this lemma then follows from an argument completely analogous to that presented in the proof of Lemma 3: there exists a sequence of nondeterministic choices leading to the simulation of k instances of h sending out at least k objects if and only if at least k objects are actually sent out by the P system being simulated. □

We can finally show that monodirectional P systems using non-elementary division (and dissolution) also do not exceed the upper bound $\mathbf{P}^{\mathbf{NP}}$.

Theorem 3. $\mathrm{PMC}^{\star}_{\mathcal{M}} \subseteq \mathbf{P}^{\mathbf{NP}}$.

Proof. We use an algorithm similar to the one described in the proof of Theorem 1. However, instead of using the oracle to compute the output of the elementary membranes, we use it to compute the output of the label-subforests. This requires first asking all queries for the label-subforests of height 0 (with an empty table T), then using the results as the table T for the queries involving label-subforests of height 1, and so on, until reaching the non-divisible membranes; these can be simulated directly by using the results of the queries involving the label-subforests immediately contained in them. Notice that the queries involving label-subforests of a given height can always be asked in parallel (across all values of a, k, t); the queries must be asked sequentially only when involving different heights. □

4 Simulation of $\mathbf{P}^{\mathbf{NP}}$ machines

In order to prove the converse inclusions between complexity classes, we describe a simulation of any Turing machine M with an **NP** oracle by means of monodirectional

P systems (an adaptation of [4]). Let Q be the set of states of M; we assume, without loss of generality, a binary alphabet $\{0,1\}$ for M. Finally, we denote by $\delta\colon Q \times \Sigma \to Q \times \Sigma \times \{\triangleleft, \triangleright\}$ the transition function of M.

Suppose that the configuration of M at a certain time step is the following: the tape contains the string $x = x_1 \cdots x_m$, the state of the machine is q, and the tape head is located on cell i. This configuration is encoded as a multiset located in a single membrane h of the P system, as follows. There is one object 1_{j-i} for each $1 \leq j \leq m$ such that $x_j = 1$; that is, each 1 in the string x is represented as an object indexed by its position in x, shifted by i; the 0s of x are not represented by an object, but rather by the absence of the corresponding 1. The object 1_0 (resp., its absence) represents a 1 (resp., a 0) located under the tape head; the indices will be updated (increased or decreased) when simulating a tape head movement. Finally, the state q of M is encoded as an object q with the same name. Further objects, not part of the encoding of the configuration of M, may also appear for simulation purposes.

A transition step of M is simulated by 7 steps of the P system. We assume that the membrane h containing the encoding of the configuration of M also contains the object $\ominus$.

Step 1. The object $\ominus$ is sent out (as the "junk" object #) in order to change the charge of h to negative:

$$[\ominus]_h^0 \to [\,]_h^- \,\# \tag{1}$$

Step 2. When h is negative, the object 1_0 is sent out, if appearing, in order to change the charge to positive. If 1_0 does not appear, the membrane remains negative.

$$[1_0]_h^- \to [\,]_h^+ \,\# \tag{2}$$

The remaining tape-objects are primed:

$$[1_i \to 1_i']_h^- \qquad\qquad \text{for } i \neq 0 \tag{3}$$

The state-object q is also primed, and produces the object $\odot$:

$$[q \to q' \odot]_h^- \tag{4}$$

Step 3. The system can now observe the charge of h and establish whether 1_0 appeared (i.e., whether the symbol under the tape head was 1) or not (i.e., the symbol was 0); this corresponds to a positive or negative charge, respectively. The object q' is rewritten accordingly:

$$[q' \to (q,1)]_h^+ \qquad\qquad [q' \to (q,0)]_h^- \tag{5}$$

At the same time, the neutral charge of h is restored by $\odot$:

$$[\odot]_h^\alpha \to [\,]_h^0 \,\# \qquad\qquad \text{for } \alpha \in \{+,-\} \tag{6}$$

Step 4. For the sake of example, suppose the transition function of M on state q is defined by $\delta(q,0) = (r,1,\triangleright)$ and $\delta(q,1) = (s,0,\triangleleft)$; the other cases are similar. The object $(q,0)$ or $(q,1)$ is rewritten accordingly:

$$[(q,0) \to (r,1,\triangleright)]_h^0 \qquad [(q,1) \to (s,0,\triangleleft)]_h^0 \tag{7}$$

Simultaneously, the tape-objects are primed again:

$$[1_i' \to 1_i'']_h^0 \qquad \text{for } i \neq 0 \tag{8}$$

Step 5. Now the triple generated in the previous step is "unpacked" into its components, which include an object that will be eventually rewritten into the new state-object, the object $1_0''$ (or nothing), and an object to be used to change the charge according to the direction of the movement of the tape head:

$$[(r,1,\triangleleft) \to \hat{r}\ 1_0''\ \oplus]_h^0 \qquad \text{for } r \in Q \tag{9}$$
$$[(r,1,\triangleright) \to \hat{r}\ 1_0''\ \ominus]_h^0 \qquad \text{for } r \in Q \tag{10}$$
$$[(r,0,\triangleleft) \to \hat{r}\ \oplus]_h^0 \qquad \text{for } r \in Q \tag{11}$$
$$[(r,0,\triangleright) \to \hat{r}\ \ominus]_h^0 \qquad \text{for } r \in Q \tag{12}$$

Step 6. The object $\oplus$, if appearing, changes the charge of the membrane to positive:

$$[\oplus]_h^0 \to [\ ]_h^+ \ \# \tag{13}$$

If $\ominus$ appears, it behaves similarly, according to rule (1). Simultaneously, the object $\hat{r}$ is primed and produces $\odot$:

$$[\hat{r} \to \hat{r}'\ \odot]_h^0 \qquad \text{for } r \in Q \tag{14}$$

Step 7. Now the charge of h is negative if the tape head is moving right, and the indices of the tape-objects have to be decremented, or positive if the tape head is moving left, and the indices must be incremented; the primes are also removed:

$$[1_i'' \to 1_{i-1}]_h^- \qquad [1_i'' \to 1_{i+1}]_h^+ \qquad \text{for } -(m-1) \leq i \leq m-1 \tag{15}$$

The object $\hat{r}'$ is now rewritten into the state-object r, and produces the $\ominus$ object to be used in Step 1 of the simulation of the next step of M:

$$[\hat{r}' \to r\ \ominus]_h^\alpha \qquad \text{for } r \in Q \text{ non final and } \alpha \in \{+,-\} \tag{16}$$

Finally, the neutral charge of h is restored by $\odot$ through rule (6). The configuration of the membrane now encodes the next configuration of M, and the system can begin simulating the next computation step. The process is depicted in Figure 2.

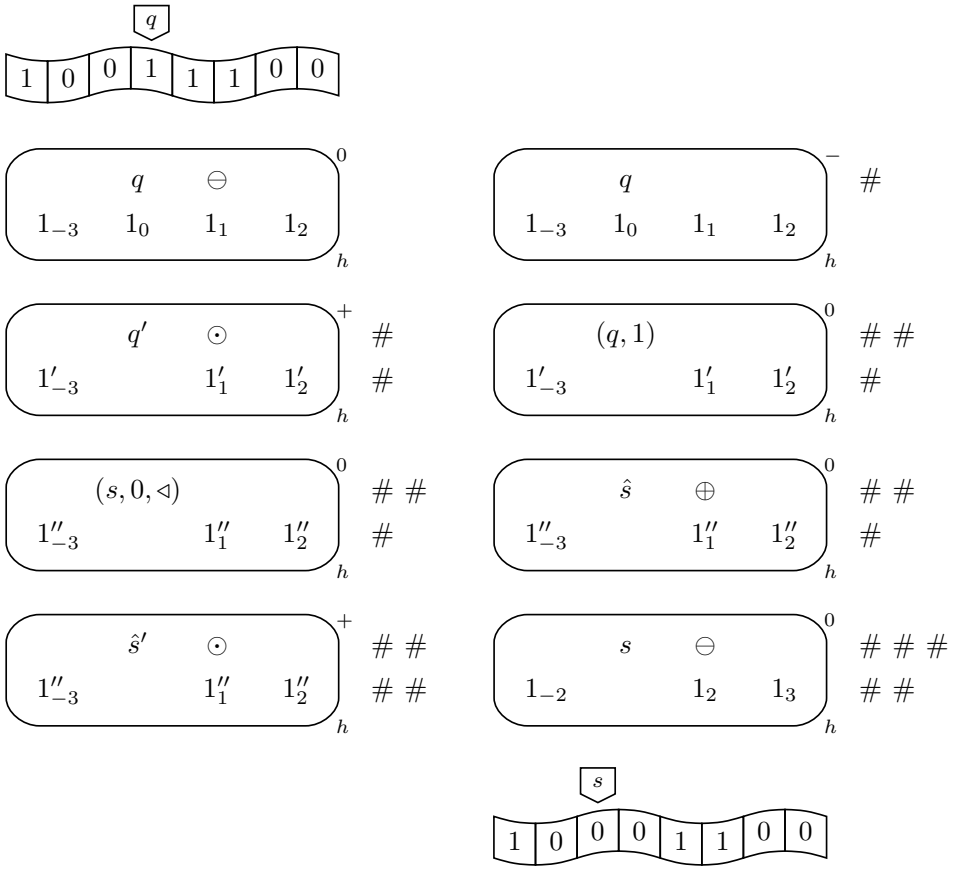


Fig. 2. Two successive Turing machine configurations, and the configurations of the P system simulating the transition step (in left-to-right, top-to-bottom order).

When $r \in Q$ is a final state (accepting or rejecting), instead of applying rule (16) the system rewrites the object $\hat{r}'$ as *yes* or *no*:

$$[\hat{r}' \to yes]_h^\alpha \qquad \text{for } r \in Q \text{ accepting and } \alpha \in \{+, -\} \tag{17}$$

$$[\hat{r}' \to no]_h^\alpha \qquad \text{for } r \in Q \text{ rejecting and } \alpha \in \{+, -\} \tag{18}$$

The object *yes* or *no* is then sent out as the result of the computation of the P system in the next step:

$$[yes]_h^0 \to [\,]_h^0\, yes \qquad\qquad [no]_h^0 \to [\,]_h^0\, no \tag{19}$$

It is easy to see that this simulation provides us with a uniform family of P systems $\boldsymbol{\Pi}_M = \{\Pi_x : x \in \{0,1\}^\star\}$, each consisting of a single membrane h and simulating the deterministic Turing machine M on all possible inputs.

4.1 Simulating oracle queries

If membrane h is not the outermost membrane of the system, then we can use division rules to simulate nondeterminism with parallelism. Suppose, for the sake of example, that the transition function of M describes nondeterministic binary choices such as $\delta(q, 0) = \{(r, 1, \triangleright), (s, 0, \triangleleft)\}$. Then, instead of the rules (7), we define the elementary division rule

$$[(q, 0)]_h^0 \to [(r, 1, \triangleright)]_h^0 \; [(s, 0, \triangleleft)]_h^0 \tag{20}$$

The two resulting copies of membrane h can then evolve in parallel according to the two possible choices.

This construction allows us to simulate polynomial-time deterministic Turing machines M with an **NP** oracle. In this section, we use the following conventions: the machine M simulates a work tape and a query tape with a single tape, by using the odd and even positions, respectively. When making a query, M writes the query string in the even positions of its tape, then enters a query state $q_?$. The oracle answers by erasing the query string (i.e., overwriting it with zeros), except for the first cell, where it writes 0 or 1 according to the result. The machine M then resumes its computation in state $q_!$, with the tape head located on the answer.

The oracle can be simulated by a polynomial-time nondeterministic Turing machine M', having initial state $q_?$ and deciding the oracle language. This machine uses only the even positions of the tape, and ends its computation in the post-query configuration described above. We assume that M' performs a series of nondeterministic choices leading to acceptance, if an accepting computation exists at all.

This combination of M and M' can be simulated by linearly nested membranes of a P system, one membrane for each query to be asked. The computation begins inside the innermost membrane, where we place a multiset encoding the initial configuration of M on its input x; whenever a query is performed, the computation moves one level higher in the membrane structure. In the following description we refer to all nested membranes as h, for brevity; the labels can be made unique, and the rules replicated for each label, with a polynomial-time preprocessing. The P system simulates the computation steps of M as described above, until M enters the query state $q_?$. Now the system pauses the simulation of M. Instead of producing $q_?$ and $\ominus$, as in rule (16), the system produces $q_?$ and $\tilde{q}_{!,t}$, where t is the maximum number of steps required by M' on query strings written by M. This number can be bounded above by considering the polynomial running time of M' on the longest possible query string, which is at most as long as the running time of M on its input x. The object $\tilde{q}_{!,t}$ is sent out from h as $q_{!,t}$, setting its charge to negative as $\ominus$ does, and upon reaching the parent membrane it begins counting down:

$$[\tilde{q}_{!,t}]_h^0 \to [\,]_h^- \; q_{!,t} \tag{21}$$

$$[q_{!,j} \to q_{!,j-1}]_h^0 \qquad \text{for } 1 \le j \le t \tag{22}$$

In the internal membrane, the nondeterministic Turing machine M' is now simulated. Since M' is allowed to make nondeterministic choices, in general there will be a number of membranes simulating M' after the first simulated step. When one of these membranes is simulating the last step of a computation of M', the object $\hat{q}'_!$ is produced by rule (14): then, instead of having a rule of type (16), the object $\hat{q}'_!$ is used to dissolve the membrane and release the tape-objects to the parent membrane:

$$[\hat{q}'_!]^{\alpha}_h \to \# \qquad \text{for } \alpha \in \{+,-\} \tag{23}$$

After t steps, all membranes simulating M' have completed the simulation, and have released their contents to the parent membrane. This membrane now contains:

- the object $q_{!,0}$;
- objects 1_i corresponding to the 1s contained in the odd positions of the tape of M (which are left unchanged by the simulation of M'); each of these objects has a multiplicity equal to the number of computations of M' on the previous query string;
- zero or more occurrences of 1_1, one for each accepting computation of M' on the query string; in particular, there is at least one occurrence of 1_1 if and only if the query string is accepted by the oracle. Notice that this object has index 1 even if it is on the first even position of the tape, since index 0 is reserved to the tape cell under the head (tape cell 1).

Before resuming the simulation of M, the system needs to eliminate any duplicate copies of objects 1_i. First of all, the object $q_{!,0}$ is rewritten into $q_!$, the next state of M:

$$[q_{!,0} \to q_!]^0_h \tag{24}$$

We then change the behaviour of M in such a way that, before continuing its original computation after receiving the answer to the oracle query, it sweeps its entire tape left-to-right and back to the first cell. This behaviour, in conjunction with the following extra rule of the P system:

$$[1_0 \to \epsilon]^+_h \tag{25}$$

erases any duplicate of 1_i for all i. Indeed, if a copy of 1_0 appears when h is positive, then another copy has been sent out in the previous step by rule (2); rule (25) eliminates such duplicates.

When the tape head of M moves back to the leftmost cell, the machine can resume its original behaviour, and the encoding of the configuration of M in the P system is now correct according to the description given at the beginning of this section.

Further queries by M are simulated analogously, by exploiting another level of the membrane structure. Notice that simulating a query actually "consumes" one level of the membrane structure, due to the dissolution rule (23). For this reason, the initial membrane structure of the P system simulating M consists of an outermost membrane, containing as many nested membranes as the number of queries performed by M.

Theorem 4. *A deterministic polynomial-time Turing machine which asks $p(n)$ queries to an* **NP** *oracle on inputs of length n can be simulated by a uniform family of monodirectional P systems of depth $p(n)$ without non-elementary division rules.*

Proof. The family of P systems $\boldsymbol{\Pi} = \{\Pi_x : x \in \{0,1\}^\star\}$ simulating M on input x can be constructed uniformly in polynomial time, since only the initial multiset depends on the actual string x, while the set of rules and the membrane structure only depend on $|x|$. We only need to make sure that the indices of the tape-objects are large enough to ensure that both the tape of M and the tape of M' can be represented at the same time. □

Corollary 1. $\mathbf{P}^{\mathbf{NP}} \subseteq \mathbf{PMC}_{\mathcal{M}(-\mathrm{wn})}$. □

Instead of using membrane dissolution as in rule (23), we can use the object $\hat{q}'_!$ to produce $\oplus$:

$$[\hat{q}'_! \to \oplus]^\alpha_h \qquad \text{for } \alpha \in \{+,-\} \tag{26}$$

which ensures that the charge of h is positive instead of negative two steps later. The tape-objects are then sent out, one at a time, by using the following rules:

$$[1_i]^+_h \to [\,]^+_h\, 1_i \qquad \text{for } -(m-1) \le i \le m-1 \tag{27}$$

The timer t of the object $\tilde{q}_{!,t}$ has to be increased appropriately, in order to take into account the time needed to send out all the tape-objects. However, since the membrane where the simulation of M is non-elementary after the first query, rule (20) is now a weak non-elementary division rule. As a consequence, we have:

Theorem 5. *A deterministic polynomial-time Turing machine which asks $p(n)$ queries to an* **NP** *oracle on inputs of length n can be simulated by a uniform family of monodirectional P systems of depth $p(n)$ without dissolution rules.* □

Corollary 2. $\mathbf{P}^{\mathbf{NP}} \subseteq \mathbf{PMC}_{\mathcal{M}(-\mathrm{d})}$. □

In order to prove the converse of Theorem 2, we introduce an auxiliary complexity class (a variant of the class of optimisation problems **OptP** [3]).

Definition 3. *Define* **OrP** *to be the class of functions $f\colon \{0,1\}^\star \to \{0,1\}^\star$ having a polynomial-time nondeterministic Turing machine M such that, for all $x \in \{0,1\}^\star$, we have $f(x) = \bigvee M(x)$, where $M(x)$ denotes the set of possible output strings of M on input x, and $\bigvee$ denotes bitwise disjunction of strings; here we assume that the bitwise disjunction of strings of different lengths is performed by padding the shortest ones with zeros.*

The purpose of the class **OrP** is to capture a polynomial number of parallel **NP** queries with a single query to a function over binary strings.

Proposition 1. $\mathbf{P}^{\mathbf{NP}}_{\parallel} = \mathbf{P}^{\mathbf{OrP}[1]}$.

Proof. A polynomial number of parallel queries $y_1, \ldots, y_m$ to an oracle for $L \in \mathbf{NP}$ can be replaced by a single query to an oracle for the function $f(y_1, \ldots, y_m) = z_1 \cdots z_m$, where $z_i = 1$ if and only if $y_i \in L$. Let M be an **NP** machine deciding L, and let M' be the following nondeterministic machine: on input $y_1, \ldots, y_m$ simulate M on each y_i and record the corresponding output bit z_i; finally, output $z_1 \cdots z_m$. For all $1 \leq i \leq m$, if y_i is accepted by the oracle, then there exists a computation of M' such that $z_i = 1$: thus, by taking the bitwise disjunction of all possible output strings of M', we obtain the i-th bit of $f(y_1, \ldots, y_m)$; this proves that $f \in \mathbf{OrP}$. Notice that this proof requires the query strings $y_1, \ldots, y_m$ to be fixed in advance, i.e., the queries cannot be performed adaptively.

Vice versa, a single query to an oracle for $f \in \mathbf{OrP}$ with query string y can be replaced by the following polynomial number of parallel queries, one for each $1 \leq i \leq |f(y)|$: "is the i-th bit of $f(y)$ a 1?". These queries are in **NP**, since they can be answered by simulating an **OrP** machine M for f and selecting only its i-th output bit; the answer will be positive if and only if there exists a computation of M having a 1 as the i-th output bit, which (by definition of **OrP**) is equivalent to the i-th bit of $f(y)$ being 1. □

Simulating an **OrP** query by means of a P system is completely analogous to simulating an **NP** query, except that, instead of a single output bit, we have a polynomial number of them. These binary strings are automatically combined by bitwise disjunction when the tape-objects are sent out of the membrane simulating the nondeterministic Turing machine. Furthermore, since a single **OrP** query suffices to capture $\mathbf{P}_{\parallel}^{\mathbf{NP}}$, we obtain the following results:

Theorem 6. *A deterministic polynomial-time Turing machine which asks a polynomial number of* parallel *queries to an* **NP** *oracle on inputs of length n can be simulated by a uniform family of monodirectional P systems of depth 1 without dissolution (and, necessarily, without non-elementary division).* □

Corollary 3. $\mathbf{P}_{\parallel}^{\mathbf{NP}} \subseteq \mathbf{PMC}_{\mathcal{M}(-\mathrm{d},-\mathrm{wn})}$. □

5 Further results

The depth of the P systems of Theorems 4 and 5 can be asymptotically reduced by exploiting the equivalence of a logarithmic number of adaptive queries and a polynomial number of parallel queries [7, Theorem 17.7], formally $\mathbf{P}_{\parallel}^{\mathbf{NP}} = \mathbf{P}^{\mathbf{NP}[\log n]}$. Suppose a deterministic polynomial-time Turing machine performs $p(n)$ sequential **NP** queries, and divide these queries into $\Theta(p(n)/\log n)$ blocks of $\Theta(\log n)$ queries. Each block can then be replaced by a polynomial number of parallel **NP** queries or, by Proposition 1, by a single **OrP** query. Hence, $p(n)$ sequential **NP** queries can be simulated by $\Theta(p(n)/\log n)$ sequential **OrP** queries, and each of the latter can be simulated by one level of depth in a P system:

Corollary 4. *A deterministic polynomial-time Turing machine which asks $p(n)$ queries to an* **NP** *oracle on inputs of length n can be simulated by a uniform family of monodirectional P systems of depth $\Theta(p(n)/\log n)$ without non-elementary division rules (resp., without division rules).* □

Theorem 3 can be sharpened by making the intra-level query parallelism explicit with **OrP** queries:

Corollary 5. *Let $\boldsymbol{\Pi}$ be a family of semi-uniform polynomial-time monodirectional P systems of depth $f(n)$. Then $\boldsymbol{\Pi}$ can be simulated by a polynomial-time deterministic Turing machine with $f(n)$ queries to an* **OrP** *oracle.* □

We can also prove that monodirectional families of P systems of *any* constant depth, even with dissolution and non-elementary division rules (in symbols $\mathcal{M}(\mathrm{O}(1))$), are always equivalent to families of depth *one* without dissolution and without non-elementary division (in symbols $\mathcal{M}(1, -\mathrm{d}, -\mathrm{wn})$), and thus only able to simulate parallel **NP** queries.

Theorem 7. $\mathbf{PMC}^{[\star]}_{\mathcal{M}(\mathrm{O}(1))} = \mathbf{PMC}^{[\star]}_{\mathcal{M}(1,-\mathrm{d},-\mathrm{wn})} = \mathbf{P}^{\mathbf{NP}}_{\parallel}$.

Proof. By Theorem 6, we already know that $\mathbf{P}^{\mathbf{NP}}_{\parallel} \subseteq \mathbf{PMC}_{\mathcal{M}(\mathrm{O}(1))}$, even when limited to depth 1; the inclusion $\mathbf{PMC}_{\mathcal{M}(\mathrm{O}(1))} \subseteq \mathbf{PMC}^{\star}_{\mathcal{M}(\mathrm{O}(1))}$ holds by definition. The inclusion $\mathbf{PMC}^{\star}_{\mathcal{M}(\mathrm{O}(1))} \subseteq \mathbf{P}^{\mathbf{NP}}_{\parallel}$ can be proved as follows. By Theorem 3, a family of P systems of constant depth k can be simulated in polynomial time by asking k sets (one per level) of $p(n)$ parallel queries, for some polynomial p. Each set of $p(n)$ parallel queries can be converted into $\Theta(\log n)$ sequential queries [7, Theorem 17.7], for a total of $k \times \Theta(\log n)$ sequential queries. These can be converted back into a polynomial number of parallel queries. □

Finally, observe that Theorem 3 also trivially holds for monodirectional P systems *without charges*. This implies a better upper bound than previously known [5] for a monodirectional variant of the P conjecture [9, Problem F], which states that P systems without charges and without non-elementary division characterise **P**.

6 Conclusions

In this paper we confirmed the importance of the direction of the information flow in P systems with active membranes with respect to their computing power. Indeed, when working in polynomial time and using only outward-bound communication, the corresponding complexity class decreases from **PSPACE** to $\mathbf{P}^{\mathbf{NP}}$, or from $\mathbf{P}^{\#\mathbf{P}}$ to $\mathbf{P}^{\mathbf{NP}}_{\parallel}$ when non-elementary division and dissolution rules are disallowed. It is interesting to notice that, unlike with other restrictions such as removing membrane division [10] or charges and dissolution [2], the resulting P systems are still more powerful than **P** (unless, of course, $\mathbf{P} = \mathbf{NP}$).

The role of strong non-elementary division (which is replaced in this paper by weak non-elementary division) in the absence of send-in rules is still unclear. Even if it provides a way to convey information from a parent membrane to its children, we do not know whether this is sufficient to altogether replace send-in communication while maintaining a polynomial run-time.

Finally, it would be interesting to investigate monodirectional P systems where the information flow is reversed, i.e., send-out communication and dissolution rules (as well as strong non-elementary division rules) are disallowed. A first issue to overcome is choosing an appropriate acceptance condition for the P systems, to replace sending out *yes* or *no* from the outermost membrane. The acceptance condition most similar "in spirit" to the original one is probably accepting (resp., rejecting) by having at least one *yes* (resp., *no*) object appear, either anywhere in the system, or inside a distinguished (and possibly dividing) membrane, during the last computation step; we also add the restriction that *yes* and *no* can never appear together, since giving the priority to one of them would allow us to solve **NP**-complete (or **coNP**-complete) problems "for free". Such monodirectional P systems appear to be very weak when working in polynomial time; indeed, even though exponentially many membranes can still be created by division, they can never communicate. Is **P** actually an upper bound to the class of problems they can solve?

References

1. Alhazov, A., Martín-Vide, C., Pan, L.: Solving a PSPACE-complete problem by recognizing P systems with restricted active membranes. Fundamenta Informaticae 58(2), 67–77 (2003)
2. Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J., Riscos-Nuñez, A., Romero-Campero, F.J.: Computational efficiency of dissolution rules in membrane systems. International Journal of Computer Mathematics 83(7), 593–611 (2006)
3. Krentel, M.W.: The complexity of optimization problems. Journal of Computer and System Sciences 36, 490–509 (1988)
4. Leporati, A., Manzoni, L., Mauri, G., Porreca, A.E., Zandron, C.: Membrane division, oracles, and the counting hierarchy. Fundamenta Informaticae (2015), in press
5. Leporati, A., Manzoni, L., Mauri, G., Porreca, A.E., Zandron, C.: Simulating elementary active membranes, with an application to the P conjecture. In: Gheorghe, M., Rozenberg, G., Sosík, P., Zandron, C. (eds.) Membrane Computing, 15th International Conference, CMC 2014, Lecture Notes in Computer Science, vol. 8961, pp. 284–299. Springer (2015)
6. Murphy, N., Woods, D.: Active membrane systems without charges and using only symmetric elementary division characterise P. In: Eleftherakis, G., Kefalas, P., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing, 8th International Workshop, WMC 2007. Lecture Notes in Computer Science, vol. 4860, pp. 367–384 (2007)
7. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1993)
8. Păun, Gh.: P systems with active membranes: Attacking NP-complete problems. Journal of Automata, Languages and Combinatorics 6(1), 75–90 (2001)

9. Păun, Gh.: Further twenty six open problems in membrane computing. In: Gutíerrez-Naranjo, M.A., Riscos-Nuñez, A., Romero-Campero, F.J., Sburlan, D. (eds.) Proceedings of the Third Brainstorming Week on Membrane Computing. pp. 249–262. Fénix Editora (2005)
10. Zandron, C., Ferretti, C., Mauri, G.: Solving NP-complete problems using P systems with active membranes. In: Antoniou, I., Calude, C.S., Dinneen, M.J. (eds.) Unconventional Models of Computation, UMC'2K, Proceedings of the Second International Conference, pp. 289–301. Springer (2001)
11. Zandron, C., Leporati, A., Ferretti, C., Mauri, G., Pérez-Jiménez, M.J.: On the computational efficiency of polarizationless recognizer P systems with strong division and dissolution. Fundamenta Informaticae 87, 79–91 (2008)

Parallel Simulation of PDP Systems: Updates and Roadmap

Miguel Ángel Martinez-del-Amor[1], Luis Felipe Macías-Ramos[2], Mario J. Pérez–Jiménez[2]

[1] Moving Picture Technologies, Fraunhofer IIS
Am Wolfsmantel 33, 91058 Erlangen, Germany
[2] Research Group on Natural Computing, Universidad de Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mails: mdelamor@us.es, lfmaciasr@us.es, marper@us.es

Summary. PDP systems are a type of multienvironment P systems, which serve as a formal modeling framework for Population Dynamics. The accurate simulation of these probabilistic models entails large run times. Hence, parallel platforms such as GPUs has been employed to speedup the simulation. In 2012 [14], the first GPU simulator of PDP systems was presented. In this paper, we present current updates made on this simulator, and future developments to consider.

1 Introduction

P systems[16, 17] have become good candidates for computational modeling thanks to the compartmental and discrete features, both in Systems Biology [19, 20] and Population Dynamics [3]. In this concern, it is worth to mention the achieved success in real ecosystem modeling through probabilistic P systems, such as the Bearded Vulture in the Catalan Pyrenees (endangered species) [2], and the zebra mussel in Ribarroja reservoir (exotic invasive species) [1]. These works have lead to a formal, computational modeling framework called Population Dynamics P systems (PDP systems) [4].

In order to experimentally validate these P systems based models, the development of simulators is requested [17]. P-Lingua [5, 25] is a simulation framework for P systems, which aims to be generic, multi-platform (it is written in Java) and to provide a standard description language for P systems. It has been used to develop simulators for many variants of P systems, specially for PDP systems. Furthermore, experts and model designers are able to run virtual experiments in an abstracted way (without the need of accessing to details of P systems) through a special software called MeCoSim [18, 24]. MeCoSim uses P-Lingua as the simulation core.

The run times offered by these general simulation frameworks are high for some scenarios involving large and complex models. This lack of efficiency is mainly given from the facts of both using Java Virtual Machine and implementing sequential algorithms [10]. Indeed, simulating massively parallel devices like P systems in a sequential fashion is twice inefficient. This issue is can be addressed by harnessing the highly parallel architecture within modern processors to map the massively parallelism of P systems [10, 11].

Whereas commodity CPUs can contain dozens of processors, current graphic processors (GPUs) [8, 15] provide thousands of computing cores. They can be programmed using general-purpose frameworks such as CUDA [9, 23], OpenCL and OpenAcc. GPUs exploit data parallelism by using a very fast memory and simplistic cores. Given the high level of parallelism within modern GPUs (up to 3500 cores per device [23]), they have provided a platform to implement real parallelism of P systems in a natural way. Many P system models have been considered to be simulated with CUDA [11]: P systems with active membranes, SAT solutions with families of P systems with active membranes and of tissue P systems with cell division, Enzymatic Numerical P systems, Spiking Neural P systems without delays, and Population Dynamics P systems [14], among others. Most of these simulators are within the scope of PMCGPU (Parallel simulators for Membrane Computing on the GPU) software project [26], which aims to gather efforts on parallelizing P system simulators with GPU computing.

As shown by all of these research works, the development of a new P system simulator requires a big research and development effort. For example, in the case of the simulator for PDP systems, the simulation algorithm called DCBA [13, 3] was implemented. It is based on 4 different phases with completely different characteristics, and the parallelization effort is also different in each one (e.g. second phase of DCBA is a random sequential loop that cannot be easily parallelized). Therefore, the different semantical and syntactical elements of each P system variant lead to completely different GPU-based simulators. Not only does the GPU code depend on the simulated variant, but its efficiency also depend on the simulated P system within the variant [14].

In this paper, we show new developments on the GPU simulator for PDP systems. In summary, a new input module received binary files has been created, allowing to run real ecosystem models defined with P-Lingua. Moreover, we present a road map proposal, a set of research lines for future work that is going to be addressed.

The paper is structured as follows: Section 2 provides an overview of the required concepts to understand this paper. Section 3 presents the new feature of the simulator consisting in a input module to read binary files, and also some preliminary results. Finally, Section 4 discuses future developments to take into consideration.

2 Preliminaries

In this section we briefly provide the minimum concepts for the understandability of the paper. We will not introduce the model of PDP systems and GPU computing into detail. Instead, we provide short descriptions along with useful references.

2.1 PDP systems model and simulation: DCBA

PDP systems [4, 3] are a branch of multienvironment P systems [6], which consists in a directed graph whose nodes are called environments. Each environment contains a single cell-like P system. Moreover, the arcs of the graph is implicitly given by a set of communication rules which allow the movement of objects between environments in a one-to-many fashion. Thus, these rules are of the form: $(x)_{e_j} \xrightarrow{pr} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$. All the P systems within a PDP system have the same skeleton: the same membrane structure (with three polarizations), the same working alphabet, and the same set of (skeleton) evolution rules. These rules are of the form: $u\ [\ v\ ]_h^\alpha \to u'\ [\ v'\ ]_h^\beta$. It can be seen that these P systems are an extension of the active membranes model. However, no dissolution neither division are allowed, and special care on the consistency of rules has to be taken.

PDP systems have also a probabilistic flavor in terms of probabilities associated to the rules. On the one hand, a probability is associated to each skeleton rule for each environment, thus being of the following form: $u\ [\ v\ ]_h^\alpha \xrightarrow{f_{r,j}} u'\ [\ v'\ ]_h^\beta$. On the other hand, a probability is associated to each communication rule globally to the PDP system. Rules are executed in a maximal parallel way according to the probabilities. Rules having the same left-hand side must satisfy the following condition: the sum of their probabilities has to be 1. Eventually, rules having an "unique" left-hand side have associated the probability 1. Inherently to the model is the concept of rule block: a block is formed by rules having the same left-hand side.

For the syntax of the models, refer to [3, 4] and [6]. Concerning the semantics of the model, several simulation algorithms have been proposed since the introduction of PDP systems. Each new algorithm aimed at improving the accuracy in which the reality is mapped to the models. Perhaps, the most difficult feature to handle by the simulation algorithms is the competition of objects between rules from different blocks (note that rules within a block have a the same left-hand side, and the objects are consumed according to the probabilities) [10].

The latest introduced algorithm for PDP system is called *Direct distribution based on Consistent Blocks Algorithm (DCBA)* [13]. The approach taken in it is based on the idea of distributing the objects along the rule blocks in a proportional way. After this distribution, the rules within the blocks are selected according to their probabilities using a multinomial distribution. In summary, DCBA consists in 4 phases: 3 for selecting rules and the last one for performing the execution. The scheme of DCBA is the following:

1. Initialization of the algorithm: *static distribution table* (**columns:** blocks, **Rows:** (objects,membrane))
2. **Loop over Time**
3. **Selection** stage:
4. **Phase 1** (Distribution of objects along rule blocks)
5. **Phase 2** (Maximality selection of rule blocks)
6. **Phase 3** (Probabilistic distribution, blocks to rules)
7. **Execution** stage

The proportional distribution of objects along the blocks is carried out through a table which implements the relations between blocks (columns) and objects in membranes (rows). We always start with a static (general) table, and depending on the current configuration of the PDP system, the table is dynamically modified by deleting columns related to non-applicable blocks. Note that after phase 1, we have to assure that the maximality condition still holds. This is normally conveyed by a random loop over the remaining blocks.

Finally, DCBA also handles the consistency of rules by defining the concept of consistent blocks [13, 10]: rules within a block have the same left-hand side and the same charge in the right-hand side. There is a further restriction within phase 1: if two non-consistent blocks (having different associated right-hand charge) can be selected in a configuration, the simulation algorithm will return an error, or optionally non-deterministically choose a subset of consistent blocks.

2.2 GPU computing

Today, PC's processors offer from 2 to 16 computing cores, and this number can be increased to 64 or even 128 in high end equipments. These cores are complex enough to run threads simultaneously, each one with its own context, exploiting a coarse grain level of parallelism. For example, OpenMP [22] is a threading library for multicore processors, which can be used in C/C++.

High Performance Computing world has changed in the past years. The introduction of the GPU [8] as a co-processor unit to compute and render 3-D graphics, encouraged the change of trend in HPC solutions and start to consider heterogeneous platforms having CPUs and co-processors. The GPU has been devised as a highly parallel processor since it was conceived, and now, GPGPU enables the GPU to be used for general purpose scientific applications [21].

A GPU consists in SIMD multiprocessors interconnected to a fast bus with the main memory system [15, 9]. Each multiprocessor has a set of computing cores that execute instructions synchronously (they always perform the same instruction over different data) and a small portion of sketchpad memory (similar to caches in CPUs, but manually managed by programmers), among other elements. Current GPUs also implement cache memories (one L2 at the level of the memory system, and a L1 cache which resides within the sketchpad memory).

Fortunately, all these aspects are abstracted to the programmer with high level programming models such as CUDA [9, 23]. Introduced by NVIDIA in 2007, CUDA

allows to run thousands of lightweight *threads* concurrently arranged in *blocks*. Threads belonging to the same block can cooperate and easily be synchronized. Threads from different blocks can only be synchronized by finishing their execution. All these threads execute the same code, called *kernel*, in a SPMD (single-program, multiple-data) fashion, since they can access to different pieces of data by using the identifiers associated to each thread and block. Moreover, each thread can also take different branches of execution, but this is penalized when happened within a *warp* (a group of 32 threads), given that it will makes the execution to be serialized. The largest but slowest memory system is called global memory, whereas the smallest but fastest sketchpad memory belonging to each block is called shared memory. The access to these memories should be done carefully, since best bandwidth is achieved when threads access to memory in coalesced (to contiguous addresses) and aligned way [15].

Finally, the GPU architecture has been improving by the different releases. GT800, Fermi, Kepler and Maxwell are the codename of each NVIDIA GPU generation. Each one has been associated to a Compute Capability (CC), 1.X, 2.X, 3.X, and 5.X, respectively [23].

2.3 PDP systems parallel simulation on the GPU

As mentioned above, the main objective of DCBA is to improve the accuracy of the algorithm. However, it comes at expenses of low efficiency. Currently, P-Lingua framework implements the algorithm, but it is usually not recommended when dealing with large models because of the large simulation times. This lack of efficiency is mainly due to the use of Java Virtual Machine and sequential algorithms. Indeed, simulating massively parallel devices like P systems in a sequential fashion is twice inefficient. A solution to outcome this issue is by harnessing the highly parallel architecture within modern processors to map the massively parallelism of P systems [10].

GPUs provide a good platform to implement real parallelism of P systems in a natural way, by using their high level parallelism [11]. Most of P systems simulators based on GPU are within the scope of PMCGPU (*Parallel simulators for Membrane Computing on the GPU*) software project [26], which aims to gather efforts on parallelizing P system simulators with GPU computing. Specifically, there is a subproject for PDP systems, called ABCD-GPU.

ABCD-GPU started with a multi-core version [12, 10], based on C++ and OpenMP, in which the environments and/or the simulations are distributed along the processors. Experiments showed that parallelizing by simulations leads to better speedups; that is, in a multiprocessor CPU, it is better to parallelize coarsely. In order to deal with finer-grain parallelism, a CUDA version has been also developed [14, 10]. In general, these parallel simulators are based on the following principles:

- Efficient representation of the data, both for PDP system syntactical elements and auxiliary structures of DCBA. In this concern, the static and dynamic

tables for phase 1 are not really implemented. Instead, the operations over these tables are translated to operations over the syntactical elements of the PDP system, together with much smaller structures. This approach is called virtual table, and has shown to dramatically decrease the required amount of data and time in DCBA.
- Exploiting levels of parallelism presented in the simulation of PDP systems: processing of rule blocks and rules, evolution of environments, and conducting several simulations to extract statistical data from the probabilistic model.

As mentioned in previous section, CUDA requires a large amount of parallelism to effectively use GPUs resources [9]. Parallelizing only by simulations as in the OpenMP version is not enough, and the parallelism level is coarse. Instead, the solution was to extract more parallelism from the PDP systems as follows [14]:

- Thread blocks: they are assigned to each environment and each simulation. For each transition step, there is a minimal communication along environments (only when executing communication rules), and each simulation can be executed independently.
- Threads: each thread is assigned to each rule block/column in selection phases (1, 2 and 3). In execution phase (4), threads will execute rules in parallel. As it is possible to have more rule blocks than threads per thread block, they perform a loop over rule blocks in tiles.

So far, ABCD-GPU simulator has been tested by using randomly generated PDP systems. The goal of this was to provide a flexible way to construct benchmarks for performance analysis, by stressing the simulator with different topologies. For example, Table 1[14] shows the performance of the simulator with PDP systems having different lengths of the left-hand sides (in terms of number of different objects in the multisets u and v) in average, and running on a NVIDIA Tesla C1060 GPU, which has 240 cores and CC 1.3. These results clearly show that phase 2 is the bottleneck of the simulator, since it is the less parallel phase consisting in a random sequential loop. Moreover, when the competition for objects increase (having more objects in the LHS leads to more competitions), overall performance drastically decreases.

	Test with average LHS length of 1.5		
	% CPU	% GPU	Speedup
Phase 1	53.7%	30.1%	14.23x
Phase 2	12.6%	47%	2.13x
Phase 3	22.6%	13.7%	13.2x
Phase 4	11.1%	9.2%	9.7x

Table 1. Performance testing through randomly generated PDP systems.

3 A new input module: binary files

After the first version of ABCD-GPU [14], the efforts were focused on creating a input module to read PDP system descriptions. In this section, we briefly present the new features of the ABCD-GPU simulator, which is a module to read binary files defining PDP systems models. We also show preliminary results of the simulator with a real ecosystem model.

3.1 Format definition

Similarly to the simulator of P systems with active membranes [10, 11], the design decision for the input file was a binary format. The reason for this is twofold:

- *Size of files*: the GPU simulator is conceived for running very large models. Otherwise, it is not worth to be used. Thus, the communication with the simulator should be as efficient as possible to avoid overheads. Since we use P-Lingua for describing PDP system models, it makes sense to use pLinguaCore to parse the files. In this concern, P-Lingua is used as the parser and compiler which send a file to the simulator with unwrapped rules (recall that rules in P-Lingua can be defined in a symbolic way). Thus, in order to reduce the size of the file as much as possible, we have defined a binary format which assign the less bits to each syntactic element.
- *Efficiency*: related with the latter, the binary file is also organized in such a way that it fits well with the initialization of structures in the simulator. This helps the efficiency of the parser, while reducing the size of the files.

Although using this kind of format lead to a coupled design (between the P-Lingua parser and the simulator), it will allow to use the GPU engine while reducing the communication/storage cost.

Next, we show the structure of the format for the binary file, which is divided into 5 sections:

- Header: unequivocally identify this file as a binary description file for PDP systems.
- Sub-header: defines the accuracy used along the file, for the different fields. This allows to use the exact number of bytes according to the number of objects, rules, etc.
- Global sizes: define the size of alphabet, number of rules, membranes, environments and membrane structure.
- Rule blocks: their information is given in 3 subsections, each one giving information for allocating space related with the next one.
- Initial configuration description.

```
1 ######################################################################
2 # Binary file format for the input of the simulator: PDP systems
3 # (revision 16-09-2014). The encoded numbers must be in big-endian
```

```
4
5
6   # Header (4 Bytes):
7   0xAF
8   0x12
9   0xFA
10  0x21 (Last Byte: 4 bits for P system model, 4 bits for file version)
11
12  # Sub-header (3 Bytes):
13  Bit-accuracy mask (2 Bytes, 2 bits for each number N (meaning a precision
14  of 2^N Bytes)), for:
15  - Num. of objects (2 bits (meaning 2^0 -- 2^2 Bytes))
16  - Num. of environments (2 bits (meaning 2^0 -- 2^2 Bytes))
17  - Num. of membranes (2 bits (meaning 2^0 -- 2^2 Bytes))
18  - Num. of skeleton rules (2 bits (meaning 2^0 -- 2^2 Bytes))
19  - Num. of environment rules (2 bits (meaning 2^0 -- 2^2 Bytes))
20  - Object multiplicities in rules (2 bits (meaning 2^0 -- 2^2 Bytes))
21  - Initial num. of objects in membranes (2 bits (meaning 2^0 -- 2^2 Bytes))
22  - Multiplicities in initial multisets (2 bits (meaning 2^0 -- 2^2 Bytes))
23  Listing char strings (1 Byte, 5 bits reserved + 3 bits), for:
24  - Reserved (5 bits)
25  - Alphabet (1 bit)
26  - Environments (1 bit)
27  - Membranes (1 bit)
28
29
30  #---- Global sizes
31
32  # Alphabet
33  Number of objects in the alphabet (1-4 Bytes)
34  ## For each object (implicit identificator given by the order)
35  Char string representing the object (finished by '\0')
36
37
38  # Environments
39  Number of environments, m parameter (1-4 Bytes)
40  ## For each environment (implicit identificator given by the order)
41  Char string representing the environment (finished by '\0')
42
43
44  # Membranes (including the environment space as a membrane)
45  Number of membranes, q parameter + 1 (1-4 Bytes)
46  ## For each membrane (implicit identificator given by the order,
47                        from 1 (0 denotes environment))
48  Parent membrane ID (1-4 Bytes)
49  Char string representing the label (finished by '\0')
50
51
```

```
# Number of rule blocks
Number of rule blocks of Pi/Skeleton (1-4 Bytes)
Number of rule blocks of the environments (1-4 Bytes)


#---- Information of rule blocks: number rules and length LHS

# For each rule block of Pi (skeleton)
Information Byte (1 Byte: 2 bits for precision of multiplicity in L/RHS
                  (2^0 -- 2^2 Bytes) + 1 bit precision number of objects
                  in LHS (2^0 -- 2^1 Bytes) + 1 bit precision number of
                  objects in RHS (2^0 -- 2^1 Bytes) + 2 bits precision
                  number of rules in the block (2^0 -- 2^2 Bytes) + 1 bit
                  don't show probability for each environment + 1 bit show
                  parent membrane)
Number of rules inside the block (1-4 Bytes)
Number of objects in LHS; that is, length U + length V (1-2 Bytes)
Active Membrane (1-4 Bytes)
# If show parent membrane flag is active (deprecated)
Parent Membrane (1-4 Bytes, this is deprecated)
Charges (1 Byte: 2 bits for alpha, 2 bits for alpha', 4 bits reserved,
         using 0=0, +=1, -=2)

# For each rule block of environment
Information Byte (1 Byte: 2 bits for precision of multiplicity in LHS
                  (2^0 -- 2^2 Bytes) + 1 bit precision number of objects
                  in LHS (2^0 -- 2^1 Bytes) + 1 bit precision number of
                  objects in RHS (2^0 -- 2^1 Bytes) + 2 bits precision of
                  number of rules in the block (2^0 -- 2^2 Bytes) + 1 bit
                  probability for each environment + 1 bit show parent
                  membrane)
Number of rules inside the block (1-2 Bytes)
Environment (1-4 Bytes)


#---- Information of rule blocks: length RHS, probabilities and LHS

# For each rule block of Pi
## For each rule
Number of objects in RHS; that is, length U' + length V' (1-2 Bytes)
### For each environment
Probability first 4 decimals (prob*10000) (2 Bytes)
## For LHS U: multiset in the LHS in the parent membrane U [ V ]_h^a
Number of objects in U (1-2 Bytes)
### For each object
Object ID (1-4 Bytes)
Multiplicity (1-4 Bytes)
## For LHS V: multiset in the LHS in the active membrane U [ V ]_h^a
```

```
Number of objects in V (1-2 Bytes)
### For each object
Object ID (1-4 Bytes)
Multiplicity (1-4 Bytes)

# For each rule block of environment
Object in LHS (1-4 Bytes)
## For each rule
Number of objects (involved environments) in RHS (1-2 Bytes)
Probability first 4 decimals (prob*10000) (2 Bytes)


#---- Information of rule blocks: RHS

# For each rule block of Pi
## For each rule
### For RHS U': multiset in the RHS in the parent membrane U' [ V' ]_h^a'
Number of objects in U' (1-2 Bytes)
#### For each object
Object ID (1-4 Bytes)
Multiplicity (1-4 Bytes)
### For RHS V': multiset in the RHS in the active membrane U' [ V' ]_h^a'
Number of objects in V' (1-2 Bytes)
#### For each object
Object ID (1-4 Bytes)
Multiplicity (1-4 Bytes)

# For each rule block of environment
## For each rule
#### For each object in RHS
Object ID (1-4 Bytes)
Environment (1-4 Bytes)


#---- Initial multisets and sekeleton states

# For each environment
## For each membrane (membrane 0 for environment)
Charge (1 Byte: 2 bits, 6 bits reserved, using 0=0, +=1, -=2)
Number of different objects in the membrane (1-4 Bytes)
## For each object:
Object ID (1-4 Bytes)
Multiplicity (1-4 Bytes)

```

3.2 Input/output parsers

The ABCD-GPU simulator has been extended with a input module which is able of reading the above described binary format. Currently, the version is still experimental, and in order to decouple the input parser from the simulator structures, the module creates temporal data structures. Of course, in the final version, these structures should be avoided, making the reading of input files more efficient. The input PDP systems can be used both by the CPU and the GPU simulators.

On the other side, a first output module has been also developed. So far, the results were printed on screen. Today, it is possible to generate CSV (Comma Separated Values) files, which can be opened by statistics software such as R and Excel.

3.3 Preliminary results: a real ecosystem model

Thanks to this input module, we have been able to test our simulator with a real ecosystem model. We have chosen the model of the Bearded Vulture ecosystem in the Pyrenees, presented in [2], for its simplicity, allowing us to perform debugging and performance testing.

We have run two tests with 100 simulations, and 47 time steps (as required by the model for 10 years), using two GPUs from different generations: (a) Tesla C1060 (GT800 architecture), and (b) GeForce GTX550 (Fermi architecture). Unfortunately, we couldn't use our Tesla K40 (Kepler architecture) yet since some artifacts happened in the simulator, that requires more debugging and testing. Table 2 shows the preliminary results extracted from our simulators.

	Tesla C1060			GTX550		
	% CPU	% GPU	Acc	% CPU	% GPU	Acc
Phase 1	53.8%	56%	4.2x	53.8%	61.2%	12.6x
Phase 2	1.6%	2%	3.4x	1.6%	6.5%	3.5x
Phase 3	37%	9.4%	17.2x	37%	22.8%	23.3x
Phase 4	7.6%	32.6%	1.02x	7.6%	9.5%	11.6x
Total			4.38x			14.4x

Table 2. Profiling the Bearded Vulture ecosystem model (2008)

At first glance, the results show that the GPU (b) (GTX550) achieves better performance, up to 14.4x of speedup with respect to the sequential version, while the GPU (a) achieves barely 4.38x. As we have shown before, GPU (a) can achieve up to 7x of speedup with randomly generated P systems. However, we can see two new behaviors that were not expected before:

- Phase 2 is not the bottleneck: it is easy to see that the considered model has no competition for objects. Thus, phase 2 is not required for its simulation (the only mechanism carried out is the checking of remaining active blocks).

- Phase 4 is the bottleneck: this results is completely unexpected at first glance. However, a deeper analysis shows that since a few ratio of rules is executed, and most of them has common objects in the right hand side, the generation of objects is not performed completely efficiently. The main reason is the usage of atomic operations for adding new objects.

However, the behavior is completely different on GPU (b):

- Phase 2 is again the bottleneck. Although it is not required, the first phase of the kernel is run, which checks the remaining active blocks.
- The rest of phases are well accelerated. This demonstrates that the better bandwidth and the L2 cache of this GPU help to achieve better speedups when simulating PDP systems.

4 Road map

In what follows, we will discuss some of the future development lines under our consideration for next versions. In fact, this list is the road map for the ABCD-GPU project.

1. Making available Phase1_filter Kernel for GPUs with CC 3.x. In a Tesla K40 GPU, the consistency checking between rule blocks fails randomly.
2. More work on the obtaining the results from the GPU:
   (a) Use asynchronous copy from the GPU (it will require a double buffer for the multisets).
   (b) Filtering the multisets on the GPU, according to some parameters defined by the user.
   (c) Finishing the output module for binary files.
   (d) Development of a module that uploads the results into a database (interoperability with MeCoSim framework).
3. Phase 4 is becoming a bottleneck when running real ecosystem models. We have to change the scatter strategy into a gather one. That is, the threads reads the selection number for each rule, and create the corresponding objects. Why not using hybrid approaches, or a queue-levels approximation? That is, perform some atomics operations on shared memory, and then dump them to global memory. However, this is not easy, because the multiset structure might not fit into shared memory.
4. Phase 2 is also very slow.
   (a) Auto-detect if Phase 2 is really required. For example, if we know that the model has no competition of objects. Or if we analyze the number of active blocks remaining after Phase 1. Otherwise, we can skip it.
   (b) Compact active blocks after phase 1 for more efficiency.
   (c) Real (random) disorder of rule blocks (maybe taking some ideas from [7]). Currently, the random order is given by the thread scheduler (not a really random).

5. Avoid current synchronization of DCBA phases. That is, run all the phases with one single kernel (perhaps one global kernel which calls to __device__ versions of current kernels). It could be convenient to maintain the original version for GPUs that are used for the graphic system on the computer (limitation of kernel time).
6. In PDP systems, the working alphabet for the skeleton and for the environments are disjoint. That is, $\Gamma \cap \Sigma = \emptyset$. Therefore, we can work with all the communication rules apart from the virtual table.
7. Implement a variant of DCBA, called μDCBA:
   (a) It will allow to extract more parallelism within each environment. If we pre-calculate the group of rules that really depend on each other because they compete for objects, we will be able to apply DCBA separately to each group, i.e. more locally and in parallel. Moreover, there will be less resources to handle (and perhaps we would be able to move more data into shared memory, such as the multisets).
   (b) We define a transitive relation between rule blocks, called *competition*: block b_i *directly* compete for objects with block b_j if they have overlapping but not equal left-hand side. Moreover, if b_k directly compete with B_j, but not with B_i, then B_i and B_k also compete for objects (however, indirectly through B_j).
   (c) The idea is to define disjoint sets of rule blocks holding the competition relation, and apply DCBA to each one.
   (d) It would be desirable to use this variant only when the sets are balanced. We could also assign different "small" sets to one thread block.
8. Improving the data structures. Concerning the storage of objects appearing in the left-hand side of the rule (blocks), the current implementation on the GPU could be improved.
   (a) Current implementation is based on CSR representation of sparse matrices. All the objects of all the left-hand sides (LHS) are stored consecutively in a single array (see Figure 1), the array at the bottom). Each rule block has a corresponding entry in an array that contains a pointer (the index) to the array for LHS objects. This index says the first object of the LHS, and the end is given by the index of the next rule block. In this way, for example, ruleblock 4 has no object in LHS (what is weird, but can take place in our implementation), since idx5-idx4=0. However, rule block 2 has 3 objects, since idx3-idx2=3.
   (b) The problem is that each thread will iterate the objects of the LHS of each rule block. However, the access to the array of LHS is not coalesce, what is really bad for performance.
   (c) The idea would be to use the ELL representation of sparse matrices, and compact the objects in the LHS by chunks of consecutive rule blocks. Rule blocks with short LHS will need to replicate with dummy objects. In this way, the access is made coalesced (see Figure 2). The array of

objects should be linear (that's why there are lines connecting each chunk). Moreover, the array of lengths could be avoided.

(d) This will entail an interesting research. Is it good to do it or not? How much is the waste of memory? Can we use it only when LHS lengths are more or less balanced? Otherwise, can we use the COO representation? Is it really much more faster? What about Fermi architecture? Will their L2 cache improve the results?, is it good or not?

9. Implement model-oriented optimizations. That is, to analyze the PDP system model prior to the simulation and extract properties that will help to the efficiency. For example, test if there are competition for objects, inconsistent rule blocks, etc.
10. *Parallel P-Lingua*. Moreover, it would be interesting to let the model designer to provide the above mentioned properties to the simulator. For example, to allow in P-Lingua the usage of directives for defining modules of rules that can be executed in parallel, similarly to the `pragma` directives in OpenMP.
11. Hybrid simulation of PDP systems, by using both the CPU and GPU platforms at the same time, and implement a merge module of simulations at the end of the process.

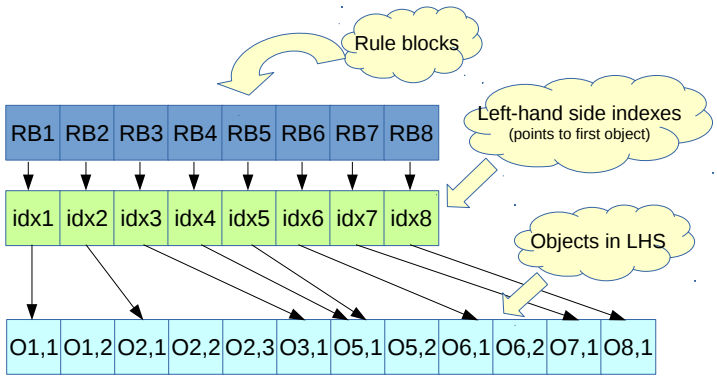


Fig. 1. Data structure for storing the information of left-hand sides of rule blocks, as currently implemented.

5 Conclusions

In this paper, we have shown the preliminary results related with the input module for the CPU/GPU simulators of PDP systems (ABCD-GPU). This module supports files with a binary format, which we have introduced here. The purpose

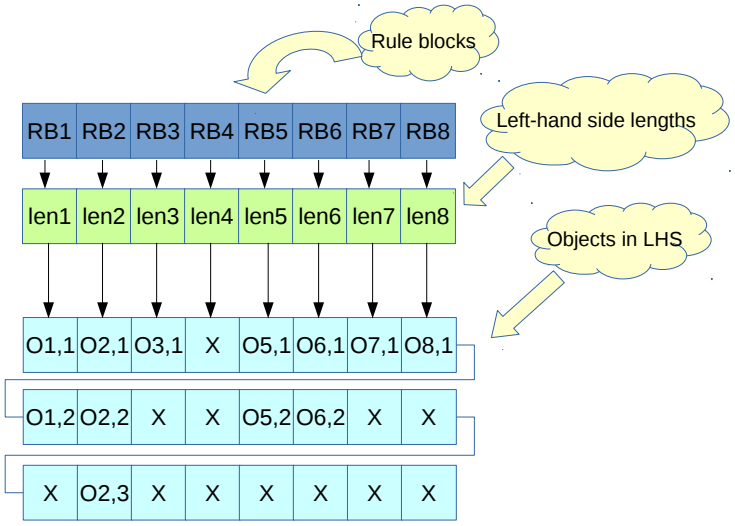


Fig. 2. Data structure for storing the information of left-hand sides of rule blocks, as proposed.

of using a restricted, binary format is for efficiency. We have also shown that simulating a real ecosystem model leads to different behaviors, depending on the GPU generation. Specifically, we have seen that phase 2 is the bottleneck for a Tesla C1060, while phase 4 is for a GTX 550 (Fermi).

Figure 3 shows the current structure of the project. The simulation engine implements DCBA in both multicore (CPU) and manycore (GPU) platforms. The input files are generated by pLinguaCore, which acts as a parser in the creation of binary files. The output files will be both in CSV and binary formats soon, and a module to upload results to a database is also under consideration. Moreover, the platform still support the input of randomly generated PDP systems and the output of corresponding profiling and debugging information, in order to conduct performance benchmarks to new versions of the simulator.

Finally, it is noteworthy that in this case, Parallel Computing is not only used to get faster solutions, but also, to obtain better results, because it enables the users to run DCBA-based simulations in an affordable time.

Acknowledgements

The authors acknowledge the support of the project TIN2012-37434 of the "Ministerio de Economía y Competitividad" of Spain, co-financed by FEDER funds. Miguel A. Martínez-del-Amor also acknowledges the support of the Alain Ben-

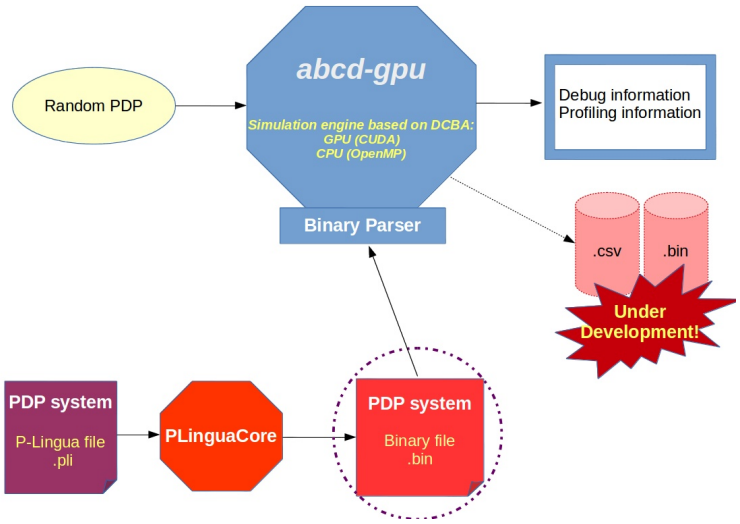


Fig. 3. Structure of ABCD-GPU project.

soussan Fellowship programme of ERCIM, and the hosting institution Fraunhofer IIS.

References

1. M. Cardona, M.A. 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.
2. M. Cardona, M.A. Colomer, A. Margalida, I. Pérez-Hurtado, M.J. Pérez-Jiménez, D. Sanuy. A P system based model of an ecosystem of some scavenger birds, *LNCS*, **5957**, (2010), 182–195.
3. M.A. Colomer-Cugat, M. García-Quismondo, L.F. Macías-Ramos, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez–Jiménez, A. Riscos-Núñez, L. Valencia-Cabrera. Membrane system-based models for specifying Dynamical Population systems. In P. Frisco, M. Gheorghe, M.J. Pérez-Jiménez (eds.), *Applications of Membrane Computing in Systems and Synthetic Biology. Emergence, Complexity and Computation series*, **Volume 7**. Chapter 4, pp. 97–132, 2014, Springer Int. Publishing.
4. M.A. 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)
5. M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M.J. Pérez-Jiménez, Agustín Riscos-Núñez. An overview of P-Lingua 2.0, *LNCS*, **5957** (2010), 264–288.
6. M. García-Quismondo, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez. Probabilistic Guarded P systems: A new formal modelling framework. *LNCS*, **8961** (2014), 194–214.

7. A. Gastalver-Rubio. *Simulation of probabilistic P systems on GPUs*, Final Research Project, University of Seville, September 2012.
8. M. Harris. Mapping computational concepts to GPUs, *ACM SIGGRAPH 2005 Courses*, NY (USA), 2005.
9. D. Kirk, W. Hwu. *Programming Massively Parallel Processors: A Hands On Approach*, MA (USA), 2010.
10. M.A. Martínez-del-Amor. *Accelerating Membrane Systems Simulators using High Performance Computing with GPU*, Ph.D. thesis, University of Seville, May 2013.
11. M.A. Martínez-del-Amor, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez. Simulating P Systems on GPU Devices: A Survey. Fundamenta Informaticae, **136**, 3 (2015), 269–284
12. M.A. Martínez-del-Amor, I. Karlin, R.E. Jensen, M.J. Pérez-Jiménez, A.C. Elster. Parallel simulation of probabilistic P systems on multicore platforms, *Proc. Tenth Brainstorming Week on Membrane Computing*, Sevilla, Spain, **Volume II**, 2012, pp. 17–26.
13. M.A. Martínez-del-Amor, I. Pérez-Hurtado, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, A. Romero-Jiménez, C. Graciani, A. Riscos-Núñez, M.A. Colomer, M.J. Pérez-Jiménez. DCBA: Simulating Population Dynamics P Systems with Proportional Object Distribution, *LNCS*, **7762** (2012), 27–56.
14. M.A. Martínez-del-Amor, I. Pérez-Hurtado, A. Gastalver-Rubio, A.C. Elster, M.J. Pérez-Jiménez. Population Dynamics P systems on CUDA. *10th Conference on Computational Methods in Systems Biology*, *LNBI*, **7605** (2012), 247–266.
15. J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips. GPU Computing, *Proceedings of the IEEE*, **96**, 5 (2008), 879-899.
16. 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
17. Gh. Păun, Gz. Rozenberg, A. Salomaa (eds.). *The Oxford Handbook of Membrane Computing*, Oxford University Press, USA, 2010.
18. I. Pérez-Hurtado, L. Valencia-Cabrera, M.J. Pérez-Jiménez, M.A. Colomer, A. Riscos-Núñez. MeCoSim: A general purpose software tool for simulating biological phenomena by means of P Systems, *Proceedings IEEE Fifth International Conference on Bio-inpired Computing: Theories and Applications (BIC-TA 2010)*, **volume I** (2010), pp. 637–643.
19. M.J. Pérez-Jiménez, F.J. Romero-Campero. P systems, a new computational modelling tool for Systems Biology. *Transactions on Computational Systems Biology VI. LNBI*, **4220** (2006), 176–197.
20. M.J. Pérez-Jiménez, F.J. Romero-Campero. A model of the Quorum Sensing system in Vibrio fischeri using P systems. *Artificial Life*, **14**, 1 (2008), 95–109.
21. *GPGPU organization*. `http://www.gpgpu.org`
22. *OpenMP webiste*. `http://www.openmp.org`
23. *NVIDIA CUDA website*, 2015. `https://developer.nvidia.com/cuda-zone`
24. *The MeCoSim web page*. `http://www.p-lingua.org/mecosim`
25. *The P-Lingua web page*. `http://www.p-lingua.org`
26. *The PMCGPU project*, 2013. `http://sourceforge.net/p/pmcgpu`

Some Quick Research Topics

Gheorghe Păun

Institute of Mathematics of the Romanian Academy
PO Box 1-764, 014700 Bucureşti, Romania
gpaun@us.es, curteadelaarges@gmail.com

Summary. Some research topics are suggested, in a preliminary form, in most cases dealing with (somewhat nonstandard) extensions of existing types of P systems.

1 Introduction

Almost at every edition of the Brainstorming, lists of open problems and research topics were circulated – the present note should be seen as a step in this tradition, part of a ritual. The interested reader can check the Brainstorming volumes, or, for a more systematic list of research suggestions, (s)he should consult the "mega-paper" [7]. Of course, many open problems and research topics can be found in [11] and at the domain web site [14].

The list which follows contains several suggestions (this time I do not count them...) which might look strange at the first sight, but which have their motivation, they are natural at least from a mathematical point of view. Just one example: multisets with negative multiplicity seems to be artificial objects, but they appear already in computer science, see, e.g., [2].

Of course, the reader is assumed to be familiar with membrane computing, so that the presentation is minimal, both in what it concerns the details and the references.

2 "Negative" Extensions

It is about negative numbers, as already mentioned above...

In several places in membrane computing we have functions $f : X \longrightarrow \mathbf{N}$. The multiplicity of objects in multisets, the time associated with rules in timed P systems, weights associated with synapses in SN P systems, life duration of objects in P systems with object decay (what else?) are of this form. At least as a mathematical challenge, we can try to extend these functions to $f : X \longrightarrow \mathbf{Z}$, where $\mathbf{Z}$ is the set of integers, both positive and negative.

Negative multiplicities can be interpreted in various ways – see also [2] where such multisets are considered. A connection can be made with anti-spikes in SN P systems, in general, with anti-matter, in the sense of [9] – with the interesting observation that anti-matter (plus the priority of the annihilation rules over evolution rules, with the annihilation rules applied in no time) is useful, it speeds-up the functioning of P systems, see, e.g., [6]. What happens when the annihilation does not have priority?

Of course, passing to negative integers in other cases raises problems concerning the definition of the functioning of the systems. For instance, what means to apply a rule having associated a negative time? Moving back in time the produced objects is a possibility, but this means separating the objects associated to several time moments, which would imply that objects can travel back and forth in time and only objects of the same time can react. What happens with the observer time? Naturally, it has to grow continuously with unit steps. How the internal times of objects interact with the external time of the observer? Finding a good definition of the computations in such a system is already a first task. I forecast, however, that the interplay of the internal and the external times will lead to interesting results. Remember also that the observer plays a crucial role in computations – see, e.g., [3].

Of course, one further extension is to replace natural numbers with numbers in larger classes than $\mathbf{Z}$, why not?, with real numbers or even with complex numbers. The results are not easy to forecast, but the next section can give some hints and motivation.

3 Hypercomputing

Going beyond the "Turing barrier" is a constant preoccupation of computer scientists. I recall only three surveys, [4], [8] and [13]. In membrane computing there are only a few attempts to achieve a hypercomputation power, see [1] and [12].

No result of this type was reported for SN P systems, in spite of the fact that the motivation of these systems comes from the brain, and the brain is (supposed to be) a non-Turing "computing device".

The problem is much more general: in the hypercomputability area there are several tricks (Martin Davis would say even "dishonest tricks", as the power is introduced from the beginning in the system and then we prove that the system is powerful...) used in order to increase the power of of the obtained machineries beyond the power of usual Turing machines. I list the ten ideas mentioned in [8]: 1. O-machines (Turing machine with oracles), 2. Turing machines with initial inscriptions (infinitely many cells of the input tape contain already symbols), 3. Coupled Turing machines (input channels are provided which bring information into the machine during the computation, as a possibly non-recursive sequence of bits), 4. Asynchronous networks of Turing machines (timing functions are provided, not necessarily recursive), 5. Error prone Turing machines (the errors appear according

to a function, again, not necessarily recursive), 6. Probabilistic Turing machines (not so easy to describe), 7. Infinite state Turing machines (infinite sets of states and transitions, but only a finite number of transitions leading from a given state), 8. Accelerated Turing machines (each step takes half of the time needed for performing the previous step – like in [1]), 9. Infinite time Turing machines, 10. Fair non-deterministic Turing machines.

Summarizing: infinite resources, specification or functioning, real numbers, non-recursive functions involved in the computations. Similar tricks are described in [13].

Which of these ideas can be (naturally) extended to P systems? Which of them have even a remote biological support/motivation? Which if these ideas can also speed-up the computations so that computationally hard problems could be solved in a polynomial time? (At least the acceleration is doing it, as in two external time units any computation halts...)

4 Extensions of SN P Systems

Many modifications of the structure and the functioning of SN P systems can be imagined. Here are a few of them.

Consider SN P systems as devices computing functions $f : \mathbf{N}^k \longrightarrow \mathbf{N}^l$, for some $k, l \geq 1$ (take k input neurons and l output neurons etc.). What about the efficiency of this way to compute functions? Any application (similar to the application of numerical P systems in robot control)?

What about SN P systems with astrocytes, with the astrocytes controlling the flow of spikes not according to thresholds associated with them, but using regular expressions in a similar way as the spiking rules use them: couples $(E_i, action_i)$ are associated with the astrocytes and $action_i$ is performed when the number of spikes on the controlled synapses belongs to $L(E_i)$.

In [5], the notion of *white holes* is introduced in membrane computing, as regions where rules which expel all objects are present. What about SN P systems with "white hole neurons", i.e., containing spiking rules $a^n \to a^n$ for all n? Systems where all neurons are of this type have an interesting behavior: just consider the SN P system in Figure 1, with all neurons being white holes, and follow its functioning. Three increasing sequences of numbers describing the number of spikes in the three neurons are obtained, with an intriguing growth. Can you characterize these sequences? Ca you compute in this way known sequences, such as the Fibonacci one?

Finally, let us return to the brain. Usually, it is considered as working at two levels, the conscious one and the subconscious one. One model of the brain functioning claims that the cortex formulates problems to the subconscious level, this one works "silently", in a great extent nondeterministically, proposing solutions to the conscious level. The cortex evaluates the proposed solutions, accepts the good one, if any, or returns the problem to the "lower" level, and so on, until either the problem is solved or the problem is abandoned or... the brains gets into troubles.

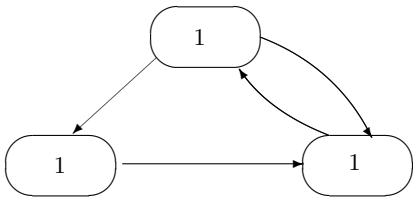


Fig. 1. An SN P system composed of white holes

Can this strategy be implemented in terms of SN P systems? Which is its computing power and, also, its computing efficiency? Nondeterminism is powerful, one can expect interesting results, provided that such an "SN P brain", with two modules (somewhat like in dP systems, [10]), a nondeterministic one and a deterministic one, connected with the environment/user, could be defined.

5 Numerical P Systems

This is a class of P systems which I feel still keeps undiscovered many nice results and, possibly, applications. Just to recall the attention about them, I am formulating here two problems, one theoretical (and also formulated in other contexts) – (1) consider numerical P systems as decidability devices and investigate their efficiency (complexity classes), both in the original setup and, if they are not efficient enough, after introducing membrane division or other tools for producing an exponential working space in linear time – and one applicative: (2) these systems were used to build controllers for robots. 2D robots. What about passing to 3D robots? This is mainly a programming issue, but it could find good applications – for instance, in controlling the drones, so popular in the last time.

6 Final Remarks

References

1. C. Calude, Gh. Păun: Bio-steps beyond Turing. *BioSystems*, 77 (2004), 175–194.
2. J. Carette, A.P. Sexton, V. Sorge, S.M. Watt: Symbolic domain decomposition. *AISC/Calculemus/MKM 2010* (S. Autexier et al., eds.), LNAI 6167, Springer, 2010, 172–188.
3. M. Cavaliere, P. Leupold: Evolution and observation. A new way to look at membrane systems. *Proc. WMC 2003*, LNCS 2933, Springer, 2003, 70–87.
4. B.J. Copeland: Hypercomputation. *Minds and Machines*, 12, 4 (2002), 461–502.
5. E. Csuhaj-Varjú, M. Gheorghe, Gy. Vaszil, M. Oswald: P systems for social networks. *Ninth Brainstorming Week on Membrane Computing*, Sevilla, 2011, 113–124.

6. D. Díaz-Pernil, F. Peña-Cantillana, A. Alhazov, M.A. Gutiérrez-Naranjo, R. Freund: Antimatter as a frontier of tractability in membrane computing. *Fundamenta Informaticae*, 134, 1-2 (2014), 83–96.
7. M. Gheorghe, Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg: Frontiers of membrane computing: Open problems and research topics, *Intern. J. Found. Computer Sci.*, 24, 5 (2013), 547–623 (first version in *Proc. Tenth Brainstorming Week on Membrane Computing*, Sevilla, January 30 – February 3, 2012, vol. I, 171–249).
8. T. Ord: *Hypercomputation: Computing More Than the Turing Machine*. Honours Thesis, Department of Computer Science, University of Melbourne, 2003.
9. Gh. Păun: Four (somewhat nonstandard) research topics. *12th BWMC*, Sevilla, February 2014, 305–309.
10. Gh. Păun, M.J. Pérez-Jiménez: Solving problems in a distributed way in membrane computing: dP systems, *Int. J. of Computers, Communication and Control*, 5, 2 (2010), 238–252.
11. Gh. Păun, G. Rozenberg, A. Salomaa, eds.: *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.
12. P. Sosík, O. Valik: On evolutionary lineages of membrane systems. *Membrane Computing, International Workshop, WMC6, Vienna, Austria, 2005, Selected and Invited Papers*, LNCS 3850, Springer, Berlin, 2006, 67–78.
13. A. Syropoulos: *Hypercomputation: Computing Beyond the Church-Turing Barrier*. Springer, Berlin, 2008.
14. The P Systems Website: `http://ppage.psystems.eu`.

Looking for Computers in the Biological Cell. After Twenty Years*

Gheorghe Păun

Institute of Mathematics of the Romanian Academy
PO Box 1-764, 014700 Bucureşti, Romania
gpaun@us.es

1 Preliminary Cautious and Explanations

The previous title needs some explanations which I would like to bring from the very beginning.

On the one hand, it promises too much, at least with respect to my scientific preoccupations in the last two decades and with respect to the discussion which follows. It is true that there are attempts to use the cell as it is (bacteria, for instance) or parts of it (especially DNA molecules) to compute, but a research direction which looks more realistic, at least for a while, and which has interested me, is to look in the cell for *ideas* useful to computer science, for *computability models* which, passing from biological structures and processes to mathematical models, of a computational type, can not only ensure a better use of the existing computers, the electronic ones, but they can also return to the starting point, as tools for biological investigations.

Looking to the cell through the mathematician-computer scientist glasses, this is the short description of the present approach, and in this area it is placed the personal research experience which the present text is based on.

On the other hand, the title announces already the autobiographical intention. Because a Reception Speech is a synthesis moment, if not also a career summarizing moment, it cannot be less autobiographical than it is, one uses to say, any novel or poetry volume. And, let us not forget, the life in the *purity and signs world* (a syntagma of Dan Barbilian-Ion Barbu, a Romanian mathematician and poet)

* This is the English version of the Reception Speech I have delivered on October 24, 2014, at the Romanian Academy, Bucharest, and printed by the Publishing House of the Romanian Academy in December 2014. The answer to this speech was given by acad. Solomon Marcus. Some ideas and some paragraphs of the text have appeared, in a preliminary version, in the paper Gh. Păun "From cells to (silicon) computers, and back", published in the volume *New Computational Paradigms. Changing Conceptions of what is Computable* (B.S. Cooper, B. Lowe, A. Sorbi, eds.), Springer, New York, 2008, 343–371.

of mathematics assumes/imposes a great degree of loneliness, as acad. Solomon Marcus reminded us in his Reception Speech (2008), while the loneliness (it is supposed to) make(s) us wiser, but it also moves us farther from the "world-as-it-is", so that at some stage you no longer know how much from a mathematician belongs to the "world" and how much belongs to mathematics. That is why we can consider that a mathematician is autobiographical both in his/her theorems and in the proofs of his/her theorems, as well as in the models (s)he proposes.

Looking back in time, I find that I am now at the end of two periods of two decades each, the second one completely devoted to "searching computers in the cell", while the first period was almost systematically devoted to preparing the tools needed/useful to this search. The present text describes mainly the latter of these two periods.

2 Another Possible Title

For a while, I had in mind also another title, much more general, namely, *From bioinformatics to infobiology*. It was at the same time a proposal and a forecast, and the pages which follow try to bring consistency to this forecast. Actually, the idea does not belong to me, in several places there were discussions about a new age of biology – the same was predicted also for physics – based on using the informational-computational paradigms, if not also based on further chapters of mathematics, not developed yet. The idea is not to apply computer science, be it theoretical or practical, to biology, but to pass to a higher level, to a systematic approach to biological phenomena in terms of computability, with the key role of information being understood. Attempts which illustrate this possibility, also advocating for its necessity, can be found in many places, going back in time to Erwin Schrödinger and John von Neumann. In a recent book, *Infobiotics. Information in Biotic Systems* (Springer-Verlag, 2013), Vincenzo Manca also pleads for "a new biology", which he calls *infobiotics*, starting from the observation that *the life is too important to be investigated only by biologists*. I would reformulate in more general terms: *the life is too important and too complex to be investigated only by the traditional biology* – with the important emphasis that exactly the biologists are called to not only benefit, but also to provide consistency to infobiology. Together with the computer scientists and, more plausibly and more efficient, borrowing from the computer scientists ideas, models, techniques, making them their own ideas, models and techniques and developing them. There is here also a plead for multi-trans-inter-disciplinarity (starting with the higher education), but also a warning: this is not only possible, but, it seems, this is also at the right time, on the verge to become urgent.

3 The Framework

Having in mind the title before and looking for an "official" enveloping area, the first syntagma which appears is *natural computing* – with the mentioning, however, that it covers a very large variety of research areas, including the bioinformatics and also moving towards infobiology. For an authoritative description, let us consider the *Handbook of Natural Computing*, edited by Grzegorz Rozenberg, Thomas Bäck and Joost N. Kok, published, in four large volumes, by Springer-Verlag, in 2012. From the beginning of the Preface, we learn that *Natural Computing is the field of research that investigates human-designed computing inspired by nature as well as computing taking place in nature, that is, it investigates models and computational techniques inspired by nature, and also it investigates, in terms of information processing, phenomena taking place in nature.* The generality is obvious, adding to the desire to identify in nature (important: not only in biology) ideas useful to computer science, a position which, as I have already said, although it is not completely new, if it is systematically applied, it can lead to a new paradigm in biological research and in other frameworks too: the informational approach, hence surpassing the traditional approach, the chemical-physical one.

The idea was formulated also in other contexts: the computational point of view (to the information processing one adds the essential aspect of computability) can also lead to a new physics – among others, this is the forecast of Jozef Gruska, an active promoter of quantum computing and a *pioneer of computer science*, rewarded with a diploma of this kind by the Computer Science Section of IEEE (let us remind the fact that also Grigore C. Moisil has been awarded such a diploma and title). On the same idea is grounded also the collective volume *A Computable Universe. Understanding and Exploring Nature as Computation*, edited by Hector Zenil and published by World Scientific in 2013. Many chapters have exciting-enthusiastic titles: *Life as Evolving Software, The Computable Universe Hypothesis, The Universe as Quantum Computer*, etc. There also is a chapter-long Preface, by sir Roger Penrose, not always fully agreeing with the hypotheses from the book.

Actually, also the *Handbook of Natural Computing* mentioned before includes the quantum computing among the covered domains. Here is its contents (the main sections, without specifying the chapters): *Cellular Automata, Neural Computation, Evolutionary Computation, Molecular Computation, Quantum Computation, Broader Perspective – Nature-Inspired Algorithms, Broader Perspective – Alternative Models of Computation.* There is some degree of "annexationism" here (for instance, cellular automata are not too much related to the biological cells), but let us mention that the section devoted to the molecular computation covers DNA computing, membrane computing, and gene assembly in ciliates, the former two areas being exactly what we are interested in here.

4 The Popularity of a Domain

Even remaining only at the editorial level and at the level of conferences (without considering also the research projects, hence the financial support), one can say that there is a real fashion of natural computing – more general, of unconventional computing, more restricted, of bioinformatics.

Here are only a few illustrations. Springer-Verlag has a separate series of books dedicated to natural computing monographs, named exactly in this way, it also has a journal, *Natural Computing*. There is an international conference, *Unconventional Computing*, which became, in the last year, slightly pleonastic, *International Conference on Unconventional Computation and Natural Computation*. *BIC-TA*, that is, *Bio-Inspired Computing – Theory and Applications*, is another conference of a real success, at least in what concerns the number of participants, a meeting whose format I has established, together with colleagues from Spain and China, in 2005, and which is organized since then each year, in China or in the neighboring countries – this can explain the massive participation, as the Chinese researchers are very active in this area.

We have reached the closest upper envelope of the area discussed here: the computability inspired from biology. It is important to note that the term "bioinformatics" (bio-computer science) has a double meaning, with, one can say, a geographical determination. In the "pragmatic West", it mainly covers the computer science applications to biology (in the "standard" scenario, one goes from problems towards tools, without too much theory). In Europe, both directions of influence are taken into consideration, from biology towards computer science and conversely. Although it is just natural that both these two research directions should be developed together, in collaboration, the reality is not always so. In search of solutions for current questions, some of them really urgent, for instance from the biomedical area, mathematics and computer science often provide tools prepared and developed in other areas. The typical example is that of differential equations, with a glorious history in physics, astronomy, mechanics, meteorology, and which are "borrowed" to biology, not always checking their adequacy. I will return to this issue, of a great importance for promoting new tools for biology.

"The European strategy", of constructing a mathematical theory which looks for applications after it is developed, has its appeal and advantages – but also its traps. Being an European, being a mathematician, I have been especially attracted by this strategy, but, in time, I became more and more interested by "reality", by applications.

5 What Means to Compute?

Let us come back to the title, with the fundamental question concerning the definition of the notions of computation and computer. This is a question of the same type as "what is mathematics?", with many different answers, none of them

complete, none of them fully agreed. If information processing is a computation, then we can see computations everywhere. With a very important detail, hidden in the previous formulation: *we can see*. We, the human beings. Otherwise stated, an observer, which interprets a process as being a computation. I do not want to push the discussion as far as asking questions of the form "does a tree which falls in the water of a lake, in the middle of an uninhabited forest, produce any noise, taking into account that there is nobody there to hear it?" – I mention the fact that this question was the topic of a paper accepted some years ago by a conference on unconventional computing, that is why I recall it – and, on the other hand, I also do not want to involve God in this issue, the omnipresent, omniscient, omnipotent God, considered as an universal observer (at least, not for observing computations, maybe only for noticing noises in desert forests...).

A somewhat exaggerated but rather suggestive example is that of a drop of liquid which falls freely in the air. During its falling down, the drop instantaneously "solves" on its surface, by the form it takes, complex differential equations. Is this a computation? I would not go so far. Similarly with what happens continuously in the cells of a leaf or of the human body, at the biochemical or even at the informational level.

The idea of a computation as a process considered so by an observer is not at all new. One of the conclusions of the John Searle book *The Rediscovery of the Mind* (MIT Press, 1992), is exactly this – a computation is not an intrinsic property of a process, but it is *observer-relative*.

A very suggestive formulation of the role of the observer in considering a process as being a computation belongs to Tommaso Toffoli. The quotation which follows appears in a paper with a statement-title: "Nothing makes sense in computing except in the light of evolution" (*Journal of Unconventional Computing*, vol. 1, 2005, pages 3–29).

"We'we just seen that it is not useful to call *computation* just any nontrivial yet somewhat disciplined coupling between state variables. We also want this coupling to have been *intentionally* set up for the purpose of predicting or manipulating – in other words, for *knowing* or *doing* something. This is what shall distinguish bona–fide computation from other intriguing function–composition phenomena such as weather patterns or stock–exchange cycles. But now we have new questions, namely, 'Set up by whom or what?', 'What is it good for?', and 'How do we recognize intention?'

Far from me to want to sneak animistic, spiritualistic, or even simply anthropic considerations into the makeup of computation! *The concept of computation must emerge as a natural, well–characterized, objective construct, recognizable by and useful to humans, Martians and robots alike*" (my emphasis, Gh.P.).

Toffoli's questions should be remembered and discussed, but they move us far from our subject. Let us return to John Searle, namely, to a more technical reading of the idea of implying an observer in the definition of a computation. This was the approach of Matteo Cavaliere and Peter Leupold, both of them my students in the PhD school in Tarragona, Spain, the former one being my first PhD student

there. They have published a series of papers with this subject, I cite here only a recent one, by Peter Leupold, "Is computation observer-relative?", presented at the *Sixth Workshop on Non-Classical Models of Automata and Applications*, Kassel, Germany, July 2014. Actually, in the Cavaliere-Leupold approach there appear two observers, one of them – we can call it observer of the first order – following a simple process and "translating" the steps of the process in an external language, and the second observer, closer to the Searle-Toffoli observer, interpreting as a computation the results of the activity of the first observer. Cavaliere and Leupold consider a series of process-observer (of the first order) pairs which, separately, have a reduced (computing) power, but which, together, lead to the computing power of Turing machines from the point of view of the external observer.

6 The Turing Machine

Let us start also from another direction, from the meaning given by mathematics to the notion of computation. Already from the thirties of the previous century we have a definition of what is computable, the answer Alan Turing gave to the question "what is mechanically computable?", formulated by David Hilbert at the beginning of the twentieth century. "Mechanically", i.e., "algorithmically" in our today reformulation. There were many proposed answers (I recall only the recursive functions and the lambda-calculus), given by great names of mathematics-computer science (I recall here only Alonzo Church, Stephen Kleene, Emil Post), but the solution given by Turing, what we call now *Turing machine*, has been accepted as the most convincing one (a fact certified even by the highly exigent Gödel). This is now in computer science the standard model of an algorithm (I have not said *definition*, because we have only an intuitive understanding of the idea of an *algorithm*, but we can say that in this way we have a definition of what is *computable*).

Without entering into details, I mention only that Hilbert's problem was more general. It started from the algorithmic resolution of diophantine equations, those with integer coefficients (the tenth problem in Hilbert's 1990 list), but in its later (in 1928) formulation Hilbert was saying that "the *Entscheidungsproblem* [the decision problem in the first order logic] would be solved if we would have a procedure which, for any logical expression we would decide through a finite number of operations whether it is satisfiable... *Entscheidungsproblem* should be considered the main problem of the mathematical logic". At this general level, Gödel theorems answer negatively Hilbert's program. Negative answers gave also Church and Turing, while Hilbert tenth problem was solved – also negatively – in 1970, by Yurii Matijasevich (after many efforts of several mathematicians: Julia Robinson, Hilary Putnam, Martin Davis). Turing not only gives a negative answer, moreover, he not only defines "the frontiers of computability", but he also produces an example of a problem placed behind these frontiers, a problem which is not algorithmically solvable, *the halting problem* (there is no algorithm, hence a Turing machine, which, taking as input an arbitrary Turing machine, can tell us, in a finite number of

steps, whether the given input machine halts or not when starting from an arbitrarily given initial data). To the halting problems reduce, directly or indirectly, most if not all undecidability results obtained after that.

The Turing machine is so important for computer science, including the natural/unconventional computability, that it is worth discussing it a little bit more.

7 Some More Technical Details

It is interesting to note that when he defined his "machine", Turing explicitly started – he states this at the beginning of the paper – from the attempt to abstract the way a human being computes, reducing to the minimum the resources used and the operations made. In this way, in the end one obtains a "computer" which consists of a potentially infinite *tape*, bounded to the left, divided in *cells* where one can write *symbols* from a given finite *alphabet*; these symbols can be read and rewritten by a *read-write head*, which can "see" only one cell, can read the symbol written there, can change it, then it can move to the neighboring left or right cell or it can stay in the same place; the activity of the read-write head is controlled by the finitely many *states* of a *memory*. Thus, we get *instructions* of the form $s_1a \to s_2bD$ with the following meaning: in state s_1, with the head reading symbol a, the machine passes to state s_2, modifies a to b (in particular, a and b can be identical), and moves the read-write head as indicated by D. One starts with the tape empty, with the machine in a special initial state s_0; one writes the initial data on the tape (for instance, two numbers which have to be multiplied), one places the head on the first cell of the tape (the leftmost one), and one follows the instructions of the (e.g., multiplication) "program" until one reaches a *final state* and the machine *halts*, no further instruction can be applied. The contents of the tape at that moment is the result of the computation.

Extremely reductionistic, but this is the most general model of an algorithmic computation – because no previous definition of what is computable is known, this assertion is only a hypothesis, called the *Turing-Church thesis*. However, what made Turing machine so attractive were not only the *simplicity* of its definition and its *power* (it was proved that the Turing machine can simulate any other computing model), but also its *robustness* (the computing power is not changed if we add further ingredients to the architecture or to the functioning, such as further tapes, if we infinitely prolong the tape also to the left, if we consider non-deterministic computations, etc.), and, mainly, the existence of *universal Turing machines*: there exists a fixed Turing machine TMU which can simulate any particular Turing machine TU, in the following sense. If a code of the machine TM (let us denote it by $code(TM)$) is placed on the tape of TMU together with an input x of TU, then TMU will provide the same result as that provided by TU when starting from input x. A little bit more formally (but still omitting some details – e.g., codifications), we can write $TMU(code(TM), x) = TM(x)$. And Turing proved that there are universal Turing machines. This was done in 1936, in the paper "On

computable numbers, with an application to the *Entscheidungsproblem*", published in *Proceedings of the London Mathematical Society*, Ser. 2, vol. 42, 1936, 230–265, with an erratum in vol. 43, 1936, pages 544–546.

This is the "birth certificate" of the today computers, consequently called of Turing-von Neumann type (in forties, when he has participate in the designing of the first programmable electronic computers, von Neumann was influenced by Turing ideas).

A couple of things deserve to be mentioned: the code of machine TM is the program to be executed/simulated on TMU, starting from the data x; the instructions of TMU form the "operating system" of our "computer"; the data and the programs are written in the same place, on the tape of the universal Turing machines (in the "computer memory") – from here it follows the possibility to process programs in the same way as we process data, hence the vulnerability of programs to computer viruses.

Several details are important from the point of view of natural computing. The work of the Turing machine is sequential, in each time unit one performs only one instruction. In many places in nature, if not in most of them, in particular, in biology, the processes develop in parallel, which is a very appealing feature for computer science, but these processes are not necessarily synchronized, which, in turn, raises difficulties for computer science.

There also are further differences between Turing machines, the "biological computers", and the electronic computers, but we will discuss these differences later.

For the time being we keep in mind that in what follows *to compute* has the meaning suggested by Turing machines: there are an input and an output, between them there is an algorithm which bridges inputs and outputs, and the result of a computation is obtained in the moment when the machine halts. Very restricted, but precise. With such a framework at hand, we can look around for computations, moreover, we can investigate them in a well developed context, the computability theory – actually, a set of several theories, such as automata theory, formal language (grammar) theory, complexity theory and others.

8 Computer Science and Mathematics

This is maybe the place to remind a debate which motivated many discussions and points of view, often biased, concerning the relation between computer science and mathematics. Discussions of this kind have appeared also in the Romanian Academy, they appeared in the higher education (in the sixties-seventies of the last century, at the time of sputniks and hydroelectrical plants, we had many faculties of "mathematics-mechanics", now mechanics was replaced by computer science), the issue is often debated in mass media. Actually, the context is larger, sometimes it is put in question the relation of mathematics with other sciences, with school education, with the society. There are persons who are proud of the fact that they

"were not good in math". It was even expressed the opinion that mathematics is a luxury, a "national fetish" (this expression has recently appeared in the title of a Romanian newspaper article), in short, that one makes too much fuss of mathematics and one teaches too much mathematics. This opinion is getting more and more popular, supported also by the ubiquitous penetration of computers ("we no longer need to know the multiplication table, the computer knows it for us").

Of course, there is a problem with the mathematical education. *What, how much*, and, mainly, *how*? – and there also are further questions; we can find them, often also together with solutions, in the papers dedicated in the last years by professor Solomon Marcus to education. The problem cannot be solved from bottom up, the mathematicians involved in research and in higher education should consider it – this is, for instance, the opinion of Juraj Hromkovic, from ETH Zürich, formulated in an article published in the *Curtea de la Argeş* journal (`www.curteadelaarges.ro`, August 2014), based on the practical activity in this respect carried out in the institute where J. Hromkovic works (among others, this activity was materialized in mathematical school books of a new type). In general, the mathematicians should enter public debates and plead for their discipline, mainly they are guilty if the domain loses its popularity. It is true that for a mathematician mathematics is a great game, which, like any game, has an intrinsic rewarding, in the very development of the game, therefore it is natural that the interest for "popularization" is low among mathematicians, but the persons who are proud of their mathematical infirmity, be it real or only claimed, are always much more visible, more vocal, and the danger which comes from this is obvious.

Having in mind only the relation between mathematics and computer science, let us mention that the theoretical computer science, placed at the intersection of the two domains, is often considered by computer scientists as a part of mathematics, and by mathematicians as a part of computer science. Sometimes, theoretical computer science has problems even inside computer science – as it happens also with other theoretical branches of science with a strong practical dimension. Of course, all these are false problems by themselves, but they can have unpleasant practical consequences.

Being of the same opinion, I cite here an authoritative voice, that of Edsger W. Dijkstra, one of the classics of computer science, in fact, of the practical computer science: it is sufficient to remind that during sixties he has worked for implementing the Algol language in the Amsterdam Mathematical Center, and, furthermore, he was the promoter of structured programming, well-known among the software practitioners. (Maybe it is good to add here that the first four years after graduation I have intensively written computer programs, in Cobol and Fortran, realizing even the programs for computing the salaries of the workers in a large Bucharest factory – I remember, therefore, what practical computer science means...)

"The end of computer science?", asks Dijkstra, ironically-rhetorically, already in the title of a note published in *Communications of the ACM* (vol. 44, March 2001, page 92), which starts with the following phrase: "In academia, in industry, and in the commercial world, there is a widespread belief that computing science as

such has been all but completed and that, consequently, computing has matured from a theoretical topic for the scientists to a practical issue for the engineers, the managers, and the entrepreneurs." Then, it adds: "This widespread belief, however, is only correct if we identify the goals of computer science with what has been accomplished and forget those goals that we failed to reach, even if they are too important to be ignored."

Much more explicit is Dijkstra in the speech he delivered in May 2000 at a symposium (*In Pursuit of Simplicity*) organized at the Austin-Texas University, on the occasion of his retirement. The title of the speech (published in *Information Processing Letters*, vol. 77, February 2011, pages 53–61) is relevant: "Under the spell of Leibniz's dream". I recall a couple of aphoristic phrases: "What is theoretically beautiful tends to be eminently useful." "In the design of sophisticated digital systems, elegance is not a dispensable luxury but a matter of life and death, being a major factor that decides between success and failure." "These days there is so much obsession with application that, if the University is not careful, external forces, which do make the distinction [between theory and practice], will drive the wedge between *theory* and *practice* and may try to banish the *theorists* to a ghetto of separate departments and separate buildings. A simple extrapolation will tell us that in due time the isolated practitioners will have little to apply; this is well-known, but has never prevented the financial mind from killing the goose that lays the golden eggs. The worst thing with institutes explicitly devoted to applied science is that they tend to become institutes of second-rate theory."

The plead to place us under the spell of Leibniz is obvious, because, Leibniz said it, "the symbols direct the reason", and, after having a language where "all reason truths will be reduced to a kind of calculus", "the errors will only be computation errors". (Leibniz program, continued and formulated in more precise terms by David Hilbert, cannot be realized, on the one hand, mathematics is too exact-rigorous while the reality is too complex and nuanced to can transform everything in formal computations, on the other hand, Gödel theorems proved that even the Hilbert program is not realizable.)

Of course, the mathematics-computer science relationship is much more complex, but we cannot explore it further here. I close the discussion returning to the starting point: the today computers, programmable, of Turing-von Neumann type, are born from the Turing universality theorem from 1936. It is interesting to note (and comfortable for Dijkstra position) that, by means of a vote through Internet, in 2013, looking for the most important scientific and technological British discovery, the first place was won, surprisingly for our pragmatic times, by the Turing machine and Turing universality theorem, which were placed ahead of the steam engine, the telephone, the cement, the carbon fiber and other similarly important things.

9 Does Nature Compute?

Having in mind the computability in the sense of Turing, the previous question becomes more restrictive, but the discussion above provides us the borderlines in between which we have to look for the answer: yes, nature computes at least at the level of... humans, and yes, nature computes whenever there is a process which can be interpreted as a computation by a suitable observer. Opinions which are placed closer to the former or the latter of these limits can be easily found, I cite here only one from the very permissive extreme, even passing over the borderline, because the observer is not mentioned anymore.

At the beginning of Chapter 2 ("Molecular Computation") of the collective volume *Non-Standard Computation* (T. Gramss, S. Bornholdt, M. Gross, M. Mitchel, Th. Pellizzari, eds., Wiley-VCH, Weinheim, 1998), M. Gross says: "Life is computation. Every single living cell reads information from a memory, rewrites it, receives data input (information about the state of its environment), processes the data and acts according to the results of all this computation. Globally, the zillions of cells populating the biosphere certainly perform more computation steps per unit of time than all man made computers put together."

In what follows, I adopt a more conservative and, at the same time, more productive position: bearing in mind the mathematical definition of computability, more precisely, the Turing approach, let us look around, especially in biology, in search of ideas, data structures, operations with them, ways to control the operations, "computer" architectures, which can suggest (1) new computability models, (2) ways to better use the existing computers, (3) possibilities of improving the existing computers at the hardware level, maybe even (4) new types of computers, based on biological materials. It should be noticed the increased ambition from a point to the next one. It is worth remembering that DNA computing started from the very beginning from the attempt to compute in a test tube, thus directly addressing the fourth goal in the list above.

We mainly had here in mind the goals of computer science, but the first objective also covers the second direction of research mentioned in the preface of the *Handbook of Natural Computing*, the investigation of processes taking place in nature in terms of computability, and this research direction should be explicitly and separately emphasized, especially for pointing to a "side effect" of this approach, namely the return to biology, delivering models useful to the biologist.

At this moment, DNA computing was not too much useful to practical computer science, it was useful to biology and much useful to nano-technology, suggesting new research questions. Membrane computing has significant applications to computer science and biology, with higher promises in the latter area, including biomedicine and ecology among the application directions.

A detail: "the goal of computer science" also covers the theoretical interest, which is not supposed to necessarily lead to applications, in the restricted meaning of the term. Let us think, for instance, to ciliates. In the division process, when passing from the micronuclear genes to the macronuclear genes, these unicellular beings complete complex operations of list processing, and they are doing this since

millions of years, much before the computer scientists gave name and investigated these data structures. Of course, the ciliates are not thinking to computations when doing this, but we, the humans, can build beautiful theories starting from their activity, including computability models, sometimes equivalent in power with Turing machines. Detains and references can be found in the monograph *Computation in Living Cells. Gene Assembly in Ciliates* (Springer-Verlag, 2004), by A. Ehrenfeucht, T. Harju, I. Petre, D.M. Prescott, and G. Rozenberg.

10 An Eternal Dilemma

The previous discussion inevitably pushes us towards the long debate concerning the relation between invention and discovery. The bibliography is huge, I cite here only the book of acad. Solomon Marcus *Invention or Discovery*, Cartea Românească Publishing House, Bucharest, 1989 (in Romanian). How much is invention and how much is discovery in computer science – with particularization to natural computing? I do not try to provide an answer, there are as many answers as many view points, personal experiences, philosophical positions. The models we work with are of a mathematical nature, the Platonic point of view ensures us that everything is discovery, because mathematics itself is a revealed reality. Yes, but it is already agreed that notions, concepts, theories, and models are inventions, the theorems are discovered, the proofs are invented. We can continue the alternating sequence by adding that the applications are discovered. Therefore, the models are considered inventions.

However, I would like to introduce a nuance. The models are based on structures which already exists, but they have not received yet a name. Moreover, differently from a wall which can be discovered both by an archaeologist who knows what he is searching for, but also by a fruit trees farmer who digs the soil with other goals than finding the basement of an old church, a computability model can be "seen" in a cell only by a computer scientist who has already in mind computability models. For instance, the processes called by biologists symport and antiport exist, they function since long ages in their ingenious ways, but they *compute* only for a mathematician who is looking for a computing model based on passing "objects" from a cell compartment to another one. "Computing by communication" – I have searched for a while something like that, having the intuition that it exists, and I had the solution when a biologist (Ioan Ardelean) told me about symport and antiport operations. This was a model mostly discovered than invented. Actually, a discovery which was not done by bringing to light the discovered object, but by means of superposing the intuition of a model over a piece of reality. The imagined model, similar to other existing computability models, was actualized during the dialogue between reality and the formal framework. I can say that this is at the same time invention and discovery.

11 Another Endless Discussion

I am not continuing with other similarly delicate questions, always of interest in spite of any given answers. (For instance, providing us the opportunity to ask how much *art* and how much *science* is in computer science, Donald Knuth entitled an impressing editorial project, planned to have a dozen of volumes, *The Art of Programming*.) However, I touch here another very sensitive topic, with which I was confronted sometimes in the form of the newspaper question (but not completely nonsensical): "During your research in the cell area, have you ever met God?" Of course, the expected answer is something different from "yes" or "no", and similarly obvious is that, if we take the question seriously, we will get lost on the slippery sands of personal options, beliefs, metaphors.

If God is the order, the organization, the good and the beautiful, Spinoza's God, visible in the harmony of the Universe laws, as Einstein would say, then yes, I meet Him continuously, both in cells and outside them. Furthermore: in the title of a book originally published in 2009 by Simon & Schuster, and translated in Romanian in 2011, Mario Livio asks *Is God a Mathematician?* I answer in the style of Plato: no, God is not a mathematician, He is mathematics itself (the "grammar of the world") – hence, again, I meet Him every moment.

If, however, God is what the Book proposes to me, then I go in line with Galileo Galilei, who, in a letter sent to don Benedetto Castelli, on December 21, 1613, said (I recall it following Edmond Constantinescu, *God Does not Play Dice*, MajestiPress Publishing House, Arad, 2008; in Romanian): "God has written two books, the Bible and the Book of Nature. The Bible is written in the language of men. The Book of Nature is written in the language of mathematics. That is why the language of the Bible is not suitable for speaking about nature. The two books must be studied independently from each other." And Galileo added: the Book of Nature teaches us "how the Sky/Heaven goes", while the Bible teaches us "how to go to the Sky/Heaven".

After centuries of separation – mainly dogmatical, from both sides –, alternating with attempts, most of them pathetical, of reconciliation of science with religion, the words of Galileo can look too simple or opportunistic, but they cut in an efficient way a continuously regenerated Gordian knot. Let me mention also a more sophisticated, but somewhat symmetrical position, of Francis S. Collins, not only contemporary with us, but also connected to the topic of these pages, as he was the director of National Human Genome Research Institute, one of the leaders of the famous Human Genome Project. In 2006 he has published a book, *The Language of God. A Scientist Presents Evidence for Belief*, Simon & Schuster, translated in Romanian in 2009. The syntagma "language of God" was used also by Bill Clinton, in 2001, when he has announced the completion of "the most important, most wondrous map ever produced by humankind", the map of the around three billions of "letters" of the "book of life". Even if the title seems to suggest this, Collins is neither a creationist, nor an adept of the intelligent creation, but he is an "evolutionary deist" and the conclusion of his book is that "the God of Bible is also the God of the genome" (page 222 in the Romanian version),

while "science can be a form of religiosity" (page 240). This is a very comfortable positioning, but, in what follows, I remain near Galileo.

12 The Limits of Today Computers

The fashion of natural computing and especially of the computing inspired from biology does not have only the internal motivation, of the numerous research directions explored in the last decades and proved to be theoretically interesting and at least promising if not directly useful in practice, but it has also an external motivation, related to the limits of the current computers, some of them rather visible. Indeed, the computers are the twentieth century invention with the widest impact, with implications in all components of our life, from communication to the functioning of the financial system, from the health system to the army, from the numerous gadgets around us to Internet. In spite of all these – actually, just because of that – the computers which we have now have limits which we reach often (with the mentioning that also here, like in most things, there is something bad in the good and something good in the bad: powerful computers can be used both in positive ways, but also for bad goals, such as breaking security systems and cryptographic protocols on which, for instance, the protected communication is based.) Let us however think positively and note that there are many tasks which the today computers cannot carry out, but which we would like to have performed.

The processors become continuously faster and more compact, the memory storage larger and larger. Sure, but how much this tendency will last? It was much invoked the so-called *Moore law*, stated in 1965 by Gordon A. Moore, co-founder of Intel Corporation, with respect to the number of transistors which can be placed on an integrated circuit, extended then to the cost of information unit stored, formulated sometimes even in the form "in each year, the computers become two times smaller, two times faster, and two times more powerful". Exponential in all the three directions, thus tending fast towards the quantum limit in the dimension of processors. Even at the more technical level, confirmed for a couple of decades, of doubling the capacity of processors, the law – actually, only an observation, followed by a forecast – has been adjusted several times, with the doubling/halving moved first at one year and a half, then at two years, then at three years. Still, it is not too bad, but one cannot continue too much even at this pace.

In fact, the real problem is a different one. Progresses are made continuously at the technological level, but the current computers have intrinsic limits, which cannot be overcome only by means of technological advances. The computer recognizes fingerprints, but not human faces, it plays chess at the level of the world champion, but (on the standard board, not on reduced boards) it plays GO only at the level of a beginner, it proves propositional calculus theorems, but cannot go over this level (and definitely cannot distinguish trivial and non-interesting theorems from theorems which deserve to be collected). All these and many more, mainly

because these computers are... of Turing-von Neumann type. That is, sequential. Uniprocessor. (It also has other weaknesses, less restrictive in the current applications – for instance, it is a considerable energy consumer.) It computes whatever can be computed, but this is true in principle, at the *competence* level. There is here also a historical aspect. In the beginning, we were interested in what it can be computed, in the frontiers of computability, of algorithmic decidability. All these are important mathematical questions, but in applications it is of a direct relevance the *performance*, the resources needed for a given computation, what we can compute now and here, in specified conditions. How much electricity consumes a computer and how much space it needs are no longer questions of current interest, as they were in sixties (and still are in special frameworks, such as in cosmos and robotics), but the time we have to wait before receiving the answer to a given problem or the result of a computation is a crucial aspect in any application. And, I already mentioned it, in this respect not the technological promises are crucial, but the mathematical limits, the borderline between feasible and non-feasible.

13 A Great Challenge: the Exponential Complexity

A powerful theory was developed dealing with this subject, the computational complexity theory. Since the very beginning, it has defined as tractable the problems which can be solved in a polynomial time with respect to the size of the problem. (An example: consider a graph – a map with localities and roads among them – with n nodes. Which is the time necessary for an algorithm to tell us whether or not the graph contains a Hamiltonian path, i.e., a path which visits all nodes, passing only once through each of them? If this time is bounded by a polynomial in n, then we say that the algorithm is of a polynomial complexity.) The problems of an exponential complexity, those which need a time of the type $2^n, 3^n$, etc. for an input of size n were considered intractable. The former class was denoted by **P**, the latter one with **NP**, with the abbreviations coming from "polynomial" and "non-deterministically polynomial", respectively: a problem belongs to **NP** if we can decide in a polynomial time whether a proposed solution for it is indeed a solution or not (otherwise stated, we "guess" a solution, then we check whether it is correct; more technically, the solution is found by a non-deterministic Turing machine, one which has several possible transitions at a computation step and we rely on the fact that it always chooses the right continuation, without exhaustively checking all possibilities). For precise details the reader can consult the monograph of C.H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.

Let us recall that in the class **NP** there is a subclass, of "the most difficult problems in **NP**", the **NP**-*complete* problems: a problem is of this type if any other problem in **NP** can be reduced to it in a polynomial time. Consequently, if an **NP**-complete problem could be solved in a polynomial time, then *all* problems in **NP** could be solved in a polynomial time. The problems used in cryptography are in most cases **NP**-complete.

A beautiful theory, which, however, in its basic version has three weaknesses: (1) it cannot tell us yet whether or not **P** = **NP**, whether or not polynomial solutions can be found also for the problems which are now supposed to be of an exponential complexity, (2) the theory does not take into account such "details" as the coefficients and the degree of the polynomials and which, at the practical level, can have a crucial influence on the computation time, and (3) the theory takes into consideration the extreme cases, it is of the *worst case* type, it counts the steps of computations which solve the most difficult instances of a problem, while the reality is placed in most cases in the middle, near the "average". Here is an example with a practical relevance: the linear programming problem is in **P**, because the ellipsoid algorithm of Leonid Khachiyan (1979) solves the problem in a polynomial time, but this algorithm is so complex that practically, in most cases, it is less efficient than the old simplex algorithm, proposed during the Second World War, considered one of the most important ten algorithms ever imagined, but which is, theoretically, of an exponential complexity.

For these reasons, the complexity theory was refined and diversified (average complexity, approximate algorithms – these algorithms have a direct connection with natural computing), while the definition of tractability was carefully redefined.

Anyway, the general feeling was transformed in a slogan: *the Turing-von Neumann computers cannot solve in a reasonable time problems of an exponential complexity.*

The interest for the **P** = **NP** problem is enormous. On the one hand, most of the (no-trivial) practical problems are in the class **NP** and are not known to be in **P**, hence they are (considered) intractable, cannot be efficiently solved, on the other hand, most of the cryptographic systems in use are based on problems of an exponential complexity, hence solving them in a polynomial time would lead to breaking these systems. The problem whether **P** is or not equal to **NP** was already formulated in 1971 (by Stephen Cook), and in the year 2000 it was included by Clay Mathematical Institute, Cambridge, Massachusetts, in the list of the seven "millennium problems", with a prize of one million dollars for a solution.

While the importance of this problem for the theoretical computer science cannot be overestimated, it is not clear which would be the practical consequences of a solution, whichever this will be. There were many discussions on this topic – see, for instance, S. Cook, "The importance of the **P** versus **NP** question", *Journal of the ACM*, vol. 50, 2003, pages 27–29. If a proof of the strict inclusion of **P** in **NP** will be obtained, as most computer scientists (but not all of them!) believe, then almost nothing will be changed at the level of the practical computer science. If the equality will be proved in a non-constructive manner, or the proof will be a constructive one, but in a non-feasible manner (polynomial solutions to problems in **NP** will be found, but with polynomials of very high degrees or with very large coefficients), then the practical consequences will not be significant (but a race will start for ad-hoc solutions, having the time estimated by polynomials with reasonable degrees and coefficients). If, however, a "cheap" passage from **NP**

to **P** would be found, then the consequences for the practical computer science will be spectacular – in the good sense, excepting the cryptography, where the consequences will be dramatic.

At the level of the software there is one further problem, which I recall here in the formulation of Edsger W. Dijkstra (from "The end of computer science?", the above mentioned paper): "Most of our systems are much more complicated than can be considered healthy, and are too messy and chaotic to be used in comfort and confidence. The average customer of the computing industry has been served so poorly that he expects his system to crash all the time." The lack of robustness of the complex software systems is today a concern of the same interest as it was in the year 2000.

In order to illustrate the fact that not by means of technological progresses one can face the exponential complexity, let us examine a simple case: let us consider a problem of exponential complexity of the range of 2^n, for instance, a graph problem, which can be solved on a usual computer, say, for graphs with 500 nodes, in approximately one quarter of hour; let us suppose that the technology provides us a computer which is 1000 times faster than the ones we have, which is a totally nontrivial advance, not very frequently met. Using the new computer, we will solve the same problems as before in about one second (around 15 minutes means approximately 1000 seconds), but if we try to address the same problem for graphs with more than 500 nodes, the progresses are negligible: with the new computer we will solve in a quarter of hour only problems for graphs with at most 510 nodes. The simple reason for that is the fact that 2^{10} is already bigger than 1000. If the problem were of complexity 3^n, then we will stop around 506 nodes...

14 Promises of Natural Computing

In order to cope with the exponential complexity, but also for other reasons which I will mention later, computer science has imagined several research directions, most of them also related to the natural computing, even to bio-inspired computing: (1) looking for massively parallel computers, (2) looking for non-deterministic computers/computations, (3) looking for approximate/probabilistic solutions to computationally hard problems.

All these three research directions were explored already in the framework of the "standard" computer science, both at the theoretical level and at the technology level, the electronic one. Multiprocessor computers are available since several years – but without reaching the massive parallelism which is supposed to solve complex problems. If a large number of processors are put together, there appear other problems, some of them technological (e.g., high temperature dissipation), others, maybe more important, theoretical, concerning the synchronization of the processors. A distinct research area deals with the synchronization complexity – see, for instance, Juraj Hromkovic monograph *Communication Complexity and Parallel Computing*, Springer-Verlag, 1997. One of the conclusions of this theory

says that, for a large number of processors, the synchronization cost (measured by the number of bits necessary to this aim) becomes larger than the cost of the computation itself, which suggests to get rid of synchronization, but then other problems appear, as we are not accustomed to use asynchronous computers.

Even less used we are to construct and utilize "non-deterministic computers". In exchange, the last of the three ideas mentioned above is rather attractive, and in this respect of a great help is the "brute force" of existing computers. The approach is useful especially in addressing complex optimization problems: exploring randomly the candidate solutions space, for a large enough time, with a sufficiently high probability we will reach optimal or nearly optimal solutions. Approximate solutions, possibly found with a known probability of being optimal.

Here it enters the stage, with great promises, the natural computing. From now on I will only refer to the one having a biological inspiration.

In a cell, a huge number of "chemical objects" (ions, simple molecules, macromolecules, DNA and RNA molecules, proteins) evolve together, in an aqueous solutions, at a high degree of parallelism, and, at the same time, of non-determinism, in a robust manner, controlled in an intricate way, successfully facing the influences coming from the environment, and getting in time very attractive characteristics, such as adaptation, learning, self-healing, reproduction. Many other details are of interest, such as the reversibility of certain processes or the energy efficiency, with the number of operations per Joule much bigger than in the case of the electronic processing of information (erasing consumes energy, that is why the reversible computers are of interest; see, e.g., R. Landauer, "Irreversibility and heat generation in the computing process", *IBM Journal of Research and Development*, vol. 5, 1961, pages 183–191, and C.H. Bennett, "Logical reversibility of computation", *Idem*, vol. 17, 1973, pages 525–532).

It seems, therefore, that during millions of years of evolution nature has polished many processes (and material supports for them) which wait to be identified and understood by the computer scientists, in order to learn new computability methods and paradigms, maybe for constructing computers of a new kind. And, the computer scientists have started to work singe a long time...

Here are a few steps on this road, very shortly: *Genetic algorithms*, as a way to organize the search through the space of candidate solutions, imitating the Darwinian evolution, in order to solve optimization problems. Generalization to *evolutionary computing* and *evolutionary programming*. *Neural networks*, trying to imitate the functioning of the human brain, also used for finding approximate solutions, especially for pattern recognition problems. A little bit later, *DNA computing*, which has proposed a new hardware, massively parallel, based on using the DNA molecules as a support for computations. Even younger, *membrane computing*, taking as the starting point the biological cell itself and cell populations.

In turn, the evolutionary computing, in general, the area of approximative algorithms inspired from biology, is spectacularly ramified, in the most diverse (in certain cases, also picturesque) directions: *immune computing, ant colony algorithm, bee colony algorithm, swarm computing, water flowing computing, cultural*

algorithm, cuckoo algorithm, strawberry algorithm – and it is highly probable that in the meantime further algorithms have been proposed...

It is important to note that all the above mentioned branches of natural computing, with the exception of DNA computing, are meant to be implemented on the usual computer, in the aim of having a better use of it; one proposes new types of software/algorithms, not to change the computers architecture or new types of hardware.

15 Everything Goes Back to Turing

In a certain sense and in a certain extent, the whole history of theoretical computer science is related to biology, it has searched and has found inspiration in biology. I have already mentioned that, in 1935-1936, when he has defined the machine which bears now his name, Turing tried to imitate the way the humans are computing.

After one decade, McCullock, Pitts, Kleene have founded the theory of finite automata starting from the modeling of neurons and of neural networks. Later, the same starting point led to what is called today neural computing.

It is interesting to note that the beginnings of this research area can be identified in unpublished papers of the same Allan Turing. We have here an interesting case which can illustrate the influence of psychology and sociology on the development of science, telling about uninspired group leaders and about researchers interested more in their research than in the publication of the obtained results. Specifically, in 1948, Turing has written a short paper, called "Intelligent machinery", which has remained unpublished until 1968, because his boss from the London National Physical Laboratory, ironically, named sir Charles Darwin, the grandson of the famous biologist with the same name, has written on the corner of the first page of the paper "schoolboy essay", thus preventing the publication.

"In reality, this farsighted paper was a manifesto of the field of artificial intelligence. In the work (...) the British mathematician not only set out the fundamentals of connectionism but also brilliantly introduced many of the concepts that were later to become central to AI, in some cases after reinvention by others." – I have cited from B.J. Copeland, D. Proudfoot, "Alan Turing's forgotten ideas in computer science", *Scientific American*, April 1999, pages 77–81. Among others, Turing paper introduces two types of "neural networks", with the neurons randomly connected. This was proposed as a first step towards an intelligent machine, one of the key features of these networks being that of learning, of getting trained for solving problems. This is neural computing *avant la lettre*, with the main ideas rediscovered later, without referring to Turing. Details about Turing "unorganized machines" can also be found in C. Teuscher, ed., *Alan Turing. Life and Legacy of a Great Thinker*, Springer-Verlag, 2003, and in C. Teuscher, E. Sánchez, "A revival of Turing's forgotten connectionist ideas: exploring unorganized machines", from *Proc. Connectionist Models of Learning, Development and Evolution Conf.*, Liége, Belgium, 2000 (R.M. French, J.J. Sougne, eds.), Springer-Verlag, 2001, pages

153–162. Furthermore, at the address `http://www.AlanTuring.net` one can find details about Turing unpublished manuscripts and about the recent efforts to reintroduce them in circulation.

The same Turing, in the same year 1948, has proposed the "genetic or evolutionary search", the first ideas of the evolutionary computing developed later, a domain which contains now several powerful branches, (re)launched during the years: evolutionary programming (L.J. Fogel, A.J. Owens, M.J. Walsh), genetic algorithms (J.H. Holland), evolutionary strategies (I. Rechenberg, H.P. Schwefel), all three initiated in the sixties, genetic programming (J.R. Koza, the years 1990). The first experiment of computer "optimization through evolution and recombination" was carried out in 1962, by Bremermann. Details can be found in A.E. Eiben, J.E. Smith, *Introduction to Evolutionary Computing*, Springer-Verlag, 2003.

It would not be completely surprisingly if among Turing manuscripts we would discover also ideas related to DNA computing – let us remember that Turing died in June 1654, and the *Nature* paper where J.D. Watson and F.H.C. Crick described the double helix structure of the DNA molecule was published one year before ("A structure for deoxyribose nucleic acid", vol. 171, April 25, pages 737–738).

It is worth mentioning that two other concepts with a high career in computer science come from Turing, thus supporting the assertion that "everything starts with Turing". First, Turing himself raised the question whether or not one can compute... more than the Turing machine, imagining Turing machines with oracles, which is a much investigated topic in the current computer science. Then, Turing can be considered not only a founder of artificial intelligence, but also a forerunner of what is called now *artificial life*: in the last years of his life, Turing was interested in morphogenesis, in modeling the evolution from the genes of a fertilized egg to the structure of the resulting animal.

16 An Encouraging Example: The Genetic Algorithms

Before passing to the DNA and membrane computing, topics which I will describe in more details, let us spend some time discussing a branch of natural computing inspired from biology which is, at the first sight, surprisingly efficient. This is the *genetic algorithms* area, used for solving complex optimization problems for which there do not exist deterministic optimal algorithms or these algorithms are not efficient. The implicit slogan can look confusing: *if you do not know where to go, then go randomly* – with the mentioning that the "randomness" here is directed, the "random walk" is done "like in nature, in species evolution".

Everything is a metaphorical imitation of some elements from the Darwinian evolution. Let us assume we have a two variables function (we can suggestively represent it as a ground surface, with valleys and hills) for which we need to find the maximum (one of them, if there are several). If we cannot analytically address the problem, then we can choose to walk randomly through the definition domain, looking for the greatest value of the function. To this aim, we represent the

domain points as "chromosomes", binary strings of a constant length, we choose (randomly or through other methods) a given number of starting points, and we compute the function value for all of them. Then we pass to "evolution": we take two by two the "chromosomes" and we recombine them (by crossover), that is, we cut them at a specified position and then we recombine the fragments, the prefix of one "chromosome" with the suffix of the other one and conversely. In this way, we obtain two new "chromosomes", describing two new individuals of the next "generation". We repeat this procedure for a specified number of times, we select the best solution obtained so far, and we stop.

Nothing guarantees that in this way we reach the solution of the problem, that, for instance, we do not get stuck in a local maximum, without being able to escape, but, and this is the (pleasant) surprise, in a large number of practical applications, this strategy works. Sure, there are a lot of variations of the previous scenario, it is even said that the monographs in this area are a sort of "cooking books", collections of recipes, lists of ingredients and suggestions of improvements of the algorithms: besides recombination, similar to the biological evolution, one also uses the local mutation operation, the passing from a generation to another one can be done in many ways, the "chromosomes" population can be distributed, we can evolve it locally, communicating in a way or another among regions, there are several halting criteria, and so on and so forth.

We have here at work the brute force of the computers and the evolutionary metaphor – with results, I repeat and stress it, unexpectedly good: non-intuitive solutions, rapid initial convergence, in many cases succeeding to avoid local maxima. The only "explanation" for these good results is the "bio-mystical" one: genetic algorithms are so good because they involve ingredients which nature has polished for many millions of years in the species evolution.

All these induce, at the same speculative level, a rather optimistic conclusion: if the genetic algorithms are so useful, in spite of the lack of any mathematical argument for their usefulness, let us try to imitate biology also in other aspects, with a great probability that, if we are similarly inspired to extract the right ideas, to obtain other fruitful suggestions for improving the use of the existing computers and, maybe, for imagining computers of other kinds, more efficient.

However, this optimism should be cooled down by the observation that a famous result in the area of evolutionary computing, in the area of approximate optimization algorithms in general, is the so-called *no free lunch theorem* of David Wolpert and William Macready (1997), which, informally, says that any two methods of approximate optimization are equally good, in average, over all optimization problems. "Equally good" can be also read "equally bad", for each method there are problems for which the method does not provide satisfactory solutions.

17 A Coincidence

Before passing to DNA computing, an autobiographical intermezzo. In April 1994 I was in Graz, Austria, attending a conference, and there I got a copy of the paper

by Tom Head, from State University of New York at Binghamton, USA, soon after that a friend and collaborator, "Formal language theory and DNA: An analysis of the generative capacity of specific recombinant behaviors", published in *Bulletin of Mathematical Biology*, vol. 49, 1987, pages 737–759. It was a revelation. I was then after twenty years of formal language theory research and I immediately felt that it is open there a large area of application of what I have done before. It is true, I should had a similar revelation earlier, namely when I have read professor Solomon Marcus paper "Linguistic structures and generative devices in molecular genetics", from *Cahiers de Linguistique Thèorique et Appliquée*, vol. 11, 1974, pages 77–104, but probably it was too early, at that time I had not passed yet through the twenty years of preparation for natural computing which will be shortly described in a forthcoming section. In his paper, Tom Head introduces a formal operation with strings which formalizes the operation of recombination of DNA molecules. He calls it *splicing*, and I will call it in the same way, thus distinguishing it from the recombination operation from the genetic algorithms. The two operations are related, but they are not identical. Still being in Graz, I have imagined a sort of grammar based on the splicing operation, in fact, a variant simpler than that of Tom Head and closer to the string operations in language theory. The paper emerged in this way was published in *Discrete Applied Mathematics* and it consecrated the splicing version I have proposed. After a few weeks, I was in Leiden, The Netherlands, where I have written a paper together with Grzegorz Rozenberg and Arto Salomaa, the latter one from Turku, Finland, the place where I have spent after that several years, initially devoted to DNA computing and then to membrane computing. As usually well inspired, G. Rozenberg gave to our paper the title "Computing by splicing". Because, starting then, we have named H systems the computing devices based on splicing, thus reminding the name of the one who has introduced (invented of discovered?...) the respective operation, we have sent the paper, in manuscript, to Tom Head. He has immediately replied, by phone, asking us rather excited: have you known that right now it was carried out a successful experiment of computing with DNA?! No, we did not know – this was only a coincidence, which I place in the category of significant coincidences.

18 The First Computation in a Test Tube

Tom Head was talking about Leonard Adleman experiment, published in the autumn of 1994 in *Science*, nr. 226, November 1994, pages 1021–1024: "Molecular computation of solutions to combinatorial problems". Speculations about the possibility of using DNA molecules for computing were made already in seventies of the last century (Ch. Bennett, M. Conrad, even R. Feynman, with his much invoked phrase "there is plenty of space at the bottom", referring to physics but also extended to biology). Adleman has confirmed these expectations, solving in a laboratory the problem whether a Hamiltonian path exists or not in a given graph (I have mentioned it in a previous section). The problem is known to be **NP**-complete, hence among the most difficult intractable problems, of an exponential

complexity (we assume that **P** is not equal to **NP**), but Adleman solved it in a number of steps which is linear with respect to the size of the graph. It is true, these steps are biochemical operations, performed by making use of a massive parallelism, even of non-determinism, all these made possible by the characteristics of the DNA molecules and the related biochemistry.

In short, millions of copies of one stranded sequences of nucleotides, codifying the nodes and the edges of the graph, were placed in an aqueous solution. Then, by decreasing the temperature of the solution, these sequences annealed, forming double stranded molecules, corresponding to the paths in the graph. Because there were used sufficiently many copies of the initial sequences, with a high probability we obtain in this way *all* paths in the graph. From them, the paths were selected which pass through all nodes, and this was done by usual laboratory procedures: gel electrophoresis for separating the molecules according to their length, then selection through denaturation and amplification by PCR of the paths passing through all nodes (hence Hamiltonian).

This procedure assumes a number of biochemical operations which is linear with respect to the number of the nodes in the graph. The problem is **NP**-complete, hence this is an extraordinary achievement – and the consequences were accordingly sound. Already in the next year, 1995, it was organized in Princeton a meeting with the title "DNA Computing", which became an international conference which is still continuing. However...

19 Pro and Against Arguments

Adleman experiment was a historical achievement, the proof that *it is possible*. However, the experiment has considered a graph with only 7 nodes, for which the problem can be solved by a simple visual inspection. In comparison, at the beginning of the nineties, the computers were already able to solve the Hamiltonian path problem for graphs with several hundred nodes, sufficient for current practical applications (in the meantime, the progresses continued).

Moreover, the solution was obtained by means of a space-time trade-off, the number of molecules used was exponential with respect to the number of nodes. Juris Hartmanis, an authoritative name in computer science, after expressing his enthusiasm (Hartmanis compares computer science with physics: while the latter progresses by means of crucial experiments, the former progresses by means of proofs that something can be done, by *demos*; Adleman has produced such a *demo*!), has computed the quantity of ADN which is necessary in order to apply Adleman's procedure for a graph with 200 nodes and he has found that the weight of the ADN would be greater than the weight of the Earth...

From a practical point of view, DNA computing is, in a certain extent, in the same point even now. Numerous experiments, but all of them always dealing with "toy problems", a lot of theory, a lot of lab experience gained in dealing with DNA molecules, with results of interest for the general lab technology (just one example:

an improved version of PCR, the Polymerase Chain Reaction, called XPCR, was proposed; one of the inventors is a mathematician, Vincenzo Manca, mentioned already in the first pages of this text), but the domain has moved towards nanotechnology, no computability practical applications were reported (unless if, and this is plausible, there were applications in cryptography which are still classified).

However, the list of possible advantages of using DNA molecules for computing is large: a very good efficiency as a data support, with one bit at the level of a nucleotide; energy efficiency; parallel and non-deterministic behavior, two dreams of computer science (with the mentioning that the non-determinism also brings problems, for instance, providing false solutions); a very developed laboratory technology; robustness, predictability, reversibility of certain processes.

20 The Marvelous Double Helix

The DNA molecule has surprising properties at the informational and computational level. Let us remind that, formulated in "syntactic" terms, we have two strings of letters A, C, G, T, the four nucleotides, placed face to face, in Watson-Crick complementary pairs, always A being paired with T and C with G. The two strings are oriented, in opposite directions with respect to each other; the biochemists indicate the directionality by marking one end of a string with $3'$ and the other end with $5'$. There already appear here a surprise, first pointed out by G. Rozenberg and A. Salomaa in *Technical Report* 96-28 of Leiden University, The Netherlands (October 1996), "Watson-Crick complementarity, universal computations, and genetic engineering": the structure of the DNA molecule "hides", in a codified manner, the computing power of Turing machines! The formulation above is not precise, it however corresponds to the following observation. Already in 1980, it was proved (J. Engelfriet and G. Rozenberg) that *any language whose strings can be recognized by a Turing machine can be written as the image of a specified fixed language, let us denote it with* $TS(0,1)$*, by means of a sequential transducer.*

The previous language is the so-called "twin-shuffle" over 0, 1 (hence the used notation). Shuffle is the operation of mixing the letters of the two words, without changing their ordering (exactly as in the case of shuffling two decks of playing cards of different back colors). Here we shuffle the letters of two "twin" words, one string of symbols 0, 1 and the second string identical with this one, but changing the "color" of each symbol (for instance, we can add an upper bar or a prime to each symbol in order to get the twin string). In turn, the sequential transducers are the simplest transducers, with a finite memory and with a head which scans the string from left to right. Let us note that we work with four symbols, let us say 0, 1 and their pairs $0', 1'$. Exactly the number of the nucleotides, four. Let us also note that $TS(0,1)$ is a fixed language. Given an arbitrary language, if it is recognized by a Turing machine, then it can be obtained from this unique language $TS(0,1)$, only the transducer depends on the language.

The nice and significant surprise is that the language $TS(0,1)$ can be obtained by means of "reading" the DNA molecules, in the following way: let us walk along the two Watson-Crick complementary sequences, from the left to the right, advancing randomly along the two strands, and associating with the four nucleotides A, C, G, T symbols 0, 1 according to the following rule: $A = 0, G = 1, T = 0', C = 1'$. Collecting all these strings over $0, 1, 0', 1'$, for all readings of all DNA molecules, we get a set which is exactly $TS(0,1)$!

Consequently, any language which can be defined by a Turing machine can be obtained by translating these readings of the DNA molecules by means of the simplest transducer, the sequential one, with a finite memory. The transducer depends on the language, it "extract" from $TS(0,1)$ the result of the computations of a Turing machine. The power is there, what we have to do is only to make it visible. (In a certain sense, we have again the coupling of a simple process, the "reading" of the DNA molecule, and an observer of the first order, a simple one, the sequential transducer, like in the papers of M. Cavaliere and P. Leupold mentioned before, with the result reaching the highest level of computability, the power of Turing machines.)

Two questions arise in this framework. For instance, we mentioned the different orientation of the two strands of the DNA molecule, but in the previous reading we pass along the two strands in the same direction, from the left to the right. There is no problem, the reading of the double stranded DNA molecules can proceed in opposite directions and the result is the same. Second: nature is redundant, are all the four nucleotides (the four symbols $0, 1, 0', 1'$) necessary in order to cover, in the sense discussed above, the power of Turing machines? No, three symbols are sufficient – but not two! Proofs for all these results can be found in the monograph (translated in Japanese, Chinese, and Russian) Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing. New Computing Paradigms*, Springer-Verlag, 1998.

Speaking about computations and redundancy, let us remember that a large part of the DNA molecule is "residual", it does not codify genes and we do not exactly know what it is used for. We can then speculate: if in the cell, at the genetic level, one performs computations (the viruses are strings of nucleotides, hence their identification is a parsing operation, hence a computation), and these computations are supposed to be complex, why not?, even of the level of Turing machines, then we need a "workspace", a "tape" which in the end remains empty in most of its length, with the result placed in a finite part of it (at the beginning in the case of the Turing machine tape). Can then the DNA without an apparent usefulness be the workspace for complex computations, which we cannot yet understand?

21 Computing by Splicing

In his experiment, Adleman has not used the splicing operation, but the biochemical ingredients specific to the splicing have been used in many other cases:

restriction enzymes, which cut the DNA molecules in well specified contexts, ligases which glue back the nucleotides thus repairing the strands, recombination on the basis of the "sticky ends" of the molecules with the strands of different lengths, hence with nucleotides which do not have their Watson-Crick pairs.

I do not recall biochemical details or mathematical details concerning the splicing operation. In short, two molecules (represented as simple strings, because the nucleotides of a strand are precisely identified by their complementary nucleotides placed on the other strand) are cut in two parts each, in the middle of a context specified by a pair of substrings, and the fragments obtained are recombined crosswise, thus obtaining two new strings. Starting from an initial set of strings and applying this operations repeatedly (with respect to a given finite set of contexts, hence of *splicing rules*), we obtain a computing device, a language generator, similar to a grammar. We obtain an *H system*. A large part of the monograph *DNA Computing. New Computing Paradigms* cited before is dedicated to the study of these systems: variants, extensions, generative power, properties.

Always when a new computing model is introduced, the first question to clarify concerns its power, in comparison with the automata theory and language theory classifications – the Turing machine and its restrictions, the Chomsky grammars, the Lindenmayer systems. Let us only note that the two "poles" of computability are the power of Turing machines, through the Turing-Church thesis the maximal level of algorithmic computability, and the power of the finite automata, the minimal level. In terms of grammars and languages, the maximal class is that of unrestricted Chomsky grammars and of recursively enumerable languages, while the minimal one corresponds to regular grammars and languages.

The H systems with a finite number of starting strings and a finite number of splicing rules generate only regular languages. This is not sufficient as computing power, moreover, a "computer" of this level cannot have (convenient) universality properties, hence it cannot be programmable.

Interesting and attractive enough is the fact that, adding a minimal control on the splicing operation, with many controls of this kind suggested by the area of regulated rewriting or coming from biology (example: associate a promoter, a symbol, with each splicing rule and the rule is applied only to strings which contain that symbol; a variant – the symbol does not appear, it acts as an inhibitor), then we obtain H systems which are equivalent with the Turing machine. The proof is constructive, therefore we "import" in this way from the Turing machines the existence of the universal machine, which means that we get an universal H system, a programmable one.

Unfortunately, so far, no such universal "computer" based on splicing was realized in a laboratory. The passage from the natural case, with an uncontrolled splicing operation (thus with the power under the power of the finite automaton), to the controlled case was not yet done in a laboratory and it is not clear whether it can be realized in the near future. The construction of the universal computer based on splicing has to still wait...

22 An Important Detail: The Autonomous Functioning

Let us not forget that a universal, programmable computer should work autonomously, that is, after starting a program, the computer continues without any external control. This is completely different from the usual DNA computing experiments, where the human operator (or a robotic operator) controls the whole process. For instance, in the case of the 1994 experiment, Adleman was, in fact, the "computer", he has only used the DNA molecules as a support for the computation, while the computation complexity counted the lab steps performed by the biochemist, not the internal steps, the DNA operations, performed in parallel.

There are, however, promising progresses towards the implementation of autonomous computations, the key-word, very much promoted in the last years, being *self-assembly*. Remarkable achievements in this direction has obtained Erik Winfree and his group from Caltech, Pasadena, USA, and his approach is worth mentioning also because it starts (pleasantly enough for the discussion concerning the usefulness of mathematics for computer science) from an old chapter of theoretical computer science, the domino calculus of Wang Hao, developed in the beginning of sixties of the last century. In short, square dominos, with the edges colored (marked), can be used for computing (by placing the dominos adjacently, in such a way that the neighboring dominos have the contact edges of the same color), thus simulating the work of a Turing machine. We obtain once again a computing model which is universal.

Erik Winfree has constructed "dominos" from DNA molecules, with the edges marked with suitable sequences of nucleotides, he has left them free in a solution, such that the dominos glued together according to the Watson-Crick affinity of the nucleotides "coloring" the edges. The approach worked well, the experiments were successful – but everything has remained once again, in Hartmanis terms, at the level of a *demo*. It is important to underline that this time it was not addressed a given problem, as in Adleman case and as in most of the experiments reported in the DNA computing literature, but it was implemented in a laboratory a Turing machine, hence an universal computing model – that is why this *demo* is perhaps farther reaching than that of Adleman (however, Adleman was the first one...).

There also are other attempts to obtain autonomous "computers" in a laboratory. I mention here only the simulation of a finite automaton, an achievement of a team from Weizmann-Rehovot and Tehnion-Haifa, Israel: Y. Benenson, T. Paz-Elizur, R. Adar, E. Keinan, Z. Livneh, E. Shapiro, "Programmable and autonomous computing machine made of biomolecules", *Nature*, vol. 414, November 2001, pages 430–434, with the mentioning that one deals with a finite automaton with only two states. Again, only a *demo*...

23 What Means to Compute *in a Natural Way*?

The *DNA Computing* monograph has also chapters dedicated to other ways of computing, inspired from the DNA biochemistry, for instance, by insertion and

deletion of substrings (in given contexts), by means of a "domino game" with DNA molecules which are coupled on the basis of the Watson-Crick complementarity, a model different from the Wang Hao one.

Splicing, insertion-deletion, prolongation of strings. In membrane computing we use the multiset processing. The evolution itself is mainly based on recombination/splicing, the local mutations appear only accidentally. In contrast, the existing computers and almost all theoretical models of computing use the string rewriting operation. One works locally, on strings of arbitrary length. This observation is valid for automata, grammars, Post systems, Markov algorithms. All these operations, both the rewriting and the "natural" ones (the splicing only with an additional control), so different among them, lead to computability models of the same power, that of the Turing machine.

The question is obvious: what means to compute *in a natural way*? With many continuations: Why computer science has not considered (with rare exceptions) also other operations different from rewriting? Can we devise (electronic) computers based on "natural operations" (for instance, using the splicing or other forms of recombination)? When Hilbert has formulated the question "what is mechanically computable?", he probably had in mind formal logical systems, where the substitution is a central inference rule, and Turing has proposed an answer in the same language. Were we influenced in this way to think in the same terms when we have designed the first computers? I have never heard that the engineers have said that we cannot imagine, maybe also construct, computers based on different operations.

It remains the question whether or not such new types of computers would be better than the existing computers or not. Theoretically, they will have the same power, hence the differences should be looked for on different coordinates: computational efficiency, easiness of use, learning possibilities and so on.

I said above that the H systems are either of the power of finite automata or equivalent with Turing machines. Similar situations are met in the membrane computing. Can we then say that the classes of automata and grammars which lie in between finite automata and Turing machines – and there are many such classes investigated in the theoretical computer science – are not "natural"? In some sense, this is the case. For instance, the context-free languages have a definition which has a mathematical-linguistic motivation, while the context-sensitive languages have a definition with a motivation coming from the complexity theory (it refers to the space needed for generating or recognizing the strings of a language).

24 Let Us Pass to the Cell!

In spite of the theoretical achievements, of numerous successful experiments (however, dealing with problems of small dimensions) and of the continuous progresses in what concerns the lab techniques, the DNA computing has not confirmed the enthusiasm of the twenty years ago, after the announcement of the Adleman experiment – if not having, as I have suggested before, application in cryptography

which will be declassified only after several decades. There are elements which can support this assumption. For instance, during the first DNA Computing Conference, Princeton, 1995, a communication was presented (D. Boneh, C. Dunworth, R. Lipton: "Breaking DES using a molecular computer") which described a possibility to break Data Encryption Standard, DES, the system used by the American administration, using DNA, in four months. Next year, the subject was discussed by a team containing also Adleman, and the proposed DNA experiment was supposed to can break DEA in five days, provided that the lab operations would be done by robots. A further paper of this kind was presented in 1997, the year when DES was broken also with electronic computers and then abandoned.

Anyway, at some years after Adleman experiment it was clear that one cannot go essentially further, it was necessary to have one more innovative idea, one more "breakthrough" in order to make an essential step towards applications (towards a "killer-app", as the Americans use to say), and one of the "explanations" of this situation was the fact that DNA molecules behave better *in vivo* (more predictable, more robustly) than *in vitro*. The suggestions is just natural: let us go to the cell!

At the personal level, this moment coincided with the writing of the *DNA Computing* monograph, a fact which repeated almost systematically in the first two decades of my research career: after approximately five years of work in a branch of theoretical computer science, I have put together, alone or in collaboration, the results, publishing a monograph, and after that I have passed to another topic – still remaining in the framework of theoretical computer science, especially of formal language and automata theory. A lack of perseverance or an excess of curiosity? Maybe a part of each of them, but a lucky combination: all chapters of theoretical computer science which I have explored before passing to the membrane computing area were used, sometimes in a decisive extent, in this last domain – with which I have discontinued the tradition of a change at each five years: after sixteen years dedicated almost exclusively to membrane computing, in spite of the fact that I have written, as usually, a monograph after about five years from the first paper, there is no sign of decreasing the interest for this area.

25 The Fascinating Cell

The cell is really fascinating for a mathematician-computer scientist. I am sure that this is true also for biologists. The smallest entity which is unanimously considered *alive*. The topic is not trivial: at the middle of years 1980, at the Santa Fe Institute for complexity studies a new research vista was initiated, under the name of *artificial life*, as an extension of artificial intelligence, aiming to investigate the life per se, to simulate it on non-biological supports, on computer and in mathematical terms. The starting point was, of course, the attempt to have a definition for what we intuitively call *life*, but the progresses have not went too far: all definitions either left out something alive, or they ensured that, for instance, the computer viruses are alive (they have "metabolism", self-reproduction etc.).

Let us also remember that already Erwin Schrödinger has a book whose title asks *What is Life?* (Cambridge Univ. Press., 1967, translated in Romanian in 1980).

The cell passes this test. It is an extraordinarily small "factory", with a complex, intricate and efficient internal structure, where an enormous number of agents interact, from ions to large macromolecules like that of ADN, and where informational processes are carried out at each place and in each moment. Some cells live alone (I am not saying "isolated"), as unicellular organisms, other cells form tissues, organs, organisms.

It is a topic of interest the one concerning the role of the cells in making possible the life itself. I am only citing the reference book B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, P. Walter, *Molecular Biology of the Cell*, 4th ed. Garland Science, New York, 2002, the paper of Jesper Hoffmeyer "Surfaces inside surfaces. On the origin of agency and life", *Cybernetics and Human Knowing*, vol. 5, 1998, pages 33–42, also important for what follows because it proposes the slogan "life means surfaces inside surfaces", referring to the membranes which define the inner structure of the cells, and I end with a paragraph from the book of S. Kauffman, *At Home in the Universe*, Oxford University Press, 1995: "The secret of life, the wellspring of reproduction, is not to be found in the beauty of Watson–Crick pairing, but in the achievements of collective catalytic closure."

I am adding also a suggestive equation-slogan, which acad. Solomon Marcus has launched during one of the first Workshops on Membrane Computing, the one in Curtea de Argeş, 2002:

Life = DNA software + membrane hardware.

26 The Membrane. From Biology to Computability

We have thus arrived to a fundamental ingredient – the membrane. One can speak very much about it, and the biologists and the experts in bio-semiotics have done it. The cell itself exists because it is separated from the neighboring environment by a membrane. Not only metaphorically, any entity exists because it is delimited by a "membrane", actual or virtual, from the world around.

The (eukaryotic) cell also has a number of membranes inside: the one which encloses the nucleus, the complicated Golgi apparatus, vesicles, mitochondria. From a computational point of view, the main role of these membranes is to define "protected reactors", compartments where a specific biochemistry takes place. There also are other features-functions of the biological membranes which are important for membrane computing: in membranes are placed protein channels which allow the selective communication among compartments; on membranes are bound enzymes which control many of the biochemical processes which take place around them; the membranes are useful also for creating reaction spaces small enough so that the molecules swimming in solution can get in contact so that they can react. It is said that when a compartment is too large for the local biochemistry to be efficient, nature creates new membranes, in order to obtain small enough "reac-

tors" (so that, by Brownian movement, the molecules collide sufficiently frequent and react) and for creating new "reaction surfaces".

I stress the fact that I look here to the cell, its structure, and processes inside it through the glasses of the mathematician-computer scientist, ignoring many biochemical details (for instance, the structure itself of the membranes) and interpreting the selected ingredients according to the goal of this approach: to define a computing model.

Let us give some details, starting with the essential role of membranes in communication. If, in the biological cell or in the model we are going to define, the compartments delimited by membranes evolve separately, then we will not have one "reactor", but a number of neighboring "reactors", evolving independently. However, the membranes ensure the integration. The polarized molecules or those of great dimensions cannot pass through the (phospholipid, with a polarized "head" and two hydrophobic "legs") molecules of the membranes, but they can pass across a membrane through the protein channels embedded in it. This passage is selective and sometimes it is done against the gradient, from a smaller concentration to a higher concentration. A very interesting case is that of the simultaneous passage through a protein channel of two or more molecules: the respective molecules cannot pass separately, but they can do it together, either in the same direction (*symport*) or in opposite directions, one molecule entering the respective compartment and the other one going out, simultaneously (*antiport*). An important chapter of membrane computing is based on these operations and the interest comes from the particularity of this process: there is no rewriting, but only object transport across the borders defined by the membranes, there is no erasing, but only communication. *Computing by communicating* (objects). We can formulate also in this context the question *what means to compute in a natural way?*

We can read in many places about the informational processes taking place in a cell, in most cases with the involvement of membranes, too.

"Many proteins in living cells appear to have as their primary function the transfer and processing of information, rather than the chemical transformation of metabolic intermediates or the building of cellular structures. Such proteins are functionally linked through allosteric or other mechanisms into biochemical 'circuits' that perform a variety of simple computational tasks including amplification, integration, and information storage."

This is the abstract of the D. Bray paper "Protein molecules as computational elements in living cells", published in *Nature*, vol. 376, July 1995, pages 307–312. In their turn, S.R. Hameroff, J.D. Dayhoff, R. Lahoz-Beltra, A.V. Samsonovich, S. Rasmussen, in a paper from *Computer*, November 1992, pages 30–39, interpret the cytoskeleton as an automaton, while W.R. Loewenstein, in *The Touchstone of Life. Molecular Information, Cell Communication, and the Foundations of Life*, Oxford University Press, 1999, constructs a whole theory starting from the informational aspects of the cell life. About the bio-semiotics of the cell has elaborated in many places Jesper Hoffmeyer, already mentioned at the previous pages. I am citing here only his paper "Semiosis and living membranes", presented at *Seminário*

Avançado de Comunicaçao e Semiótica. Biossemiótica e Semiótica Cognitiva, Sao Paolo, Brasil, 1998.

In this context we can also remember the essential role of the water in the life of the cell, as well as the processes of moving water molecules across membranes through the dedicated channels, the *aquaporines*, in whose discovery our colleague Gheorghe Benga had pioneering contributions.

27 A Terminology-History Parenthesis

Before passing to a quick description of membrane computing, let me point out a few preliminary things.

First, about the name of the domain. I have called it *membrane computing*, starting from the position of the membrane in the life of the cell, in its architecture and functioning, but the choice was not the best one. "Cellular computing" was probably the most "marketable" choice, but I have discarded it as being too comprehensive.

Then, the name of the models: in the first papers, I used "membrane systems", but soon those who started to investigate these models have called them "P systems", continuing the line of other computing devices having a name (H systems are the closest ones). In the beginning this induced to me some public discomfort, for instance, during conferences, but the letter P soon became autonomous, completely neutral to me. What is interesting is that there are papers which use the syntagma "P system", sometimes even in the title, without citing any paper of mine. Of course, it would be a great success if this syntagma will became largely folkloric...

The domain has grown very rapidly and it is still active after more than sixteen years since its initiation. I have sometimes asked myself which were the explanations, and what I can do for enhancing the growth. Several aspects concurred to the interest for the membrane computing: the favorable context (the natural computing "fashion" mentioned in the beginning); the right moment, on the one hand, with respect to the DNA computing (which, in some sense, is covered and generalized by membrane computing), on the other hand, with respect to the theoretical computer science in general and the formal language theory in particular.

There are several things to be mentioned here. After four decades since the introduction of Chomsky grammars, the formal language theory became "classical" enough and got somewhat retired from the front research (almost completely in USA), even if there still exist specialized conferences (for instance, about finite automata and their applications) or more general conferences (DLT – Developments in Language Theory). Membrane computing appeared as a continuation and an extension of formal language theory: the main investigation objects are no longer the strings of symbols and the languages, but (I anticipate) the multisets of symbols and the sets of multisets. Strings, without taking into account the ordering of the symbols, more technically speaking, strings "seen" through the Parikh application,

the one which tells us the number of occurrences of each symbol in a given string. The consequence was that a large number of researchers in formal language theory became interested in the new research area. Among them, since the very beginning, important names, such as Arto Salomaa (Finland) and Grzegorz Rozenberg (The Netherlands), Oscar H. Ibarra (USA), Sheng Yu (Canada), Kamala Krithivasan (India), Takashi Yokomori (Japan), Mario J. Pérez-Jiménez (Spain), as well as very active researchers from my generation, such as Jürgen Dassow (Germany), Erzsébet Csuhaj-Varjú (Hungary), Jozef Kelemen (The Czech Republic), Rudolf Freund (Austria), Gheorghe Marian and Gabriel Ciobanu (Romania), Yurii Rogozhin (Republic of Moldova), Linqiang Pan (China), many of them coagulating around them research groups dedicated to membrane computing.

Somewhat surprising was the rapidly growing number of PhD students – now doctors – who have presented theses in membrane computing. There are over 50 at this moment. I mention only the first two, Shankara Narayanan Krishna (India) and Claudio Zandron (Italy), with the theses presented in 2001, respectively, in 2002. Starting with the summer of 2014, C. Zandron is the chairman of the steering committee of membrane computing.

A comprehensive information about the membrane computing area can be found at the domain website from the address `http://ppage.psystems.eu`, hosted in Vienna (it is the successor of a page which has functioned for many years in Milan, Italy, at the address `http://psystems.disco.unimib.it`).

Of course, it has counted very much the "sociology" of the domain. A *community* was soon created, and this is very important, not only in science, but in culture in general. They have contributed to that the seniors mentioned above, the yearly conferences (started in 2000, with the first three editions organized in Curtea de Argeş, Romania, where the meeting returned for the tenth edition and where I intend to also organize the twentieth edition) as well as a series of meetings which I would like to specially emphasize, one of a unusual format, which I have organized for the first time in Tarragona, Spain, in 2003. After that, it took place every year in Seville, also in Spain. Because it had to have a name, I called it "Brainstorming Week on Membrane Computing". One week when researchers interested in membrane computing work together, far from the current preoccupations, teaching or bureaucratic tasks. A very fruitful idea was to collect in advance open problems and research topics and to circulate them among the participants before the meeting in Seville, then addressed, in collaboration, during the Brainstorming. Very useful meetings – in the website of membrane computing one can find the yearly volumes, with the papers written or only started during the Brainstorming.

Very useful was, of course, the Internet. The first paper, "Computing with membranes", has waited more than one year before it was published in *Journal of Computer and System Sciences* (vol. 61, 2000, pages 108–143), but, because I was in Turku, Finland, in the autumn of 1998, I made the paper available on Internet, in the form of an internal report of TUCS, Turku Center for Computer Science (*Report No.* 208, 1998, www.tucs.fi). Until 2000, when the journal paper

has appeared, there were written some dozens of papers, making possible the organization of the first meeting dedicated to this topic, in Curtea de Argeş.

28 A Quick View on Membrane Computing

Let us not forget: we want to start from the cell and to construct a computing model. The result (the one proposed in the fall of 1988) is something of the following form. We look to the cell and we abstract it until we only see the *structure* of the hierarchically arranged *membranes*, defining *compartments* where *multisets of objects* are placed (I am using a generic term, abstract, free of any biochemical interpretation); these objects evolve according to given *reactions*. A multiset is a set with multiplicities associated with its elements, hence it can be described by a string; for instance, *aabcab* describes the multiset which contains three copies of a, two of b, and one of c. All permutations of the string *aabcab* describe the same multiset. The reactions, in their turn, are described by multiset "rewriting" rules, of the form $u \to v$, where u and v are strings which identify multisets. Initially (in the beginning of a computation), in the compartments of our system we have given multisets of objects. The evolution rules start to be applied, like biochemical reactions, in parallel, simultaneously, making evolve all objects which can evolve – and thus the multisets change. Using a rule $u \to v$ as above means to "consume" the objects indicated by u and to introduce the objects indicated by v. We have to notice that the objects and the rules are localized, placed in compartments, the rules in a given compartment are applied only to objects from that compartment. Certain objects can also pass through membranes. We proceed by applying rules until (like in the case of a Turing machine) we get stuck, no rule can be applied, and then the computation halts. The result of a halting computation is "read", for instance, in the form of the number of objects placed in a compartment specified in advance.

Processing of multisets (of symbols), in parallel, in the compartments defined by a hierarchical structure of membranes – this is the short description of a "P system". A distributed grammar, working with multisets of symbols – this is the direct connection with the formal language theory.

The working site starting here looks endless.

First, one can introduce a large number of variations of P systems, with a mathematical, computer science, biological motivation, or motivated by applications.

From the point of view of mathematics, the models should be minimalistic, they have to contain the smallest number of ingredients. For computer science, a computing model is good to be as powerful as possible, in the best case universal, equivalent with the Turing machine, and as efficient as possible, in the best case able to solve **NP**-complete problems in polynomial time.

Biology and applications provide a long list of alternatives, starting with the way of arranging the membranes (hierarchical, as in a cell, or placed in the nodes

of an arbitrary graph, as in tissues and other populations of cells), the types of objects (symbols as before, strings or even more complex data structures, such as graphs or bidimensional arrays), the form of the evolution rules (also dependent on the type of objects), the strategies of applying them, the way of defining the result of a computation.

I have mentioned before the multiset rewriting rules. They can be *arbitrary, non-cooperative* (with the left hand multiset consisting of a single object, which corresponds to context-free rules in Chomsky grammars), or, an intermediate case, *catalytic* (of the form $ca \to cv$, where c is a catalyst, an object which assists object a in its transformation to the multiset v). Then, we have the *symport* and *antiport* rules, which move objects from a compartment into another one (example: the antiport rule $(u, out; v, in)$, associated with a membrane, moves the objects indicated by u from this membrane to the surrounding compartment and the objects indicated by v in the opposite direction). Very important are the rules which *divide* membranes, because they increase, even exponentially, the number of membranes in the system. Many other types of rules were investigated (for instance, with a control on their application – with promoters, inhibitors, etc.), but I do not mention them here, the presentation would become too technical for the intentions of this text.

If the objects in the compartments of a system are strings, then they evolve by means of operations specific to strings: rewriting, insertion and deletion, or, in order to make the model more uniform from a biological point of view, by the splicing operation from the DNA computing.

An interesting situation is that when we work with symbol objects, hence with numbers, but the result of a computation is "read" outside the system, in the form of the string of the objects which are expelled from the system. It is worth noticing the qualitative difference between the internal data structure, the multiset, and the external one, the string, which carries out positional information.

In turn, the applications need a completely different strategy of constructing the models – far from minimalistic, but adequate to the modeled piece of reality; this time not the computing power is of interest, but the evolution in time of the system. I will come back to applications.

Over this small jungle of models one superposes the investigation program of the classic computer science: computing power, normal forms, descriptional complexity, computational complexity, simulation programs, etc., etc.

29 Classes of Results (and Problems)

Of course, I will not recall precise theorems, but I will only mention the two main classes of results in membrane computing and their general form.

Computational completeness/universality: most of the classes of P systems considered so far are equivalent with Turing machines, they are computationally complete. Because the proofs are constructive, in this way one also brings to membrane

computing the universality property in the sense of Turing (that is why we speak about computational completeness and universality as they would be synonymous). In most cases, this result is obtained for systems of a reduced, particular form, with a small number of membranes. For instance, cell-like P systems with only two membranes, using catalytic rules (hence not of the general form) can compute whatever the Turing machines can compute.

An important detail: *two* catalysts are sufficient. It is an open problem whether the P systems with only one catalyst are universal. The conjecture is that the answer is negative, but the proof still fails to appear. This is one of the most interesting types of open problems in membrane computing (many of them still open): identifying the precise borderline between universality and non-universality.

Efficiency: the classes of P systems which can grow (exponentially) the number of membranes can solve **NP**-complete problems in a polynomial time. The idea is to generate, in a polynomial time, an exponential working space and then to use it, in parallel, for examining the possible solutions to a problem. Membrane division helps, similarly the membrane creation, similarly other operations. Like in the case of the Adleman experiment, we have again a space-time trade-off, but in our case the space is not provided in advance, but it is created during the computation, through "mitosis" or by means of other "realistic" biological operations.

There are also in this area open problems concerning the borderline between efficiency and non-efficiency, but more difficult to be stated in plain words.

Interesting is a somewhat unexpected fact. Using rules of the form $a \to aa$, applied in parallel, we can produce an exponential number of copies of a in a linear number of steps. (In n steps, we get 2^n copies of a.) However, such an exponential working space is not of any help in solving high complexity problems in a feasible time– this is what the so-called *Milan theorem*, from Claudio Zandron PhD thesis, says. If these objects are localized, placed in an exponentially large number of membranes, then the situation is different. Otherwise stated, not only the size of the working space matters, but also its structure, the possibility to apply different rules in different compartments. This is a subtle aspect, which I do not know whether it has been met also in other frameworks.

For details, the reader is refereed to the monograph *Membrane Computing. An Introduction*, published by Springer-Verlag in 2002 (and recently translated in Chinese) and, especially, to *The Oxford Handbook of Membrane Computing*, edited by Gh. Păun, G. Rozenberg, and A. Salomaa and published by Oxford University Press, in 2010.

30 Significations for Computer Science and for Biology

A computing model which has the same power as the Turing machine is a good thing, such a computer is universal not only in the intuitive sense, but it is also programmable. Moreover we have here a distributed, parallel computer, with a great degree of non-determinism, controlled in various biologically inspired ways.

Let us, however, observe the similarities and the differences between a usual computer program, a set of instructions of a Turing machine, and a set of evolution rules of a P system. In the programming languages, the programs consist of precisely ordered instructions, perhaps labeled and addressed by means of these labels. In the case of the Turing machine, the sequence of instructions to be applied is determined by the states of the machine and by the contents of the tape. In the cell case, the reactions are potential, their set is completely unstructured, and their application depends on the available molecules. The evolution rules are just waiting for the data to which they can be applied, there is a competition between rules with respect to the objects to process.

The differences are visible and they suggest once again the question *what means to compute in a natural way?*, adding now the question whether we can work with programs in the form of completely unstructured sets of instructions.

On the other hand, in the first moment, it is expected that the biologist reaction to results of the type of the equivalence with the Turing machine is indifference, a raising of the shoulders. Another domain, another language, another book... But: if the cell is so powerful from a computational point of view, then, according to an old result, the Rice theorem ("all nontrivial problems – having both instances with a positive answer and instances with a negative answer – about a computing model equivalent with Turing machines are algorithmically undecidable"), no nontrivial question about the cell can be solved in an algorithmic way, by means of a program. The biologists formulate every day such questions: How a cell, a cell population, an organ or an organism evolves in time? Is there a substance which gets accumulated over a given threshold, in a given compartment? What happens if we add a multiset of molecules (a medicine), does the state of an organ improves (from specified points of view)? – and so on. If a model of the cell would be decidable, then we could find the answer to such questions by (algorithmically) examining the model, at a given initial state. But, because this is not possible (cannot be done in principle, not only we cannot do it now, here), what remains to do are the laboratory experiment (expensive and time consuming), the computer experiment (cheap, fast, but with the relevance depending on the quality of the model), and, theoretically, the non-algorithmic, ad-hoc, approach.

The previous paragraphs can be seen also as a plead for biology to learn new languages, in particular, the language of theoretical computer science, thus having the possibility of raising problems and of finding solutions which cannot appear, cannot be even formulated in the previous language. This would be an essential step towards infobiology.

31 Three Novel Computer Science Problems

In the continuation of the discussion about the significance for computer science, let us point out a remarkable fact: natural computing in general and membrane computing in particular raise theoretical questions which were not considered in

the framework of the classical computer science. Here are three questions of this kind, all three pertaining to complexity theory.

Like in the case of Adleman, most experiments of DNA computing started from an instance of a problem and constructed a "computer" associated with that instance. The standard complexity theory does not allow such an approach, it asks for *uniform* solutions, for programs/algorithms which start from the problem (and its size) and solve all instances of the problem. The idea is that during the programming stage one can already work on solving the problem, so that one can then pretend that the solution was found faster than it was the case in reality. That is why, also for the uniform solutions one limits the time allowed for programming, for constructing the algorithm. Let us then place a bound also on the programming time in the case when we start from an instance, so that we cannot cheat here either. The relationship between uniform solutions and semi-uniform (with a limited time for programming) solutions is not clarified yet, in spite of its importance for the natural computing. In membrane computing there were reported a series of related results – see, for instance, recent papers by Damien Woods (Caltech, USA), Niall Murphy (Microsoft Research, Cambridge, UK), Mario J. Pérez-Jiménez (Seville University, Spain).

Second: in DNA computing and in many models in membrane computing, at least part of the steps of a computation are of a non-deterministic type, but in the end the experiment/computation provides a unique result. The idea is to organize the computation in such a way that it is *confluent*, with two variants: either the system evolves non-deterministically for a while, then it "converges" to a unique configuration and then it continues in a deterministic way, or the system "converges logically", it gives the same result irrespective how it evolves. Again, the complexity theory lacks a study of these situations, of the cases intermediate between determinism and non-determinism.

Finally, the biology provides situations where extended resources wait for external challenges which activate a suitable portion of the resource. The examples of the brain and of the liver, from which we use at any given time only part of the huge number of available cells, are the most known. We can then imagine "computers" – for instance, neural networks – with an arbitrarily large number of cells/neurons, but containing only a limited quantity of information (not to hide there the solution of a problem); after introducing a problem in the system, one activates the necessary number of cells/neurons for solving it. There is no theory dealing with this strategy (of using *pre-computed resources*). How the pre-computed working space should look in order to contain only "a limited quantity of information", how this information can be defined and measured, when a system with pre-computed resources is acceptable/honest, it cannot hide the solution of a problem in its structure?

Natural computing not only motivates the improvement of old results in computer science, but it also makes necessary new developments, which were not imagined before.

32 About the Tools Used in Membrane Computing

In order to stress once again the relationships between various branches of theoretical computer science which, at the first sight, look far from each other, and the fact that membrane computing, the natural computing in general, use many old techniques and results, let me remind some details from my personal experience.

In the first universality proof for P systems I have used the result of Yuri Matijasevich mentioned also before, of characterizing the sets of numbers computed by Turing machines as solutions of diophantine equations. I have, however, soon realized that a simpler proof can be obtained starting from the characterization of the same sets of numbers with the help of the matrix grammars. The initial paper was published in this form. In this context it appears the necessity of improving some old results in this area. After a while, also the matrix grammars were replaced, the proofs are now based mainly on register machines, investigated already in the sixties.

A technique even older was useful in the first universality proof for H systems, namely the way of functioning of Post systems, which were introduced at the beginning of the years 1940. Adapted to the splicing operation, this has led to a technique called *rotate-and-simulate*, which has become almost standard for H systems and their variants.

In the first years of my research activity, I was much interested in matrix grammars and I have concluded this research with a monograph (published in Romanian, in 1981), extended after a while to a book (published by Springer-Verlag, in 1989), in collaboration with Jürgen Dassow, from Magdeburg, Germany, dedicated to all restrictions in the derivation of context-free grammars. The same happened with other domains which were useful in the membrane computing; the Marcus contextual grammars and the grammar systems are the most important of them.

In mathematics and computer science it is not possible to say in advance whether and when a subject or a result will be useful...

33 Spiking Neural P (SNP) Systems

A class of P systems inspired from the brain structure and functioning deserves to be separately discussed. It was introduced later than other models (M. Ionescu, Gh. Păun, T. Yokomori: "Spiking neural P systems", *Fundamenta Informaticae*, vol. 71, 2006, pages 279–308), but it seems that it will get earlier hardware implementations useful to computer science (details about this possibility can be found in the paper "The stochastic loss of spikes in spiking neural P systems: Design and implementation of reliable arithmetic circuits", by Zihan Xu, Matteo Cavaliere, Pei An, Sarma Vrudhula, published in *Fundamenta Informaticae*, vol. 134, issue 1-2, January 2014, pages 183–200).

In a few words, such a system consists of "neurons" linked through "synapses" along which circulate electrical impulses, produced in the neurons by means of specific rules. Like in the case of the real neurons (see, for instance, W. Maass: "Networks of spiking neurons: The third generation of neural network models", *Neural Networks*, vol. 10, 1997, pages 1659–1671), the communication among neurons is done by means of identical electrical impulses, *spikes*, for which the frequency is relevant, codifying information. Otherwise stated, important is the distance in time between spikes. In each moment, the axons are a sort of "bar codes", sequences of 0 and 1 which move from a neuron to another one. Obviously, the model ignores many neuro-biological details, but even at this reductionistic level we can formulate a series of questions concerning the relevance for computer science. In a certain sense, the SNP systems use the time as a support of information. The distance between two events, two spikes here, codifies a number. Can we construct a computer with such a "memory"? I mention the question only as a speculation – provocative at the theoretical level.

A result which deserves to be recalled refers to the search of SNP systems which are universal in the Turing sense, that is, they can be programmed in such a way to simulate any other SNP system. From the equivalence with the Turing machine, it follows immediately that such a system exists. The problem of interest concerns the number of neurons of an "universal brain" of this kind, able to simulate any computation in any particular system. This number is not at all too large. In the paper "Small universal spiking neural P systems", *BioSystems*, vol. 90, 2007, pages 48–60, by Andrei Păun and Gh. Păun, one uses 50 – 80 neurons, depending on the type of rules for producing spikes, but these numbers were subsequently decreased. In newspaper terms, we can say that "there are computationally universal brains consisting of only a few tens of neurons". From here we can either infer that a computing model of the form of SNP systems is very powerful, actually, that the neurons of these systems are *too powerful*, or that the Turing computability level is not very high – or both these conclusions. Of course, the human brain does not function as a Turing machine – but the computational paradigm was useful, in a certain extent, in modeling the brain functioning.

34 About Implementations

The DNA computing started by the definition of the splicing operation, in 1987, but about the possibility of using DNA molecules for computing there were discussions already one decade before. However, the domain became popular after Adleman experiment in 1994. An example was thus created, so that the question whether or not there are implementations of P systems is both natural and frequent. It is understood that one speaks about implementations on a biological substrate. The answer is negative. There were some attempts, but no successful experiment was reported.

An experiment of this kind was designed in the group of professor Ehud Keinan (with well known research both in chemistry and biology) from the Technion Insti-

tute in Haifa, Israel, where I have spent one week in 2006, exactly with this purpose. Two main related problems were identified from the beginning: finding a P system plausible to be implemented in a laboratory and, of course, finding the biochemical techniques necessary. We did not intend to solve an **NP**-complete problem, we have not found a reasonable one, but we have looked for a system whose behavior was illustrative for membrane computing (compartments, multisets, parallel processing), and we have chosen a system generating numbers in the Fibonacci sequence. The lab implementation seemed to be only a time issue – as well a question of money, for buying the laboratory equipments and the... DNA molecules. The plan was to simulate the membranes by means of the micro-chambers of a reconfigurable lab installation, with the objects being DNA molecules.

The first experiments did not succeed, then the... sociology of science struck again: the two PhD lady students who were in charge with this experiment moved to USA. In the meantime, an USA patent has appeared, on the name of Ehud Keinan, for implementing a P system, but using another technique, based on three non-miscible liquids placed in a common space. As far as I know, it is about a "theoretical implementation", no successful experiment was reported.

The question which naturally arises is whether or not such an experiment would bring something useful from the point of view of applications. Recalling a saying of Benjamin Franklin, "it is impossible to say what will become a newborn baby", but, having in mind the case of DNA computing, it is highly possible that this will only be a *demo*, at the level of simple calculations.

Completely different is the situation of implementations on an electronic hardware. There are several promising implementations on a parallel hardware (on NVIDIA graphic cards, in Seville, Spain), on a hardware especially designed for membrane computing (Madrid – Spain and Adelaide – Australia), on networks of computers, even on web. All these succeed in a great extent to capture the essential characteristics of P systems, the parallelism. Having in mind the parallelism, I do not call implementations, but *simulations* the cases when one uses standard sequential computers.

On the other hand, both the simulation programs and, still more, the implementations are useful in applications.

35 Applications

Membrane computing confirms an observation already made in several situations: when a mathematical theory, starting from a piece of reality, is sufficiently developed at the abstract, theoretical level, there are high chances to find applications not only in the domain which has inspired it, but in other areas too, some of them far away, at the first sight, from the reality from where the theory emerged (but having a common deep structure). It is, very convincingly, the present case.

It was just natural to return to the cell. Biology needs tools and models, the cell is not easy to model. It was stated that, after completing the human genome

reading, the main challenge for the bioinformatics is the modeling of the cell (M. Tomita: "Whole-cell simulation: A grand challenge of the 21st century", *Trends in Biotechnology*, vol. 19, 2001, pages 205–210). I have already mentioned that many of the models currently used in biology are based on differential equations. In many cases they are adequate, in many cases not. Differential equations belong to the mathematics of the continuum, they are appropriate to very large populations of molecules, uniformly stirred. However, in a cell, many molecules can be found in small numbers, therefore the approximation of the finite through the infinite, as necessary for applying differential equations, can lead to wrong results. This makes necessary the discrete models, in particular, the P systems, which also have other characteristics which are attractive for the biologist: they come from biology, hence they are easily understandable, which is an aspect which should not be underestimated; furthermore, P systems are algorithmic models, directly programmable in order to simulate them on the computer; can be easily extended, are scalable, adding new components, of any type, does not change the simulation program; their behavior is emergent, cannot be predicted by just looking to the components.

There are many applications of membrane computing in biology and biomedicine. From the individual cell, the applications passed to populations of cells (e.g., of bacteria) and then to... ecosystems. Here is only one title, a suggestive one: "Modeling ecosystems using P systems: The bearded vulture, a case study", by Mónica Cardona, M. Angels Colomer, Mario J. Pérez-Jiménez, Delfi Sanuy, and Antoni Margalida, the last two being biologists, experts in the ecology of the bearded vulture and animal protection from Lleida, Spain. Of course, the ecosystem is a metaphoric cell, while the "molecules" are the vultures, goats, wolves, hunters, all these in discrete quantities, small known numbers, with no possibility to be modeled with the instruments of the continuous mathematics. Other ecosystems which were investigated concern Panda bears in China and the zebra mussel from the water basins of the Spanish hydroelectrical plants.

So far, plausible applications. Not so expected are the applications in computer graphics (but in this respect we have a previous example, that of Lindenmayer systems), cryptography (in the organization of the attack against certain cryptographic systems), approximate optimization (distributed evolutionary computing, with the distribution organized like in a cell; the number of papers in this area is very large, the topic being popular in China, and the results are surprisingly and pleasantly good – with the mentioning that the famous no free lunch theorem should cool down also here the enthusiasm), economic modeling (a metaphorical extension similar to that to ecosystems), robot control.

These two last areas of applications are part of a potentially larger one, based on the use of the so-called *numerical P systems*, where, in a cell-like framework there evolve numerical variables, not molecules; the evolution is done by means of certain *programs*, consisting of a *production function* and a *repartition protocol*. The inspiration comes from economics (Gh. Păun, Radu Păun: "Membrane computing and economics: Numerical P systems", *Fundamenta Informaticae*, vol. 73, 2006,

pages 213–227). The systems of this kind compute functions of several variables, in a parallel way, and this computation is rather efficient, that is why it is expected that this somewhat exotic class of P systems will find further applications.

Details about applications can be found in the webpage of membrane computing, in the mentioned *Handbook*, as well as in the collective volumes *Applications of Membrane Computing* (edited by G. Ciobanu, Gh. Păun, M.J. Pérez-Jiménez) and *Applications of Membrane Computing in Systems and Synthetic Biology* (edited by P. Frisco, M. Gheorghe, M.J. Pérez-Jiménez), both of them published by Springer-Verlag, in 2006 and 2014, respectively.

36 Doubts, Difficulties, Failures

During ceremonies like the today one [delivering a Reception Speech in the Academy] or with the occasion of periodical reports, it is not usual, even not appropriate, to also speak about difficult moments, even if this would be instructive for the reader and useful for the domain.

On the other hand, the hesitations and the doubts are continuously a component of the researcher life. For instance, I can compile a long list of moments where my expectations were of a certain type and the results were different.

This happened starting with the mathematical results. For instance, in the beginning I did not believe that the catalytic P systems are universal, furthermore, that they are universal even in the case of using only two catalysts. Similarly, for a while I have expected to find a class of systems for which the number of membranes induces an infinite hierarchy (of the classes of sets of computed numbers). In exchange, almost always the universality is obtained with only one or two membranes. One membrane means no structure of the system, a trivial architecture. Of course, we can see here the positive fact: the (catalytic) processing of multisets is powerful enough in order to simulate a Turing machine.

Because I have in mind the case of the DNA computing, I do not count as a failure the fact that there are no biological implementations of P systems (although such an event would have a great publicity impact), but I still wait for an implementation on a parallel or a dedicated hardware having a "commercial" value. Such an implementation is necessary and, I believe, it is also possible. For instance, some years ago, a team of biologists and computer scientists from Nothingham, Sheffield (UK), and Seville have tried to simulate on a computer the communication among bacteria, modeling the so-called *quorum sensing*. The simulation programs were able to deal with hundreds of bacteria, the biologists wanted to pass to populations of thousands of bacteria. My expectation is that the implementations, for instance, on NVIDIA cards, will reach soon this level of magnitude requested by the biologists.

Concerning the applications in general, although they were not of interest at the beginning of membrane computing, at some moment it was clear that the domain cannot pass over a certain level of development and notoriety without "real"

applications. For a while, there were applications, but of the *postdiction*, not of the *prediction* type. The frequent scenario is the following: we take a biological phenomenon, discussed in a paper or in a book, we formalize it as a P system, we write a simulation program (or we take one available – at this moment we also have a specialized programming language, *P-lingua*, realized in the Seville University), we perform experiments with data from the paper or the book, and if the results are similar to those obtained in a laboratory or through other methods, we are happy. Postdiction, nothing new for the biologists, we only get more confidence in the new model and we can tune it with real data. In order to pass over this stage it is necessary to have a biologist in the team, who should come with a research question, with hypotheses which need to be checked. In turn, the computer scientist should come with sufficiently versatile models and with sufficiently efficient programs, in order to cope with the complexity of biological processes. After sixteen years, the bibliography of membrane computing applications is rather large – see the references from the previous section – although still we need biologists who have to come towards the computer scientists, maybe to learn membrane computing or, at least, to learn to use the instruments which the computer scientists have already realized (and tested).

I said before that I was continuously interested in forming a community – initially, this was an intuitive desire, later it became conscious, as this was a way to stabilize the domain against the dynamics of the groups. This looks as an external aspect, but we do not have to ignore the influence of the psycho-sociology on science, especially in the case of young branches. A group which is broken can mean a group less (it depends where its members land, whether they continue or not the research activity) or the apparition of several new groups, in new places. I have been the witness of both these two types of consequences. Fortunately, at the present time, the membrane computing community has dimensions which provide it with a comfortable inertia – which, however, does not mean that membrane computing will not get dissolved into infobiology, it already works for that...

37 At the Frontier of Science-Fiction

The main promise of natural computing is a better use of the existing computers, pushing forward the frontier of feasibility, by providing solutions, perhaps approximate, to problems which cannot be solved by means of traditional techniques. The DNA computing came with a more ambitious goal, that of providing a new type of hardware, of "biological chips", "wet processors", efficient not only in computational terms, but also in what concerns the energy consumption, or making plausible very attractive features, self-healing, adaptation, learning. Biology can suggest also new computer architectures or ideas for implementing other dreams of computer science, such as the parallel computation, the unsynchronized one, the control of distributed processes, the reversible computation and so on.

All these are somewhat standard expectations, but there also are some ideas which point out to the science of tomorrow, if not directly to science-fiction.

One of these directions is that which aims to *hypercomputability*, to "compute the uncomputable", to pass beyond the "Turing barrier". The domain is well developed, there are over one dozen basic ideas which lead to computability models stronger that the Turing machine – while physics does not forbid any one of these ideas, moreover, it even suggests ideas which look genuinely SF, like, e.g., the use of an internal time of the model which contains cycles or of a bidimensional time. It is true, Martin Davis ("The myth of hypercomputation", in *Alan Turing: The Life and Legacy of a Great Thinker*, C. Teuscher, ed., Springer, 2004, pages 195-212) considers all of them tricks by which the computing power is introduced in the model from the very beginning, in disguise, and then one proves that the model passes beyond the Turing machine (for example, one considers real numbers, which can codify, in their infinite sequence of decimals, all possible computations), but there also are some ideas which look more realistic that others.

One of them is that of *acceleration*, already discussed several decades ago, not only in computer science: R. Blake (1926), H. Weyl (1927), B. Russell (1936), have imagined processes which need one time unit (measured by an external clock) for performing the first step, half of a time unit for the second step (the process "learns"), and so on, at every step, half of the time needed by the previous step. In this way, in two time units (I insist: external, measured by the observer) one performs infinitely many (internal) steps. Such an accelerated Turing machine can solve the halting problem, hence it is more powerful than a usual Turing machine.

Let us now remember the observation that nature creates new membranes in order to get small reactors, where the reactions are enhanced, because of the higher possibilities of molecules to collide. Consequently, *smaller is faster*. The biochemistry in an inner membrane is faster than in the surrounding membrane. Let us push the speculation to the end and assume that the "life" in a membrane is twice faster than in the membrane containing it. Exactly the acceleration we have mentioned above. One can prove (C. Calude, Gh. Păun: "Bio-steps beyond Turing", *BioSystems*, vol. 77, 2004, pages 175–194) that, exactly as in the case of the accelerated Turing machine, an accelerated P system (able to repeatedly create inner membranes) can decide the halting problem.

Hypercomputability can seem to be only a mathematical exercise, but it is estimated that passing beyond the Turing barrier could have more important consequences than finding a proof, even an efficient one, of the $\mathbf{P} = \mathbf{NP}$ equality; see, for instance, B.J. Copeland: "Hypercomputation", *Minds and Machines*, vol. 12, 2002, pages 461–502.

Let us get closer to the laboratory. I have mentioned the lab implementation of a finite automaton with an autonomous functioning. A finite automaton can parse strings. The genes are strings, the viruses are strings (of nucleotides). A hope of medicine is to cure illnesses by editing genes, to eliminate viruses by identifying them and then cutting them in pieces. A more efficient idea than to introduce medicines in out body is to construct a "machinery" which can recognize and edit the necessary sequences of nucleotides, genes or viruses. To this aim, we need a carrying vector, to bring the gene editor in the right place. The identification of

that place can be done by an automaton, possibly a finite one, while the vector can be a sort of nano-carrier which can be also built from DNA molecules. In short, un nano-robot, suitably multiplied, which can move from a cell to another one, curing what it is necessary to be cured. A pre-project of such a nano-robot was presented in 2004, by Y. Benenson, E. Shapiro, B. Gill, U. Ben-Dor, R. Adar ("Molecular computer. A 'smart drug' in a test tube"), to the tenth edition of the DNA Computing Conference organized in Milan, Italy. In a great extent, it was the same team which has implemented the autonomous finite automaton mentioned before.

There still are many things to be done, the possibility to have our body continuously scanned by a gene repairing robot is not at all close to us. (Such a robot can also have malevolent tasks, it can be a weapon – one can open here a discussion about the ethics of research, but there are sufficiently many debates of this type, even in bio-computer science. Also Francis S. Collins speaks about bioethics in *The Language of God*, the book mentioned several pages before.) On the other hand, there are numerous nano-constructions made of DNA, "motors", "robots", etc. The nano-technology based on DNA biochemistry is spectacularly developed. I cite, as a reference, the paper J.H. Reif, T.H. LaBean, S. Sahu, H. Yan, P. Yin: "Design, simulation, and experimental demonstration of self-assembled DNA nanostructures and motors", *Proceedings of the Workshop on Unconventional Programming Paradigms*, UPP04, Le Mont Saint-Michel, September 2004.

It is worth mentioning here also an observation made by Jana Horáková and Jozef Kelemen in "Capek, Turing, von Neumann, and the 20th century evolution of the concept of machine", from *Proceedings of the International Conference in Memoriam John von Neumann*, Budapest Polytechnic, 2003, pages 121–135, with respect to the evolution of computers, somewhat in parallel with the evolution of the idea of a robot: from organic to electromagnetic, then to electronic, and in the end tending to return to organic.

Further speculations? Without any limits, starting from facts with a solid scientific background. In the extreme edge, one can mention Frank Tipler, with his controversial eternal life, in informational terms, which is nothing else than artificial life at the scale of the whole universe (F. Tipler: *The Physics of Immortality*, Doubleday, New York, 1994). In any case, we have to be conscious that all these are plans for tomorrow formulated today in the yesterday language, to cite a saying of Antoine de Saint-Exupéry. The progresses in bioengineering can bring surprises which we cannot imagine in this moment.

38 Do We Dream Too Much?

Let us come down on the Earth, to the reality, to the natural computing as we have it now and how it is plausible to have it in the near future, adopting a lucid position, even a skeptical one, opposed to the enthusiasm from the previous section and to the enthusiasm of many authors. (I am not referring here to newspaper authors, which too often use big words when talking about bioinformatics.)

In order to promote an young scientific branch, the enthusiasm is useful and understandable – but natural computing is no longer an young research area. Let us oppose here to the previous optimism a more realistic position, starting from the differences, many and significant, between computer science and biology, from the difficulties to implement bio-ideas in computer science and computations in cells: the goal of life is life, not the computations, we, the computer scientists, see everywhere computations and try to use them for us; in a certain sense, life has unbounded time and resources, it affords to make experiments and to discard the results of unsuccessful attempts – all these are difficult to extend to computers, even if they are based on biomolecules. Similarly, life has a great degree of redundancy and non-determinism. Then, the biological processes have a high degree of complexity, moreover, they seem to mainly use the mathematics of approximations, probabilities, fuzzy sets, all of which are difficult to be captured in a computing model, not to speak about the difficulty to implement them.

Still more important: we perhaps dream too much even from the theoretical point of view. First, the space-time trade-off does not redefine the complexity classes, at most it can enlarge the feasibility space (see again Hartmanis remarks about Adleman experiment).

Then, there is a theorem of Michael Conrad ("The price of programmability", in the volume *The Universal Turing Machine: A Half-Century Survey*, R. Herken, ed., Kammerer and Unverzagt, Hamburg, 1988, pages 285–307) which says that three desired characteristics of a computer, *programmability* (universality), *efficiency*, and *evolvability* (the capacity to adapt and learn), are contradictory, there is no computer which can have all these three features at the same time. We can interpret this result as a general *no free lunch* theorem for the natural computing.

A similar theorem of limitation of "what can be done in principle" belongs to Robin Gandy, a student and collaborator of Turing, which offers general mathematical arguments to Martin Davis: the hypercomputability is a difficult thing to reach (see, for instance, the paper by R. Gandy "Church's thesis and principles for mechanisms", in the volume *The Kleene Symposium*, J. Barwise et al., eds., North-Holland, Amsterdam, 1980, pages 123–148). Gandy wanted to free the Turing-Church thesis of any anthropic meaning (in Turing formulation, the thesis says that "everything which can be computed by a human being can be computed by a Turing machine"). To this aim, he has defined a general notion of a "computing machine", described by four properties formulated mathematically and which any "computer", an actual or a theoretical one, should possess. Then, Gandy proved that any machine having these properties can be simulated by a Turing machine.

Passing from theoretical computer science to applications, let me notice that there are visible limitations also in this respect. I am even convinced that, if one will make lists with the properties the models and the simulations we would like to have (adequacy, relevance, accuracy, efficiency, understandability, programmability, scalability and so on), then impossibility theorems similar to Arrow, Conrad,

Gödel theorems will be proved concerning the modeling and the simulation of the cell – the very task which M. Tomita formulated.

39 Everything is New and Old All Are...

(The title of this section reproduces a verse from a poem by Mihai Eminescu, the national poet of Romania.)

In spite of what was said above, there is a more and more visible interest in the modeling of the cell. Actually, a dedicated research direction was proposed, the *systems biology*, with several programmatic papers, published in high visibility journals, such as *Science* and *Nature*. The main promotor was H. Kitano ("Systems biology: A brief overview", *Science*, vol. 295, March 2002, pages 1662–1664, "Computational systems biology", *Nature*, vol. 420, November 2002, pages 206–210), which has in mind a general model of the cell, meant to be simulated on a computer and then used, in relation also with other computer science and biological instruments, in such a way "to transform biology and medicine in a precise engineering". The goal is important and probably feasible in a medium-long term, but the insistence with which one speaks about "systems biology" as about a novel idea made Olaf Wolkenhauer to ask already in the title of his paper from *Bioinformatics* (vol. 2, 2001, pages 258–270) whether this is not only "the reincarnation of systems theory applied in biology". The paper recalls the efforts in this respect made in the years 1960, with the disappointments appeared at that time, due, among others, to the limits of the computers (but also to the limits of biology: let us remember that the Singer-Nicolson model of the membrane as a "fluid mosaic" dates only from 1972). But, besides the computing power, it is possible that something else was missing, which is perhaps missing even today, both in computer science and in biology. The last paragraph from Olaf Wolkenhauer paper invokes the name of Mihailo Mesarovic, a classic of systems theory, which, in 1968, said: "in spite of the considerable interest and efforts, the application of systems theory in biology has not quite lived up the expectations... One of the main reasons for the existing lag is that systems theory has not been directly concerned with some of the problems of vital importance in biology". His advice for biologists, continues Olaf Wolkenhauer, is that such a progress can only be obtained by means of a stronger direct interaction with the systems theory researchers. "The real advance in the applications of systems theory to biology will come about only when the biologists start *asking questions* which are based on the system-theoretic concepts rather than using these concepts to represent in still another way the phenomena which are already explained in terms of biophysical or biochemical principles... then we will not have *the application of engineering principles to biological problems*, but rather a field of *systems biology* with its own identity and in its own right." (M.D. Mesarovic: "System theory and biology – view of a theoretician", in *System Theory and Biology*, M.D. Mesarovic, ed., Springer, New York, 1968, pages 59–87)

Mesarovic words can be taken as a motto of infobiology in favor of which the whole present text pleads.

The transformation of biology and medicine in "a precise engineering" can be also related with the current difficulties to understand what is life, materialized, among others, in the current limits of the artificial intelligence and artificial life. One says, for instance, that up to now the computers are good in IA, the intelligence amplification, but not equally good in AI, artificial intelligence. Still less progresses were made in what concerns the artificial life. In terms of Rodney Brooks ("The relationship between matter and life", *Nature*, vol. 409, January 2001, pages 409–411), this suggests that "we might be missing something fundamental and currently unimagined in our models of biology". Computers are good in crunching numbers, but "not good at modeling living systems, at small or large scale". The intuition is that life is more than biophysics and biochemistry, but what else it is can be something which we cannot imagine today, "some aspects of living systems which are invisible to us right now". "It is not completely impossible that we might discover some new properties of biomolecules or some new ingredient". An example of such a "new stuff", R. Brooks says, can be the quantum effects from the microtubules of the neural cells, which, according to Penrose, "might be the locus of consciousness at the level of the individual cell" (citation from R. Brooks).

A similar opinion was expressed by another great name of the artificial intelligence, John McCarthy ("Problems and projection in CS for the next 49 years", *Journal of the ACM*, vol. 50, 2003, pages 73–79): "Human–level intelligence is a difficult scientific problem and probably needs some new ideas. These are more likely to be invented by a person of genius than as part of a Government or industry project".

Anyway, the progresses related to the collaboration between computer science and biology should not be underestimated. If we do it, then we take a risk which has struck big names of science and cultures. I close with a funny example of this kind, some statements (dated around 1830) of the French philosopher Auguste Comte: "Every attempt to employ mathematical methods in the study of biological questions must be considered profoundly irrational and contrary to the spirit of biology. If mathematical analysis should ever hold a prominent place in biology – an aberration which is happily almost impossible – it would occasion a rapid and widespread degeneration of that science."

Thanks to God, the philosopher was wrong – but we needed about two hundred years to see that...

40 (Provisory) Last Words

I hope that this quick description was convincing in showing that the way from biology to computer science and back to biology is intellectually fascinating and useful to both sciences.

A few things should be remembered: (i) in all its history, computer science tried to learn from biology, (ii) and this effort brought important benefits to computer science and equally to biology; (iii) the progresses in this area should not be underestimated, (iv) but, in general, it is plausible that we expect too much (and too fast) from the computer science-biology symbiosis, (v) because we ignore the essential differences between the two universes, the inherent limits of computability and the fact that biology is not a mathematically formalized science, (vi) with the mentioning that it is possible to need a new mathematics in order to model and simulate life and intelligence; finally, (vii) let me anticipate a new age of biology, beyond the today bioinformatics and the today natural computing, and let me also propose a name for it, *infobiology*.

Should we wait two further decades in order to see it taking shape?

From an intellectual point of view, during the forty years which I have told about here I have lived around academician Solomon Marcus, a "big tree" which invalidates the phrase ("In the shadow of big trees not even the grace is growing.") by which Constantin Brancusi motivated his decision to refuse to work under the guidance of Rodin: professor Solomon Marcus never puts shadow on his numerous students and collaborators, but on the contrary. I repeat, in order to stress it: on the contrary. I witness this and I dedicate to him this discourse, thanking him once again.

Minimal Cooperation in P Systems with Symport/Antiport: A Complexity Approach

Luis Valencia-Cabrera[1], Bosheng Song[2], Luis F. Macías-Ramos[1], Linqiang Pan[2], Agustín Riscos-Núñez[1], and Mario J. Pérez-Jiménez[1]

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

[2] Key Laboratory of Image Information Processing and Intelligent Control,
School of Automation, Huazhong University of Science and Technology,
Wuhan 430074, Hubei, China
E-mail: boshengsong@163.com, lqpan@mail.hust.edu.cn

Summary. Membrane systems with symport/antiport rules compute by just moving objects among membranes, and not by changing the objects themselves. In these systems the environment plays an active role because, not only it receives objects from the system, but it also sends objects into the system. Actually, in this framework it is commonly assumed that an arbitrarily large number of copies of some objects are initially available in the environment. This special feature has been widely exploited for the design of efficient solutions to computationally hard problems in the framework of tissue like P systems able to create an exponential workspace in polynomial time (e.g. via cell division or cell separation rules).

This paper deals with cell-like P systems which use symport/antiport rules as communication rules, and the role played by the *minimal cooperation* is studied from a computational complexity point of view. Specifically, the limitations on the efficiency of P systems with membrane separation whose symport/antiport rules involve at most two objects are established. In addition, a polynomial time solution to HAM-CYCLE problem, a well known **NP**-complete problem, by using a family of such kind of P systems with membrane division, is provided. Therefore, in the framework of cell-like P systems with minimal cooperation in communication rules, passing from membrane separation to membrane division amounts to passing from tractability to **NP**–hardness.

1 Introduction

The **P** versus **NP** problem is one of the most important open problems in theoretical computer science. Broadly speaking, we can say that this problem analyzes whether or not *finding solutions* is harder than *checking the correctness* of possible

solutions. It is widely believed that it is harder *to solve* a problem than *to check* that a solution is valid/good; that is, it is widely believed that $\mathbf{P} \neq \mathbf{NP}$. The classical approach to solve this problem consists on considering a single **NP**–complete problem and trying to prove whether that problem belongs to the class **P** or not. In the first case, the answer of the conjecture is negative. If the **NP**–complete problem considered does not belong to **P**, then the answer of the conjecture is positive.

In this paper we follow the lines of previous works [3, 4, 5, 6, 8, 10, 13, 14], and new tools to tackle the **P** versus **NP** problem are given in the framework of *Membrane Computing* paradigm. Specifically, we deal with cell-like P systems whose communication is implemented by means of symport/antiport rules abstracting trans-membrane transport of couples of chemical substances, in the same or in opposite directions. Besides, in order to achieve the efficiency of these models, membrane division rules abstracting cell division process and membrane separation rules inspired by membrane fission process, are also included. It is worth pointing out some relevant differences of cell-like approach with respect to tissue-like approach. First, communication rules are not given in a single set within the description of the model, but are organized into subsets, each one of them associated with a membrane label. Second, the structure of the system is a rooted tree given in an explicit way, instead of a directed graph given by means of the set of rules of the system. Third, communication is only produced between inner compartments if they have a parent-child relationship, and the communication with the environment is restricted to the skin membrane. Finally, only elementary membranes can be divided.

In the framework of cell-like P systems which use symport/antiport rules working with minimal cooperation (at most two objects are involved in these rules), we analyze the role played by membrane division and membrane separation as a tool to create an exponential workspace in linear time. On the one hand, we study the limitations on the efficiency of this kind of P systems with membrane separation; that is, we prove that the corresponding polynomial complexity class, denoted by $\mathbf{PMC}_{\mathbf{CSC}(\mathbf{2})}$, is equal to class **P**. On the other hand, we analyze the efficiency of the systems that use membrane division instead of membrane separation, by giving a polynomial time solution to `HAM-CYCLE` problem (that is, showing that $\texttt{HAM-CYCLE} \in \mathbf{PMC}_{\mathbf{CDC}(\mathbf{2})}$). Therefore, in the framework of cell-like P systems with minimal cooperation in communication rules, passing from membrane separation to membrane division amounts to passing from tractability to **NP**–hardness.

The paper is structured as follows. We first recall some preliminaries concerning definitions, concepts and results needed in order to make the paper self-contained. The specific models of cell-like P systems with symport/antiport rules that we use in this work and the corresponding complexity classes are introduced in Section 3.1. Next section is devoted to analyze the limitations about the computational efficiency of P systems with minimal cooperation which use membrane separation rules. Section 5 presents a polynomial time solution of `HAM-CYCLE` problem by means of a family of P systems with membrane division using symport/antiport

rules with length at most 2. Conclusions and some open problems are formulated at the last section.

2 Preliminaries

2.1 Languages and Multisets

An *alphabet* Γ is a non-empty set and their elements are called *symbols*. A *string* u over Γ is a mapping from a natural number $n \in \mathbb{N}$ onto Γ. Number n is called *length* of the string u and it is denoted by $|u|$. The empty string (with length 0) is denoted by λ. A *language* over Γ is a set of strings over Γ.

A *multiset* over an alphabet Γ is an ordered pair (Γ, f), where f is a mapping from Γ onto the set of natural numbers $\mathbb{N}$. For each $x \in \Gamma$ we say that $f(x)$ is the *multiplicity* of x in that multiset. The *support* of a multiset $m = (\Gamma, f)$ is defined as $supp(m) = \{x \in \Gamma \mid f(x) > 0\}$. A multiset is finite if its support is a finite set. The *size* of a finite multiset m, denoted by $|m|$, is the sum of the multiplicities of each element of Γ (obviously that sum is a natural number). We denote by $\emptyset$ the empty multiset. Let us note that a set is a particular case of a multiset where each symbol of the support has multiplicity 1.

Let $m_1 = (\Gamma, f_1)$, $m_2 = (\Gamma, f_2)$ be multisets over Γ. Then, the union of m_1 and m_2, denoted by $m_1 + m_2$, is the multiset (Γ, g), where $g(x) = f_1(x) + f_2(x)$ for each $x \in \Gamma$. We say that m_1 is contained in m_2, and we denote it by $m_1 \subseteq m_2$, if $f_1(x) \leq f_2(x)$ for each $x \in \Gamma$. The relative complement of m_2 in m_1, denoted by $m_1 \setminus m_2$, is the multiset (Γ, g), where $g(x) = f_1(x) - f_2(x)$ if $f_1(x) \geq f_2(x)$, and $g(x) = 0$ otherwise.

2.2 Graphs and Hamiltonian cycles

Let us recall that a *free tree* (*tree*, for short) is a connected, acyclic, undirected graph. A *rooted tree* is a tree in which one of the vertices (called *the root of the tree*) is distinguished from the others. In a rooted tree, the concepts of ascendants and descendants are defined in a usual way. Given a node x (different from the root), if the last edge on the (unique) path from the root of the tree to the node x is $\{x, y\}$ (in this case, $x \neq y$), then y is **the** *parent* of node x and x is **a** *child* of node y. The root is the only node in the tree with no parent. A node with no children is called a *leaf* (see [1] for details).

Let $G = (V, E)$ be a directed graph, where $V = \{1, \ldots, n\}$ and the set of arcs is $E = \{(u_1, v_1), \ldots, (u_m, v_m)\} \subset V \times V$. We say that a finite sequence $\gamma = (u_{\alpha_1}, u_{\alpha_2}, \ldots, u_{\alpha_r}, u_{\alpha_{r+1}})$ of nodes of G is a *simple path of G of length* $r \geq 1$ if the following holds:

- $\forall i \ (1 \leq i \leq r \rightarrow (u_{\alpha_i}, u_{\alpha_{i+1}}) \in E)$.
- $|\{u_{\alpha_1}, u_{\alpha_2}, \ldots, u_{\alpha_r}\}| = r$.

If $u_{\alpha_{r+1}} \notin \{u_{\alpha_1}, u_{\alpha_2}, \ldots, u_{\alpha_r}\}$, then we say that γ is a simple path of length r from u_{α_1} to $u_{\alpha_{r+1}}$. If $u_{\alpha_{r+1}} = u_{\alpha_1}$ and $r \geq 2$, then we say that γ is a *simple cycle* of length r.

A *Hamiltonian path* of G from $a \in V$ to $b \in V$ $(a \neq b)$ is a simple path $\gamma = (u_{\alpha_1}, u_{\alpha_2}, \ldots, u_{\alpha_r}, u_{\alpha_{r+1}})$ from a to b such that $a = u_{\alpha_1}$, $b = u_{\alpha_{r+1}}$, and $V = \{u_{\alpha_1}, u_{\alpha_2}, \ldots, u_{\alpha_r}, u_{\alpha_{r+1}}\}$. A *Hamiltonian cycle* of G is a simple cycle $\gamma = (u_{\alpha_1}, u_{\alpha_2}, \ldots, u_{\alpha_r}, u_{\alpha_{r+1}})$ of G such that $V = \{u_{\alpha_1}, u_{\alpha_2}, \ldots, u_{\alpha_r}\}$.

If $\gamma = (u_{\alpha_1}, u_{\alpha_2}, \ldots, u_{\alpha_r}, u_{\alpha_{r+1}})$ is a simple path of G then we also denote it by the set $\{(u_{\alpha_1}, u_{\alpha_2})_1, (u_{\alpha_2}, u_{\alpha_3})_2, \ldots, (u_{\alpha_r}, u_{\alpha_{r+1}})_r\}$. That is, $(u_{\alpha_k}, u_{\alpha_{k+1}})_k$ can be interpreted as the k-th arc of the path γ, for each k $(1 \leq k \leq r)$.

Let $G = (V, E)$ be a directed graph with $V = \{1, \ldots, n\}$. Throughout this paper, $A_G = \{(i,j)_k \mid i, j, k \in \{1, \ldots, n\} \wedge (i,j) \in E\}$, $A'_G = \{(i,j)'_k \mid (i,j)_k \in A_G\}$ and $A''_G = \{(i,j)''_k \mid (i,j)_k \in A_G\}$.

Proposition 2.1 *Let $G = (V, E)$ be a directed graph such that $V = \{1, \ldots, n\}$. If $B \subseteq A_G$ then the following assertions are equivalent:*

1. *B is a Hamiltonian cycle.*
2. *$|B| = n$ and the following holds: for each $i, i', j, j', k, k' \in \{1, \ldots, n\}$,*
   (a) $[(i,j)_k \in B \wedge (i',j')_{k'} \in B \wedge (i,j)_k \neq (i',j')_{k'} \rightarrow k \neq k']$
   (b) $[(i,j)_k \in B \wedge (i',j')_{k'} \in B \wedge (i,j)_k \neq (i',j')_{k'} \rightarrow i \neq i']$
   (c) $[(i,j)_k \in B \wedge (i',j')_{k'} \in B \wedge (i,j)_k \neq (i',j')_{k'} \rightarrow j \neq j']$
   (d) $[(i,j)_k \in B \wedge (i',j')_{k+1} \in B \rightarrow j = i']$

Proof: Let $B = \{(u_{\alpha_1}, u_{\alpha_2})_1, (u_{\alpha_2}, u_{\alpha_3})_2 \ldots, (u_{\alpha_m}, u_{\alpha_{r+1}})_n\}$ be a Hamiltonian cycle of G. Then, $|B| = n$ and conditions (a), (b), (c) and (d) from (2) hold.

Let $B \subseteq A_G$ such that $|B| = n$ and conditions (a), (b), (c) and (d) from (2) hold. Then, from (a) the set B must to be of the form

$$B = \{(u_{\alpha_1}, v_{\alpha_1})_1, (u_{\alpha_2}, v_{\alpha_2})_2 \ldots, (u_{\alpha_n}, v_{\alpha_n})_n\}$$

where:

- From (d) we deduce that $\forall s$ $(1 \leq s \leq n-1 \rightarrow v_{\alpha_s} = u_{\alpha_{s+1}})$.
- From (b) we have $V = \{u_{\alpha_1}, u_{\alpha_2}, \ldots, u_{\alpha_n}\}$.

Finally, on the one hand we have $v_{\alpha_n} \in \{u_{\alpha_1}, u_{\alpha_2} \ldots, u_{\alpha_n}\}$. On the other hand, by condition (c) we deduce that $v_{\alpha_n} \notin \{v_{\alpha_1}, \ldots, v_{\alpha_{n-1}}\} = \{u_{\alpha_2}, \ldots, u_{\alpha_n}\}$. Thus, $v_{\alpha_n} = u_{\alpha_1}$.

□

Remark 1: Let $B \subseteq A_G$ be a Hamiltonian cycle of G. For each $i, i', j, j', k, k' \in \{1, \ldots, n\}$ the following holds:

1. If $(i,j)_k \in B$ and $j \neq j'$ then $(i,j')_{k'} \notin B$.
2. If $(i,j)_k \in B$ and $i \neq i'$ then $(i',j)_{k'} \notin B$.
3. If $(i,j)_k \in B$ and $(i,j) \neq (i',j')$ then $(i',j')_k \notin B$.
4. If $(i,j)_k \in B$ and $(i',j')_{k+1} \in B$ then $j = i'$.

Remark 2: Let us notice that if $(u_{\alpha_1}, u_{\alpha_2}, \ldots, u_{\alpha_n}, u_{\alpha_1})$ is a Hamiltonian cycle of G of length n, then we can describe it by the following subset of A_G:

$$B_1 = \{(u_{\alpha_1}, u_{\alpha_2})_1, (u_{\alpha_2}, u_{\alpha_3})_2, \ldots, (u_{\alpha_n}, u_{\alpha_1})_n\}$$

But $(u_{\alpha_2}, u_{\alpha_3}, \ldots, u_{\alpha_m}, u_{\alpha_1}, u_{\alpha_2})$ also represents the same Hamiltonian cycle. It can be described as follows: $B_2 = \{(u_{\alpha_2}, u_{\alpha_3})_1, (u_{\alpha_3}, u_{\alpha_4})_2, \ldots, (u_{\alpha_1}, u_{\alpha_2})_n\}$. Thus, given a Hamiltonian cycle γ of G, there are exactly n different subsets of A_G codifying that cycle.
Remark 3: Let us suppose that the total number of Hamiltonian cycles of G is q. Then, the number of different subsets B of A_G verifying conditions (a), (b), (c), and (d) from Proposition 2.1 is exactly $n \cdot q$.

2.3 Encoding ordered pairs of natural numbers

The *pair function* $\langle n, m\rangle = ((n + m)(n + m + 1)/2) + n$ is a polynomial–time computable function from $\mathbb{N} \times \mathbb{N}$ onto $\mathbb{N}$ which is also a primitive recursive and bijective function.

3 P systems with symport/antiport rules

In this section we introduce a kind of cell-like P systems that use communication rules capturing the biological phenomenon of trans-membrane transport of several chemical substances. Specifically, two processes have been considered. The first one allows a multiset of chemical substances to pass through a membrane in the same direction. In the second one, two multisets of chemical substances, located in different biological membranes, only pass with the help of each other (yielding an *exchange* of objects between both membranes).

Next, we introduce an abstraction of these operations in the framework of P systems with symport/antiport rules following [9]. In these models, the membranes are not polarized.

Definition 1. *A P system with symport/antiport rules* (*SA P system, for short*) *of degree* $q \geq 1$ *is a tuple* $\Pi = (\Gamma, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \ldots, \mathcal{R}_q, i_{in}, i_{out})$, *where:*

1. Γ *is a finite alphabet;*
2. $\mathcal{E} \subseteq \Gamma$*;*
3. Σ *is an (input) alphabet strictly contained in* Γ *such that* $\mathcal{E} \subseteq \Gamma \setminus \Sigma$*;*
4. μ *is a rooted tree whose nodes are injectively labelled by* $1, \ldots, q$ (*the root of the tree is labelled by* 1)*;*
5. $\mathcal{M}_1, \ldots, \mathcal{M}_q$ *are finite multisets over* $\Gamma \setminus \Sigma$*;*
6. $\mathcal{R}_i$, $1 \leq i \leq q$, *are finite sets of communication rules over* Γ *of the form:*
   (*a*) Symport rules*:* (u, out) *or* (u, in), *where* u *is a finite multiset over* Γ *such that* $|u| > 0$*;*

(*b*) Antiport rules*:* $(u, out; v, in)$*, where* u, v *are finite multisets over* Γ *such that* $|u| > 0$ *and* $|v| > 0$*;*

7. $i_{in} \in \{1, \ldots, q\}$ *and* $i_{out} \in \{0, 1, \ldots, q\}$.

A SA P system of degree q $\Pi = (\Gamma, \mathcal{E}, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \ldots, \mathcal{R}_q, i_{out})$ can be viewed as a set of q membranes, labelled by $1, \ldots, q$, arranged in a hierarchical structure μ (given by a rooted tree whose root is called the *skin membrane*), such that: (a) $\mathcal{M}_1, \ldots, \mathcal{M}_q$ represent the finite multisets of objects initially placed into the q membranes of the system; (b) $\mathcal{E}$ is the set of objects initially located in the environment of the system (labelled by 0), all of them available in an arbitrary number of copies; (c) $\mathcal{R}_1, \ldots, \mathcal{R}_q$ are finite sets of communication rules over Γ ($\mathcal{R}_i$ is associated with the membrane i of μ); and (d) i_{out} represents a distinguished *region* which will encode the output of the system. We use the term *region* i ($0 \leq i \leq q$) to refer to membrane i in the case $1 \leq i \leq q$ and to refer to the environment in the case $i = 0$. The length of rule (u, out) or (u, in) (resp. $(u, out; v, in)$) is defined as $|u|$ (resp. $|u| + |v|$).

For each membrane $i \in \{2, \ldots, q\}$ (different from the skin membrane) we denote by $p(i)$ the parent of membrane i in the rooted tree μ. We define $p(1) = 0$, that is, by convention the "parent" of the skin membrane is the environment.

An *instantaneous description* or a *configuration* at an instant t of a SA P system is described by the membrane structure at instant t, all multisets of objects over Γ associated with all the membranes present in the system, and the multiset of objects over $\Gamma \setminus \mathcal{E}$ associated with the environment at that moment. Recall that we assume that there are infinitely many copies of objects from $\mathcal{E}$ in the environment, and hence it does not make sense to keep record of their multiplicity along the computation. The *initial configuration* of the system is $(\mu, \mathcal{M}_1, \ldots, \mathcal{M}_q; \emptyset)$.

A symport rule $(u, out) \in \mathcal{R}_i$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant t if membrane i is in $\mathcal{C}_t$ and multiset u is contained in that membrane. When applying a rule $(u, out) \in \mathcal{R}_i$, the objects specified by u are sent out of membrane i into the region immediately outside (the parent $p(i)$ of i), which can be the environment in the case of the skin membrane. A symport rule $(u, in) \in \mathcal{R}_i$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant t if membrane i is in $\mathcal{C}_t$ and multiset u is contained in the parent of i. When applying a rule $(u, in) \in \mathcal{R}_i$, the multiset of objects u is taken from the parent membrane of i and enters into the region defined by membrane i.

An antiport rule $(u, out; v, in) \in \mathcal{R}_i$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant t if membrane i is in $\mathcal{C}_t$ and multiset u is contained in that membrane, and multiset v is contained in the parent of i. When applying a rule $(u, out; v, in) \in \mathcal{R}_i$, the objects specified by u are sent out of membrane i into the parent of i and, at the same time, the objects specified by v are brought into membrane i.

With respect to the semantics of SA P systems, the rules of such P systems are applied in a non-deterministic maximally parallel manner.

Let Π be a P system with symport/antiport rules. We say that configuration $\mathcal{C}_t$ yields configuration $\mathcal{C}_{t+1}$ in one *transition step*, denoted by $\mathcal{C}_t \Rightarrow_\Pi \mathcal{C}_{t+1}$, if we can pass from $\mathcal{C}_t$ to $\mathcal{C}_{t+1}$ by applying the rules from the system following the

semantics described above. A *computation* of Π is a (finite or infinite) sequence of configurations such that: (a) the first term is the initial configuration of the system; (b) for each $n \geq 2$, the n-th configuration of the sequence is obtained from the previous configuration in one transition step; and (c) if the sequence is finite (called *halting computation*) then the last term is a *halting configuration* (a configuration where no rule of the system is applicable to it). All the computations start from an initial configuration and proceed as stated above; only a halting computation gives a result, which is encoded by the objects present in the output region i_{out} associated with the halting configuration. If $\mathcal{C} = \{\mathcal{C}_t\}_{t<r+1}$ of Π is a halting computation, then the *length* of $\mathcal{C}$, denoted by $|\mathcal{C}|$, is r. For each i $(1 \leq i \leq q)$, we denote by $\mathcal{C}_t(i)$ the finite multiset of objects over Γ contained in all membranes labelled by i (by applying division rules different membranes with the same label can be created) at configuration $\mathcal{C}_t$.

Definition 2. *A P system with symport/antiport rules and membrane division (SAD P system, for short) of degree $q \geq 1$ is a tuple*

$$\Pi = (\Gamma, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \ldots, \mathcal{R}_q, i_{in}, i_{out}),$$

where:

1. $\Pi = (\Gamma, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \ldots, \mathcal{R}_q, i_{in}, i_{out})$ *is a P system with symport/antiport rules of degree q;*
2. $\mathcal{R}_i$, $1 \leq i \leq q$, *are finite sets of rules over Γ of the following types:*
   (*a*) Symport/antiport rules.
   (*b*) Division rules*:* $[\,a\,]_i \rightarrow [\,b\,]_i\,[\,c\,]_i$, *where $a, b, c \in \Gamma$, $i \in \{2, \ldots, q\}$, $i \neq i_{out}$, and i is the label of a leaf of the tree μ;*
3. $i_{in} \in \{1, \ldots, q\}$ *and* $i_{out} \in \{0, 1, \ldots, q\}$.

A SAD P system of degree q is a P system with symport/antiport rules of degree q where membrane division rules (for only elementary membranes) are allowed.

A division rule $[a]_i \rightarrow [b]_i[c]_i \in \mathcal{R}_i$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant t if the following holds: (a) membrane i is in $\mathcal{C}_t$; (b) object a is contained in that membrane; and (c) membrane i is elementary, and it is neither the skin membrane nor the output membrane (if $i_{out} \in \{1, \ldots, q\}$). When applying a division rule $[a]_i \rightarrow [b]_i[c]_i$, under the influence of object a, the membrane with label i is divided into two membranes with the same label; in the first copy, object a is replaced by object b, and in the second one, object a is replaced by object c; all the other objects residing in the membrane are replicated, and a copy of each one of them is placed in each of the two new membranes.

With respect to the semantics of SAD P systems, the rules of such P systems are applied in a non-deterministic maximally parallel manner with the following important remark: when a membrane i is divided by a division rule at a computation step, this is the only one from $\mathcal{R}_i$ which can be applied to that membrane at that step. The new membranes resulting from division could participate in the interaction with other membranes or the environment by means of communication rules at the next step – providing that they are not divided once again.

Definition 3. *A P system with symport/antiport rules and membrane separation (SAS P system, for short) of degree $q \geq 1$ is a tuple*

$$\Pi = (\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \ldots, \mathcal{R}_q, i_{in}, i_{out}),$$

where

1. $\Pi = (\Gamma, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \ldots, \mathcal{R}_q, i_{in}, i_{out})$ *is a P system with symport/antiport rules of degree q;*
2. $\{\Gamma_0, \Gamma_1\}$ *is a partition of* Γ*, that is,* $\Gamma = \Gamma_0 \cup \Gamma_1$*,* $\Gamma_0, \Gamma_1 \neq \emptyset$*,* $\Gamma_0 \cap \Gamma_1 = \emptyset$*;*
3. $\mathcal{R}_i$*,* $1 \leq i \leq q$*, are finite sets of rules over* Γ *of the following types:*
   (*a*) Symport/antiport rules.
   (*b*) Separation rules: $[\,a\,]_i \rightarrow [\,\Gamma_0\,]_i\,[\,\Gamma_1\,]_i$*, where* $a \in \Gamma$*,* $i \in \{2, \ldots, q\}$*,* $i \neq i_{out}$*, and i is the label of a leaf of the tree;*
4. $i_{in} \in \{1, \ldots, q\}$ *and* $i_{out} \in \{0, 1, \ldots, q\}$.

A SAS P system of degree q is a P system with symport/antiport rules of degree q where membrane separation rules (for only elementary membranes) are allowed.

A separation rule $[a]_i \rightarrow [\Gamma_0]_i[\Gamma_1]_i \in \mathcal{R}_i$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant t, if there exists an elementary membrane labelled by i in $\mathcal{C}_t$, different from the skin membrane and from the output membrane, such that it contains an object a. When applying a separation rule $[a]_i \rightarrow [\Gamma_0]_i\,[\Gamma_1]_i \in \mathcal{R}_i$ to a membrane labelled by i in a configuration $\mathcal{C}_t$, that membrane is separated into two membranes with the same label; at the same time, object a is consumed, and the rest of the contents of the membrane are distributed as follows: the objects from Γ_0 are placed in the first membrane, while those from Γ_1 are placed in the second membrane. In this way, several membranes with the same label $i \neq 1$ can be present in the new membrane structure μ' of the system: a new node i and a new arc $(p(i), i)$ are added to μ' each time a membrane separation rule $[a]_i \rightarrow [\Gamma_0]_i[\Gamma_1]_i$ is applied.

With respect to the semantics of these variants, the rules of such P systems are applied in a non-deterministic maximally parallel manner with the following important remark: when a membrane i is separated, the membrane separation rule is the only one from $\mathcal{R}_i$ which is applied for that membrane at that step. The new membranes resulting from separation could participate in the interaction with other membranes or the environment by means of communication rules at the next step – providing that they are not separated once again.

3.1 Recognizer P systems with symport/antiport rules

Recognizer P systems were introduced in [12], and they provide a natural framework to solve decision problems by means of computational devices in membrane computing (i.e., P systems).

Definition 4. *A recognizer P system with symport/antiport rules (and membrane division or membrane separation) of degree $q \geq 1$ is a P system with symport/antiport rules (and membrane division or membrane separation) such that:*

1. *Alphabet Γ has two distinguished symbols* `yes` *and* `no`*;*
2. *initial multisets are finite multisets over $\Gamma \setminus \Sigma$ such that at least one copy of* `yes` *or* `no` *is present in some of them;*
3. *the output region is the environment ($i_{out} = 0$);*
4. *all computations halt;*
5. *if $\mathcal{C}$ is a computation of the system, then either symbol* `yes` *or symbol* `no` *(but not both) must have been released to the environment, and only at the last step of the computation.*

Let us notice that, if a recognizer P system has a symport rule of the type $(u, in) \in \mathcal{R}_1$, then the multiset u must contain some object from $\Gamma \setminus \mathcal{E}$; otherwise there might exist non-halting computations of Π.

We say that a computation $\mathcal{C}$ of a recognizer P system is an *accepting computation* (respectively, *rejecting computation*) if object `yes` (respectively, object `no`) appears in the environment associated with the corresponding halting configuration of $\mathcal{C}$, and neither object `yes` nor `no` appears in the environment associated with any non–halting configuration of $\mathcal{C}$.

We denote by $\mathbf{CDC}(k)$ (respectively, $\mathbf{CSC}(k)$) the class of all recognizer P systems with symport/antiport rules and membrane division (respectively, membrane separation) for elementary membranes such that the length of the communication rules of the system is at most k.

3.2 Polynomial complexity classes of recognizer P systems with symport/antiport rules

Next, according to [11], we define what solving a decision problem by a family of recognizer P systems with symport/antiport rules and membrane division or membrane separation means.

Definition 5. *A decision problem $X = (I_X, \theta_X)$ is solvable in polynomial time by a family $\mathbf{\Pi} = \{\Pi(n) \mid n \in \mathbb{N}\}$ of recognizer P systems with symport/antiport rules and membrane division or membrane separation, if the following holds:*

- *the family $\mathbf{\Pi}$ is polynomially uniform by Turing machines; that is, there exists a deterministic Turing machine working in polynomial time which constructs the system $\Pi(n)$ from $n \in \mathbb{N}$;*
- *there exists a pair (cod, s) of polynomial-time computable functions over I_X such that:*
  - *for each instance $u \in I_X$, $s(u)$ is a natural number and $cod(u)$ is an input multiset of the system $\Pi(s(u))$;*
  - *for each $n \in \mathbb{N}$, $s^{-1}(n)$ is a finite set;*
  - *the family $\mathbf{\Pi}$ is polynomially bounded with regard to (X, cod, s); that is, there exists a polynomial function p, such that for each $u \in I_X$ every computation of $\Pi(s(u)) + cod(u)$ is halting and it performs at most $p(|u|)$ steps;*
  - *the family $\mathbf{\Pi}$ is sound with regard to (X, cod, s); that is, for each $u \in I_X$, if there exists an accepting computation of $\Pi(s(u)) + cod(u)$, then $\theta_X(u) = 1$;*

– *the family* $\mathbf{\Pi}$ *is complete with regard to* (X, cod, s)*; that is, for each* $u \in I_X$*, if* $\theta_X(u) = 1$*, then every computation of* $\Pi(s(u)) + cod(u)$ *is an accepting one.*

According to Definition 5, we say that the family $\mathbf{\Pi}$ provides a *uniform solution* to the decision problem X. We also say that ordered pair (cod, s) is a polynomial encoding from X in $\mathbf{\Pi}$ and s is the size mapping associated with that solution. It is worth pointing out that, for each instance $u \in I_X$, the P system $\Pi(s(u)) + cod(u)$ is *confluent*, in the sense that all possible computations of the system must give the same answer.

If $\mathbf{R}$ is a class of recognizer P systems, then we denote by $\mathbf{PMC_R}$ the set of all decision problems which can be solved in polynomial time (and in a uniform way) by means of recognizer P systems from $\mathbf{R}$. The class $\mathbf{PMC_R}$ is closed under complement and polynomial–time reductions (see [11] for details). Besides, we have $\mathbf{P} \subseteq \mathbf{PMC_R}$. Indeed, if $X \in \mathbf{P}$, then we consider the family $\mathbf{\Pi} = \{\Pi(n) \mid n \in \mathbb{N}\}$ where $\Pi(n) = \Pi(0)$, for each $n \in \mathbb{N}$, and $\Pi(0)$ is a P system from $\mathbf{R}$ of degree 1 containing only two rules $(\texttt{yes}, out)$ and $(\texttt{no}, out)$. Let us consider the polynomial encoding from X in $\mathbf{\Pi}$ defined as follows: (a) $s(u) = 0$, for each $u \in I_X$; and (b) $cod(u) = \texttt{yes}$ if $\theta_X(u) = 1$ and $cod(u) = \texttt{no}$ if $\theta_X(u) = 0$. Then, the family $\mathbf{\Pi}$ solves X according to Definition 5.

4 Computational efficiency of systems in CSC(2)

In this section, we study the limitations on the computational efficiency (ability to solve hard problems in polynomial time) of systems from $\mathbf{CSC}(2)$. Specifically, we show that only problems in class $\mathbf{P}$ can be efficiently solved in polynomial time by means of families of recognizer P systems with membrane separation that use symport/antiport rules involving at most two objects (i.e., with *minimal cooperation*). Hence, we prove that $\mathbf{P} = \mathbf{PMC_{CSC(2)}}$.

Let us first introduce a new representation for the membrane structure of recognizer P systems with membrane separation. Let $\Pi = (\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \ldots, \mathcal{R}_q, i_{in}, i_{out})$ be a recognizer P system of degree $q \geq 1$ from $\mathbf{CSC}(2)$. In order to identify the membranes created by the application of a separation rule, we modify the labels of the new membranes in the following recursive manner:

- The label of a membrane will be a pair (i, σ), where $1 \leq i \leq q$ and σ is a string over $\{0, 1\}$. At the initial configuration, the labels of the membranes are $(1, \lambda), \ldots, (q, \lambda)$.
- If a separation rule from $\mathcal{R}_i$ is applied to a membrane labelled by (i, σ), then the new created membranes will be labelled by $(i, \sigma 0)$ and $(i, \sigma 1)$, respectively. Membrane $(i, \sigma 0)$ will only contain the objects of membrane (i, σ) which belong to Γ_0, and membrane $(i, \sigma 1)$ will only contain the objects of membrane (i, σ) which belong to Γ_1. The skin membrane cannot be separated, so the label of

the skin membrane, $(1, \lambda)$, is not changed along any computation. Note that we can consider a lexicographical order over the set of labels of cells in the system along any computation.

If a membrane labelled by (i, σ) is engaged by a communication rule, then, after the application of the rule, the membrane keeps its label.

A configuration at an instant t of a P system from $\mathbf{CSC}(2)$ is described by the current membrane structure, the multisets of objects over Γ contained in each membrane, and the multiset of objects over $\Gamma \setminus \mathcal{E}$ currently in the environment. Hence, a configuration of Π can be described by a multiset of labelled objects

$$\{(a, i, \sigma) \mid a \in \Gamma \cup \{\lambda\}, 1 \leq i \leq q, \sigma \in \{0,1\}^*\} \cup \{(a, 0) \mid a \in \Gamma \setminus \mathcal{E}\}.$$

Let us notice that the number of labels we need to identify all membranes appearing along any computation of a P system from $\mathbf{CSC}(2)$ is quadratic in the size of the initial configuration of the system and the length of the computation.

Let $r = (ab, out) \in \mathcal{R}_i$, $2 \leq i \leq q$, be a symport rule of Π and $n \in \mathbb{N}$. We denote by $n \cdot LHS(r, (i, \sigma), (p(i), \tau))$ the multiset of objects $(a, i, \sigma)^n (b, i, \sigma)^n$, and we denote by $n \cdot RHS(r, (i, \sigma), (p(i), \tau))$ the multiset $(a, p(i), \tau)^n (b, p(i), \tau)^n$. In a similar way, $n \cdot LHS(r, (i, \sigma), (p(i), \tau))$ and $n \cdot RHS(r, (i, \sigma), (p(i), \tau))$ are defined when r is of the form $(a, out) \in \mathcal{R}_i$. Note that, at a given instant of the computation, for each membrane (i, σ) there is a unique parent membrane $(p(i), \tau)$, according to the current membrane structure.

Let $r = (ab, out) \in \mathcal{R}_1$ be a symport rule of Π and $n \in \mathbb{N}$. We denote by $n \cdot LHS(r, (1, \lambda), 0)$ the multiset of objects $(a, 1, \lambda)^n (b, 1, \lambda)^n$. We denote by $n \cdot RHS(r, (1, \lambda), 0)$ the following multiset of objects:

$$\begin{cases} (a,0)^n (b,0)^n, & \text{if } a, b \in \Gamma \setminus \mathcal{E}; \\ (a,0)^n, & \text{if } a \in \Gamma \setminus \mathcal{E} \text{ and } b \in \mathcal{E}; \\ (b,0)^n, & \text{if } b \in \Gamma \setminus \mathcal{E} \text{ and } a \in \mathcal{E}; \\ \emptyset, & \text{if } a, b \in \mathcal{E}. \end{cases}$$

In a similar way, $n \cdot LHS(r, (1, \lambda), 0)$ and $n \cdot RHS(r, (1, \lambda), 0)$ are defined when r is of the form $(a, out) \in \mathcal{R}_1$.

Let $r = (ab, in) \in \mathcal{R}_i$, $2 \leq i \leq q$, be a symport rule of Π and $n \in \mathbb{N}$. We denote by $n \cdot LHS(r, (i, \sigma), (p(i), \tau))$ the multiset of objects $(a, p(i), \tau)^n (b, p(i), \tau)^n$. We denote by $n \cdot RHS(r, (i, \sigma), (p(i), \tau))$ the multiset of objects $(a, i, \sigma)^n (b, i, \sigma)^n$. In a similar way, $n \cdot LHS(r, (i, \sigma), (p(i), \tau))$ and $n \cdot RHS(r, (i, \sigma), (p(i), \tau))$ are defined when r is of the form $(a, in) \in \mathcal{R}_i$.

Let $r = (ab, in) \in \mathcal{R}_1$ be a symport rule of Π and $n \in \mathbb{N}$. We denote by $n \cdot LHS(r, (1, \lambda), 0)$ the following multiset of objects:

$$\begin{cases} (a,0)^n (b,0)^n, & \text{if } a, b \in \Gamma \setminus \mathcal{E}; \\ (a,0)^n, & \text{if } a \in \Gamma \setminus \mathcal{E} \text{ and } b \in \mathcal{E}; \\ (b,0)^n, & \text{if } b \in \Gamma \setminus \mathcal{E} \text{ and } a \in \mathcal{E}; \\ \emptyset, & \text{if } a, b \in \mathcal{E}. \end{cases}$$

We denote by $n \cdot RHS(r,(1,\lambda),0)$ the multiset of objects $(a,1,\lambda)^n(b,1,\lambda)^n$. In a similar way, $n \cdot LHS(r,(1,\lambda),0)$ and $n \cdot RHS(r,(1,\lambda),0)$ are defined when r is of the form $(a,in) \in \mathcal{R}_1$.

Let $r = (a,out;b,in) \in \mathcal{R}_i$, $2 \leq i \leq q$, be an antiport rule of Π and $n \in \mathbb{N}$. We denote by $n \cdot LHS(r,(i,\sigma),(p(i),\tau))$ the multiset of objects $(a,i,\sigma)^n(b,p(i),\tau)^n$. Similarly, we denote by $n \cdot RHS(r,(i,\sigma),(p(i),\tau))$ the multiset of objects $(a,p(i),\tau)^n(b,i,\sigma)^n$.

Let $r = (a,out;b,in) \in \mathcal{R}_1$ be an antiport rule of Π. We denote by $n \cdot LHS(r,(1,\lambda),0)$ the following multiset of objects:

$$\begin{cases} (a,1,\lambda)^n(b,0)^n, & \text{if } b \in \Gamma \setminus \mathcal{E}; \\ (a,1,\lambda)^n, & \text{if } b \in \mathcal{E}. \end{cases}$$

Similarly, we denote by $n \cdot RHS(r,(1,\lambda),0)$ the following multiset of objects:

$$\begin{cases} (a,0)^n(b,1,\lambda)^n, & \text{if } a \in \Gamma \setminus \mathcal{E}; \\ (b,1,\lambda)^n, & \text{if } a \in \mathcal{E}. \end{cases}$$

If $\mathcal{C}_t$ is a configuration of Π, then we denote by $\mathcal{C}_t + \{(x,i,\sigma)/\sigma'\}$ the multiset obtained by replacing in $\mathcal{C}_t$ every occurrence of (x,i,σ) by (x,i,σ'). Besides, $\mathcal{C}_t + m$ (resp., $\mathcal{C}_t \setminus m$) is used to denote that a multiset m of labelled objects is added (resp., removed) to the configuration.

4.1 Characterizing class P by means of systems from CSC(2)

In order to show that only tractable problems can be solved efficiently by using families of P systems from **CSC**(2), we first state a technical result concerning recognizer P systems from **CSC**(2) (see [7] for more details).

Lemma 4.1 *Let $\Pi = (\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \ldots, \mathcal{R}_q, i_{in}, i_{out})$ be a recognizer P system of degree $q \geq 1$ from* **CSC**(2). *Let $M = |\mathcal{M}_1 + \ldots + \mathcal{M}_q|$ and let $\mathcal{C} = \{\mathcal{C}_0, \ldots, \mathcal{C}_r\}$ be a computation of Π. Then, we have*

(1) $|\mathcal{C}_0^*| = M$, and for each t, $0 \leq t < r$, $\mathcal{C}_{t+1}^* \cap (\Gamma \setminus \mathcal{E}) \subseteq \mathcal{C}_t^* \cap (\Gamma \setminus \mathcal{E})$;
(2) for each t, $0 \leq t \leq r$, $\mathcal{C}_t^* \cap (\Gamma \setminus \mathcal{E}) \subseteq (\mathcal{M}_1 + \ldots + \mathcal{M}_q) \cap (\Gamma \setminus \mathcal{E})$, and $|\mathcal{C}_t^* \cap (\Gamma \setminus \mathcal{E})| \leq M$;
(3) for each t, $0 \leq t < r$, $|\mathcal{C}_{t+1}^*| \leq |\mathcal{C}_t^*| + M$;
(4) for each t, $0 \leq t \leq r$, $|\mathcal{C}_t^*| \leq M \cdot (1+t)$;
(5) the number of membranes created along computation $\mathcal{C}$ by the application of separation rules is bounded by $2M \cdot (1+r)$.

Next, we present a deterministic algorithm $\mathcal{A}$ working in polynomial time that receives as an input a P system Π from **CSC**(2) and an input multiset m of Π, in such manner that algorithm $\mathcal{A}$ reproduces the behaviour of a computation of $\Pi + m$. In particular, if Π is confluent, then algorithm $\mathcal{A}$ will provide the same answer of the system Π.

The pseudocode of the algorithm $\mathcal{A}$ is described as follows:

Input: A P system Π from **CSC**(2) and an input multiset m
  Initialization phase: $\mathcal{C}_0$ is the initial configuration of $\Pi + m$
  $t \leftarrow 0$
  while $\mathcal{C}_t$ is a non halting configuration **do**
    Selection phase: Input $\mathcal{C}_t$, Output $(\mathcal{C}'_t, A)$
    Execution phase: Input $(\mathcal{C}'_t, A)$, Output $\mathcal{C}_{t+1}$
    $t \leftarrow t + 1$
  end while
Output: *Yes* if object yes appears in the environment associated with the halting configuration $\mathcal{C}_t$, *No* otherwise

The algorithm $\mathcal{A}$ receives a recognizer P system

$$\Pi = (\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \ldots, \mathcal{R}_q, i_{in}, i_{out})$$

from **CSC**(2) and an input multiset m. Let $M = |\mathcal{M}_1 + \ldots + \mathcal{M}_q|$, $p \in \mathbb{N}$ be a natural number such that any computation of $\Pi + m$ performs, at most, p transition steps. Hence, from Lemma 4.1, we know that the number of membranes in the system along any computation is bounded by $2M(1 + p) + q$.

A transition step of a recognizer P system $\Pi + m$ is performed by the selection and the execution phases. Specifically, the selection phase receives as an input a configuration $\mathcal{C}_t$ of $\Pi + m$ at an instant t. The output of this phase is a pair $(\mathcal{C}'_t, A)$, where A encodes a multiset of rules selected to be applied to $\mathcal{C}_t$, and $\mathcal{C}'_t$ is the configuration obtained from $\mathcal{C}_t$ once the labelled objects corresponding to the left-hand side of the rules from A have been consumed. The execution phase receives as an input the pair $(\mathcal{C}'_t, A)$, and the output of this phase is the next configuration $\mathcal{C}_{t+1}$ of $\mathcal{C}_t$. More precisely, configuration $\mathcal{C}_{t+1}$ is obtained from $\mathcal{C}'_t$ by adding the labelled objects produced by the application of rules from A; that is, the labelled objects corresponding to the right-hand side of the rules from A.

Selection phase.

Input: A configuration $\mathcal{C}_t$ of $\Pi + m$ at instant t
  $\mathcal{C}'_t \leftarrow \mathcal{C}_t$; $A \leftarrow \emptyset$; $B \leftarrow \emptyset$
  for $r = (u, out; v, in) \in \mathcal{R}_i, 2 \leq i \leq q$ according to the order chosen **do**
    for each membrane (i, σ) of $\mathcal{C}'_t$ according to the lexicographical order **do**
      $n_r \leftarrow$ maximum number of times that r is applicable to (i, σ)
      if $n_r > 0$ **then**
        $\mathcal{C}'_t \leftarrow \mathcal{C}'_t \setminus n_r \cdot LHS(r, (i, \sigma), (p(i), \tau))$
        $A \leftarrow A \cup \{(r, n_r, (i, \sigma), (p(i), \tau))\}$
        $B \leftarrow B \cup \{(i, \sigma), (p(i), \tau)\}$
      end if
    end for
  end for

for $r = (u, out; v, in) \in \mathcal{R}_1$ `according to the order chosen` **do**
  $n_r \leftarrow$ `maximum number of times that` r `is applicable to` $(1, \lambda)$
  if $n_r > 0$ **then**
    $\mathcal{C}'_t \leftarrow \mathcal{C}'_t \setminus n_r \cdot LHS(r, (1, \lambda), 0)$
    $A \leftarrow A \cup \{(r, n_r, (1, \lambda), 0)\}$
  end if
end for
for $r = [\, a \,]_i \rightarrow [\, \Gamma_0 \,]_i [\, \Gamma_1 \,]_i \in \mathcal{R}_i$ $(i \neq 1)$ `according to the order chosen` **do**
  for each $(a, i, \sigma) \in \mathcal{C}'_t$ `according to the lexicographical order, and such that` $(i, \sigma) \notin B$ **do**
    $\mathcal{C}'_t \leftarrow \mathcal{C}'_t \setminus \{(a, i, \sigma)\}$
    $A \leftarrow A \cup \{(r, 1, (i, \sigma))\}$
    $B \leftarrow B \cup \{(i, \sigma)\}$
  end for
end for

This algorithm is deterministic and works in polynomial time. Indeed, the running time of the previous algorithm is polynomial in the size of Π because: the number of cycles of the first main loop **for** is of order $O(|\mathcal{R}| \cdot M \cdot p \cdot q)$; the number of cycles of the second main loop **for** is of order $O(|\mathcal{R}|)$; and the number of cycles of the third main loop **for** is of order $O(|\mathcal{R}| \cdot M \cdot p \cdot q \cdot |\Gamma|)$.

Execution phase.

Input: `The output` $(\mathcal{C}'_t, A)$ `of the selection phase`
  for each $(r, n_r, (i, \sigma), (p(i), \tau)) \in A$ **do**
    $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + n_r \cdot RHS(r, (i, \sigma), (p(i), \tau))$
  end for
  for each $(r, n_r, (1, \lambda), 0) \in A$ **do**
    $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + n_r \cdot RHS(r, (1, \lambda), 0)$
  end for
  for each $(r, 1, (i, \sigma)) \in A$ **do**
    $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + \{(\lambda, i, \sigma)/\sigma 0\}$
    $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + \{(\lambda, i, \sigma 1)\}$
    for each $(x, i, \sigma) \in C'_t$ `according to the lexicographical order` **do**
      if $x \in \Gamma_0$ **then**
        $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + \{(x, i, \sigma)/\sigma 0\}$
      else
        $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + \{(x, i, \sigma)/\sigma 1\}$
      end if
    end for

end for
$\mathcal{C}_{t+1} \leftarrow \mathcal{C}'_t$

This algorithm is deterministic and works in polynomial time. Indeed, the running time of the previous algorithm is polynomial in the size of Π because: the number of cycles of the first main loop **for** is of order $O(|\mathcal{R}| \cdot M \cdot p \cdot q)$; the number of cycles of the second main loop **for** is of order $O(|\mathcal{R}|)$; and the number of cycles of the third main loop **for** is of order $O(|\mathcal{R}| \cdot M \cdot p \cdot q \cdot |\Gamma|)$.

Theorem 4.2 $\mathbf{P} = \mathbf{PMC}_{\mathbf{CSC}(2)}$.

Proof. It suffices to show that $\mathbf{PMC}_{\mathbf{CSC}(2)} \subseteq \mathbf{P}$. Let $X \in \mathbf{PMC}_{\mathbf{CSC}(2)}$ and let $\mathbf{\Pi} = \{\Pi(n) \mid n \in \mathbb{N}\}$ be a family of recognizer P systems from $\mathbf{CSC}(2)$ solving X, according to Definition 5. Let (cod, s) be a polynomial encoding associated with that solution. If $u \in I_X$ is an instance of the problem X, then u will be processed by the system $\Pi(s(u)) + cod(u)$.
Let us consider the following deterministic algorithm $\mathcal{A}'$:

Input: `an instance` u `of the problem` X
`Construct the system` $\Pi(s(u)) + cod(u)$.
`Run algorithm` $\mathcal{A}$ `with input` $\Pi(s(u)) + cod(u)$.
Output: *Yes* `if algorithm` $\mathcal{A}$ `returns` *Yes*,
No `otherwise.`

The algorithm $\mathcal{A}'$ receives as an input an instance u of the decision problem $X = (I_X, \theta_X)$ and works in polynomial time with respect to the size of the input. The following assertions are equivalent:

- $\theta_X(u) = 1$; that is, the answer of problem X to instance u is affirmative.
- Every computation of $\Pi(s(u)) + cod(u)$ is an accepting computation.
- The output of algorithm $\mathcal{A}'$ with input u is *Yes*.

Hence, $X \in \mathbf{P}$.

□

5 Computational efficiency of systems in CDC(2)

In this section we study the ability to solve **NP**–complete problems of families of recognizer P systems with membrane division whose communication rules (of type symport/antiport) use a minimal cooperation (i.e., communication rules involving at most two objects). Specifically, we give a polynomial time solution to `HAM-CYCLE` problem, a well known **NP**-complete problem [2], by means of a family of such kind of recognizer P systems, according to Definition 5 (see [15] for more details).

Let us recall that `HAM-CYCLE` problem is the following: *Given a directed graph, determine whether or not there exists a Hamiltonian cycle in the graph.*

5.1 A polynomial time solution of HAM-CYCLE problem in CDC(2)

For each $n, m \in \mathbb{N}$, we consider the recognizer P system with symport/antiport rules and membrane division of degree $11 + 2n + n^3$

$$\begin{aligned}\Pi(\langle n,m\rangle) = (\Gamma, \mathcal{E}, \Sigma, \mu, \mathcal{M}_r\,(1 \le r \le 11)\,, \mathcal{M}_{a_{1,j}}\,(1 \le j \le n),\ \mathcal{M}_{a_{2,j}}\,(1 \le j \le n)\,,\\ \mathcal{M}_{e_{i,j,k}}(1 \le i,j,k \le n)\,, \mathcal{R}_r\,(1 \le r \le 11)\,, \mathcal{R}_{a_{1,j}}\,(1 \le j \le n)\,,\\ \mathcal{R}_{a_{2,j}}\,(1 \le j \le n)\ \mathcal{R}_{e_{i,j,k}}(1 \le i,j,k \le n))\end{aligned}$$

defined as follows:

(1) Working alphabet:
$\Gamma = \Sigma \cup \mathcal{E} \cup \{\beta_r \mid 0 \le r \le n^3+7\} \cup \{b'_r, b''_r, b'''_r, c'_r, c''_r, c'''_r, c''''_r \mid 1 \le r \le n^3\} \cup$
$\{(i,j)'_k\,,\ (i,j)''_k \mid 1 \le i,j,k \le n\} \cup \{(i,j)''_{k,r} \mid 1 \le i,j,k \le n \wedge 1 \le r \le n^3\} \cup$
$\{\alpha_0, a, a', a'', b, b', b'', b''', c, c', c'', c''', c'''', \mathtt{yes}, \mathtt{no}\}$,
where the input alphabet is $\Sigma = \{(i,j)_k \mid 1 \le i,j,k \le n\}$, and the alphabet of the environment is $\mathcal{E} = \{\alpha_r \mid 1 \le r \le n^3+6\}$

(2) Membrane structure μ: the root is labelled by 1, and the remaining nodes are children of the root, being labelled by

$$2, 3, \ldots, 11, a_{1,j}\,(1 \le j \le n), a_{2,j}\,(1 \le j \le n), e_{i,j,k}\,(1 \le i,j,k \le n),$$

respectively.

(3) Initial multisets:
$\mathcal{M}_1 = \{\alpha_0\} \cup \{\beta_r \mid 1 \le r \le n^3+7\} \cup \{b'_r, b''_r, b'''_r, c'_r, c''_r, c'''_r, c''''_r \mid 1 \le r \le n^3-1\}$;
$\mathcal{M}_2 = \{a^n, b, c\}$;
$\mathcal{M}_3 = \{b'_{n^3}\}$; $\mathcal{M}_4 = \{b''_{n^3}\}$; $\mathcal{M}_5 = \{b'''_{n^3}\}$;
$\mathcal{M}_6 = \{c'_{n^3}\}$; $\mathcal{M}_7 = \{c''_{n^3}\}$; $\mathcal{M}_8 = \{c'''_{n^3}\}$; $\mathcal{M}_9 = \{c''''_{n^3}\}$;
$\mathcal{M}_{10} = \{\mathtt{yes}\}$; $\mathcal{M}_{11} = \{\mathtt{no}, \beta_0\}$;
$\mathcal{M}_{a_{1,j}} = \{a'_{n^3}\}$, $\mathcal{M}_{a_{2,j}} = \{a''_{n^3}\}, 1 \le j \le n$;
$\mathcal{M}_{e_{i,j,k}} = \{(i,j)''_{k,n^3}\}, 1 \le i,j,k \le n$.

(4) Rules of the system:

• Rules in $\mathcal{R}_1$:

1.1 Rules to control the output of the computations by counters of type α_r.

$$(\alpha_r\,,\, out\,;\, \alpha_{r+1}\,,\, in)\ ,\ 0 \le r \le n^3+5.$$

Rules 1.2 and 1.3 produce the output of the computations:

1.2 $(\mathtt{yes}\,,\, out)$

1.3 $(\mathtt{no}\,\alpha_{n^3+6}\,,\, out)$

• Rules in $\mathcal{R}_2$:

2.1 Rules to produce all possible subsets of A'_G in membranes labelled by 2 at configuration $\mathcal{C}_{n^3+1}$:

$$[\,(i,j)_k\,]_2 \to [\,(i,j)'_k\,]_2\ [\,\#\,]_2,\ 1 \le i,j,k \le n.$$

Rules 2.2, 2.3, 2.4 and 2.5 allow to introduce objects a', a'', b', b'', c''', c', c'', c''' and c'''' in membranes labelled by 2 at configurations $\mathcal{C}_{n^3+2}$, $\mathcal{C}_{n^3+3}$, $\mathcal{C}_{n^3+4}$ and $\mathcal{C}_{n^3+5}$, respectively:

2.2 $(a\,,\,out\,;\,a'\,,\,in)$; $(a'\,,\,out\,;\,a''\,,\,in)$;
2.3 $(b\,,\,out\,;\,b'\,,\,in)$; $(b'\,,\,out\,;\,b''\,,\,in)$; $(b''\,,\,out\,;\,b'''\,,\,in)$;
2.4 $(c\,,\,out\,;\,c'\,,\,in)$; $(c'\,,\,out\,;\,c''\,,\,in)$; $(c''\,,\,out\,;\,c'''\,,\,in)$; $(c'''\,,\,out\,;\,c''''\,,\,in)$;
2.5 $(a''\,b'''\,,\,out)$; $(b'''\,c''''\,,\,out)$.
2.6 Rules to produce in each membrane labelled by 2 at configuration $\mathcal{C}_{n^3+2}$ a subset of A''_G from a subset of A'_G at configuration $\mathcal{C}_{n^3+1}$:

$$((i,j)'_k\,,\,out\,;\,(i,j)''_k\,,\,in)\ ,\ 1 \leq i,j,k \leq n.$$

2.7 Rules to generate in each membrane labelled by 2 at configuration $\mathcal{C}_{n^3+1}$ a subset of A''_G encoding a possible Hamiltonian cycle.
$((i,j)''_k\ (i,j')''_{k'}\ ,\,out),\ 1 \leq i,i',j,j',k,k' \leq n;$
$((i,j)''_k\ (i',j)''_{k'}\ ,\,out),\ 1 \leq i,i',j,j',k,k' \leq n;$
$((i,j)''_k\ (i',j')''_{k+1}\ ,\,out),\ 1 \leq i,i',j,j',k,k' \leq n,\ j \neq i';$
$((i,j)''_k\ (i',j')''_k\ ,\,out),\ 1 \leq i,i',j,j',k,k' \leq n.$
2.8 Rules to check if the subset represented by each membrane with label 2 at configuration $\mathcal{C}_{n^3+3}$ encodes a Hamiltonian cycle of the input graph:

$$(a''\,(i,j)''_k\,,\,out),\ 1 \leq i,j,k \leq n.$$

• Rules in $\mathcal{R}_3$:
Rules to produce $2^{n \cdot p}$ copies of objects b' in the skin membrane of configuration $\mathcal{C}_{n^3+1}$:
3.1 $(b'_r\,,\,out\,;\,b'_{r-1}\,,\,in),\ n \cdot m + 1 \leq r \leq n^3;$
3.2 $[\,b'_r\,]_3 \rightarrow [\,b'_{r-1}\,]_3\ [\,b'_{r-1}\,]_3,\ 2 \leq r \leq n \cdot m;$
3.3 $[\,b'_1\,]_3 \rightarrow [\,b'\,]_3\ [\,b'\,]_3;$
3.4 $(b'\ ,\,out)$.
• Rules in $\mathcal{R}_4$:
Rules to produce $2^{n \cdot p}$ copies of objects b'' in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:
4.1 $(b''_r\,,\,out\,;\,b''_{r-1}\,,\,in),\ n \cdot m + 1 \leq r \leq n^3;$
4.2 $[\,b''_r\,]_4 \rightarrow [\,b''_{r-1}\,]_4\ \ b''_{r-1}\,]_4,\ 2 \leq r \leq n \cdot m;$
4.3 $[\,b''_1\,]_4 \rightarrow [\,b''\,]_4\ [\,b''\,]_4;$
4.4 $(b''\ ,\,out)$.
• Rules in $\mathcal{R}_5$:
Rules to produce $2^{n \cdot p}$ copies of objects b''' in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:
5.1 $(b'''_r\,,\,out\,;\,b'''_{r-1}\,,\,in),\ n \cdot m + 1 \leq r \leq n^3;$
5.2 $[\,b'''_r\,]_5 \rightarrow [\,b'''_{r-1}\,]_5\ [\,b'''_{r-1}\,]_5,\ 2 \leq r \leq n \cdot m;$
5.3 $[\,b'''_1\,]_5 \rightarrow [\,b'''\,]_5\ [\,b'''\,]_5;$
5.4 $(b'''\ ,\,out)$.
• Rules in $\mathcal{R}_6$:
Rules to produce $2^{n \cdot p}$ copies of objects c' in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:

6.1 $(c'_r\,,\,out\,;\,c'_{r-1}\,,\,in),\ n\cdot m+1\leq r\leq n^3$;
6.2 $[\,c'_r\,]_6\rightarrow[\,c'_{r-1}\,]_6\,[\,c'_{r-1}\,]_6,\ 2\leq r\leq n\cdot m$;
6.3 $[\,c'_1\,]_6\rightarrow[\,c'\,]_6\,[\,c'\,]_6$;
6.4 $(c'\,,\,out)$.

• Rules in $\mathcal{R}_7$:
Rules to produce $2^{n\cdot p}$ copies of objects c'' in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:
7.1 $(c''_r\,,\,out\,;\,c''_{r-1}\,,\,in),\ n\cdot m+1\leq r\leq n^3$;
7.2 $[\,c''_r\,]_7\rightarrow[\,c''_{r-1}\,]_7\,[\,c''_{r-1}\,]_7,\ 2\leq r\leq n\cdot m$;
7.3 $[\,c''_1\,]_7\rightarrow[\,c''\,]_7\,[\,c''\,]_7$;
7.4 $(c''\,,\,out)$.

• Rules in $\mathcal{R}_8$:
Rules to produce $2^{n\cdot p}$ copies of objects c''' in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:
8.1 $(c'''_r\,,\,out\,;\,c'''_{r-1}\,,\,in),\ n\cdot m+1\leq r\leq n^3$;
8.2 $[\,c'''_r\,]_8\rightarrow[\,c'''_{r-1}\,]_8\,[\,c'''_{r-1}\,]_8,\ 2\leq r\leq n\cdot m$;
8.3 $[\,c'''_1\,]_8\rightarrow[\,c'''\,]_8\,[\,c'''\,]_8$;
8.4 $(c'''\,,\,out)$.

• Rules in $\mathcal{R}_9$:
Rules to produce $2^{n\cdot p}$ copies of objects c'''' in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:
9.1 $(c''''_r\,,\,out\,;\,c''''_{r-1}\,,\,in),\ n\cdot m+1\leq r\leq n^3$;
9.2 $[\,c''''_r\,]_9\rightarrow[\,c''''_{r-1}\,]_9\,[\,c''''_{r-1}\,]_9,\ 2\leq r\leq n\cdot m$;
9.3 $[\,c''''_1\,]_9\rightarrow[\,c''''\,]_9\,[\,c''''\,]_9$;
9.4 $(c''''\,,\,out)$.

• Rules in $\mathcal{R}_{10}$:
Rules to produce an affirmative answer:
10.1 $(\alpha_{n^3+6}\,c''''\,,\,in)\ ;\ (c''''\,\texttt{yes}\,,\,out)$

• Rules in $\mathcal{R}_{11}$:
Rules to control the negative answer of the computations by counters β_r:
11.1 $(\beta_r\,out\,;\,\beta_{r+1}\,,\,in),\ 0\leq r\leq n^3+6$;
11.2 $(\beta_{n^3+7}\,\texttt{no}\,,\,out)$.

• Rules in $\mathcal{R}_{a_{1,j}},\ 1\leq j\leq n$:
Rules to produce 2^{n^3} copies of objects a' in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:
$\mathbf{a_{1,j}.1}$ $[\,a'_r\,]_{a_{1,j}}\rightarrow[\,a'_{r-1}\,]_{a_{1,j}}\,[\,a'_{r-1}\,]_{a_{1,j}},\ 2\leq r\leq n^3$;
$\mathbf{a_{1,j}.2}$ $[\,a'_1\,]_{a_{1,j}}\rightarrow[\,a'\,]_{a_{1,j}}\,[\,a'\,]_{a_{1,j}}$;
$\mathbf{a_{1,j}.3}$ $(a'\,,\,out)$.

• Rules in $\mathcal{R}_{a_{2,j}},\ 1\leq j\leq n$:
Rules to produce 2^{n^3} copies of objects a'' in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:
$\mathbf{a_{2,j}.1}$ $[\,a''_r\,]_{a_{2,j}}\rightarrow[\,a''_{r-1}\,]_{a_{2,j}}\,[\,a''_{r-1}\,]_{a_{2,j}},\ 2\leq r\leq n^3$;

$\mathbf{a_{2,j}.2}$ $[\, a_1'' \,]_{a_{2,j}} \rightarrow [\, a'' \,]_{a_{2,j}}\ [\, a'' \,]_{a_{2,j}}$;
$\mathbf{a_{2,j}.3}$ (a'', out).

• Rules in $\mathcal{R}_{e_{i,j,k}}$, $1 \leq i, j, k \leq n$:

Rules to produce 2^{n^3} copies of objects $(i,j)_k''$ in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:

$\mathbf{e_{i,j,k}.1}$ $[\, (i,j)_{k,r}'' \,]_{e_{i,j,k}} \rightarrow [\, (i,j)_{k,r-1}'' \,]_{e_{i,j,k}}\ [\, (i,j)_{k,r-1}'' \,]_{e_{i,j,k}}$, $2 \leq r \leq n^3$;
$\mathbf{e_{i,j,k}.2}$ $[\, (i,j)_{k,1}'' \,]_{e_{i,j,k}} \rightarrow [\, (i,j)_k'' \,]_{e_{i,j,k}}\ [\, (i,j)_k'' \,]_{e_{i,j,k}}$;
$\mathbf{e_{i,j,k}.3}$ $((i,j)_k'', out)$.

(5) The input membrane is the membrane labelled by 2 and the output region is the environment of the system (labelled by 0).

5.2 An overview of the computations

Now we briefly show how each system $\Pi(\langle n, m \rangle)$ works in order to process any directed graph with n nodes and m arcs.

We consider the ensuing polynomial encoding (cod, s) from HAM-CYCLE in $\mathbf{\Pi}$: for each instance $G = (V, E)$ of HAM-CYCLE problem, with $V = \{1, \ldots, n\}$ and $E = \{(i_1, j_1), \ldots, (i_m, j_m)\}$, we define $s(G) = \langle n, m \rangle$ and $cod(G) = \{(i,j)_k \mid (i,j) \in E, 1 \leq k \leq n\}$. The expression $(i,j)_k$ in $cod(G)$ can be interpreted as follows: arc (i,j) is "placed" in "position k" in a potential path. According to this polynomial encoding, graph G will be processed by system $\Pi(s(G))$ with input multiset $cod(G)$. In what follows, we informally describe how system $\Pi(s(G)) + cod(G)$ works. The solution is structured in the following stages:

- *Generation Stage*: All possible combinations of arcs from the input graph, including a code of their position in potential paths, are generated by using cell division in an adequate way.
- *Checking Stage*: It is checked whether or not the different combinations of arcs generated in the previous stage encode Hamiltonian cycles of the input graph.
- *Output Stage*: The system sends the right answer to the environment according to the results obtained in the previous stage.

Generation stage

At this stage, the system generates all the possible subsets of arcs of the graph (in fact, subsets of A'_G) which contain their potential positions in a path according to the notations introduced in Subsection 2.2. In this way, by applying rules of type 2.1 at configuration $\mathcal{C}_{2^{n \cdot m}}$, there will be $2^{n \cdot m}$ membranes labelled by 2 such that each of them encodes a different combination of arcs from the input graph. Simultaneously, by applying rules of types 1, 2 and 3 from $\mathcal{R}_3$, $\mathcal{R}_4$, $\mathcal{R}_5$, $\mathcal{R}_6$, $\mathcal{R}_7$, $\mathcal{R}_8$ and $\mathcal{R}_9$, $2^{n \cdot m}$ copies of objects $b', b'', b''', c', c'', c'''$ and c'''' are produced in membranes labelled by $3, 4, 5, 6, 7, 8, 9$, respectively, and 2^{n^3} copies of objects a', a'' and $(i,j)_k''$ are produced in membranes labelled by $a_{1,j}, a_{2,j}$, and $e_{i,j,k}$, respectively. The generation stage takes n^3 steps.

Checking stage

At this stage, the system checks whether or not there exists a membrane labelled by 2 at configuration $\mathcal{C}_{n^3+5}$ containing a subset of A''_G that encodes a Hamiltonian cycle of G. This is done in 4 steps.

At step n^3+1, the contents of membranes labelled by $3, 4, 5, 6, 7, 8, 9$, $a_{1,j}$ $(1 \leq j \leq n), a_{2,j}$ $(1 \leq j \leq n)$ and $e_{i,j,k}$ $(1 \leq i, j, k \leq n)$ are sent to the skin membrane by applying rules 3.4, 4.4, 5.4, 6.4, 7.4, 8.4, 9.4, $a_{1,j}.2, a_{2,j}.2, e_{i,j,k}.3$. From this moment on, none of these membranes will participate in the evolution of the configurations.

At step n^3+2, objects a, b, c in membrane labelled by 2 at configuration $\mathcal{C}_{n^3+1}$ are replaced by objects a', b', c' from the skin membrane by applying rules 2.2, 2.3, and 2.4. Simultaneously, by applying rules 2.6, each subset of A'_G contained in a membrane labelled by 2 at configuration $\mathcal{C}_{n^3+1}$ produces the "corresponding" subset of A''_G. Besides, $\mathcal{C}_{n^3+2}(10) = \{\texttt{yes}\}$ and $\mathcal{C}_{n^3+2}(11) = \{\beta_{n^3+2}\,, \texttt{no}\}$.

At step n^3+3, by applying rules 2.3 and 2.4, objects a', b', c' in membranes labelled by 2 at configuration $\mathcal{C}_{n^3+2}$ are replaced by objects a'', b'', c'' from the skin membrane. Simultaneously, by applying rules of type 2.7, each subset contained in a membrane labelled by 2 at configuration $\mathcal{C}_{n^3+2}$ is transformed into a subset encoding each possible path in the input graph. This way, according to Proposition 2.1, we have that the input graph (with n nodes and m arcs) has a Hamiltonian cycle if and only if at configuration $\mathcal{C}_{n^3+3}$ there exists some membrane labelled by 2 at configuration $\mathcal{C}_{n^3+3}$ such that the subset of A''_G contained in it has size equal to n. Besides, $\mathcal{C}_{n^3+3}(10) = \{\texttt{yes}\}$ and $\mathcal{C}_{n^3+3}(11) = \{\beta_{n^3+3}\,, \texttt{no}\}$.

At step n^3+4, by applying rules 2.3 and 2.4, objects b'', c'' in membranes labelled by 2 are substituted by objects b''', c''' from the skin membrane. Simultaneously, by applying rules 2.8, each object contained in the subset associated with each membrane labelled by 2 at configuration $\mathcal{C}_{n^3+3}$ is sent to the skin membrane cooperating with an object a''. Therefore, the number of copies of object a'' appearing in a membrane labelled by 2 at configuration $\mathcal{C}_{n^3+4}$ is equal to $n - \gamma$, where γ is the size of the path in the input graph encoded by that membrane. Then, the input graph (with n nodes and m arcs) has a Hamiltonian cycle if and only if there exists a membrane labelled by 2 at configuration $\mathcal{C}_{n^3+4}$ such that it does not contain any object a''.

At step n^3+5, by applying rules of type 2.5, objects a'' and b''' in membrane labelled by 2 at configuration $\mathcal{C}_{n^3+5}$ are sent to the skin membrane. Simultaneously, rule $(c''', out\,; c'''', in)$ produces an object c'''' in each membrane labelled by 2 at configuration $\mathcal{C}_{n^3+5}$.

Output stage

Finally, the output stage takes 4 steps. Only membranes labelled by 2 at configuration $\mathcal{C}_{n^3+5}$ containing some object b''' (i.e., membrane encoding a Hamiltonian cycle) can evolve, and only rule $(c''', out\,; c'''', in) \in \mathcal{R}_2$ is applicable to that membrane. In this case, an object c'''' will appear in each membrane labelled by 2

at that configuration. Besides, if a membrane with label 2 at the mentioned configuration does not encode a Hamiltonian cycle of the input graph, then it contains objects b'', so rule $(a''\, b''' , out) \in \mathcal{R}_2$ will be applied. That is, the input graph has a Hamiltonian cycle if and only if some object c'''' appears in the skin membrane at configuration $\mathcal{C}_{n^3+6}$. Besides, $\mathcal{C}_{n^3+6}(10) = \{\texttt{yes}\}$ and $\mathcal{C}_{n^3+6}(11) = \{\beta_{n^3+6} , \texttt{no}\}$.

If the input graph has a Hamiltonian cycle, then only rules $(\alpha_{n^3+6}\, c'''' , in) \in \mathcal{R}_{10}$ and $(\beta_{n^3+6} , out ; \beta_{n^3+7} , in) \in \mathcal{R}_{11}$ are applicable to configuration $\mathcal{C}_{n^3+6}$. Otherwise, only rule $(\beta_{n^3+6}\, out ; \beta_{n^3+7} , in)$ is applicable to that configuration. Therefore, the answer of the problem is affirmative if and only if $\mathcal{C}_{n^3+7}(10) = \{\alpha_{n^3+6}\, c'''' , \texttt{yes}\}$. Besides, in any case, $\mathcal{C}_{n^3+7}(11) = \{\beta_{n^3+7} , \texttt{no}\}$. Then, if there exists a Hamiltonian path, then rules $(c''''\, \texttt{yes} , out) \in \mathcal{R}_{10}$ and $(\beta_{n^3+7}\, \texttt{no} , out) \in \mathcal{R}_{11}$ are applicable to configuration $\mathcal{C}_{n^3+7}$. Otherwise, only rule $(\beta_{n^3+7}\, \texttt{no} , out) \in \mathcal{R}_{11}$ is applicable to that configuration. Hence, the answer of the problem is affirmative if and only if the skin membrane at configuration $\mathcal{C}_{n^3+8}$ contains object $\texttt{yes}$ (together with objects $c'''', \beta_{n^3+7}, \texttt{no}$), but no object α_{n^3+6}. Otherwise, the skin membrane at configuration $\mathcal{C}_{n^3+8}$ contains objects $\beta_{n^3+7}, \texttt{no}, \alpha_{n^3+6}$, but no object $\texttt{yes}$.

At the last step, in cases when an affirmative answer results, rule $(\texttt{yes} , out)$ is applied to configuration $\mathcal{C}_{n^3+8}$, producing an object $\texttt{yes}$ in the environment, and the computation halts. Otherwise, rule $(\texttt{no}\, \alpha_{n^3+6} , out)$ is applied to that configuration, thus producing a negative answer.

5.3 Main result

Theorem 5.1 $\texttt{HAM-CYCLE} \in \mathbf{PMC}_{\mathbf{CDC(2)}}$.

Proof. The family of P systems with symport/antiport rules and membrane division constructed in Section 3.2 verifies the following:

(a) Every system of the family $\mathbf{\Pi}$ is a recognizer P system with membrane division and symport/antiport rules of length at most 2.

(b) The family $\mathbf{\Pi}$ is polynomially uniform by Turing machines because, for each $n, m \in \mathbb{N}$, the rules of $\mathit{\Pi}(\langle n, m\rangle)$ of the family are recursively defined from $n, m \in \mathbb{N}$, and the amount of resources needed to build an element of the family is of a polynomial order in n, as shown below:
- Size of the alphabet: $n^6 + 12n^3 + 29 \in \Theta(n^6)$;
- Initial number of membranes: $n^3 + 2n + 11 \in \Theta(n^3)$;
- Initial number of objects: $9n^3 + 3n + 13 \in \Theta(n^3)$;
- Number of rules: $n^6 + 4n^5 + n^4 + 13n^3 + 2n + 30 \in \Theta(n^6)$;
- Maximal length of a rule: $2 \in \Theta(1)$.

(c) The pair (cod, s) of polynomial–time computable functions defined in Subsection 5.2 is a polynomial encoding from $\texttt{HAM} - \texttt{CYCLE}$ to $\mathbf{\Pi}$.

(d) The family $\mathbf{\Pi}$ is polynomially bounded, sound and complete with regard to $(\texttt{HAM-CYCLE}, cod, s)$ (see Subsection 5.2).

Therefore, according to Definition 5, the family $\mathbf{\Pi}$ from $\mathbf{CDC(2)}$ solves HAM-CYCLE problem in polynomial time with respect to the number of nodes.

□

Corollary 5.2 $\mathbf{NP} \cup \mathbf{co\text{-}NP} \subseteq \mathbf{PMC_{CDC(2)}}$.

Proof. It suffices to notice that HAM-CYCLE problem is an **NP**-complete problem, HAM-CYCLE$\in \mathbf{PMC_{CDC(2)}}$, and the complexity class $\mathbf{PMC_{CDC(2)}}$ is closed under polynomial-time reduction and under complement.

□

6 Conclusions and open problems

The ability of cell-like P systems with symport/antiport rules involving at most two objects to efficiently solve computationally hard problems, has been studied. Specifically, if further membrane separation rules are allowed (while keeping the minimal cooperation restriction), then only problems in **P** can be solved in polynomial time. Nevertheless, if membrane division rules are allowed (instead of membrane separation rules), then **NP**–complete problems can be solved in polynomial time. In summary, we have two important results concerning the polynomial complexity classes associated with these kind of systems: (a) $\mathbf{P} = \mathbf{PMC_{CSC(2)}}$; and (b) $\mathbf{NP} \cup \mathbf{co\text{-}NP} \subseteq \mathbf{PMC_{CDC(2)}}$.

Therefore, assuming that **P** is different from **NP**, a new frontier of the efficiency has been obtained in Membrane Computing in terms of the kind of rules (separation versus division) able to produce an exponential workspace in linear time. That is, passing from allowing membrane separation rules to allowing membrane division rules in the framework of P systems with symport/antiport rules which use minimal cooperation, amounts to passing from non–efficiency to efficiency.

Acknowledgements

The work of L. Valencia-Cabrera, L.F. Macías-Ramos, A. Riscos-Núñez , and M.J. Pérez-Jiménez was supported by Project TIN2012-37434 of the Ministerio de Economía y Competitividad of Spain, cofinanced by FEDER funds. The work of B. Song and L. Pan was supported by National Natural Science Foundation of China (61033003, 91130034 and 61320106005).

References

1. T.H. Cormen, C.E. Leiserson, R.L. Rivest. *An Introduction to Algorithms*. The MIT Press, Cambridge, Massachussets, 1994.

2. M.R. Garey, D.S. Johnson. *Computers and Intractability A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, (1979).
3. M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez, F.J. Romero-Campero. On the efficiency of cell-like and tissue-like recognizing membrane systems. *International Journal of Intelligent Systems*, **24**, 7 (2009), 747-765.
4. M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez, F.J. Romero-Campero. On the power of dissolution in P systems with active membranes. In R. Freund, Gh. Paun, Gr. Rozenberg, A. Salomaa (eds.) Membrane Computing, 6th International Workshop, WMC 2005, Vienna, Austria, July 18-21, 2005, Revised Selected and Invited Papers. *Lecture Notes in Computer Science*, **3850** (2006), 224-240.
5. L.F. Macías-Ramos, M.J. Pérez-Jiménez, A. Riscos-Núñez, M. Rius-Font. The efficiency of tissue P systems with cell separation relies on the environment. In E. Csuhaj-Varjú, M. Gheorghe, G. Rozenberg, A. Salomaa, G. Vaszil (eds.) Membrane Computing- 13th International Conference CMC 2012 Budapest, Hungary, August 28-31, 2012, Revised Selected Papers. *Lecture Notes in Computer Science*, **7762** (2013), 243-256.
6. L.F. Macías-Ramos, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez, A. Riscos-Núñez, L. Valencia-Cabrera. The Role of the Direction in Tissue P Systems with Cell Separation. *Journal of Automata, Languages and Combinatorics*, **19**, 1-4 (2014), 185-199.
7. L.F. Macías-Ramos, L. Valencia-Cabrera, B. Song, T. Song, L. Pan, M.J. Pérez-Jiménez. Membrane Fission: A Computational Complexity Perspective. *Complexity*, 2015, in press.
8. L. Pan, M.J. Pérez-Jiménez. Computational complexity of tissue–like P systems. *Journal of Complexity*, **26**, 3 (2010), 296–315.
9. A. Păun, Gh. Păun. The power of communication: P systems with symport/antiport, *New Generation Computing*, **20**, 3 (2002), 295–305.
10. I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez, M.A. Gutiérrez, M. Rius-Font. On a partial affirmative answer for a Paun's conjecture. *International Journal of Foundations of Computer Science*, **22**, 1 (2011), 55-64.
11. M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini, F. Complexity classes in models of cellular computing with membranes. *Natural Computing*, **2**, 3 (2003), 265–285.
12. M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini, A polynomial complexity class in P systems using membrane division, *Journal of Automata, Languages and Combinatorics*, **11**, 4 (2006) 423–434.
13. M.J. Pérez-Jiménez, A. Riscos-Núñez, M. Rius-Font, F.J. Romero- Campero. A polynomial alternative to unbounded environment for tissue P systems with cell division. *International Journal of Computer Mathematics*, **90**, 4 (2013), 760-775.
14. M.J. Pérez-Jiménez, A. Riscos-Núñez, M. Rius-Font, L. Valencia-Cabrera. The relevance of the environment on the efficiency of tissue P systems. In A. Alhazov, S. Cojocaru, M. Gheorghe, Y. Rogozhin G. Rozenberg, A. Salomaa (eds.) Membrane Computing- 14th International Conference CMC 2013 Chisinau, Republic of Moldova, August 20-23, 2013, Revised Selected Papers. *Lecture Notes in Computer Science*, **8340** (2014), 308-321.
15. L. Valencia-Cabrera, B. Song, T. Song, L.F. Macías-Ramos, L. Pan, M.J. Pérez-Jiménez. The Role of Cooperation in the Efficiency of Bioinspired Computing Devices, submitted 2015.

Computational Efficiency of P Systems with Symport/Antiport Rules and Membrane Separation

Luis Valencia-Cabrera[1], Bosheng Song[2], Luis F. Macías-Ramos[1], Linqiang Pan[2], Agustín Riscos-Núñez[1], and Mario J. Pérez-Jiménez[1]

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

[2] Key Laboratory of Image Information Processing and Intelligent Control,
School of Automation, Huazhong University of Science and Technology,
Wuhan 430074, Hubei, China
E-mail: boshengsong@163.com, lqpan@mail.hust.edu.cn

Summary. Membrane fission is a process by which a biological membrane is split into two new ones in such a way that the contents of the initial membrane is separated and distributed between the new membranes. Inspired by this biological phenomenon, membrane separation rules were considered in membrane computing. In this paper we deal with cell-like P systems with membrane separation rules that use symport/antiport rules (such systems compute by changing the places of objects with respect to the membranes, and not by changing the objects themselves) as communication rules. Specifically we study a lower bound on the length of communication rules with respect to the computational efficiency of such kind of membrane systems; that is, their ability to solve computationally hard problems in polynomial time by trading space for time. The main result of this paper is the following: communication rules involving at most three objects is enough to achieve the computational efficiency of P systems with membrane separation. Thus, a polynomial time solution to SAT problem is provided in this computing framework. It is known that only problems in **P** can be solved in polynomial time by using minimal cooperation in communication rules and membrane separation, so the lower bound of the efficiency obtained is an optimal bound.

1 Introduction

In a eukaryotic cell, the lipid membranes serve as concentration barriers allowing to incorporate material from its environment (in the case of the cell membrane), or exchange material between compartments. This is done by means of a simple

three-step process whose last step is *membrane fission*, consisting in splitting it into two new membranes [6].

The biological phenomenon of membrane fission process was incorporated in *membrane computing* [11] as a new kind of computational rules, called *membrane separation rules*, in the framework of polarizationless P systems with active membranes [1]. These rules were associated with different subsets of the working alphabet. In [7], a new definition of separation rules in the framework of P systems with active membranes was introduced, where there exists a distinguished partition of the working alphabet into two subsets such that each separation rule is associated with that predefined partition. By applying such a rule, two new membranes are created, the object triggering it is consumed and the remaining objects are *distributed* among the newly created membranes. A uniform and polynomial time solution to `SAT` problem by a family of P systems with active membranes and membrane separation rules was given in [1].

Networks of membranes, which compute by communication only in the form of symport/antiport rules, were considered in [9]. These networks aim to abstract the biological phenomenon of trans-membrane transport of couples of chemical substances, in the same or opposite directions. Such rules are used both for communication with the environment and for direct communication between different membranes. Membrane fission was introduced into tissue-like P systems with symport/antiport rules through *cell separation* rules yielding *tissue P systems with cell separation* [8]. The computational efficiency of these systems was investigated and a tractability border in terms of the length of communication rules was obtained: passing from 1 to 8 amounts to passing from tractability to **NP**–hardness [8]. Furthermore, in [15], that frontier was refined in an optimal sense with respect to communication rules length (passing from 2 to 3).

Cell-like P systems with symport/antiport rules were introduced in [10], and their computational completeness (five membranes are enough if at most two objects are used in the rules) was shown. In this work, we investigate the computational efficiency of this kind of P systems when membrane separation rules are allowed. Specifically, a polynomial time solution to `SAT` problem by using a family of such systems that use communication rules with length at most 3, is provided. The hardness of the design is high and a P-Lingua simulator [4] has been helpful to check the validity of some modules in which the solution was structured

The paper is organized as follows. Section 2 briefly describes some preliminaries in order to make the paper self-contained. In Section 3, the modeling framework of P systems with symport/antiport rules and membrane separation is introduced. Section 4 describes in detail the design of a family solving `SAT` problem efficiently. The solution presented is informally outlined in Section 5. Then, a formal verification of the solution is exhaustively presented in Section 6. The paper ends with a summary of the results and some conclusions.

2 Preliminaries

2.1 Languages and Multisets

An *alphabet* Γ is a non-empty set and their elements are called *symbols*. A *string* u over Γ is a mapping from a natural number $n \in \mathbb{N}$ onto Γ. Number n is called *length* of the string u and it is denoted by $|u|$. The empty string (with length 0) is denoted by λ. A *language* over Γ is a set of strings over Γ.

A *multiset* over an alphabet Γ is an ordered pair (Γ, f), where f is a mapping from Γ onto the set of natural numbers $\mathbb{N}$. For each $x \in \Gamma$ we say that $f(x)$ is the *multiplicity* of x in that multiset. The *support* of a multiset $m = (\Gamma, f)$ is defined as $supp(m) = \{x \in \Gamma \mid f(x) > 0\}$. A multiset is finite if its support is a finite set. We denote by $\emptyset$ the empty multiset. Let us note that a set is a particular case of a multiset when each symbol of the support has multiplicity 1.

Let $m_1 = (\Gamma, f_1)$, $m_2 = (\Gamma, f_2)$ be multisets over Γ, then the union of m_1 and m_2, denoted by $m_1 + m_2$, is the multiset (Γ, g), where $g(x) = f_1(x) + f_2(x)$ for each $x \in \Gamma$. We say that m_1 is contained in m_2 and we denote it by $m_1 \subseteq m_2$, if $f_1(x) \leq f_2(x)$ for each $x \in \Gamma$. The relative complement of m_2 in m_1, denoted by $m_1 \setminus m_2$, is the multiset (Γ, g), where $g(x) = f_1(x) - f_2(x)$ if $f_1(x) \geq f_2(x)$, and $g(x) = 0$ otherwise.

2.2 Graphs

Let us recall that a *free tree* (*tree*, for short) is a connected, acyclic, undirected graph. A *rooted tree* is a tree in which one of the vertices (called *the root of the tree*) is distinguished from the others. In a rooted tree the concepts of ascendants and descendants are defined in a usual way. Given a node x (different from the root), if the last edge on the (unique) path from the root of the tree to the node x is $\{x, y\}$ (in this case, $x \neq y$), then y is **the** *parent* of node x and x is **a** *child* of node y. The root is the only node in the tree with no parent (see [2] for details).

2.3 Encoding ordered pairs of natural numbers

The *pair function* $\langle n, m \rangle = ((n + m)(n + m + 1)/2) + n$ is a polynomial–time computable function from $\mathbb{N} \times \mathbb{N}$ onto $\mathbb{N}$ which is also a primitive recursive and bijective function.

3 P systems with symport/antiport rules with membrane separation

In this section we introduce a kind of cell-like P systems that use communication rules capturing the biological phenomenon of trans-membrane transport of

chemical substances. Specifically, two processes have been considered. The first one allows a multiset of chemical substances to pass through a membrane in the same direction. In the second one, two multisets of chemical substances (located in different biological membranes) only pass with the help of each other (i.e., an *exchange* of objects between both membranes happens).

Next, we introduce an abstraction of these operations in the framework of P systems with symport/antiport rules following [10]. In these models, the membranes are not polarized.

Definition 1. *A P system with symport/antiport rules and membrane separation (SAS P system, for short) of degree* $q \geq 1$ *is a tuple*

$$\Pi = (\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \cdots, \mathcal{R}_q, i_{in}, i_{out}),$$

where

1. Γ *is a finite alphabet;*
2. $\{\Gamma_0, \Gamma_1\}$ *is a partition of* Γ*, that is,* $\Gamma = \Gamma_0 \cup \Gamma_1$*,* $\Gamma_0, \Gamma_1 \neq \emptyset$*,* $\Gamma_0 \cap \Gamma_1 = \emptyset$*;*
3. $\mathcal{E} \subseteq \Gamma$*;*
4. Σ *is an (input) alphabet strictly contained in* Γ *such that* $\mathcal{E} \subseteq \Gamma \setminus \Sigma$*;*
5. μ *is a rooted tree whose nodes are injectively labelled with* $1, \ldots, q$ *(the root of the tree is labelled with* 1*);*
6. $\mathcal{M}_1, \ldots, \mathcal{M}_q$ *are finite multisets over* $\Gamma \setminus \Sigma$*;*
7. $\mathcal{R}_i$*,* $1 \leq i \leq q$*, are finite sets of communication rules over* Γ *of the form:*
   (*a*) Communication rules*:*
      (*a*) Symport rules*:* (u, out) *or* (u, in)*, where* u *is a finite multiset over* Γ *such that* $|u| > 0$*;*
      (*b*) Antiport rules*:* $(u, out; v, in)$*, where* u, v *are finite multisets over* Γ *such that* $|u| > 0$ *and* $|v| > 0$*;*
   (*b*) Separation rules*:* $[\, a \,]_i \rightarrow [\, \Gamma_0 \,]_i [\, \Gamma_1 \,]_i$*, where* $a \in \Gamma$*,* $i \in \{2, \ldots, q\}$*, with* $i \neq i_{out}$ *the label of a leaf of the tree;*
8. $i_{in} \in \{1, \ldots, q\}$ *and* $i_{out} \in \{0, 1, \ldots, q\}$*.*

A P system with symport/antiport rules and membrane separation of degree $q \geq 1$

$$\Pi = (\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \cdots, \mathcal{R}_q, i_{in}, i_{out})$$

can be viewed as a set of q membranes, labelled with $1, \ldots, q$, arranged in a hierarchical structure μ given by a rooted tree whose root is called the *skin membrane*, such that: (a) $\mathcal{M}_1, \ldots, \mathcal{M}_q$ represent the finite multisets of *objects* (symbols of the working alphabet Γ) initially placed into the q membranes of the system; (b) $\mathcal{E}$ is the set of objects initially located in the environment of the system (labelled with 0), all of them available in an arbitrary number of copies; (c) $\mathcal{R}_1, \cdots, \mathcal{R}_q$ are finite sets of communication rules over Γ ($\mathcal{R}_i$ is associated with the membrane i of μ); and (d) i_{out} represents a distinguished *region* which will encode the output of the system. We use the term *region* i ($0 \leq i \leq q$) to refer to membrane i in the case

$1 \leq i \leq q$ and to refer to the environment in the case $i = 0$. The length of rule (u, out) or (u, in) (resp. $(u, out; v, in)$) is defined as $|u|$ (resp. $|u| + |v|$).

For each membrane $i \in \{2, \ldots, q\}$ (different from the skin membrane) we denote by $p(i)$ the parent of membrane i in the rooted tree μ. We define $p(1) = 0$, that is, by convention the "parent" of the skin membrane is the environment.

An *instantaneous description* or a *configuration* at an instant t of a SA P system is described by the membrane structure at instant t, all multisets of objects over Γ associated with all the membranes present in the system, and the multiset of objects over $\Gamma - \mathcal{E}$ associated with the environment at that moment. Recall that there are infinite copies of objects from $\mathcal{E}$ in the environment, so that this set is not properly changed along the computation. The *initial configuration* of the system is $(\mu, \mathcal{M}_1, \cdots, \mathcal{M}_q; \emptyset)$.

A symport rule $(u, out) \in \mathcal{R}_i$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant t if there exists a membrane labelled with i in $\mathcal{C}_t$ such that multiset u is contained in such membrane. When applying a rule $(u, out) \in \mathcal{R}_i$ to such a membrane, the objects specified by u are sent out of that membrane into the region immediately outside (the parent $p(i)$ of i). Note that this can be the environment in the case of the skin membrane.

A symport rule $(u, in) \in \mathcal{R}_i$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant t if multiset u is contained in the parent of i. When applying a rule $(u, in) \in \mathcal{R}_i$ to a membrane labelled with i, the multiset of objects u leaves the parent of such membrane and enters into the region defined by that membrane.

An antiport rule $(u, out; v, in) \in \mathcal{R}_i$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant t if there exists a membrane labelled with i in $\mathcal{C}_t$ such that multiset u is contained in such membrane, and multiset v is contained in the parent of i. When applying a rule $(u, out; v, in) \in \mathcal{R}_i$ to such a membrane, the objects specified by u are sent out of it into the parent of i and, at the same time, the objects specified by v are brought into that membrane i.

A separation rule $[a]_i \rightarrow [\Gamma_0]_i[\Gamma_1]_i \in \mathcal{R}_i$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant t, if there exists an elementary membrane labelled with i in $\mathcal{C}_t$, different from the skin membrane, such that it contains object a. When applying a separation rule $[a]_i \rightarrow [\Gamma_0]_i\,[\Gamma_1]_i \in \mathcal{R}_i$ to such a membrane in a configuration $\mathcal{C}_t$, triggered by object a, that membrane is separated into two membranes with the same label; at the same time, object a is consumed; the objects (from the original membrane) belonging to Γ_0 are placed in the first membrane, while those from belonging to Γ_1 are placed in the second membrane. This way, several membranes with the same label i can be present in the new membrane structure μ' of the system: for each membrane labelled with $i \neq 1$ we have an arc $(p(i), i)$ in μ' as a result of the application of a membrane separation rule $[a]_i \rightarrow [\Gamma_0]_i[\Gamma_1]_i$.

Regarding the semantics of these variants, the rules of such P systems are applied in a non-deterministic maximally parallel manner with the following important remark: when a membrane i is separated, the membrane separation rule

is the only one from $\mathcal{R}_i$ which is applied for that membrane at that step. The new membranes resulting from separation could participate in the interaction with other membranes or the environment by means of communication rules at the next step – providing that they are not separated once again. The label of a membrane precisely identify the rules which can be applied to it.

Let Π be a P system with symport/antiport rules and membrane separation. We say that configuration $\mathcal{C}_t$ yields configuration $\mathcal{C}_{t+1}$ in one *transition step*, denoted by $\mathcal{C}_t \Rightarrow_\Pi \mathcal{C}_{t+1}$, if we can pass from $\mathcal{C}_t$ to $\mathcal{C}_{t+1}$ by applying the rules from the system following the above semantics. A *computation* of Π is a (finite or infinite) sequence of configurations such that: (a) the first term is the initial configuration of the system; (b) for each $n \geq 2$, the n-th configuration of the sequence is obtained from the previous configuration in one transition step; and (c) if the sequence is finite (called a *halting computation*) then the last term is a *halting configuration* (a configuration where no rule of the system is applicable). All the computations start from an initial configuration and proceed as stated above; only a halting computation gives a result, which is encoded by the objects present in the output region i_{out} associated with the halting configuration. For each finite multiset w over the input alphabet Σ, a *computation of Π with input multiset w* starts from the configuration of the form $(\mu, \mathcal{M}_1, \ldots, \mathcal{M}_{i_{in}} + w, \ldots, \mathcal{M}_q, \emptyset)$, where the input multiset w is added to the content of the input membrane i_{in}. That is, we have an initial configuration associated with each input multiset w over Σ in recognizer P systems with symport/antiport rules. We denote by $\Pi + w$ the P system Π with input multiset w.

If $\mathcal{C} = (\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_r)$ of Π is a halting computation, then the *length* of $\mathcal{C}$, denoted by $|\mathcal{C}|$, is r. For each i $(1 \leq i \leq q)$, we denote by $\mathcal{C}_t(i)$ the finite multiset of objects over Γ contained in all membranes labelled with i (by applying separation rules different membranes with the same label can be created) at configuration $\mathcal{C}_t$.

3.1 Recognizer P systems with symport/antiport rules

Recognizer P systems were introduced in [14], and they provide a natural framework to solve decision problems by means of computational devices in membrane computing (i.e., P systems).

Definition 2. *A recognizer P system with symport/antiport rules and membrane separation of degree $q \geq 1$ is a P system with symport/antiport rules and membrane separation of degree q such that:*

1. *The working alphabet has two distinguished symbols* `yes` *and* `no`*;*
2. *initial multisets are finite multisets over $\Gamma \setminus \Sigma$ such that at least one copy of* `yes` *or* `no` *is present in some of them;*
3. *the output region is the environment ($i_{out} = 0$);*
4. *all computations halt;*

5. *if $\mathcal{C}$ is a computation of the system, then either symbol* `yes` *or symbol* `no` *(but not both) must have been released into the environment, and only at the last step of the computation.*

Let us notice that if a recognizer P system has a symport rule of the type $(u, in) \in \mathcal{R}_1$ then the multiset u must contain some object from $\Gamma \setminus \mathcal{E}$ because on the contrary, it might exist non-halting computations of Π.

We say that a computation $\mathcal{C}$ of a recognizer P system is an *accepting computation* (respectively, *rejecting computation*) if object `yes` (respectively, object `no`) appears in the environment associated with the corresponding halting configuration of $\mathcal{C}$, and neither object `yes` nor `no` appears in the environment associated with any non–halting configuration of $\mathcal{C}$.

We denote by $\mathbf{CSC}(k)$ the class of all recognizer P systems with symport/antiport rules and membrane separation (for elementary membranes) such that the length of the communication rules of the system is at most k.

3.2 Polynomial complexity classes of recognizer P systems with symport/antiport rules

Next, according to [13], we define what it means to solve a decision problem by a family of recognizer P systems with symport/antiport rules and membrane separation.

Definition 3. *A decision problem $X = (I_X, \theta_X)$ is solvable in polynomial time by a family $\mathbf{\Pi} = \{\Pi(n) \mid n \in \mathbb{N}\}$ of recognizer P systems with symport/antiport rules and membrane separation or membrane separation, if the following holds:*

- *The family $\mathbf{\Pi}$ is polynomially uniform by Turing machines, that is, there exists a deterministic Turing machine working in polynomial time which constructs the system $\Pi(n)$ from $n \in \mathbb{N}$;*
- *there exists a pair (cod, s) of polynomial-time computable functions over I_X such that:*
  - *for each instance $u \in I_X$, $s(u)$ is a natural number and $cod(u)$ is an input multiset of the system $\Pi(s(u))$;*
  - *for each $n \in \mathbb{N}$, $s^{-1}(n)$ is a finite set;*
  - *the family $\mathbf{\Pi}$ is polynomially bounded with regard to (X, cod, s), that is, there exists a polynomial function p, such that for each $u \in I_X$ every computation of $\Pi(s(u)) + cod(u)$ is halting and it performs at most $p(|u|)$ steps;*
  - *the family $\mathbf{\Pi}$ is sound with regard to (X, cod, s), that is, for each $u \in I_X$, if there exists an accepting computation of $\Pi(s(u)) + cod(u)$, then $\theta_X(u) = 1$;*
  - *the family $\mathbf{\Pi}$ is complete with regard to (X, cod, s), that is, for each $u \in I_X$, if $\theta_X(u) = 1$, then every computation of $\Pi(s(u)) + cod(u)$ is an accepting one.*

According to Definition 3, we say that the family $\mathbf{\Pi}$ provides a *uniform solution* to the decision problem X. We also say that ordered pair (cod, s) is a polynomial encoding from X in $\mathbf{\Pi}$ and s is the size mapping associated with that solution. It is worth pointing out that for each instance $u \in I_X$, the P system $\Pi(s(u))+cod(u)$ is *confluent*, in the sense that all possible computations of the system must give the same answer.

If $\mathbf{R}$ is a class of recognizer P systems, then we denote by $\mathbf{PMC_R}$ the set of all decision problems which can be solved in polynomial time (and in a uniform way) by means of recognizer P systems from $\mathbf{R}$. The class $\mathbf{PMC_R}$ is closed under complement and polynomial–time reductions (see [13] for details). Besides, we have $\mathbf{P} \subseteq \mathbf{PMC_R}$. Indeed, if $X \in \mathbf{P}$ then we consider the family $\mathbf{\Pi} = \{\Pi(n) \mid n \in \mathbb{N}\}$ where $\Pi(n) = \Pi(0)$, for each $n \in \mathbb{N}$, and $\Pi(0)$ is a P system from $\mathbf{R}$ of degree 1 containing only two rules $(\texttt{yes}, out)$ and $(\texttt{no}, out)$. Let us consider the polynomial encoding from X in $\mathbf{\Pi}$ defined as follows: (a) $s(u) = 0$, for each $u \in I_X$; and (b) $cod(u) = \texttt{yes}$ if $\theta_X(u) = 1$ and $cod(u) = \texttt{no}$ if $\theta_X(u) = 0$. Then, the family $\mathbf{\Pi}$ solves X according to Definition 3.

4 On Efficiency of CSC(3)

The limitations on the efficiency of P systems with membrane separation whose symport/antiport rules involve at most two objects, have been established [5]. Specifically, it has been proved that the polynomial complexity class $\mathbf{PMC_{CSC(2)}}$ is equal to class $\mathbf{P}$: only tractable problems can be efficiently solved by using families of P systems with membrane separation which make use of symport/antiport rules with length at most 2. In this Section we analyze the computational efficiency of familes of P systems from $\mathbf{CSC(3)}$, and it is given a polynomial time solution to SAT problem by means of a family of such P systems, in a uniform way, according to Definition 3.

4.1 A polynomial time solution to SAT problem in CSC(3)

Let us recall that SAT problem is the following: *given a Boolean formula in conjunctive normal form (CNF), to determine whether or not there exists an assignment to its variables on which it evaluates true.* This is a well known **NP**-complete problem [3].

We consider a family $\mathbf{\Pi} = \{\Pi(t) \mid t \in \mathbb{N}\}$ of recognizer P system from $\mathbf{CSC}(3)$, such that each system $\Pi(t)$, with $t = \langle n, m\rangle$, will process all instances of SAT problem (an instance is a Boolean formula φ in conjunctive normal form with n variables and m clauses) provided that the appropriate input multiset $cod(\varphi)$ is supplied to the system.

For each $n, m \in \mathbb{N}$, we consider the recognizer P system from $\mathbf{CSC}(3)$

$$\Pi(\langle n, m\rangle) = (\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \cdots, \mathcal{R}_q, i_{in}, i_{out})$$

defined as follows:

(1) Working alphabet:

$$\begin{aligned}\Gamma = \Sigma \cup \mathcal{E} \cup \{&\alpha_{i,0,k}, \alpha'_{i,0,k} \mid 1 \le i \le n-1 \wedge 0 \le k \le 1\} \cup \\ &\{A_1, B_1, b_1, b'_1, c_1, c'_1, v_1, q_{1,1}, \beta_0, \beta'_0, \beta''_0, \gamma_0, \gamma'_0, \gamma''_0, \gamma'''_0, f_0, \texttt{yes}, \texttt{no}\} \cup \\ &\{f'_i \mid 0 \le i \le 3n+2m+1\}, \cup \{\rho_{i,0}, \tau_{i,0} \mid 1 \le i \le n\}, \cup \{\delta_{j,0} \mid 0 \le j \le m\}\end{aligned}$$

where the input alphabet is $\Sigma = \{x_{i,j}, \overline{x}_{i,j} \mid 1 \le i \le n \wedge 1 \le j \le m\}$, and the alphabet of the environment is:

$$\begin{aligned}\mathcal{E} = \{&\alpha_{i,j,k}, \alpha'_{i,j,k} \mid 1 \le i \le n-1 \wedge 1 \le j \le 3(n-1) \wedge 0 \le k \le 1\} \cup \\ &\{\beta_j, \beta'_j, \beta''_j, \gamma_j, \gamma'_j, \gamma''_j, \gamma'''_j \mid 1 \le j \le 3(n-1)\} \cup \\ &\{\rho_{i,j}, \tau_{i,j} \mid 1 \le i \le n \wedge 1 \le j \le 3n-1\} \cup \\ &\{T_{i,j}, T'_{i,j}, F_{i,j}, F'_{i,j} \mid 1 \le i < j \wedge 1 \le j \le n\} \cup \\ &\{T_{i,i}, F'_{i,i}, T_i, F_i \mid 1 \le i \le n\} \cup \{A_i, A'_i, B_i, B'_i \mid 2 \le i \le n+1\} \cup \\ &\{b_i, b'_i, c_i, c'_i \mid 2 \le i \le n\} \cup \{v_i \mid 2 \le i \le n-1\} \cup \\ &\{y_i, a_i, w_i \mid 1 \le i \le n-1\} \cup \{z_i \mid 1 \le i \le n-2\} \cup \\ &\{q_{i,j} \mid 1 \le i \le j \wedge 2 \le j \le n-1\} \cup \{u_{i,j} \mid 1 \le i \le j \wedge 1 \le j \le n-2\} \cup \\ &\{t_{i,j}, f_{i,j}, r_{i,j}, s_{i,j} \mid 1 \le i \le j \wedge 1 \le j \le n-1\} \cup \\ &\{d_{i,j,k}, \overline{d}_{i,j,k} \mid 1 \le i \le n \wedge 1 \le j \le m \wedge 1 \le k \le n-1\} \cup \\ &\{f_r \mid 1 \le r \le 3n+2m\} \cup \{e_{i,j}, \overline{e}_{i,j} \mid 1 \le i \le n \wedge 1 \le j \le m\} \cup \\ &\{\delta_{j,r} \mid 0 \le j \le m \wedge 1 \le r \le 3n\} \cup \{E_j \mid 0 \le j \le m\} \cup \{S\}\end{aligned}$$

(2) The partition is $\{\Gamma_0, \Gamma_1\}$, where $\Gamma_0 = \Gamma \setminus \Gamma_1$ and

$$\begin{aligned}\Gamma_1 = \{&T'_{i,j} F'_{i,j} \mid 1 \le i < j \wedge 1 \le j \le n\} \cup \{F'_{i,i} \mid 1 \le i \le n\} \cup \\ &\{A'_i, B'_i \mid 2 \le i \le n+1\}\end{aligned}$$

(3) Membrane structure: $\mu = [\,[\quad]_2\,[\quad]_3]_1$. The input membrane is the membrane labelled with 1.

(4) Initial multisets:

$$\begin{aligned}\mathcal{M}_1 = \{&\alpha_{i,0,k}, \alpha'_{i,0,k} \mid 1 \le i \le n-1 \wedge 0 \le k \le 1\} \cup \{\rho_{i,0}, \tau_{i,0} \mid 1 \le i \le n\} \cup \\ &\{\beta_0, \beta'_0, \beta''_0, \gamma_0, \gamma'_0, \gamma''_0, \gamma'''_0, c_1, c'_1, b_1, b'_1, v_1, q_{1,1}, f_0, \texttt{yes}\} \cup \\ &\{\delta_{j,0} \mid 0 \le j \le m\} \cup \{f'_p \mid 1 \le p \le 3n+2m+1\} \\ \mathcal{M}_2 = \{&A_1, B_1\} \\ \mathcal{M}_3 = \{&f'_0, \texttt{no}\}\end{aligned}$$

(5) $\boxed{\text{Rules in } \mathcal{R}_1}$:

1.1 Rules to generate in the membrane 1 of configuration $\mathcal{C}_{3p+1}$ ($p = 1, \ldots, n-1$) the objects $T_{i,p+1}^{2^{p-1}}, T_{i,p+1}'^{2^{p-1}}, F_{i,p+1}^{2^{p-1}}, F_{i,p+1}'^{2^{p-1}}$:

$$\left.\begin{array}{l}(\alpha_{i,0,k}, out; \alpha_{i,1,k}, in)\\ (\alpha'_{i,0,k}, out; \alpha'_{i,1,k}, in)\\ (\alpha_{i,1,k}, out; \alpha_{i,2,k}, in)\\ (\alpha'_{i,1,k}, out; \alpha'_{i,2,k}, in)\\ (\alpha_{i,2,k}, out; \alpha_{i,3,k}, in)\\ (\alpha'_{i,2,k}, out; \alpha'_{i,3,k}, in)\end{array}\right\} 1 \leq i \leq n-1 \wedge 0 \leq k \leq 1$$

$$\begin{array}{l}(\alpha_{i,3p,k}\,, out; \alpha_{i,3p+1,k}\ \Delta^k_{i,p+1}\,, in) : 1 \leq i \leq p \wedge 1 \leq p \leq n-2 \wedge 0 \leq k \leq 1\\ (\alpha'_{i,3p,k}\,, out; \alpha'_{i,3p+1,k}\ \Delta'^k_{i,p+1}\,, in) : 1 \leq i \leq p \wedge 1 \leq p \leq n-2 \wedge 0 \leq k \leq 1\\ (\alpha_{i,3p,k}\,, out; \alpha_{i,3p+1,k}\,, in) : p+1 \leq i \leq n-1 \wedge 1 \leq p \leq n-2 \wedge 0 \leq k \leq 1\\ (\alpha'_{i,3p,k}\,, out; \alpha'_{i,3p+1,k}\,, in) : p+1 \leq i \leq n-1 \wedge 1 \leq p \leq n-2 \wedge 0 \leq k \leq 1\\ (\alpha_{i,3p+1,k}\,, out; \alpha_{i,3p+2,k}\,, in) : 1 \leq i \leq n-1 \wedge 1 \leq p \leq n-2 \wedge 0 \leq k \leq 1\\ (\alpha'_{i,3p+1,k}\,, out; \alpha'_{i,3p+2,k}\,, in) : 1 \leq i \leq n-1 \wedge 1 \leq p \leq n-2 \wedge 0 \leq k \leq 1\\ (\alpha_{i,3p+2,k}\,, out; \alpha^2_{i,3p+3,k}\,, in) : 1 \leq i \leq n-1 \wedge 1 \leq p \leq n-2 \wedge 0 \leq k \leq 1\\ (\alpha'_{i,3p+2,k}\,, out; \alpha'^2_{i,3p+3,k}\,, in) : 1 \leq i \leq n-1 \wedge 1 \leq p \leq n-2 \wedge 0 \leq k \leq 1\\ (\alpha_{i,3(n-1),k}\,, out;\ \Delta^k_{i,n}\,, in) : 1 \leq i \leq n-1 \wedge 0 \leq k \leq 1\\ (\alpha'_{i,3(n-1),k}\,, out;\ \Delta'^k_{i,n}\,, in) : 1 \leq i \leq n-1 \wedge 0 \leq k \leq 1\end{array}$$

where $\Delta^0_{i,j} = F_{i,j}$, $\Delta'^0_{i,j} = F'_{i,j}$, $\Delta^1_{i,j} = T_{i,j}$, $\Delta'^1_{i,j} = T'_{i,j}$.

1.2 Rules to generate in the membrane 1 of configuration $\mathcal{C}_{3p+1}$ ($p = 0, 1, \ldots, n-1$) the objects $B^{2^p}_{p+2}, B'^{2^p}_{p+2}, S^{2^p}$:

$$\left.\begin{array}{l}(\beta_{3p}\,, out; \beta_{3p+1}\ B_{p+2}\,, in)\\ (\beta'_{3p}\,, out; \beta'_{3p+1}\ B'_{p+2}\,, in)\\ (\beta''_{3p}\,, out; \beta''_{3p+1}\ S\,, in)\\ (\beta_{3p+1}\,, out; \beta_{3p+2}\,, in)\\ (\beta'_{3p+1}\,, out; \beta'_{3p+2}\,, in)\\ (\beta''_{3p+1}\,, out; \beta''_{3p+2}\,, in)\\ (\beta_{3p+2}\,, out; \beta^2_{3p+3}\,, in)\\ (\beta'_{3p+2}\,, out; \beta'^2_{3p+3}\,, in)\\ (\beta''_{3p+2}\,, out; \beta''^2_{3p+3}\,, in)\end{array}\right\} 0 \leq p \leq n-3$$

$$\left.\begin{array}{l}(\beta_{3(n-2)}\,, out; \beta_{3(n-2)+1}\ B_n\,, in)\\ (\beta'_{3(n-2)}\,, out; \beta'_{3(n-2)+1}\ B'_n\,, in)\\ (\beta''_{3(n-2)}\,, out; \beta''_{3(n-2)+1}\ S\,, in)\\ (\beta_{3(n-2)+1}\,, out; \beta_{3(n-2)+2}\,, in)\\ (\beta'_{3(n-2)+1}\,, out; \beta'_{3(n-2)+2}\,, in)\\ (\beta''_{3(n-2)+1}\,, out; \beta''_{3(n-2)+2}\,, in)\\ (\beta_{3(n-2)+2}\,, out; \beta^2_{3(n-2)+3}\,, in)\\ (\beta'_{3(n-2)+2}\,, out; \beta'^2_{3(n-2)+3}\,, in)\\ (\beta''_{3(n-2)+2}\,, out; \beta''^2_{3(n-2)+3}\,, in)\end{array}\right\}$$

$$\left.\begin{array}{l}(\beta_{3(n-1)}\,, out; B_{n+1}\,, in)\\ (\beta'_{3(n-1)}\,, out; B'_{n+1}\,, in)\\ (\beta''_{3(n-1)}\,, out; S\,, in)\end{array}\right\}$$

1.3 Rules to generate in the membrane 1 of configuration $\mathcal{C}_{3p+1}$ ($p = 0, 1, \ldots, n-1$) the objects $T_{p+1,p+1}^{2^p}, T_{p+1,p+1}'^{2^p}, A_{p+2}^{2^p}, A_{p+2}'^{2^p}$:

$$\left.\begin{array}{r}
(\gamma_{3p}\,, out; \gamma_{3p+1}\ T_{p+1,p+1}\,, in)\\
(\gamma'_{3p}\,, out; \gamma'_{3p+1}\ F'_{p+1,p+1}\,, in)\\
(\gamma''_{3p}\,, out; \gamma''_{3p+1}\ A_{p+2}\,, in)\\
(\gamma'''_{3p}\,, out; \gamma'''_{3p+1}\ A'_{p+2}\,, in)\\
(\gamma_{3p+1}\,, out; \gamma_{3p+2}\,, in)\\
(\gamma'_{3p+1}\,, out; \gamma'_{3p+2}\,, in)\\
(\gamma''_{3p+1}\,, out; \gamma''_{3p+2}\,, in)\\
(\gamma'''_{3p+1}\,, out; \gamma'''_{3p+2}\,, in)\\
(\gamma_{3p+2}\,, out; \gamma^2_{3p+3}\,, in)\\
(\gamma'_{3p+2}\,, out; \gamma'^2_{3p+3}\,, in)\\
(\gamma''_{3p+2}\,, out; \gamma''^2_{3p+3}\,, in)\\
(\gamma'''_{3p+2}\,, out; \gamma'''^2_{3p+3}\,, in)
\end{array}\right\} 0 \leq p \leq n-3$$

$$\left.\begin{array}{r}
(\gamma_{3(n-2)}\,, out; \gamma_{3(n-2)+1}\ T_{n-1,n-1}\,, in)\\
(\gamma'_{3(n-2)}\,, out; \gamma'_{3(n-2)+1}\ F'_{n-1,n-1}\,, in)\\
(\gamma''_{3(n-2)}\,, out; \gamma''_{3(n-2)+1}\ A_n\,, in)\\
(\gamma'''_{3(n-2)}\,, out; \gamma''_{3(n-2)+1}\ A'_n\,, in)\\
(\gamma_{3(n-2)+1}\,, out; \gamma_{3(n-2)+2}\, in)\\
(\gamma'_{3(n-2)+1}\,, out; \gamma'_{3(n-2)+2}\, in)\\
(\gamma''_{3(n-2)+1}\,, out; \gamma''_{3(n-2)+2}\, in)\\
(\gamma'''_{3(n-2)+1}\,, out; \gamma'''_{3(n-2)+2}\, in)\\
(\gamma_{3(n-2)+2}\,, out; \gamma^2_{3(n-2)+3}\, in)\\
(\gamma'_{3(n-2)+2}\,, out; \gamma'^2_{3(n-2)+3}\, in)\\
(\gamma''_{3(n-2)+2}\,, out; \gamma''^2_{3(n-2)+3}\, in)\\
(\gamma'''_{3(n-2)+2}\,, out; \gamma'''^2_{3(n-2)+3}\, in)
\end{array}\right\}$$

$$\left.\begin{array}{r}
(\gamma_{3(n-1)}\,, out;\ T_{n,n}\,, in)\\
(\gamma'_{3(n-1)}\,, out;\ F'_{n,n}\,, in)\\
(\gamma''_{3(n-1)}\,, out;\ A_{n+1}\,, in)\\
(\gamma'''_{3(n-1)}\,, out;\ A'_{n+1}\,, in)
\end{array}\right\}$$

1.4 Rules to generate in the membrane 1 of configuration $\mathcal{C}_{3n}$ the objects $T_i^{2^{n-1}}, F_i^{2^{n-1}}$ ($1 \leq i \leq n$):

$$\left.\begin{array}{r}
(\rho_{i,0}\,, out; \rho_{i,1}\,, in)\\
(\tau_{i,0}\,, out; \tau_{i,1}\,, in)\\
(\rho_{i,1}\,, out; \rho_{i,2}\,, in)\\
(\tau_{i,1}\,, out; \tau_{i,2}\,, in)\\
(\rho_{i,2}\,, out; \rho_{i,3}\,, in)\\
(\tau_{i,2}\,, out; \tau_{i,3}\,, in)
\end{array}\right\} 1 \leq i \leq n$$

$$\left.\begin{array}{r}(\rho_{i,3p}\,,out;\rho_{i,3p+1}\,,in)\\(\tau_{i,3p}\,,out;\tau_{i,3p+1}\,,in)\\(\rho_{i,3p+1}\,,out;\rho^2_{i,3p+2}\,,in)\\(\tau_{i,3p+1}\,,out;\tau^2_{i,3p+2}\,,in)\\(\rho_{i,3p+2}\,,out;\rho_{i,3p+3}\,,in)\\(\tau_{i,3p+2}\,,out;\tau_{i,3p+3}\,,in)\end{array}\right\}\ 1\leq i\leq n\wedge 1\leq p\leq n-2$$

$$\left.\begin{array}{r}(\rho_{i,3(n-1)}\,,out;\rho_{i,3(n-1)+1}\,,in)\\(\tau_{i,3(n-1)}\,,out;\tau_{i,3(n-1)+1}\,,in)\\(\rho_{i,3(n-1)+1}\,,out;\rho^2_{i,3(n-1)+2}\,,in)\\(\tau_{i,3(n-1)+1}\,,out;\tau^2_{i,3(n-1)+2}\,,in)\\(\rho_{i,3(n-1)+2}\,,out;T_i\,,in)\\(\tau_{i,3(n-1)+2}\,,out;F_i\,,in)\end{array}\right\}\ 1\leq i\leq n$$

$$\left.\begin{array}{r}(A_i\,,out;a_i\,,in)\\(A'_i\,,out;a_i\,,in)\\(B_i\,,out;a_i\,,in)\\(B'_i\,,out;a_i\,,in)\end{array}\right\}\ 1\leq i\leq n-1$$

$$\left.\begin{array}{r}(y_i\,,out;z_i\,w_i\,,in):1\leq i\leq n-2\\(y_{n-1}\,,out;w_{n-1}\,,in):\end{array}\right\}$$

$$\left.\begin{array}{r}(w_i\,,out;c_{i+1}\,c'_{i+1}\,,in):1\leq i\leq n-1\\(z_i\,,out;v_{i+1}\,,in):1\leq i\leq n-2\end{array}\right\}$$

$$\left.\begin{array}{r}(v_i\,,out;y^2_i\,,in)\\(a_i\,,out;b_{i+1}\,b'_{i+1}\,,in)\end{array}\right\}\ 1\leq i\leq n-1$$

$$\left.\begin{array}{r}(q_{1,1}\,,out;r_{1,1}\,,in)\\(q_{i,j}\,,out;r^2_{i,j}\,,in):1\leq i\leq n-1\wedge i\leq j\leq n-1\end{array}\right\}$$

$$\left.\begin{array}{r}(r_{i,j}\,,out;s_{i,j}\,u_{i,j}\,,in):1\leq i\leq n-2\wedge i\leq j\leq n-2\\(r_{i,n-1}\,,out;s_{i,n-1}\,,in):\qquad 1\leq i\leq n-1\end{array}\right\}$$

$$(s_{i,j}\,,out;t_{i,j}\,f_{i,j}\,,in)\ :\ 1\leq i\leq n-1\wedge i\leq j\leq n-1$$

$$\left.\begin{array}{r}(u_{1,j}\,,out;q_{1,j+1}\,q_{2,j+1}\,,in):\qquad 1\leq j\leq n-2\\(u_{i,j}\,,out;q_{i+1,j+1}\,,in):2\leq i\leq j\wedge 2\leq j\leq n-2\end{array}\right\}$$

$$\left.\begin{array}{r}(T_{i,j}\,t_{i,j},out)\\(T'_{i,j}\,t_{i,j},out)\\(F_{i,j}\,f_{i,j},out)\\(F'_{i,j}\,f_{i,j},out)\end{array}\right\}\ 1\leq i\leq j\wedge 1\leq j\leq n$$

1.5 Rules allowing that each object $x_{i,j}$ (meaning that $x_i \in C_j$) and $\overline{x}_{i,j}$ (meaning that $\neg x_i \in C_j$) results in the corresponding $e_{i,j}$ and $\overline{e}_{i,j}$ objects with multiplicity 2^{n-1} in membrane 1 of configuration $\mathcal{C}_{n+1}$.

$$\left.\begin{array}{r}(x_{i,j}\,, out; d^2_{i,j,1}\,; in)\\ (\overline{x}_{i,j}\,, out; \overline{d}^2_{i,j,1}\,; in)\end{array}\right\} 1 \leq i \leq n \wedge 1 \leq j \leq m$$

$$\left.\begin{array}{r}(d_{i,j,k}\,, out; d^2_{i,j,k+1}\,, in)\\ (\overline{d}_{i,j,k}\,, out; \overline{d}^2_{i,j,k+1}\,, in)\end{array}\right\} 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge 1 \leq k \leq n-2$$

$$\left.\begin{array}{r}(d_{i,j,n-1}\,, out; e_{i,j}\,, in)\\ (\overline{d}_{i,j,n-1}\,, out; \overline{e}_{i,j}\,, in)\end{array}\right\} 1 \leq i \leq n \wedge 1 \leq j \leq m$$

1.6 Output rule with **affirmative** answer: $(E_0\, f_{3n+2m}\, \texttt{yes}\,; out)$.

1.7 Output rule with **negative** answer: $(f_{3n+2m}\, \texttt{no}\,; out)$.

1.8 Rules to generate in the membrane 1 of configuration $\mathcal{C}_{3n}$ the objects $E_1^{2^n}$, and in the membrane 1 of configuration $\mathcal{C}_{3n+1}$ the objects $E_0^{2^n}, E_2^{2^n}, \ldots, E_m^{2^n}$:

$$\left.\begin{array}{r}(\delta_{j,3p}\,, out; \delta_{j,3p+1}, in)\\ (\delta_{j,3p+1}\,, out; \delta^2_{j,3p+2}, in)\end{array}\right\} 0 \leq j \leq m \wedge 0 \leq p \leq n-1$$

$$(\delta_{j,3p+2}\,, out; \delta_{j,3p+3}, in)\ 0 \leq j \leq m \wedge 0 \leq p \leq n-2$$

$$(\delta_{1,3(n-1)+2}\,, out; E_1, in)$$

$$\left.\begin{array}{r}(\delta_{j,3(n-1)+2}\,, out; \delta_{j,3(n-1)+3}, in)\\ (\delta_{j,3n}\,, out; E_j, in)\end{array}\right\} 0 \leq j \leq m \wedge j \neq 1$$

$$(f_p\,, out; f_{p+1}\,; in)\ 0 \leq p \leq 3n+2m-1$$

1.9 Rules to remove a part of the garbage:

$$\left.\begin{array}{r}(t_{i,k}\ T_{i,k}\,, out)\\ (t_{i,k}\ T'_{i,k}\,, out)\\ (f_{i,k}\ F_{i,k}\,, out)\\ (f_{i,k}\ F'_{i,k}\,, out)\end{array}\right\} 1 \leq i < k \wedge 2 \leq k \leq n$$

$$\left.\begin{array}{r}(t_{i,i}\ T_{i,i}\,, out)\\ (f_{i,i}\ F'_{i,i}\,, out)\end{array}\right\} 1 \leq i \leq n$$

$$\left.\begin{array}{r}(b_k\ B_{k+1}\,, out)\\ (b'_k\ B'_{k+1}\,, out)\\ (c_k\ A_{k+1}\,, out)\\ (c'_k\ A'_{k+1}\,, out)\end{array}\right\} n-1 \leq k \leq n$$

(6) Rules in $\mathcal{R}_2$:

2.1 Separation rule: $[\,S\,]_2 \rightarrow [\,\Gamma_0\,]_2\ [\,\Gamma_1\,]_2$.

2.2 Rules to produce objects $T_{i,i}, A_{i+1}, F'_{i,i}, A'_{i+1}$ in each membrane 2:

$$\left.\begin{array}{r}(A_i, out; c_i\, c'_i, in)\\ (A'_i, out; c_i\, c'_i, in)\\ (B_i, out; b_i\, b'_i, in)\\ (B'_i, out; b_i\, b'_i, in)\\ (b_i, out; B_{i+1}\, S, in)\\ (b'_i, out; B'_{i+1}, in)\\ (c_i, out; T_{i,i}\, A_{i+1}, in)\\ (c'_i, out; F'_{i,i}\, A'_{i+1}, in)\end{array}\right\} 1 \leq i \leq n$$

2.3 Rules to produce an object E_1 in each membrane 2 of configuration $\mathcal{C}_{3n+1}$ and an object E_0 in each membrane 2 of configuration $\mathcal{C}_{3n+2}$:

$$\begin{array}{c}(B_{n+1}, out; E_1, in)\\ (B'_{n+1}, out; E_1, in)\\ (A_{n+1}, out; E_0, in)\\ (A'_{n+1}, out; E_0, in)\end{array}$$

2.4 Rules to produce a truth assignment in each membrane 2 of configuration $\mathcal{C}_{3n+1}$:

$$\left.\begin{array}{r}(T_{i,j}, out; t_{i,j}, in)\\ (T'_{i,j}, out; t_{i,j}, in)\\ (F_{i,j}, out; f_{i,j}, in)\\ (F'_{i,j}, out; f_{i,j}, in)\end{array}\right\} 1 \leq i \leq j \wedge 1 \leq j \leq n$$

$$\left.\begin{array}{r}(t_{i,j}, out; T_{i,j+1}\, T'_{i,j+1}, in)\\ (f_{i,j}, out; F_{i,j+1}\, F'_{i,j+1}, in)\end{array}\right\} 1 \leq i \leq j \wedge 1 \leq j \leq n-1$$

$$\left.\begin{array}{r}(T_{i,n}, out; T_i, in)\\ (T'_{i,n}, out; T_i, in)\\ (F_{i,n}, out; F_i, in)\\ (F'_{i,n}, out; F_i, in)\end{array}\right\} 1 \leq i \leq n$$

2.5 Rules to check clause C_j through the truth assignment encoded by a membrane 2:

$$\left.\begin{array}{r}(E_j\, T_i, out; e_{i,j}, in)\\ (E_j\, F_i, out; \overline{e}_{i,j}, in)\end{array}\right\} 1 \leq i \leq n \wedge 1 \leq j \leq m$$

2.6 Rules to restore the truth assignment encoded by a membrane 2 which makes clause C_j true:

$$\left.\begin{array}{r}(e_{i,j}, out, E_{j+1}\, T_i, in)\\ (\overline{e}_{i,j}, out, E_{j+1}\, F_i, in)\end{array}\right\} 1 \leq i \leq n \wedge 1 \leq j \leq m-1$$

2.7 Rules to send an object E_0 to membrane 1 of configuration $\mathcal{C}_{3n+2m+1}$, meaning that some truth assignment encoded by a membrane labelled with 2 makes the input formula φ true:

$$\left.\begin{array}{l}(e_{i,m}\, E_0\,;out)\\(\overline{e}_{i,m}\, E_0\,;out)\end{array}\right\} 1 \leq i \leq n$$

(7) Rules in $\mathcal{R}_3$:

3.1 Rules to produce objects $f'_{3n+2m+1}$ and `no` in the membrane 1 of configuration $\mathcal{C}_{3n+2m+2}$.

$$(f'_p, out; f'_{p+1}, in)\ 0 \leq p \leq 3n+2m$$
$$(f'_{3n+2m+1}\,\texttt{no}\,;out)$$

5 An overview of the computations

A family of recognizer P systems with symport/antiport rules and membrane separation is constructed above. For an instance of `SAT` problem $\varphi = C_1 \wedge \cdots \wedge C_m$, consisting of m clauses $C_j = l_{j,1} \vee \cdots \vee l_{j,r_j},\ 1 \leq j \leq m$, where $Var(\varphi) = \{x_1, \cdots, x_n\}$, and $l_{j,k} \in \{x_i, \neg x_i \mid 1 \leq i \leq n\},\ 1 \leq j \leq m, 1 \leq k \leq r_j$. Let us assume that the number of variables, n, and the number of clauses, m, of the input formula φ, are greater or equal to 2.

The *size* mapping on the set of instances is defined as $s(\varphi) = \langle m, n\rangle$, for each $\varphi \in I_{\texttt{SAT}}$, and the encoding of the instance φ is the multiset

$$cod(\varphi) = \{x_{i,j} :\ x_i \in C_j\}\ \cup\ \{\overline{x}_{i,j} :\ \neg x_i \in C_j\}$$

That is, $x_{i,j}$ (respectively, $\overline{x}_{i,j}$) denotes variable x_i (respectively, $\neg x_i$) belonging to clause C_j. Then, the Boolean formula φ will be processed by the system $\Pi(s(\varphi))$ with input multiset $cod(\varphi)$.

Next, we informally describe how the system $\Pi(s(\varphi)) + cod(\varphi)$ works, in order to process the instance φ of `SAT` problem. The solution proposed follows a brute force algorithm in the framework of recognizer P systems with symport/antiport rules and membrane separation, and it consists of the following phases:

- *Generation phase*: using separation rules, all truth assignments for the variables associated with the Boolean formula $\varphi(x_1, \ldots, x_n)$ are produced. This phase exactly takes $3n + 1$ computation steps.
- *Checking phase*: checking whether or not the input formula φ is satisfied by some truth assignment generated in the previous phase. This phase takes, exactly, $3m + 1$ steps, being m the number of clauses of the formula φ.

- *Output phase*: the system sends the right answer to the environment depending on the results of the previous phase. This phase takes, exactly, 1 step if the answer affirmative, and 2 steps if the answer is negative.

Generation phase

In this phase, all truth assignments for the variables associated with the Boolean formula $\varphi(x_1, \ldots, x_n)$ are generated, by applying separation rules in membranes labelled with 2. This way, after completing the phase, there will exist 2^n membranes labelled with 2 such that each of them encodes a different truth assignment of the variables $\{x_1, \ldots, x_n\}$.

This phase consists in a loop with n iterations and one additional final step. Each iteration of the loop takes three steps and, consequently, this phase takes $3n + 1$ steps.

To do this, in the configurations of the kind $\mathcal{C}_{3p+2}$ $(0 \leq p \leq n-1)$ there exist 2^p membranes labelled with 2 containing objects

$$A_{p+2},\ A'_{p+2},\ B_{p+2},\ B'_{p+2},\ T_{p+1,p+1},\ F'_{p+1,p+1},\ S$$

along with $2p$–tuples of objects $(\pi_{1,p+1}, \pi'_{1,p+1}, \ldots, \pi_{p,p+1}, \pi'_{p,p+1})$, with $\pi \in \{T, F\}$, in such a way that the corresponding tuples are all different in the different membranes.

Thus, a separation rule can be applied to each membrane labelled with 2. As a consequence, in configuration $\mathcal{C}_{3p+3}$ $(0 \leq p \leq n-2)$ there will exist 2^{p+1} membranes labelled with 2. 2^p of them will contain objects A_{p+2} and B_{p+2}, as well as $(p+1)$–tuples $(\pi_{1,p+1}, \ldots, \pi_{p+1,p+1})$, with $\pi \in \{T, F\}$, in such a way that $\pi_{p+1,p+1} = T_{p+1,p+1}$, and the corresponding tuples of these membranes are all different. The other 2^p membranes labelled with 2 contain the objects A'_{p+2} and B'_{p+2}, as well as $(p+1)$–tuples $(\pi'_{1,p+1}, \ldots, \pi'_{p+1,p+1})$ with $\pi \in \{T, F\}$, in such a way that $\pi'_{p+1,p+1} = F'_{p+1,p+1}$ and the corresponding tuples of these membranes are all different.

Finally, in configuration $\mathcal{C}_{3n}$ there exist 2^n membranes labelled with 2. 2^{n-1} of them contain the objects A_{n+1} and B_{n+1}, as well as n–tuples $(\pi_{1,n}, \ldots, \pi_{n,n})$ with $\pi \in \{T, F\}$, in such a way that $\pi_{n,n} = T_{n,n}$ and the corresponding tuples of these membranes are all different. The other 2^{n-1} membranes labelled with 2 contain the objects A'_{n+1} and B'_{n+1}, as well as n–tuples $(\pi'_{1,n}, \ldots, \pi'_{n,n})$ with $\pi \in \{T, F\}$, in such a way that $\pi'_{n,n} = F'_{n,n}$ and the corresponding tuples of these membranes are all different.

This phase ends in the step $3n + 1$, where configuration $\mathcal{C}_{3n+1}$ contains 2^n membranes labelled with 2, each one of them containing the objects A_{n+1} and E_1, as well as n–tuples $(\pi_1, \ldots, \pi_n)$ with $\pi \in \{T, F\}$, and the corresponding tuples of these membranes are all different.

Simultaneously, during the generation phase, from the input multiset placed initially in the skin membrane, 2^{n-1} copies of each object of that multiset are generated in that membrane, corresponding to configuration $\mathcal{C}_n$. Due to technical reasons, we will change variables $x_{i,j}$ and $\overline{x}_{i,j}$ by $e_{i,j}$ and $\overline{e}_{i,j}$, respectively. This is accomplished by using the following rules from $\mathcal{R}_1$:

$$\left.\begin{array}{rl}
(x_{i,j}\,, out; d^2_{i,j,1}\,; in) & : \; 1 \leq i \leq n \wedge 1 \leq j \leq m \\
(\overline{x}_{i,j}\,, out; \overline{d}^2_{i,j,1}\,; in) & : \; 1 \leq i \leq n \wedge 1 \leq j \leq m \\
(d_{i,j,k}\,, out; d^2_{i,j,k+1}\,, in) & : \; 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge 1 \leq k \leq n-2 \\
(\overline{d}_{i,j,k}\,, out; \overline{d}^2_{i,j,k+1}\,, in) & : \; 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge 1 \leq k \leq n-2 \\
(d_{i,j,n-1}\,, out; e_{i,j}\,, in) & : \; 1 \leq i \leq n \wedge 1 \leq j \leq m \\
(\overline{d}_{i,j,n-1}\,, out; \overline{e}_{i,j}\,, in) & : \; 1 \leq i \leq n \wedge 1 \leq j \leq m
\end{array}\right\}$$

The cited multiset that codifies the input formula will be denoted by $(cod(\varphi))^{2^{n-1}}_e$.

Checking phase

This phase begins at computation step $3n + 2$ and consists in a loop with m iterations, taking each of them 2 steps. Hence, the checking phase takes $2m$ steps.

In the configuration $\mathcal{C}_{3n+1}$, the presence of an object E_1 in each membrane labelled with 2, along with the code of a truth assignment, marks the beginning of this phase. In the first iteration of the loop, the truth assignments making clause C_1 of φ true are found. To do this, the following rules of $\mathcal{R}_2$ are applied:

$$\left.\begin{array}{l}
(E_1\, T_i, out; e_{i,1}, in) \\
(E_1\, F_i, out; \overline{e}_{i,1}, in)
\end{array}\right\} 1 \leq i \leq n$$

Simultaneously, in the computation step $(3n+1)+2$, the object E_0 is incorporated to each of the membranes labelled with 2 by means of the application of the following rules of $\mathcal{R}_2$: (A_{n+1}, out, E_0, in) and (A'_{n+1}, out, E_0, in).

At this point, the presence of an object $e_{i,1}$ or an object $\overline{e}_{i,1}$ in a membrane 2 of the configuration $\mathcal{C}_{(3n+1)+1}$ indicates that this membrane codifies a truth assignment making the first clause true.

In the next computation step, those membranes will incorporate an object E_2 coming from the skin by applying the following rules from $\mathcal{R}_2$:

$$\left.\begin{array}{l}
(e_{i,1}, out, E_2\, T_i, in) \\
(\overline{e}_{i,1}, out, E_2\, F_i, in)
\end{array}\right\} 1 \leq i \leq n$$

This way, the presence of an object E_2 in a membrane 2 of the configuration $\mathcal{C}_{(3n+1)+2}$ indicates that this membrane codifies a truth assignment making true the first clause and that is ready to check the second clause of the formula. That

is, from this moment, the membranes labelled with 2 not making true the first clause will not evolve.

In the j-th iteration ($2 \leq j \leq m$) of the aforementioned loop, the truth assignments making true the clause C_j of the formula are checked, taking into account that only the truth assignments containing the object E_j will be checked, since only these membranes make clauses $C_1, \ldots, C_{j-1}$ of φ true. This is accomplished by applying the following rules from $\mathcal{R}_2$:

$$\left.\begin{array}{l}(E_j\, T_i, out; e_{i,j}, in)\\ (E_j\, F_i, out; \overline{e}_{i,j}, in)\end{array}\right\} 1 \leq i \leq n \wedge 1 \leq j \leq m$$

Then, the presence of an object $e_{i,j}$ or an object $\overline{e}_{i,j}$ in a membrane 2 of the configuration $\mathcal{C}_{(3n+1)+2\cdot(j-1)+1}$ indicates that this membrane codifies a truth assignment making clauses $C_1, \ldots, C_j$ of φ true. Following this, those membranes will incorporate an object E_{j+1} coming from the skin by applying the following rules from $\mathcal{R}_2$:

$$\left.\begin{array}{l}(e_{i,j}, out, E_{j+1}\, T_i, in)\\ (\overline{e}_{i,j}, out, E_{j+1}\, F_i, in)\end{array}\right\} 1 \leq i \leq n \wedge 1 \leq j \leq m-1$$

If the input formula φ is satisfiable, then in some membrane labelled with 2 of the configuration $\mathcal{C}_{(3n+1)+2(m-1)+1}$ there will exist an object $e_{i,m}$ or an object $\overline{e}_{i,m}$. This indicates that the truth assignment that this membrane codifies makes true all the clauses from φ and, consequently, makes true the input formula. In this case, the checking phase ends up by applying a rule from $\mathcal{R}_2$ of the kind $(e_{i,m}\, E_0\, ; out)$ or $(\overline{e}_{i,m}\, E_0\, ; out)$ making an object E_0 go to the skin membrane of the configuration $\mathcal{C}_{(3n+1)+2(m-1)+2}$, where also the object f_{3n+2m} has been produced.

If the input formula φ is not satisfiable, the no membrane labelled with 2 of the configuration $\mathcal{C}_{(3n+1)+2(m-1)+1}$ contains an object $e_{i,m}$ neither an object $\overline{e}_{i,m}$. In this case, the checking phase ends up by applying the rule $(f'_{3n+2m}, out; f'_{3n+2m+1}, in) \in \mathcal{R}_3$ (in fact, this is the only rule applicable to the configuration $\mathcal{C}_{(3n+1)+2(m-1)+1}$).

The checking phase ends at step $(3n+1) + 2(m-1) + 2 = 3n + 2m + 1$.

Output phase

If the input formula φ is satisfiable, then objects E_0 and f_{3n+2m} will appear in the input membrane of the configuration $\mathcal{C}_{3n+2m+1}$. Then, by applying the rule $(E_0\, f_{3n+2m}\ \texttt{yes}\, ; out)$ in the skin membrane, the object `yes` is released into the environment, providing and affirmative answer at computation step $(3n+1) + 2m + 1 = 3n + 2m + 2$.

If the input formula φ is not satisfiable, then objects f_{3n+2m} and `yes` are present in the skin membrane of the configuration $\mathcal{C}_{(3n+1)+2(m-1)+1} = \mathcal{C}_{3n+2m}$,

but not the object E_0. In this case, the only applicable rule in the system is $(f'_{3n+2m}, out; f'_{3n+2m+1}, in)$ in the membrane 3 and in the next computation step only the rule $(f'_{3n+2m+1}\,\texttt{no}\,; out) \in \mathcal{R}_3$ is applicable. Consequently, objects $f_{3n+2m}, \texttt{yes}, f'_{3n+2m+1}$ and $\texttt{no}$ appear in the skin membrane of the configuration $\mathcal{C}_{3n+2m+2}$. Then, by applying the rule $(f_{3n+2m}\,\texttt{no}\,; out)$ in the skin membrane, an object $\texttt{no}$ will be released into the environment, providing a negative answer in the step $3n + 2m + 3$.

Hence, the output phase takes 1 computation step in the case of an affirmative answer, and 2 computation steps in the case of a negative answer.

6 A Formal Verification

In this Section we show that the family $\mathbf{\Pi} = \{\Pi(\langle n, m\rangle) \mid n, m \in \mathbb{N}\}$ considered in the previous section provides a polynomial time solution to SAT problem according to the Definition 3.2. For that, we must prove that it is polynomially uniform by Turing machines and that there exists a polynomial encoding (cod, s) of SAT problem in the family $\mathbf{\Pi}$ such that $\mathbf{\Pi}$ is polynomially bounded, sound and complete with regards to $(\texttt{SAT}, cod, s)$.

6.1 Polynomial Uniformity of the Family

In this subsection, we shall show that the family $\mathbf{\Pi} = \{\Pi(\langle n, m\rangle) \mid n, m \in \mathbb{N}\}$ defined above is polynomially uniform by Turing machines. To this aim we prove that $\Pi(\langle n, m\rangle)$ is built in polynomial time with respect to the size parameter m and n of instances of SAT problem.

It is easy to check that the rules of a system $\Pi(\langle n, m\rangle)$ of the family are recursively defined through the values n (that represents the number of variables of the input formula) and m (that represents the number of clauses of the input formula). The amount of resources to construct $\Pi(\langle n, m\rangle)$ is of a polynomial order in the numbers n and m, as shown below:

1. The size of the working alphabet is of the order $\Theta(n^2 \cdot m)$.
2. The initial number of cells is $3 \in \Theta(1)$.
3. The initial number of objects in membranes is $9n + 3m + 17 \in \Theta(\max\{n, m\})$.
4. The total number of rules is of order $\Theta(n^2 \cdot m)$.
5. The maximum length of a rule is $3 \in \Theta(1)$.

Therefore, there exists a deterministic Turing machine that builds the system $\Pi(\langle n, m\rangle)$ in a polynomial time with respect to n and m.

6.2 Soundness and Completeness of the Family

In the first place, we are going to justify that in the skin membrane of the configuration $\mathcal{C}_n$ objects $e_{i,j}$ appear such that $x_{i,j} \in cod(\varphi)$ and objects $\overline{e}_{i,j}$ appear such that $\overline{x}_{i,j} \in cod(\varphi)$, each of them with multiplicity 2^{n-1}.

Theorem 1. *For each k $(1 \leq k \leq n-1)$, the membrane 1 of the configuration $\mathcal{C}_k$ contains the following multiset of objects:*

$$\{d_{i,j,k}^{2^k} \mid x_{i,j} \in cod(\varphi) \wedge 1 \leq i \leq n \wedge 1 \leq j \leq m\} \cup$$
$$\{\overline{d}_{i,j,k}^{2^k} \mid \overline{x}_{i,j} \in cod(\varphi) \wedge 1 \leq i \leq n \wedge 1 \leq j \leq m\}$$

Proof. By bounded induction on k. Let us start analyzing the base case $k = 1$. The membrane 1 is the input membrane of the system and, consequently, contains the multiset of objects:

$$cod(\varphi) = \{x_{i,j} \mid x_i \in C_j \wedge 1 \leq i \leq n \wedge 1 \leq j \leq m\} \cup$$
$$\{\overline{x}_{i,j} \mid \neg x_i \in C_j \wedge 1 \leq i \leq n \wedge 1 \leq j \leq m\}$$

Then, by applying the rules from R_1 of the kind

$$\left.\begin{array}{l}(x_{i,j}\,, out; d_{i,j,1}^2\,; in)\\ (\overline{x}_{i,j}\,, out; \overline{d}_{i,j,1}^2\,; in)\end{array}\right\} 1 \leq i \leq n \wedge 1 \leq j \leq m$$

to the configuration $\mathcal{C}_0$, membrane 1 of $\mathcal{C}_1$ ends containing the multiset of objects:

$$\{d_{i,j,1}^2 \mid x_{i,j} \in cod(\varphi) \wedge 1 \leq i \leq n \wedge 1 \leq j \leq m\} \cup$$
$$\{\overline{d}_{i,j,1}^2 \mid x_{i,j} \in cod(\varphi) \wedge 1 \leq i \leq n \wedge 1 \leq j \leq m\}$$

Consequently, the result holds for $k = 1$.

By induction hypothesis, let us consider k such that $1 \leq k < n-1$ and let us suppose the result holds for k, that is, let us suppose that the membrane 1 of the configuration $\mathcal{C}_k$ contains the multiset of objects:

$$\{d_{i,j,k}^{2^k} \mid x_{i,j} \in cod(\varphi) \wedge 1 \leq i \leq n \wedge 1 \leq j \leq m\} \cup$$
$$\{\overline{d}_{i,j,k}^{2^k} \mid x_{i,j} \in cod(\varphi) \wedge 1 \leq i \leq n \wedge 1 \leq j \leq m\}$$

Let us see that the result also holds for $k+1$.

By applying the rules of R_1 of the kind

$$\left.\begin{array}{l}(d_{i,j,k}\,, out; d_{i,j,k+1}^2\,, in)\\ (\overline{d}_{i,j,k}\,, out; \overline{d}_{i,j,k+1}^2\,, in)\end{array}\right\} 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge 1 \leq k \leq n-2$$

to the configuration $\mathcal{C}_k$ results that the membrane 1 of $\mathcal{C}_{k+1}$ contains the multiset of objects:

$$\{d_{i,j,k+1}^{2^{k+1}} \mid x_{i,j} \in cod(\varphi) \wedge 1 \leq i \leq n \wedge 1 \leq j \leq m\} \cup$$
$$\{\overline{d}_{i,j,k+1}^{2^{k+1}} \mid x_{i,j} \in cod(\varphi) \wedge 1 \leq i \leq n \wedge 1 \leq j \leq m\}$$

Consequently, the result holds for $k+1$. This ends up with the proof of the theorem. □

Next, we are going to analyse the evolution of the system thurough every phase in the proposed solution.

Generation phase

We will denote by $(cod(\varphi))_e$ the multiset of $(cod(\varphi))$ where the objects $x_{i,j}$ and $\overline{x}_{i,j}$ are replaced by $e_{i,j}$ and $\overline{e}_{i,j}$, respectively. If in the multiset $(cod(\varphi))_e$ each object has multiplicity 2^k, then we will denote it by $(cod(\varphi))_e^{2^k}$.

Next, let us consider the formulas $\theta_1(p)$, $\theta_2(p)$ and $\theta_3(p)$, where $p = 0, 1, \ldots, n-1$. These formulas indicate the relevant contents of the configurations $\mathcal{C}_{3p+1}$, $\mathcal{C}_{3p+2}$ and $\mathcal{C}_{3p+3}$, respectively.

The formula $\theta_1(p)$ captures the contents of configuration $\mathcal{C}_{3p+1}$, corresponding to the **first step** of each loop iteration. The formula $\theta_1(p)$ is the following:

"*In configuration* $\mathcal{C}_{3p+1}$ *the following holds:*

- *In the membrane labelled with 1 we can find as relevant objects:*
  - $\rho_{i,3p+1}$ *and* $\tau_{i,3p+1}$ *(for* $1 \leq i \leq n$*), each of them with multiplicity 1 if* $r = 0$ *and multiplicity* 2^{p-1} *if* $p \geq 1$.
  - $\delta_{j,3p+1}$ *(for* $1 \leq j \leq m$*), each of them with multiplicity* 2^p.
  - $f_{3p+1}, \texttt{yes}, f'_0, \ldots, \widehat{f'_{3p+1}}, \ldots, f'_{3n+2m+1}$, *each of them with multiplicity 1.*
  - A_{p+2} , A'_{p+2} , B_{p+2} , B'_{p+2} , S, *each of them with multiplicity* 2^p.
  - $T_{p+1,p+1}$, $F'_{p+1,p+1}$, *each of them with multiplicity* 2^p.
  - *If* $p = 0$ *it contains* A_1 , B_1, *each of them with multiplicity* 1.
  - *If* $0 \leq p \leq n-2$ *then there also exist objects*
    - $\alpha_{i,3p+1,k}$, $\alpha'_{i,3p+1,k}$ *(for* $1 \leq i \leq n-1$ *and* $0 \leq k \leq 1$*), each of them with multiplicity* 2^{p-1}, *if* $1 \leq i \leq n-2$, *and with multiplicity 1 if* $p = 0$.
    - β_{3p+1}, β'_{3p+1}, β''_{3p+1}, *each of them with multiplicity* 2^p.
    - γ_{3p+1}, γ'_{3p+1}, γ''_{3p+1}, γ'''_{3p+1}, *each of them with multiplicity* 2^p.
    - y_{p+1} *with multiplicity* 2^{p+1}.
    - $r_{1,p+1}, r_{2,p+1}, \ldots, r_{p+1,p+1}$ *with multiplicity* 2^p.

  - *If* $1 \leq p \leq n-2$, *it contains the objects* A_{p+1}, A'_{p+1}, B_{p+1}, B'_{p+1}, *each of them with multiplicity* 2^{p-1}.
  - *If* $1 \leq p \leq n-1$, *it contains the objects* $T_{i,p+1}$, $T'_{i,p+1}$, $F_{i,p+1}$, $F'_{i,p+1}$, *for* $1 \leq i \leq p$, *each of them with multiplicity* 2^{p-1}.
  - *If also* $3p+1 \geq n$, *then it contains* $(cod(\varphi))_e^{2^{n-1}}$."

- *There exist* 2^p *membranes labelled with 2, each of them containing objects* $b_{p+1}, b'_{p+1}, c_{p+1}, c'_{p+1}$ *with multiplicity 1. If* $p \geq 1$, *then each membrane labelled with 2 has a p-tuple of objects* $(\pi_{1,p}, \ldots, \pi_{p,p})$ *such that* $\pi \in \{t, f\}$ *and the corresponding tuples are all different in the different membranes. Thus, for example, for* $p = 1$, *there exist* $2^1 = 2$ *membranes labelled with 2 such that both of them contain objects* b_2, b'_2, c_2, c'_2 *and, additionally, the first of them contains the object* $t_{1,1}$ *and the second one* $f_{1,1}$. *For* $p = 2$, *there exist* $2^2 = 4$ *membranes labelled with 2 such that all of them contain the objects* b_3, b'_3, c_3, c'_3. *In addition, one of them contains* $t_{1,2}, t_{2,2}$, *the second one contains* $f_{1,2}, t_{2,2}$, *the third one contains* $t_{1,2}, f_{2,2}$ *and the fourth one contains* $f_{1,2}, f_{2,2}$.
- *The membrane labelled with 3 contains the objects* f'_{3p+1} *and* `no` *with multiplicity 1.*

The formula $\theta_2(p)$ captures the content of the configuration $\mathcal{C}_{3p+2}$ corresponding to the **second step** of each iteration of the loop. The formula $\theta_2(p)$ is the following:

"*In configuration* $\mathcal{C}_{3p+2}$ *the following holds:*

- *In membrane 1 we can find as relevant objects:*
  - *Objects* $\rho_{i,3p+1}$ *and* $\tau_{i,3p+1}$ *(for* $1 \leq i \leq n$*), each of them with multiplicity* 2^p.
  - *Objects* $\delta_{j,3p+1}$ *(for* $1 \leq j \leq m$*), each of them with multiplicity* 2^{p+1}.
  - *Objects* $f_{3p+1}, \texttt{yes}, f'_0, \ldots, \widehat{f'_{3p+1}}, \ldots, f'_{3n+2m+1}$
  - *If* $0 \leq p \leq n-2$, *then it also contains objects*
    - $\alpha_{i,3p+1,k}$, $\alpha'_{i,3p+1,k}$ *(for* $1 \leq i \leq n-1$ *y* $0 \leq k \leq 1$*), each of them with multiplicity* 2^{p-1} *if* $p \geq 1$, *and with multiplicity 1 if* $p = 0$.
    - β_{3p+1}, β'_{3p+1}, β''_{3p+1}, *each of them with multiplicity* 2^p.
    - γ_{3p+1}, γ'_{3p+1}, γ''_{3p+1}, γ'''_{3p+1}, *each of them with multiplicity* 2^p.
    - w_{p+1} *and* a_{p+1} *with multiplicity* 2^{p+1}.
    - $s_{1,p+1}$, ... , $s_{p+1,p+1}$, *each of them with multiplicity* 2^p.
  - *If* $0 \leq p \leq n-3$, *then it also contains:*

  - ⋆ *objects* z_{p+1} *with multiplicity* 2^{p+1} *and objects* $u_{1,p+1}, \ldots, u_{p+1,p+1}$, *each of them with multiplicity* 2^p.
- *If* $3p+1 \geq n$, *then it also contains* $(cod(\varphi))_e^{2^{n-1}}$."

- *There exist* 2^p *membranes labelled with 2, each of them containing objects* B_{p+2}, S, B'_{p+2}, *as well as objects* $T_{p+1,p+1}, A_{p+2}, F'_{p+1,p+1}, A'_{p+2}$, *all of them with multiplicity 1. Also, if* $p \geq 1$, *then each membrane labelled with 2 contains* $2p$*–tuples of objects* $(\pi_{1,p+1}, \pi'_{1,p+1}, \ldots, \pi_{p,p+1}, \pi'_{p,p+1})$, *with* $\pi \in \{T, F\}$, *in such a way that in the different membranes, the corresponding tuples are different with each other.*
- *The membrane labelled with 3 contains the objects* f'_{3p+1} *and* `no`.

The formula $\theta_3(p)$ captures the contents of the configuration $\mathcal{C}_{3p+3}$ corresponding to the **third step** of each loop iteration. The formula $\theta_3(p)$ is the following:

"*In the configuration* $\mathcal{C}_{3p+3}$ *the following holds:*

- *In the membrane 1 we can find as relevant objects:*
  - *If* $0 \leq p \leq n-2$, *then there exist objects*
    - ⋆ $\alpha_{i,3p+3,k}$, $\alpha'_{i,3p+3,k}$ (*for* $1 \leq i \leq n-1$ *and* $0 \leq k \leq 1$), *each of them with multiplicity* 2^p.
    - ⋆ β_{3p+3}, β'_{3p+3}, β''_{3p+3}, *each of them with multiplicity* 2^{p+1}.
    - ⋆ γ_{3p+3}, γ'_{3p+3}, γ''_{3p+3}, γ'''_{3p+3}, *each of them with multiplicity* 2^{p+1}.
    - ⋆ $\rho_{i,3p+3}$ *and* $\tau_{i,3p+3}$ (*for* $1 \leq i \leq n$), *each of them with multiplicity* 2^p.
    - ⋆ $\delta_{j,3p+3}$ (*for* $1 \leq j \leq m$), *each of them with multiplicity* 2^{p+1}.
    - ⋆ $f_{3p+3}, \mathtt{yes}, f'_0, \ldots, \widehat{f'_{3p+3}}, \ldots, f'_{3n+2m+1}$, *each of them with multiplicity 1.*
    - ⋆ $b_{p+2}, b'_{p+2}, c_{p+2}, c'_{p+2}$, *each of them with multiplicity* 2^{p+1}.
    - ⋆ $t_{1,p+1}$, $t_{p+1,p+1}$, $f_{1,p+1}$, $f_{p+1,p+1}$ *and* $q_{1,p+2}$, $q_{p+2,p+2}$, *each of them with multiplicity* 2^p.
    - ⋆ w_{p+1} *and* a_{p+1} *with multiplicity* 2^{p+1}.
  - *If* $0 \leq p \leq n-3$ *then it also contains objects*
    - ⋆ v_{p+2} *with multiplicity* 2^{p+1}.
  - *If* $p = n-1$ *then it also contains objects*
    - ⋆ $\delta_{j,3p+3}$ (*for* $1 \leq j \leq m$), *each of them with multiplicity* 2^{p+1}.
    - ⋆ $f_{3p+3}, \mathtt{yes}, f'_0, \ldots, \widehat{f'_{3p+3}}, \ldots, f'_{3n+2m+1}$, *each of them with multiplicity 1.*
    - ⋆ T_i (*for* $1 \leq i \leq n$), *each of them with multiplicity* 2^p.

  - ⋆ *E_1 with multiplicity 2^{p+1}.*
  - *If, besides, $3p+1 \geq n$, then it also contains $(cod(\varphi))_e^{2^{n-1}}$.*

- *There exist 2^{p+1} membranes labelled with 2. 2^p of these membranes contain objects A_{p+2} and B_{p+2}, as well as $(p+1)$–tuples $(\pi_{1,p+1}, \ldots, \pi_{p+1,p+1})$ with $\pi \in \{T, F\}$, in such a way that $\pi_{p+1,p+1} = T_{p+1,p+1}$ and the corresponding tuples are all different in the different membranes.*

  The other 2^p membranes labelled with 2 contain the objects A'_{p+2} and B'_{p+2}, as well as $(p+1)$–tuples $(\pi'_{1,p+1}, \ldots, \pi'_{p+1,p+1})$ with $\pi \in \{T, F\}$, in such a way that $\pi'_{p+1,p+1} = F'_{p+1,p+1}$ and the corresponding tuples are all different in the different membranes.

- *The membrane labelled with 3 contains the objects f'_{3p+3} and* `no`*".*

Next, we are going to prove that the formula $\theta(p) \equiv \theta_1(p) \wedge \theta_2(p) \wedge \theta_3(p)$ is an invariant of the loop associated to the generation phase.

Theorem 2. *For each $p = 0, 1, \ldots, n-1$, the formula $\theta(p) \equiv \theta_1(p) \wedge \theta_2(p) \wedge \theta_3(p)$ is true*

Proof. By bounded induction on p. Let us start analyzing the base case $p = 0$; that is, let us show that the formula $\theta(0)$ holds. For this, we have to prove that the formulas $\theta_1(0)$, $\theta_2(0)$ and $\theta_3(0)$ are true.

Let us recall that the initial configuration of the system, $\mathcal{C}_0$ is the following:

$$\begin{aligned}
\mathcal{C}_0(1) = {} & \{\alpha_{i,0,k}, \alpha'_{i,0,k} \mid 1 \leq i \leq n-1 \wedge 0 \leq k \leq 1\} \cup \{\rho_{i,0}, \tau_{i,0} \mid 1 \leq i \leq n\} \cup \\
& \{\beta_0, \beta'_0, \beta''_0, \gamma_0, \gamma'_0, \gamma''_0, \gamma'''_0, c_1, c'_1, b_1, b'_1, v_1, q_{1,1}, f_0, \texttt{yes}\} \cup \\
& \{\delta_{j,0} \mid 1 \leq j \leq m+1\} \cup \{f'_p \mid 1 \leq p \leq 3n+2m+1\} \cup cod(\varphi) \\
\mathcal{C}_0(2) = {} & \{A_1, B_1\} \\
\mathcal{C}_0(3) = {} & \{f'_0, \texttt{no}\}
\end{aligned}$$

Then, the following rules are applied to the membranes as stated:

- In membrane 1 the following rules from $\mathcal{R}_1$ are applied:

$$\left.\begin{array}{l}
(\alpha_{i,0,k}, out; \alpha_{i,1,k}, in) \ : \ 1 \leq i \leq n-1 \wedge 0 \leq k \leq 1 \\
(\alpha'_{i,0,k}, out; \alpha'_{i,1,k}, in) \ : \ 1 \leq i \leq n-1 \wedge 0 \leq k \leq 1 \\
(\beta_0\,, out; \beta_1\ B_2\,, in) \\
(\beta'_0\,, out; \beta'_1\ B'_2\,, in) \\
(\beta''_0\,, out; \beta''_1\ S\,, in) \\
(\gamma_0\,, out; \gamma_1\ T_{1,1}\,, in) \\
(\gamma'_0\,, out; \gamma'_1\ F'_{1,1}\,, in) \\
(\gamma''_0\,, out; \gamma''_1\ A_2\,, in) \\
(\gamma'''_0\,, out; \gamma'''_1\ A'_2\,, in) \\
(\tau_{i,0}\,, out; \tau_{i,1}\,, in) \ : \ 1 \leq i \leq n \\
(\rho_{i,0}\,, out; \rho_{i,1}\,, in) \ : \ 1 \leq i \leq n \\
(\delta_{j,0}\,, out; \delta_{j,1}\,, in) \ : \ 1 \leq j \leq m \\
(f_0\,, out; f_1\,, in) \\
(v_1\,, out; y_1^2\,, in) \\
(q_{1,1}\,, out; r_{1,1}\,, in) \\
(x_{i,j}\,, out; d^2_{i,j,1}\,; in) \ : \ 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge 0 \leq k \leq 1 \\
(\overline{x}_{i,j}\,, out; \overline{d}^2_{i,j,1}\,; in) \ : \ 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge 0 \leq k \leq 1
\end{array}\right\}$$

- In membrane 2, the following rules from $\mathcal{R}_2$ are applied:

$$\left.\begin{array}{l}
(A_1, out; c_1\, c'_1, in) \\
(B_1, out; b_1\, b'_1, in)
\end{array}\right\}$$

- In membrane 3, the following rules from $\mathcal{R}_3$ are applied: $(f'_0, out; f'_1, in)$

By applying the aforementioned rules, the configuration $\mathcal{C}_1$ holds the following:

- In the membrane labelled with 1 we have the objects:
  - B_2, S, B'_2, $T_{1,1}$, A_2, $F'_{1,1}$, A'_2, A_1, B_1, each one with multiplicity 1.
  - Objects $f_1, \texttt{yes}, f'_0, \widehat{f'_1}, f'_2, \ldots, f'_{3n+2m+1}$
  - Objects $\rho_{i,1}$ and $\tau_{i,1}$ (for $1 \leq i \leq n$), each one with multiplicity 1.
  - Objects $\delta_{j,1}$ (for $1 \leq j \leq m$), each one with multiplicity 1.
  - $\alpha_{i,1,k}$, $\alpha'_{i,1,k}$ (for $1 \leq i \leq n-1$ y $0 \leq k \leq 1$), each one with multiplicity 1.
  - β_1, β'_1, β''_1, each one with multiplicity 1.
  - γ_1, γ'_1, γ''_1, γ'''_1, each one with multiplicity 1.
  - y_1 with multiplicity 2^1.
  - $r_{1,1}$ with multiplicity 1.
  - For each $1 \leq i \leq n \wedge 1 \leq j \leq m$, objects $d^2_{i,j,1}$ such that $x_{i,j} \in cod(\varphi)$ and objects $\overline{d}^2_{i,j,1}$ such that $\overline{x}_{i,j} \in cod(\varphi)$.

- There exists 1 membrane labelled with 2 containing objects b_1, b'_1, c_1, c'_1 with multiplicity 1.
- The membrane labelled with 3 contains the objects f'_1 and `no`.

Hence, the formula $\theta_1(0)$ is true.

At configuration $\mathcal{C}_1$, the following rules are applied to the stated membranes:

- In membrane 1, the following rules from $\mathcal{R}_1$ are applied:

$$\left.\begin{array}{l}
(\alpha_{i,1,k}, out; \alpha_{i,2,k}, in) \ : \ 1 \leq i \leq n-1 \wedge 0 \leq k \leq 1 \\
(\alpha'_{i,1,k}, out; \alpha'_{i,2,k}, in) \ : \ 1 \leq i \leq n-1 \wedge 0 \leq k \leq 1 \\
(\beta_1 \, , out; \beta_2 \, , in) \\
(\beta'_1 \, , out; \beta'_2 \, , in) \\
(\beta''_1 \, , out; \beta''_2 \, , in) \\
(\gamma_1 \, , out; \gamma_2 \, , in) \\
(\gamma'_1 \, , out; \gamma'_2 \, , in) \\
(\gamma''_1 \, , out; \gamma''_2 \, , in) \\
(\gamma'''_1 \, , out; \gamma'''_2 \, , in) \\
(\tau_{i,1} \, , out; \tau_{i,2} \, , in) \ : \ 1 \leq i \leq n \\
(\rho_{i,1} \, , out; \rho_{i,2} \, , in) \ : \ 1 \leq i \leq n \\
(\delta_{j,1} \, , out; \delta^2_{j,2} \, , in) \ : \ 1 \leq j \leq m \\
(f_1 \, , out; f_2 \, , in) \\
(y_1 \, , out; z_1 \, w_1 \, , in) \\
(r_{1,1} \, , out; s_{1,1} \, u_{1,1} \, , in) \\
(d_{i,j,1} \, , out; d^2_{i,j,2} \, ; in) \ : \ 1 \leq i \leq n \wedge 0 \leq k \leq 1 \\
(\overline{d}_{i,j,1} \, , out; \overline{d}^2_{i,j,2} \, ; in) \ : \ 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge 0 \leq k \leq 1
\end{array}\right\}$$

- In membrane 2, the following rules from $\mathcal{R}_2$ are applied:

$$\left.\begin{array}{l}
(b_1, out; B_2 \, S, in) \\
(b'_1, out; B'_2, in) \\
(c_1, out; T_{1,1} \, A_2, in) \\
(c'_1, out; F'_{1,1} \, A'_2, in)
\end{array}\right\}$$

- In membrane 3, the following rule from $\mathcal{R}_3$ is applied: $(f'_1, out; f'_2, in)$.

As a result of this, configuration $\mathcal{C}_2$ verifies the following:

- Membrane 1 contains the following objects:
  - $f_1, \mathtt{yes}, f'_0, \widehat{f'_1}, f'_2, \ldots, f'_{3n+2m+1}$, with multiplicity 1.
  - $\rho_{i,1}$ and $\tau_{i,1}$ (for $1 \leq i \leq n$), each one with multiplicity 1.

- $\delta_{j,1}$ (for $1 \leq j \leq m$), each one with multiplicity 1.
- If $0 \leq n-2$, then it also contains the objects
  - $\star$ $\alpha_{i,1,k}$, $\alpha'_{i,1,k}$ (for $1 \leq i \leq n-1$ and $0 \leq k \leq 1$), each one with multiplicity 1.
  - $\star$ β_1, β'_1, β''_1, each one with multiplicity 1.
  - $\star$ γ_1, γ'_1, γ''_1, γ'''_1, each one with multiplicity 2^1.
  - $\star$ w_1 and a_1 with multiplicity 2^1.
- If $0 \leq n-3$, then it also contains:
  - $\star$ object z_1, with multiplicity 2^1, and objects $s_{1,1}$, $u_{1,1}$, each with multiplicity 2^1.

- There exists only one membrane labelled with 2 containing objects B_2, S, B'_2, as well as objects $T_{1,1}, A_2, F'_{1,1}, A'_2$.
- The membrane labelled with 3 contains the objects f'_1 and `no`.

Hence, the formula $\theta_2(0)$ is true.

At configuration $\mathcal{C}_2$, the following rules are applied to the stated membranes:

- In membrane 1 the following rules from $\mathcal{R}_1$ are applied:

$$\left.\begin{array}{l}
(\alpha_{i,2,k}, out; \alpha_{i,3,k}, in) \ : \ 1 \leq i \leq n-1 \wedge 0 \leq k \leq 1 \\
(\alpha'_{i,2,k}, out; \alpha'_{i,3,k}, in) \ : \ 1 \leq i \leq n-1 \wedge 0 \leq k \leq 1 \\
(\beta_2\,, out; \beta_3^2\,, in) \\
(\beta'_2\,, out; \beta_3'^2\,, in) \\
(\beta''_2\,, out; \beta_3''^2\,, in) \\
(\gamma_2\,, out; \gamma_3^2\,, in) \\
(\gamma'_2\,, out; \gamma_3'^2\,, in) \\
(\gamma''_2\,, out; \gamma_3''^2\,, in) \\
(\gamma'''_2\,, out; \gamma_3'''^2\,, in) \\
(\tau_{i,2}\,, out; \tau_{i,3}\,, in) \ : \ 1 \leq i \leq n \\
(\rho_{i,2}\,, out; \rho_{i,3}\,, in) \ : \ 1 \leq i \leq n \\
(\delta_{j,2}\,, out; \delta_{j,3}\,, in) \ : \ 1 \leq j \leq m \\
(f_2\,, out; f_3\,, in) \\
(a_1\,, out; b_2\, b'_2\,, in) \\
(w_1\,, out; c_2\, c'_2\,, in) \\
(u_{1,1}\,, out; q_{1,2}\, q_{2,2}\,, in) \\
(d_{i,j,2}\,, out; d^2_{i,j,3}\,; in) \ : \ 1 \leq i \leq n \wedge 0 \leq k \leq 1 \\
(\overline{d}_{i,j,2}\,, out; \overline{d}^{\,2}_{i,j,3}\,; in) \ : \ 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge 0 \leq k \leq 1
\end{array}\right\}$$

- In membrane 2 the following separation rule from $\mathcal{R}_2$ is applied: $[\ S\ ]_2 \rightarrow [\ \Gamma_0\ ]_2\ [\ \Gamma_1\ ]_2$

- In membrane 3 the following rule from $\mathcal{R}_3$ is applied: $(f'_2, out; f'_3, in)$.

Hence, configuration $\mathcal{C}_3$ verifies the following:

- In membrane 1, we can find the following relevant objects (the non-relevant objects are a_1, a'_1, b_1, b'_1, which cannot trigger any rule of the system at that instant):
  - $\alpha_{i,3,k}$, $\alpha'_{i,3,k}$ (for $1 \leq i \leq n-1$ and $0 \leq k \leq 1$), each one with multiplicity 1.
  - β_3, β'_3, β''_3, each one with multiplicity 2^1.
  - γ_3, γ'_3, γ''_3, γ'''_3, each one with multiplicity 2^1.
  - Objects $\rho_{i,3}$ and $\tau_{i,3}$ (for $1 \leq i \leq n$), each one with multiplicity 1.
  - Objects $\delta_{j,3}$ (for $1 \leq j \leq m$), each one with multiplicity 2^1.
  - Objects $f_3, \texttt{yes}, f'_0, \ldots, \widehat{f'_3}, \ldots, f'_{3n+2m+1}$
  - If $0 \leq n-2$, there also exist objects:
    - $b_2, b'_2, c_2, c'_2, v_2$, each one with multiplicity 2^1.
    - $t_{1,1}$, $f_{1,1}$ and $q_{1,2}$, $q_{2,2}$, each one with multiplicity 1.
- There exist 2 membranes labelled with 2. One of them contains objects A_2, B_2 and T_{11}. The other membrane contains objects A'_2, B'_2 and F'_{11}.
- The membrane labelled with 3 contains objects f'_3 and `no`.

Hence, the formula $\theta_3(0)$ is true and, consequently, the formula $\theta(0)$ is true; that is, the result holds for $p = 0$.

By induction hypothesis, let p be such that $0 \leq p < n-1$, and let us suppose the result holds for p; that is, the formulas $\theta_1(p), \theta_2(p)$ and $\theta_3(p)$ are true. Let us see that the result also holds for $p+1$; that is, the formulas $\theta_1(p+1), \theta_2(p+1)$ and $\theta_3(p+1)$ are also true.

Let us notice that the configuration $\mathcal{C}_{3(p+1)+1}$ is obtained from the configuration $\mathcal{C}_{3(p+1)}$ by applying the following rules:

- In membrane 1, the following rules from $\mathcal{R}_1$ are applied:

$$\left.\begin{array}{l}
(\alpha_{i,3(p+1),k}, out; \alpha_{i,3(p+1)+1,k}\ \Delta^{k}_{i,p+1}, in) \ : \ 1 \leq i \leq p \wedge 0 \leq k \leq 1 \\
(\alpha'_{i,3(p+1),k}, out; \alpha'_{i,3(p+1)+1,k}\ \Delta'^{k}_{i,p+1}, in) \ : \ 1 \leq i \leq p \wedge 0 \leq k \leq 1 \\
(\alpha_{i,3(p+1),k}, out; \alpha_{i,3(p+1)+1,k}, in) \ : \ p+1 \leq i \leq n-1 \wedge 0 \leq k \leq 1 \\
(\alpha'_{i,3(p+1),k}, out; \alpha'_{i,3(p+1)+1,k}, in) \ : \ p+1 \leq i \leq n-1 \wedge 0 \leq k \leq 1 \\
(\beta_{3(p+1)}, out; \beta_{3(p+1)+1}\ B_{p+2}, in) \\
(\beta'_{3(p+1)}, out; \beta'_{3(p+1)+1}\ B'_{p+2}, in) \\
(\beta''_{3(p+1)}, out; \beta''_{3(p+1)+1}\ S, in) \\
(\gamma_{3(p+1)}, out; \gamma_{3(p+1)+1}\ T_{1,1}, in) \\
(\gamma'_{3(p+1)}, out; \gamma'_{3(p+1)+1}\ F'_{1,1}, in) \\
(\gamma''_{3(p+1)}, out; \gamma''_{3(p+1)+1}\ A_2, in) \\
(\gamma'''_{3(p+1)}, out; \gamma'''_{3(p+1)+1}\ A'_2, in) \\
(\tau_{i,3(p+1)}, out; \tau_{i,3(p+1)+1}, in) \ : \ 1 \leq i \leq n \\
(\rho_{i,3(p+1)}, out; \rho_{i,3(p+1)+1}, in) \ : \ 1 \leq i \leq n \\
(\delta_{j,3(p+1)}, out; \delta_{j,3(p+1)+1}, in) \ : \ 1 \leq j \leq m \\
(f_{3(p+1)}, out; f_{3(p+1)+1}, in) \\
(v_{3(p+1)+1}, out; y^{2}_{3(p+1)+1}, in) \\
(q_{1,1}, out; r_{1,1}, in) \ : \ \text{if} \ \ p = 0 \\
(q_{i,j}, out; r^{2}_{i,j}, in) \ : \ \text{if} \ \ p \geq 1 \wedge 1 \leq i \leq j \leq p+1
\end{array}\right\}$$

- In membrane 2, the following rules from $\mathcal{R}_2$ are applied:

$$\left.\begin{array}{l}
(T_{i,p+1}, out; t_{i,p+1}, in) \ : \ 1 \leq i \leq n \\
(T'_{i,p+1}, out; t_{i,p+1}, in) \ : \ 1 \leq i \leq n \\
(F_{i,p+1}, out; f_{i,p+1}, in) \ : \ 1 \leq i \leq n \\
(F'_{i,p+1}, out; f_{i,p+1}, in) \ : \ 1 \leq i \leq n \\
(A_{p+2}, out; c_{p+2}\, c'_{p+2}, in) \\
(A'_{p+2}, out; c_{p+2}\, c'_{p+2}, in) \\
(B_{p+2}, out; b_{p+2}\, b'_{p+2}, in) \\
(B'_{p+2}, out; b_{p+2}\, b'_{p+2}, in)
\end{array}\right\}$$

- In membrane 3, the following rule from $\mathcal{R}_3$ is applied:

$$(f'_{3(p+1)}, out; f'_{3(p+1)+1}, in)$$

Hence, configuration $\mathcal{C}_{3(p+1)+1}$ verifies the following:

- In the membrane labelled with 1 we can find as relevant objects:
  - $\rho_{i,3(p+1)+1}$ and $\tau_{i,3(p+1)+1}$ (for $1 \leq i \leq n$), each one with multiplicity 2^p.
  - $\delta_{j,3(p+1)+1}$ (for $1 \leq j \leq m$), each one with multiplicity 2^{p+1}.
  - $f_{3(p+1)+1}, \mathtt{yes}, f'_0, \ldots, \widehat{f'}_{3(p+1)+1}, \ldots, f'_{3n+2m+1}$, each one with multiplicity 1.
  - A_{p+3}, A'_{p+3}, B_{p+3}, B'_{p+3}, S, each one with multiplicity 2^{p+1}.

- $T_{p+2,p+2}$, $F'_{p+2,p+2}$, each one with multiplicity 2^{p+1}.
- $\beta_{3(p+1)+1}$, $\beta'_{3(p+1)+1}$, $\beta''_{3(p+1)+1}$, each one with multiplicity 2^{p+1}.
- $\gamma_{3(p+1)+1}$, $\gamma'_{3(p+1)+1}$, $\gamma''_{3(p+1)+1}$, $\gamma'''_{3(p+1)+1}$, each one with multiplicity 2^{p+1}.
- y_{p+2} each one multiplicity 2^{p+2}.
- $r_{1,p+2}, r_{2,p+2}, \ldots, r_{p+2,p+2}$, with multiplicity 2^{p+1}.
- A_{p+2}, A'_{p+2}, B_{p+2}, B'_{p+2}, each one with multiplicity 2^p.
- $T_{i,p+2}$, $T'_{i,p+2}$, $F_{i,p+2}$, $F'_{i,p+2}$, for $1 \leq i \leq p+1$, each one with multiplicity 2^p.
- In the case $1 \leq p \leq n-3$, it also contains objects $\alpha_{i,3(p+1)+1,k}$, $\alpha'_{i,3(p+1)+1,k}$ (for $1 \leq i \leq n-1$ and $0 \leq k \leq 1$), each one with multiplicity 2^p.
- If $3(p+1)+1 \geq n$, then it also contains $(cod(\varphi))_e^{2^{n-1}}$.

- There exist 2^{p+1} membranes labelled with 2, each of them containing objects $b_{p+2}, b'_{p+2}, c_{p+2}, c'_{p+2}$ with multiplicity 1. Besides, each one of them contains a $(p+1)$-tuple of objects $(\pi_{1,p+1}, \ldots \pi_{p+1,p+1})$ such that $\pi \in \{t, f\}$ and the tuples are all different in the different membranes.
- The membrane labelled with 3 contains the objects $f'_{3(p+1)+1}$ and `no` with multiplicity 1.

Hence, the formula $\theta_1(p+1)$ is true. To prove that the formula $\theta_2(p+1)$ is true, we notice that configuration $\mathcal{C}_{3(p+1)+2}$ is obtained from configuration $\mathcal{C}_{3(p+1)+1}$ by applying the following rules to the stated membranes:

- In membrane 1, the following rules from $\mathcal{R}_1$ are applied:

$(\alpha_{i,3(p+1)+1,k}, out; \alpha_{i,3(p+1)+2,k}, in)$: $1 \leq i \leq n-1 \wedge 0 \leq k \leq 1$
$(\alpha'_{i,3(p+1)+1,k}, out; \alpha'_{i,3(p+1)+2,k}, in)$: $1 \leq i \leq n-1 \wedge 0 \leq k \leq 1$
$(\beta_{3(p+1)+1}, out; \beta_{3(p+1)+2}, in)$
$(\beta'_{3(p+1)+1}, out; \beta'_{3(p+1)+2}, in)$
$(\beta''_{3(p+1)+1}, out; \beta''_{3(p+1)+2}, in)$
$(\gamma_{3(p+1)+1}, out; \gamma_{3(p+1)+2}, in)$
$(\gamma'_{3(p+1)+1}, out; \gamma'_{3(p+1)+2}, in)$
$(\gamma''_{3(p+1)+1}, out; \gamma''_{3(p+1)+2}, in)$
$(\gamma'''_{3(p+1)+1}, out; \gamma'''_{3(p+1)+2}, in)$
$(\tau_{i,3(p+1)+1}, out; \tau_{i,3(p+1)+2}, in)$: $1 \leq i \leq n$
$(\rho_{i,3(p+1)+1}, out; \rho_{i,3(p+1)+2}, in)$: $1 \leq i \leq n$
$(\delta_{j,3(p+1)+1}, out; \delta^2_{j,3(p+1)+2}, in)$: $1 \leq j \leq m$
$(f_{3(p+1)+1}, out; f_{3(p+1)+2}, in)$
$(y_1, out; z_1\, w_1, in)$
$(r_{1,3(p+1)+1}, out; s_{1,3(p+1)+1}\, u_{1,3(p+1)+1}, in)$: $p+1 \leq n-2$
$(r_{1,3(p+1)+1}, out; s_{1,3(p+1)+1}\, in)$: $p+1 = n-1$
$(d_{i,j,3(p+1)+1}, out; d^2_{i,j,3(p+1)+2}; in)$: $3(p+1)+1 \leq n-1 \wedge 1 \leq i \leq n \wedge 0 \leq k \leq 1$
$(\overline{d}_{i,j,1}, out; \overline{d}^2_{i,j,2}; in)$: $3(p+1)+1 \leq n-1 \wedge 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge 0 \leq k \leq 1$
$(d_{i,j,3(p+1)+1}, out; e_{i,j}; in)$: $3(p+1)+1 = n \wedge 1 \leq i \leq n \wedge 0 \leq k \leq 1$
$(\overline{d}_{i,j,3(p+1)+1}, out; \overline{e}_{i,j}; in)$: $3(p+1)+1 = n \wedge 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge 0 \leq k \leq 1$

- In membrane 2, the following rules from $\mathcal{R}_2$ are applied:

$$\left.\begin{array}{l}(t_{i,p+1}, out; T_{i,p+2}\, T'_{i,p+2}, in)\\ (f_{i,p+1}, out; F_{i,p+2}\, F'_{i,p+2}, in)\\ (b_{p+2}, out; B_{p+3}\, S, in)\\ (b_{p+2}, out; B'_{p+3}, in)\\ (c_{p+2}, out; T_{p+2,p+2}\, A_{p+3}, in)\\ (c'_{p+2}, out; F'_{p+2,p+2}\, A'_{p+3}, in)\end{array}\right\}$$

- In membrane 3, the following rule from $\mathcal{R}_3$ is applied: $(f'_{3(p+1)+1}, out; f'_{3(p+1)+2}, in)$.

Hence, configuration $\mathcal{C}_{3(p+1)+2}$ verifies the following:

- In membrane 1, we can find as relevant objects:
  - Objects $\rho_{i,3(p+1)+1}$ and $\tau_{i,3(p+1)+1}$ (for $1 \leq i \leq n$), each of them with multiplicity 2^{p+1}.
  - Objects $\delta_{j,3(p+1)+1}$ (for $1 \leq j \leq m$), each of them with multiplicity 2^{p+2}.
  - Objects $f_{3(p+1)+1}, \texttt{yes}, f'_0, \ldots, \widehat{f'}_{3(p+1)+1}, \ldots, f'_{3n+2m+1}$
  - If $p+1 \leq n-2$ then it also contains objects
    - $\alpha_{i,3(p+1)+1,k}$, $\alpha'_{i,3(p+1)+1,k}$ (for $1 \leq i \leq n-1$ and $0 \leq k \leq 1$), each of them with multiplicity 2^p.

  - ⋆ $\beta_{3(p+1)+1}$, $\beta'_{3(p+1)+1}$, $\beta''_{3(p+1)+1}$, each of them with multiplicity 2^{p+1}.
  - ⋆ $\gamma_{3(p+1)+1}$, $\gamma'_{3(p+1)+1}$, $\gamma''_{3(p+1)+1}$, $\gamma'''_{3(p+1)+1}$, each of them with multiplicity 2^p.
  - ⋆ w_{p+2} and a_{p+2} with multiplicity 2^{p+1}.
  - ⋆ $s_{1,p+2}$, … , $s_{p+2,p+2}$, each of them with multiplicity 2^{p+1}.
- If $p+1 \leq n-3$ then it also contains objects
  - ⋆ z_{p+2} with multiplicity 2^{p+2} and objects $u_{1,p+2}$, … , $u_{p+2,p+2}$, each with multiplicity 2^{p+1}.
- If, besides, $3(p+1)+1 \geq n$, then it contains $(cod(\varphi))_e^{2^{n-1}}$.

- There exist 2^{p+1} membranes labelled with 2 each of them containing objects B_{p+3}, S, B'_{p+3}, as well as the objects $T_{p+2,p+2}, A_{p+3}, F'_{p+2,p+2}, A'_{p+3}$ all of them with multiplicity 1. Besides, each membrane labelled with 2 contains $2(p+1)$–tuples of objects $(\pi_{1,p+2}, \pi'_{1,p+2}, \ldots, \pi_{p+1,p+2}, \pi'_{p+1,p+2})$ with $\pi \in \{T, F\}$, in such a way that and the tuples are all different in the different membranes.
- The membrane labelled with 3 contains the objects $f'_{3(p+1)+1}$ and `no`.

Hence, the formula $\theta_2(p+1)$ is true. To prove that the formula $\theta_3(p+1)$ is true, we notice that the configuration $\mathcal{C}_{3(p+1)+3}$ is obtained from the configuration $\mathcal{C}_{3(p+1)+2}$ by applying the following rules to the stated membranes:

- In membrane 1, the following rules from $\mathcal{R}_1$ are applied:

$(\alpha_{i,3(p+1)+2,k}, out; \alpha^{2}_{i,3(p+1)+3,k}, in) \ : \ p+1 \leq n-2 \wedge 1 \leq i \leq n-1 \wedge 0 \leq k \leq 1$
$(\alpha'_{i,3(p+1)+2,k}, out; \alpha'^{2}_{i,3(p+1)+3,k}, in) \ : \ p+1 \leq n-2 \wedge 1 \leq i \leq n-1 \wedge 0 \leq k \leq 1$
$(\beta_{3(p+1)+2}, out; \beta^{2}_{3(p+1)+3}, in) \ : \ p+1 \leq n-2$
$(\beta'_{3(p+1)+2}, out; \beta'^{2}_{3(p+1)+3}, in) \ : \ p+1 \leq n-2$
$(\beta''_{3(p+1)+2}, out; \beta''^{2}_{3(p+1)+3}, in) \ : \ p+1 \leq n-2$
$(\gamma_{3(p+1)+2}, out; \gamma^{2}_{3(p+1)+3}, in) \ : \ p+1 \leq n-2$
$(\gamma'_{3(p+1)+2}, out; \gamma'^{2}_{3(p+1)+3}, in) \ : \ p+1 \leq n-2$
$(\gamma''_{3(p+1)+2}, out; \gamma''^{2}_{3(p+1)+3}, in) \ : \ p+1 \leq n-2$
$(\gamma'''_{3(p+1)+2}, out; \gamma'''^{2}_{3(p+1)+3}, in) \ : \ p+1 \leq n-2$
$(\tau_{i,3(p+1)+2}, out; \tau_{i,3(p+1)+3}, in) \ : \ p+1 \leq n-2 \wedge 1 \leq i \leq n$
$(\rho_{i,3(p+1)+2}, out; \rho_{i,3(p+1)+3}, in) \ : \ p+1 \leq n-2 \wedge 1 \leq i \leq n$
$(\delta_{j,3(p+1)+2}, out; \delta_{j,3(p+1)+3}, in) \ : \ p+1 \leq n-2 \wedge 1 \leq j \leq m$
$(\tau_{i,3(p+1)+2}, out; T_i, in) \ : \ p+1 = n-1 \wedge 1 \leq i \leq n$
$(\rho_{i,3(p+1)+2}, out; F_i, in) \ : \ p+1 = n-1 \wedge 1 \leq i \leq n$
$(\delta_{1,3(p+1)+2}, out; E_1, in) \ : \ p+1 = n-1$
$(\delta_{j,3(p+1)+2}, out; \delta_{j,3(p+1)+3}, in) \ : \ p+1 = n-1 \wedge 2 \leq j \leq m$
$(f_{3(p+1)+2}, out; f_{3(p+1)+3}, in)$
$(a_{p+1}, out; b_{p+2}\, b'_{p+2}, in) \ : \ p+1 \leq n-1$
$(w_{p+1}, out; c_{p+2}\, c'_{p+2}, in) \ : \ p+1 \leq n-1$
$(z_{p+1}, out; v_{p+2}, in) \ : \ p+1 \leq n-2$
$(u_{1,p+1}, out; q_{1,p+2}\, q_{2,p+2}, in) \ : \ p+1 \leq n-3$
$(u_{i,p+1}, out; q_{i,p+2}\, q_{2,2}, in) \ : \ p+1 \leq n-3 \wedge 1 \leq i \leq n-1 \wedge 1 \leq i \leq p+1$
$(s_{i,p+1}, out; t_{1,p+2}\, f_{1,p+2}, in) \ : \ p+1 \leq n-2 \wedge 1 \leq i \leq p+1$

- In membrane 2, the following separation rule from $\mathcal{R}_2$ is applied: $[\ S\ ]_2 \rightarrow [\ \Gamma_0\ ]_2\ [\ \Gamma_1\ ]_2$
- In membrane 3, the following rule from $\mathcal{R}_3$ is applied: $(f'_{3(p+1)+2}, out; f'_{3(p+1)+3}, in)$

Hence, the configuration $\mathcal{C}_{3(p+1)+3}$ verifies the following:

- In membrane 1, we can find as relevant objects:
  - If $p+1 \leq n-2$, then there exist objects:
    - $\alpha_{i,3(p+1)+3,k}$, $\alpha'_{i,3(p+1)+3,k}$ (for $1 \leq i \leq n-1$ and $0 \leq k \leq 1$), each of them with multiplicity 2^{p+1}.
    - $\beta_{3(p+1)+3}$, $\beta'_{3(p+1)+3}$, $\beta''_{3(p+1)+3}$, each of them with multiplicity 2^{p+2}.
    - $\gamma_{3(p+1)+3}$, $\gamma'_{3(p+1)+3}$, $\gamma''_{3(p+1)+3}$, $\gamma'''_{3(p+1)+3}$, each of them with multiplicity 2^{p+2}.
    - $\rho_{i,3(p+1)+3}$ and $\tau_{i,3(p+1)+3}$ (for $1 \leq i \leq n$), each of them with multiplicity 2^{p+1}.
    - $\delta_{j,3(p+1)+3}$ (for $1 \leq j \leq m$), each of them with multiplicity 2^{p+2}.

  - ⋆ $f_{3(p+1)+3}, \texttt{yes}, f'_0, \ldots, \widehat{f'}_{3(p+1)+3}, \ldots, f'_{3n+2m+1}$, each of them with multiplicity 1.
  - ⋆ $b_{p+3}, b'_{p+3}, c_{p+3}, c'_{p+3}$, each of them with multiplicity 2^{p+2}.
  - ⋆ $t_{1,p+2}$, $t_{p+2,p+2}$, $f_{1,p+2}$, $f_{p+2,p+2}$ and $q_{1,p+3}$, $q_{p+3,p+3}$, each of them with multiplicity 2^{p+1}.
  - ⋆ w_{p+2} and a_{p+2}, with multiplicity 2^{p+2}.
- If $p+1 \leq n-3$, then it also contains objects:
  - ⋆ v_{p+3}, with multiplicity 2^{p+2}.
  - ⋆ $q_{1,p+3}$, $q_{p+3,p+3}$, each of them with multiplicity 2^{p+1}.
- If $p+1 = n-1$, then it also contains objects:
  - ⋆ $\delta_{j,3(p+1)+3}$ (for $1 \leq j \leq m$), each of them with multiplicity 2^{p+2}.
  - ⋆ $f_{3(p+1)+3}, \texttt{yes}, f'_0, \ldots, \widehat{f'}_{3(p+1)+3}, \ldots, f'_{3n+2m+1}$, each of them with multiplicity 1.
  - ⋆ T_i (for $1 \leq i \leq n$), each of them with multiplicity 2^{p+1}.
  - ⋆ E_1 with multiplicity 2^{p+2}.
- If, besides, $3p+1 \geq n$, then it contains $(cod(\varphi))_e^{2^{n-1}}$.

- There exist 2^{p+2} membranes labelled by 2. 2^{p+1} of them contain objects A_{p+3} and B_{p+3}, as well as $(p+2)$–tuples $(\pi_{1,p+2}, \ldots, \pi_{p+2,p+2})$ with $\pi \in \{T, F\}$, in such a way that $\pi_{p+2,p+2} = T_{p+2,p+2}$ and all tuples are different.

  The other 2^{p+1} membranes labelled by 2 contain the objects A'_{p+3} and B'_{p+3}, as well as $(p+2)$–tuples $(\pi'_{1,p+2}, \ldots, \pi'_{p+2,p+2})$ with $\pi \in \{T, F\}$, in such a way that $\pi'_{p+2,p+2} = F'_{p+2,p+2}$ and all tuples are different.
- The membrane labelled by 3 contains objects $f'_{3(p+1)+3}$ and `no`.

Hence, formula $\theta_3(p+1)$ is true and, consequently, formula $\theta(p+1)$ is true. This completes the proof of the theorem. □

Thus, when completing the aforementioned loop that corresponds to the generation phase, the formula $\theta_3(n-1)$ is true. Consequently, at configuration $\mathcal{C}_{3(n-1)+3} = \mathcal{C}_{3n}$, we have:

- In membrane 1, we can find as relevant objects:
  - $f_{3n}, \texttt{yes}, f'_0, \ldots, \widehat{f'_{3n}}, \ldots, f'_{3n+2m+1}$, each of them with multiplicity 1.
  - $\delta_{j,3n}$ (for $2 \leq j \leq m$), each of them with multiplicity 2^n.

  - E_1, with multiplicity 2^n.
  - T_i, F_i $(1 \leq i \leq n)$, each of them with multiplicity 2^{n-1}.
  - $(cod(\varphi))_e^{2^{n-1}}$.

- There exist 2^n membranes labelled by 2. Half of these membranes contain objects A_{n+1} and B_{n+1}, as well as n–tuples $(\pi_{1,n}, \ldots, \pi_{n,n})$ with $\pi \in \{T, F\}$, in such a way that $\pi_{n,n} = T_{n,n}$ and all tuples are different.

  The other 2^{n-1} membranes labelled by 2 contain the objects A'_{n+1} and B'_{n+1}, as well as n–tuples $(\pi'_{1,n}, \ldots, \pi'_{n,n})$ with $\pi \in \{T, F\}$, in such a way that $\pi'_{n,n} = F'_{n,n}$ and all tuples are different.
- The membrane labelled by 3 contains the objects f'_{3n} and `no`.

The generation phase ends with an additional computation step that allows going from configuration $\mathcal{C}_{3n}$ to configuration $\mathcal{C}_{3n+1}$, whose content is described in the following theorem.

Theorem 3. *At configuration $\mathcal{C}_{3n+1}$ we have:*

- *In the membrane labelled by 1 we can find as relevant objects:*
  - $f_{3n+1}, \texttt{yes}, f'_0, \ldots, \widehat{f'_{3n+1}}, \ldots, f'_{3n+2m+1}$, *each of them with multiplicity 1.*
  - E_j *(for* $2 \leq j \leq m$*), each of them with multiplicity* 2^n.
  - B_{n+1}, B'_{n+1}, *with multiplicity* 2^{n-1}.
  - $(cod(\varphi))_e^{2^{n-1}}$.
- *There exist 2^n membranes labelled by 2 each of them containing objects A_{n+1} and E_1, as well as n–tuples $(\pi_1, \ldots, \pi_n)$ with $\pi \in \{T, F\}$, in such a way that in the different membranes, the corresponding tuples are different with each other.*
- *The membrane labelled by 3 contains the objects f'_{3n+1} and* `no`.

Proof. It is enough to take into account that configuration $\mathcal{C}_{3n+1}$ is obtained from configuration $\mathcal{C}_{3n}$ by applying the following rules to the stated membranes:

- In membrane 1, the following rules from $\mathcal{R}_1$ are applied:

$$\left.\begin{array}{l}(\delta_{j,3n}\,, out; E_j\,, in) \;:\; 2 \leq j \leq m \\ (f_{3n}\,, out; f_{3n+1}\,, in)\end{array}\right\}$$

- In membrane 2, the following rules from $\mathcal{R}_2$ are applied:

$$\left.\begin{array}{l}(B_{n+1}\,, out; E_1\,, in)\\ (B'_{n+1}\,, out; E_1\,, in)\\ (T_{i,n}\,, out; T_i\,, in) \;:\; 1 \leq i \leq n\\ (T'_{i,n}\,, out; T_i\,, in) \;:\; 1 \leq i \leq n\\ (F_{i,n}\,, out; F_i\,, in) \;:\; 1 \leq i \leq n\\ (F'_{i,n}\,, out; F_i\,, in) \;:\; 1 \leq i \leq n\end{array}\right\}$$

- In membrane 3, the following rule from $\mathcal{R}_3$ is applied: $(f'_{3n}, out; f'_{3n+1}, in)$.

□

Let us notice that in configuration $\mathcal{C}_{3n+1}$ each of the 2^n membranes labelled by 2 codifies a truth assignment associated with the variables $\{x_1, \ldots, x_n\}$. Thus, if one of these membranes contains object T_i (respectively, F_i), then the membrane codifies a truth assignment that associates the `true` value (resp., the `false` value) to the variable x_i.

Checking phase

As we explained in the computation overview, the checking phase starts at computation step $3n+2$, and consists of a loop with m iterations, taking 2 steps each. Hence, the checking phase takes $2m$ steps. It is worth pointing out that at step p of this loop, clause C_{p+1} is checked.

In this phase no separation rule is applied at any computation step, so all the following configurations have exactly 2^n membranes labelled by 2.

Let us consider the formula $\nu_1(p)$, for $0 \leq p \leq m-2$, defined as follows:

"*At configuration* $\mathcal{C}_{(3n+1)+2p+1}$ *we have:*

- *In membrane labelled by 1 we can find as relevant objects:*
  - $f_{(3n+1)+2p+1}, \mathtt{yes}, f'_0, \ldots, \widehat{f'}_{(3n+1)+2p+1}, \ldots, f'_{3n+2m+1}$, *each of them with multiplicity 1.*
  - E_j $(p+2 \leq j \leq m)$, *each of them with multiplicity* 2^n.
  - $e_{i,j}$, *such that* $x_{i,j} \in cod(\varphi)$ *and* $j \geq p+2$, *as well as objects* $\overline{e}_{i,j}$, *such that* $\overline{x}_{i,j} \in cod(\varphi)$ *and* $j \geq p+2$. *All these objects appear with multiplicity* 2^{n-1}.
- *Each of the* 2^n *membranes labelled by 2 contains an object* E_0.
- *A membrane labelled by 2 encodes a truth assignment* σ *making true clauses* $C_1, \ldots, C_p, C_{p+1}$ *of* φ, *if and only if it contains a (single) object* $e_{i,p+1}$ *or an object* $\overline{e}_{i,p+1}$, *for a given* i, $1 \leq i \leq n$. *Besides, in that membrane,* σ *keeps all its values,* T *and* F, *except for the* i*-th, which has been replaced by* $e_{i,p+1}$ *(if the object in its place in the previous configuration was* T_i*) or by* $\overline{e}_{i,p+1}$ *(if the object in its place in the previous configuration was* F_i*).*

- *The membrane labelled by 3 contains the objects* $f'_{(3n+1)+2p+1}$ *and* `no`*, both with multiplicity 1."*

Let us consider the formula $\nu_2(p)$, for $0 \leq p \leq m-2$, defined as follows:

"*At configuration* $\mathcal{C}_{(3n+1)+2p+2}$ *we have:*

- *In membrane 1, we can find as relevant objects:*
  - $f_{(3n+1)+2p+2}, \mathtt{yes}, f'_0, \ldots, \widehat{f'}_{(3n+1)+2p+2}, \ldots, f'_{3n+2m+1}$*, each with multiplicity 1.*
  - E_j $(p+3 \leq j \leq m)$*, each with multiplicity* 2^n*.*
  - $e_{i,j}$ *such that* $x_{i,j} \in cod(\varphi)$ *and* $j \geq p+2$*, as well as objects* $\overline{e}_{i,j}$ *such that* $\overline{x}_{i,j} \in cod(\varphi)$ *and* $j \geq p+2$*. All these objects appear with multiplicity* 2^{n-1}*.*
- *Each of the* 2^n *membranes labelled by 2 contains an object* E_0*.*
- *A membrane labelled by 2 encodes a truth assignment* σ *making true clauses* $C_1, \ldots, C_p, C_{p+1}$ *of* φ*, if and only if it contains object* E_{p+2}*. Besides, in that membrane,* σ *keeps all its values,* T *and* F*.*
- *Membrane 3 contains objects* $f'_{(3n+1)+2p+2}$ *and* `no`*, both with multiplicity 1."*

Next, we are going to prove that the formula $\nu(p) \equiv \nu_1(p) \wedge \nu_2(p)$ is an invariant of the loop associated to the checking phase.

Theorem 4. *For each* $p = 0, \ldots, m-2$*, formula* $\nu(p) \equiv \nu_1(p) \wedge \nu_2(p)$ *holds.*

Proof. By bounded induction on p. Let us start analyzing the base case $p = 0$; that is, let us show that the formula $\nu(0)$ is true. For this, we have to prove that the formulas $\nu_1(0)$ and $\nu_2(0)$ are true.

First of all, we notice that configuration $\mathcal{C}_{(3n+1)+1}$ is obtained from configuration $\mathcal{C}_{(3n+1)}$ by applying the following rules to the stated membranes:

- In membrane 1, rule $(f_{3n+1}\,, out; f_{(3n+1)+1}\,, in)$ from $\mathcal{R}_1$ is applied.
- In membranes labelled by 2, the following rules from $\mathcal{R}_2$ can be applied:

$$\left.\begin{array}{l}(E_1\,T_i\,, out; e_{i,1}\,, in)\\ (E_1\,F_i\,, out; \overline{e}_{i,1}\,, in)\\ (A_{n+1}\,, out; E_0\,, in)\\ (A'_{n+1}\,, out; E_0\,, in)\end{array}\right\}$$

  A membrane labelled by 2 encodes a truth assignment σ making true clause C_1 if and only if there exists a literal $l_{i_0,1}$ in clause C_1 that is true by σ. If $l_{i_0,1} = x_k$, then rule $(E_1\,T_k\,, out; e_{k,1}\,, in)$ is applied; if $l_{i_0,1} = \overline{x}_k$, then rule

$(E_1\, F_k\, , out; \overline{e}_{k,1}\, , in)$ is applied. To sum up, either object $e_{i,1}$ or object $\overline{e}_{i,1}$ appears in a membrane 2 if and only if the truth assignment associated to such membrane makes true clause C_1 of φ.

- In membrane 3, rule $(f'_{3n+1}\, , out; f'_{(3n+1)+1}\, , in)$ from $\mathcal{R}_1$ is applied.

Hence, configuration $\mathcal{C}_{(3n+1)+1}$ verifies the following:

- In membrane 1, we can find as relevant objects:
  - $f_{(3n+1)+1}, \mathtt{yes}, f'_0, \ldots, \widehat{f'}_{(3n+1)+1}, \ldots, f'_{3n+2m+1}$, each of them with multiplicity 1.
  - E_j $(2 \leq j \leq m)$, each of them with multiplicity 2^n.
  - $e_{i,j}$, such that $x_{i,j} \in cod(\varphi)$ and $j \geq 1$, as well as objects $\overline{e}_{i,j}$, such that $\overline{x}_{i,j} \in cod(\varphi)$ and $j \geq 1$. All these objects appear with multiplicity 2^{n-1}.
- Each of the 2^n membranes labelled by 2 contains an object E_0.
- A membrane labelled by 2 encodes a truth assignment σ making true clause C_1 of φ if and only if it contains a (single) object $e_{i,1}$ or an object $\overline{e}_{i,1}$, for a given i, $1 \leq i \leq n$. Besides, in that membrane, σ keeps all its values T and F, except for the i-th. There are two possible cases: (a) if the i-th object was T_i in the previous configuration, then it has been released to the skin membrane in this step, and replaced by $e_{i,1}$; and (b) if the i-th object was F_i in the previous configuration, then it has been released to the skin membrane in this step, and replaced by $\overline{e}_{i,1}$.
- Membrane labelled by 3 contains objects $f'_{(3n+1)+1}$ and $\mathtt{no}$, both with multiplicity 1.

Hence, formula $\nu_1(0)$ holds.

Next, let us show that formula $\nu_2(0)$ also holds. For this purpose, let us notice that configuration $\mathcal{C}_{(3n+1)+2}$ is obtained from configuration $\mathcal{C}_{(3n+1)+1}$ by applying the following rules to the stated membranes:

- In membrane 1, the rule $(f_{(3n+1)+1}\, , out; f_{(3n+1)+2}\, , in)$ from $\mathcal{R}_1$ is applied.
- In membrane 2, rules from $\mathcal{R}_2$ of the following kind are applied:

$$\left.\begin{array}{l}(e_{i,1}\, , out; T_i\, E_{1+1}\, , in)\\ (\overline{e}_{i,1}\, , out; F_i\, E_{1+1}\, , in)\end{array}\right\}$$

  Obviously, these rule will be only applied to those membranes labelled by 2 containing an object $e_{i,1}$ or an object $\overline{e}_{i,1}$; that is, to membranes codifying a truth assignment making the first clause true. In this case, by using the previous rule, (a) truth assignment value associated to such membrane is restored; and

(b) object E_2 is incorporated in order for the checking process of the second clause to start. Only membranes labelled by 2 and codifying a truth assignment making the first clause true will carry out this checking.

- In membrane 3, rule $(f'_{(3n+1)+1}\,, out; f'_{(3n+1)+2}\,, in)$ from $\mathcal{R}_1$ is applied

Hence, configuration $\mathcal{C}_{(3n+1)+2}$ verifies the following:

- In membrane 1, we can find as relevant objects:
  - Objects $f_{(3n+1)+2}, \mathtt{yes}, f'_0, \ldots, \widehat{f'}_{(3n+1)+2}, \ldots, f'_{3n+2m+1}$, each of them with multiplicity 1.
  - Objects E_j $(3 \leq j \leq m)$, each of them with multiplicity 2^n.
  - Objects $e_{i,j}$, such that $x_{i,j} \in cod(\varphi)$ and $j \geq 2$, as well as objects $\overline{e}_{i,j}$, such that $\overline{x}_{i,j} \in cod(\varphi)$ and $j \geq 2$. All these objects appear with multiplicity 2^{n-1}.
- Each of the 2^n membranes labelled by 2 contains an object E_0.
- A membrane labelled by 2 encodes a truth assignment σ making true clause C_1 of φ, if and only if it contains an object E_2. Besides, in that membrane, σ keeps all its values, T and F.
- Membrane 3 contains the objects $f'_{(3n+1)+2}$ and $\mathtt{no}$, both with multiplicity 1.

Thus, the formula $\nu_2(0)$ holds and, consequently, also the formula $\nu(0)$ does; that is, the result holds for the base case $p = 1$.

By induction hypothesis, let p be such that $0 \leq p < m-1$ and let us suppose that the result holds for p; that is, formulas $\nu_1(p)$ and $\nu_2(p)$ hold. Let us see that the result also holds for $p+1$; that is, the formulas $\nu_1(p+1)$ and $\nu_2(p+1)$ are also true.

In order to prove that the result holds for $p+1$,let us notice that configuration $\mathcal{C}_{(3n+1)+2(p+1)+1}$ is obtained the configuration $\mathcal{C}_{(3n+1)+2(p+1)} = \mathcal{C}_{(3n+1)+2p+2}$ (let us recall that the content of this configuration is known because we are assuming that formula $\nu_2(p)$ holds) by applying the following rules:

- In membrane 1, the following rule $(f_{(3n+1)+2(p+1)}\,, out; f_{(3n+1)+2(p+1)+1}\,, in)$ from $\mathcal{R}_1$ is applied.
- In membrane 2, the following rules from $\mathcal{R}_2$ are applied:

$$\left.\begin{array}{l}(E_{p+2}\,T_i\,, out; e_{i,p+2}\,, in) \;:\; 1 \leq i \leq n \\ (E_{p+2}\,F_i\,, out; \overline{e}_{i,p+2}\,, in) \;:\; 1 \leq i \leq n\end{array}\right\}$$

By using these rules, if a membrane 2 codifies a truth assignment σ making true clauses $C_1, \ldots, C_p, C_{p+1}$, then that membrane contains an object E_{p+2} at

configuration $\mathcal{C}_{(3n+1)+2(p+1)}$. Thus, if there exists a literal l_{i_0} of clause C_{p+2} that is true by the truth assignment σ, then the following holds: if $l_{i_0} = x_k$, then the rule $(E_{p+2}\, T_k\, , out; e_{k,p+2}\, , in)$ would be applicable, and if $l_{i_0} = \overline{x}_k$, then the rule $(E_{p+2}\, F_k\, , out; \overline{e}_{k,p+2}\, , in)$ would be applicable.

- In membrane 3, the rule $(f'_{(3n+1)+2(p+1)}\, , out; f'_{(3n+1)+2(p+1)+1}\, , in)$ from $\mathcal{R}_3$ is applied.

As a result of this, at configuration $\mathcal{C}_{(3n+1)+2(p+1)+1}$ we have:

- In membrane 1, we can find as relevant objects:
  - Objects $f_{(3n+1)+2(p+1)+1}, \texttt{yes}, f'_0, \ldots, \widehat{f'}_{(3n+1)+2(p+1)+1}, \ldots, f'_{3n+2m+1}$, each of them with multiplicity 1.
  - Objects E_j $(p+3 \leq j \leq m)$, each of them with multiplicity 2^n.
  - Objects $e_{i,j}$, such that $x_{i,j} \in cod(\varphi)$ and $j \geq p+3$, as well as objects $\overline{e}_{i,j}$, such that $\overline{x}_{i,j} \in cod(\varphi)$ and $j \geq p+3$. All these objects appear with multiplicity 2^{n-1}.
- Each of the 2^n membranes labelled by 2 contains an object E_0.
- A membrane labelled by 2 encodes a truth assignment σ making true clauses $C_1, \ldots, C_p, C_{p+1}, C_{p+2}$ of φ if and only if it contains a (single) object $e_{i,p+2}$ or an object $\overline{e}_{i,p+2}$, for a given i, $1 \leq i \leq n$. Besides, in that membrane, σ keeps all its values, T and F, excepting the i-th. There are two possible cases: (a) if the i-th object was T_i in the previous configuration, then it has been released to the skin membrane in this step, and replaced by $e_{i,p+2}$; and (b) if the i-th object was F_i in the previous configuration, then it has been released to the skin membrane in this step, and replaced by $\overline{e}_{i,p+2}$.
- The membrane labelled by 3 contains objects $f'_{(3n+1)+2(p+1)+1}$ and $\texttt{no}$ with multiplicity 1.

Hence, the formula $\nu_1(p+1)$ holds.

Next, let us show that the formula $\nu_2(p+1)$ is also true. For this purpose, let us notice that the configuration $\mathcal{C}_{(3n+1)+2(p+1)+2}$ is obtained from the configuration $\mathcal{C}_{(3n+1)+2(p+1)+1}$ by applying the following rules to the stated membranes:

- In membrane 1 the rule $(f_{3n+2(p+1)+1}\, , out; f_{3n+2(p+1)+2}\, , in)$ from $\mathcal{R}_1$ is applied.
- In membrane 2, the following rules $\mathcal{R}_2$ are applied:

$$\left.\begin{array}{l}(e_{i,p+2}\, , out; T_i\, E_{p+3}\, , in)\\ (\overline{e}_{i,p+2}\, , out; F_i\, E_{p+3}\, , in)\end{array}\right\}$$

In configuration $\mathcal{C}_{(3n+1)+2(p+1)+1}$, the truth assignment σ encoded by a membrane 2 makes clauses $C_1, \ldots, C_p, C_{p+1}, C_{p+2}$ true if and only if that membrane contains an object $e_{i,p+2}$ or an object $\overline{e}_{i,p+2}$. In this case, a rule of the type $(e_{i,p+2}\,, out; T_i\, E_{p+3}\,, in)$ or of the type $(\overline{e}_{i,p+2}\,, out; F_i\, E_{p+3}\,, in)$. can be applied. Besides, if neither object $e_{i,j}$ nor $\overline{e}_{i,j}$ appears in a membrane 2 of $\mathcal{C}_{(3n+1)+2(p+1)+1}$, then that membrane will not evolve any more.

- In membrane 3, rule $(f'_{3n+2(p+1)+1}\,, out; f'_{3n+2(p+1)+2}\,, in)$ from $\mathcal{R}_3$ is applied.

Hence, configuration $\mathcal{C}_{(3n+1)+2(p+1)+2}$ verifies the following:

- In membrane 1, we can find as relevant objects:
  - Objects $f_{(3n+1)+2(p+1)+2}, \texttt{yes}, f'_0, \ldots, \widehat{f'}_{(3n+1)+2(p+1)+2}, \ldots, f'_{3n+2m+1}$, each of them with multiplicity 1.
  - Objects E_j $(p+4 \leq j \leq m)$, each of them with multiplicity 2^n.
  - Objects $e_{i,j}$, such that $x_{i,j} \in cod(\varphi)$ and $j \geq p+3$, as well as objects $\overline{e}_{i,j}$ such that, $\overline{x}_{i,j} \in cod(\varphi)$ and $j \geq p+3$. All these objects appear with multiplicity 2^{n-1}."
- Each of the 2^n membranes labelled by 2 contains an object E_0.
- A membrane labelled by 2 encodes a truth assignment σ making true clauses $C_1, \ldots, C_p, C_{p+1}, C_{p+2}$ of φ, if and only if it contains an object E_{p+3}. Besides, in that membrane, σ keeps all its values, T and F.
- Membrane 3 contains objects $f'_{(3n+1)+2(p+1)+2}$ and `no` with multiplicity 1.

Thus, the formula $\nu_2(p+1)$ holds, consequently, it also formula $\nu(p+1)$ does; that is, the result holds for $p+1$. This completes the proof of the theorem. □

From the Theorem 4 we deduce that the formula $\nu(m-2)$ holds, and in particular, also formula $\nu_2(m-2)$ does. That is, at configuration $\mathcal{C}_{(3n+1)+2(m-2)+2} = \mathcal{C}_{(3n+1)+2(m-1)}$ we have the following:

- In membrane, we can find as relevant objects:
  - $f_{(3n+1)+2m}, \texttt{yes}, f'_0, \ldots, \widehat{f'}_{(3n+1)+2m}, \ldots, f'_{3n+2m+1}$, each of them with multiplicity 1.
  - $e_{i,m}$ such that $x_{i,m} \in cod(\varphi)$ and $\overline{e}_{i,m}$ such that $\overline{x}_{i,m} \in cod(\varphi)$.
- Each of the 2^n membranes labelled by 2 contains object E_0.
- A membrane labelled by 2 encodes a truth assignment σ making true clauses $C_1, \ldots, C_{m-2}, C_{m-1}$ of φ, if and only if it contains the object $E_{(m-2)+2} = E_m$. Besides, in that membrane σ, keeps all its values, T and F.

- The membrane labelled by 3 contains the objects $f'_{(3n+1)+2m}$ and no with multiplicity 1.

Then, configuration $\mathcal{C}_{(3n+1)+2(m-1)+1}$ is obtained from $\mathcal{C}_{(3n+1)+2(m-1)}$ by applying the following rules to the stated membranes:

- In membrane 1 rule $(f_{(3n+1)+2(m-1)}\,, out; f_{(3n+1)+2(m-1)+1}\,, in)$ from $\mathcal{R}_1$ is applied.
- In membranes 2, the following rules from $\mathcal{R}_2$ can be applied:

$$\left.\begin{array}{l}(E_m\, T_i\,, out;\, e_{i,m}\,, in)\\ (E_m\, F_i\,, out;\, \overline{e}_{i,m}\,, in)\end{array}\right\}$$

- In membrane 3, rule $(f'_{(3n+1)+2(m-1)}\,, out; f'_{(3n+1)+2(m-1)+1}\,, in)$ from $\mathcal{R}_3$ is applied.

Hence, at configuration $\mathcal{C}_{(3n+1)+2(m-1)+1}$ we have the following:

- In the membrane, 1 we can find as relevant objects:
  - $f_{(3n+1)+2(m-1)+1} = f_{3n+2m}, \mathtt{yes}, f'_0, \ldots, \widehat{f'}_{3n+2m}, f'_{3n+2m+1}$, each of them with multiplicity 1.
- Each of the 2^n membranes labelled by 2 contains an object E_0.
- A membrane labelled by 2 encodes a truth assignment σ making true the clauses $C_1, \ldots, C_{m-1}, C_m$ of φ if and only if it contains an object $e_{i,m}$ or an object $\overline{e}_{i,m}$; that is, the input formula φ is satisfiable if and only there exists a membrane labelled by 2 that contains an object $e_{i,m}$ or an object $\overline{e}_{i,m}$.
- The membrane labelled by 3 contains the objects $f'_{3n+2m}, \mathtt{no}$, each of them with multiplicity 1.

Then, configuration $\mathcal{C}_{(3n+1)+2(m-1)+2}$ is obtained from $\mathcal{C}_{(3n+1)+2(m-1)+1}$ by applying the following rules to the stated membranes:

- In a membrane 2, a rule of the type $(e_{i,m})\, E_0\,, out)$ or of the type $(\overline{e}_{i,m})\, E_0\,, out)$ will be applied if an only if the truth assignment σ encoded by that membrane makes the formula φ true.
- In the membranes 3, rule $(f'_{3n+2m}\,, out;\ f'_{3n+2m+1}\,, in)$ from $\mathcal{R}_3$ will be applied.

Consequently, at configuration $\mathcal{C}_{(3n+1)+2(m-1)+1}$ we have the following:

- Membrane 1 contains an object E_0 if and only if the input formula φ is satisfiable.
- Membrane 3 contains objects $f'_{3n+2m+1}, \mathtt{no}$, each of them with multiplicity 1.

Then, the checking phase has finished.

Output phase

Case 1: **Affirmative output**.

Let us assume that input formula φ is satisfiable. In this case, at configuration $\mathcal{C}_{3n+2m+1}$, the skin membrane contains some object E_0 and object f_{3n+2m}, while membrane 3 contains objects $f'_{3n+2m+1}$ and `no`.

Hence, in the next computation step (leading to configuration $\mathcal{C}_{3n+2m+2}$), rule $(E_0\, f_{3n+2m}\, \texttt{yes}\,;out) \in \mathcal{R}_1$ will be applied sending object `yes` to the environment, and providing an affirmative answer. At the same time rule $(f'_{3n+2m+1}\, \texttt{no}\,;out) \in \mathcal{R}_3$ will be applied sending to the skin object `no`. In this case, configuration $\mathcal{C}_{3n+2m+2}$ is halting since object f_{3n+2m} has been sent to the environment and, consequently rule $(f_{3n+2m}\, \texttt{no}\,;out) \in \mathcal{R}_1$ cannot be applied.

To sum up, the affirmative answer is provided in the computation step $(3n + 1) + 2m + 1 = 3n + 2m + 2$.

Case 2: **Negative output**.

If the input formula φ is not satisfiable, then in the skin membrane of configuration $\mathcal{C}_{3n+2m+1}$ objects f_{3n+2m} and `yes` will appear, but not the object E_0. In this case, rule $(E_0\, f_{3n+2m}\, \texttt{yes}\,;out) \in \mathcal{R}_1$ will not be applicable to the configuration $\mathcal{C}_{3n+2m+1}$ and, consequently, the only applicable rule to this configuration being $(f'_{3n+2m+1}\, \texttt{no}\,;out) \in \mathcal{R}_3$. Therefore, in the skin membrane of configuration $\mathcal{C}_{3n+2m+2}$ objects f_{3n+2m}, `yes`, $f'_{3n+2m+1}$ and `no` appear, but not object E_0. In this case, the rule $(E_0\, f_{3n+2m}\, \texttt{yes}\,;out) \in \mathcal{R}_1$ will not be applicable, being rule $(f_{3n+2m}\, \texttt{no}\,;out) \in \mathcal{R}_1$ the only applicable to the system. Execution of this rule will send object `no` to the environment, providing a negative answer at computation step $3n + 2m + 3$.

Hence, the output phase takes 1 step in the case of an affirmative answer, and 2 steps in the case of a negative answer.

Corollary 1. $\texttt{SAT} \in \mathbf{PMC}_{\mathbf{CSC(3)}}$.

7 P-lingua simulator as a checker of the solution

The formal verification of a solution given in the framework of a computing model is a necessary, and usually very complex to implement, task. In order to assist researchers in designing P system families to efficiently solve hard problems and verifying them, simulation tools are indispensable.

The solution to `SAT` problem by means of a family from $\mathbf{CSC}(\mathbf{3})$ presented in Section 4 has been extraordinarily complex. It is structured into several modules, each of them performing a specific task. Modules have been designed and checked separately and subsequently incorporated into the general solution. The different

modules have been checked (in several relevant instances) with the help of the P-Lingua simulator for the model **CSC** developed in [4]. Regarding the formal verification, the simulator was used to check that the identified invariants were corroborated in the corresponding configurations.

The `P-Lingua` source code that defines a cell-like P system belonging to the family specified above and the corresponding `MeCoSim` custom application source files can be found at [16].

7.1 Results of simulation

We have simulated several P systems of the defined family solving relevant instances to `SAT` problem. Simulation results are shown in Table 1.

Table 1. Formula satisfiability and simulation time

Formula	**n**	**m**	**SAT**	**Time (s)**
$(\bar{x}_1 + \bar{x}_2) \cdot x_1 \cdot x_2$	2	3	F	0,233
$(\bar{x}_1 + \bar{x}_2) \cdot x_2 \cdot (\bar{x}_1 + x_2)$	2	3	T	0,224
$(x_1 + x_2) \cdot (x_1 + x_2 + \bar{x}_3) \cdot \bar{x}_1 \cdot \bar{x}_2$	3	4	F	0,491
$(\bar{x}_1 + x_2) \cdot \bar{x}_1 \cdot x_3 \cdot (\bar{x}_1 + x_3)$	3	4	T	0,487
$(x_1 + x_4) \cdot (x_1 + \bar{x}_4) \cdot x_3 \cdot (x_2 + \bar{x}_3 + x_4) \cdot \bar{x}_1$	4	5	F	0,827
$(x_3+\bar{x}_4)\cdot(\bar{x}_1+x_2+\bar{x}_3+x_4)\cdot(x_1+x_2)\cdot(\bar{x}_1+x_2+x_3+x_4)\cdot(\bar{x}_1+x_3)$	4	5	T	0,981
$(x_1+\bar{x}_2+x_3+x_5)\cdot(\bar{x}_1+x_4)\cdot(\bar{x}_2+\bar{x}_4)\cdot x_4\cdot x_2\cdot(\bar{x}_1+x_2+\bar{x}_3+x_4)$	5	6	F	2,369
$(x_3 + x_4) \cdot (x_4 + \bar{x}_5) \cdot (\bar{x}_1 + x_2 + \bar{x}_3 + \bar{x}_4) \cdot (x_1 + \bar{x}_2 + x_4) \cdot (x_1 + \bar{x}_3 + x_4) \cdot (x_3 + x_5)$	5	6	T	2,312
$(x_3 + x_5 + x_6) \cdot (x_3 + \bar{x}_4 + x_5 + \bar{x}_6) \cdot \bar{x}_3 \cdot \bar{x}_6 \cdot (x_1 + \bar{x}_2 + \bar{x}_3 + x_5 + x_6) \cdot (x_1 + x_4 + x_5) \cdot (\bar{x}_5 + x_6)$	6	7	F	4,877
$(\bar{x}_1 + \bar{x}_2 + x_5) \cdot (x_2 + x_3) \cdot (x_3 + \bar{x}_5 + \bar{x}_6) \cdot (\bar{x}_1 + x_2 + \bar{x}_3 + x_4 + x_5 + x_6) \cdot (\bar{x}_2 + \bar{x}_3) \cdot (x_2 + x_3 + x_6) \cdot (x_1 + \bar{x}_2 + x_3 + x_4 + x_5 + x_6)$	6	7	T	4,195
$(\bar{x}_5+\bar{x}_6+\bar{x}_7)\cdot(x_3+\bar{x}_4+x_7)\cdot(\bar{x}_1+x_3+x_5+x_6+\bar{x}_7)\cdot(x_1+x_3+\bar{x}_5+x_6+x_7)\cdot(x_2+x_6)\cdot(x_2+\bar{x}_6)\cdot\bar{x}_2\cdot(x_2+x_3+x_4+\bar{x}_5+x_7)$	7	8	F	10,320
$(\bar{x}_2 + x_5 + x_6 + x_7) \cdot (x_2 + \bar{x}_4 + \bar{x}_5 + \bar{x}_7) \cdot (x_1 + x_2 + \bar{x}_3 + \bar{x}_6 + x_7) \cdot (x_1 + x_2 + x_3 + \bar{x}_5 + x_6 + \bar{x}_7) \cdot (\bar{x}_3 + \bar{x}_5 + x_6 + \bar{x}_7) \cdot (x_1 + x_2 + \bar{x}_3 + \bar{x}_7) \cdot (\bar{x}_1 + x_2 + \bar{x}_4 + \bar{x}_6) \cdot (x_3 + x_5 + x_6 + \bar{x}_7)$	7	8	T	8,862
$(x_3+x_4+\bar{x}_6+\bar{x}_8)\cdot(x_6+\bar{x}_7)\cdot(\bar{x}_2+x_3+\bar{x}_4+x_5+x_8)\cdot x_7\cdot(x_1+\bar{x}_2+x_5+\bar{x}_7+\bar{x}_8)\cdot(x_2+x_7+x_8)\cdot(\bar{x}_6+\bar{x}_7)\cdot(x_1+x_5+\bar{x}_8)\cdot(x_1+\bar{x}_4+x_5+\bar{x}_6+x_7)$	8	9	F	16,364
$(x_1+\bar{x}_5+\bar{x}_6+\bar{x}_7+\bar{x}_8)\cdot(x_2+x_3+x_4+\bar{x}_6+\bar{x}_7+x_8)\cdot(x_3+x_4+\bar{x}_5+\bar{x}_6+\bar{x}_7+\bar{x}_8)\cdot(\bar{x}_1+\bar{x}_3+\bar{x}_4+\bar{x}_5+x_6+\bar{x}_7+x_8)\cdot(\bar{x}_3+\bar{x}_7)\cdot(x_4+x_5+\bar{x}_7)\cdot(x_1+x_3+\bar{x}_4)\cdot(x_1+\bar{x}_2+\bar{x}_3+\bar{x}_4+\bar{x}_5+\bar{x}_6+\bar{x}_7)\cdot(x_4+\bar{x}_5+\bar{x}_6+x_7+\bar{x}_8)$	8	9	T	18,856

$(\bar{x}_2 + \bar{x}_3 + x_5 + x_7) \cdot (x_2 + x_5 + x_6 + x_7 + x_9) \cdot (\bar{x}_3 + x_5 + x_7 + x_8) \cdot (x_1 + \bar{x}_4 + \bar{x}_5 + x_6 + x_8) \cdot (\bar{x}_2 + x_3 + x_5 + x_7 + x_8 + \bar{x}_9) \cdot (\bar{x}_2 + \bar{x}_4 + x_7 + x_9) \cdot (\bar{x}_2 + x_4 + x_6 + x_9) \cdot x_1 \cdot x_5 \cdot (\bar{x}_1 + \bar{x}_5)$	9	10	F	34,669
$(x_3 + x_8) \cdot (x_1 + \bar{x}_2 + x_5 + \bar{x}_6 + x_9) \cdot (x_3 + x_6 + x_9) \cdot (x_3 + x_5 + \bar{x}_6 + \bar{x}_8) \cdot (x_1 + x_2 + \bar{x}_5 + x_7 + \bar{x}_8 + \bar{x}_9) \cdot (\bar{x}_1 + x_2 + \bar{x}_4 + x_5 + \bar{x}_6 + \bar{x}_7 + x_9) \cdot (x_1 + x_2 + x_4 + \bar{x}_6 + x_8 + \bar{x}_9) \cdot (\bar{x}_1 + x_2 + \bar{x}_3 + \bar{x}_4 + x_7 + \bar{x}_8) \cdot (\bar{x}_1 + x_2 + x_3 + x_5 + \bar{x}_6 + x_8 + \bar{x}_9) \cdot (x_2 + \bar{x}_3 + x_4 + \bar{x}_6 + \bar{x}_7 + \bar{x}_9)$	9	10	T	36,450

Let us recall that in P systems of **CSC**(3), there is no replication of objects, but a distribution of them. Consequently, in order to generate an exponential amount of some objects, it is necessary to use the skin membrane, interacting with the environment by using antiport rules with length 3 (in a computation step, an object is released into the environment and, simultaneously, two objects enter the system).

8 Conclusions

In this paper we have studied the computational efficiency of cell-like P systems with symport/antiport rules and membrane separation. A uniform polynomial time solution to `SAT` problem by a family of such P systems which uses communication rules involving at most three objects is given, and the formal verification is shown.

Bearing in mind that $\mathbf{PMC}_{\mathbf{CSC(2)}} = \mathbf{P}$(that is, only tractable problems are efficiently solved by families of P systems with symport/antiport rules and membrane separation which uses communication rules with length at most two) an optimal frontier of the efficiency has been obtained with respect to the length of such rules. Specifically, we have shown that, in the framework of P systems with symport/antiport rules and membrane separation, passing from 2 to 3 amounts to passing from non–efficiency to efficiency, assuming that $\mathbf{P} \neq \mathbf{NP}$.

Acknowledgements

The work of L. Valencia-Cabrera, L.F. Macías-Ramos, A. Riscos-Núñez and M.J. Pérez-Jiménez was supported by Project TIN2012-37434 of the Ministerio de Economía y Competitividad of Spain. The work of B. Song and L. Pan was supported by National Natural Science Foundation of China (61033003, 91130034 and 61320106005).

References

1. A. Alhazov, T.O. Ishdorj. Membrane operations in P systems with active membranes. In Gh.Păun, A. Riscos-Núñez, A. Romero-Jiménez, F. Sancho-Caparrini (eds.) *Proceedings of the Second Brainstorming Week on Membrane Computing*, Sevilla, 2-7

February 2004, Research Group on Natural Computing, TR 01/2004, University of Seville, 37-44.
2. T.H. Cormen, C.E. Leiserson, R.L. Rivest. *An Introduction to Algorithms*. The MIT Press, Cambridge, Massachussets, 1994.
3. M.R. Garey, D.S. Johnson. *Computers and Intractability A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, (1979).
4. L.F. Macías-Ramos, L. Valencia-Cabrera, B. Song, T. Song, L. Pan, M.J. Pérez-Jiménez. P-Lingua based software for cell-like P systems with symport/antiport rules. *Fundamenta Informaticae*, 2015, in press.
5. L.F. Macías-Ramos, L. Valencia-Cabrera, B. Song, T. Song, L. Pan, M.J. Pérez-Jiménez. Membrane Fission: A Computational Complexity Perspective. *Complexity*, 2015, in press.
6. S. Morlot, A. Roux. Mechanics of dynamic-mediated membrane fission. *Annhual Review of Biophysics*, **42**, (2013), 629–649.
7. L. Pan, T.-O. Ishdorj. P systems with active membranes and separation rules. *Journal of Universal Computer Science*, **10**, 5 (2004), 630–649.
8. L. Pan, M.J. Pérez-Jiménez. Computational complexity of tissue–like P systems. *Journal of Complexity*, **26**, 3 (2010), 296–315.
9. A. Păun, Gh. Păun, G. Rozenberg. Computing by communication in networks of membranes, *International Journal of Foundations of Computer Science*, **13**, 6 (2002), 779–798
10. A. Păun, Gh. Păun. The power of communication: P systems with symport/antiport, *New Generation Computing*, **20**, 3 (2002), 295–305.
11. Gh. Păun, G. Rozenberg, A. Salomaa (eds.). *The Oxford Handbook of Membrane Computing*, Oxford University Press, Oxford, 2010.
12. Gh. Păun. Attacking **NP**-complete problems. In *Unconventional Models of Computation, UMC'2K* (I. Antoniou, C. Calude, M. J. Dinneen, eds.), Springer-Verlag, 2000, 94-115.
13. M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini, F. Complexity classes in models of cellular computing with membranes. *Natural Computing*, **2**, 3 (2003), 265–285.
14. M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini, A polynomial complexity class in P systems using membrane division, *Journal of Automata, Languages and Combinatorics*, **11**, 4 (2006) 423–434.
15. M.J. Pérez-Jiménez, P. Sosík. An optimal frontier of the efficiency of tissue P systems with cell separation. *Fundamenta Informaticae*, in press, 2015.
16. The MeCoSim Web Site: `http://www.p-lingua.org/mecosim/`.

Author Index