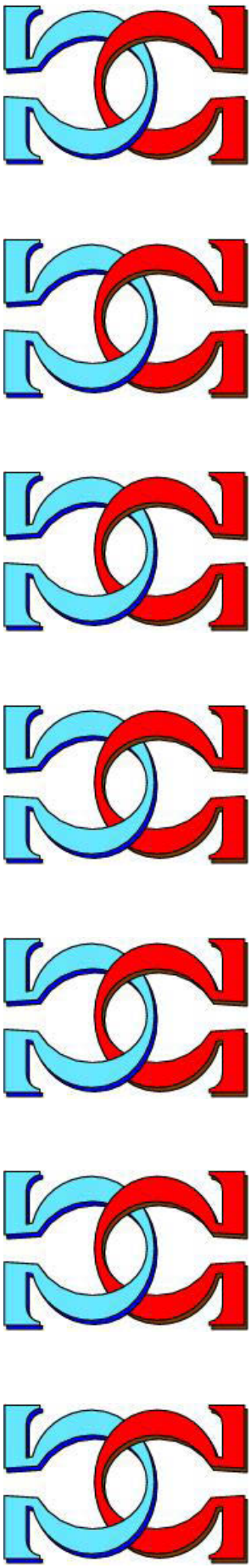


CDMTCS Research Report Series

Proceedings of the Workshop on Membrane Computing 2015 (WMC2015)

Michael J. Dinneen (Editor)
Department of Computer Science
University of Auckland
Auckland, New Zealand

CDMTCS-487 (Satellite workshop of UCNC2015)
August 2015

Centre for Discrete Mathematics and
Theoretical Computer Science

Small Catalytic P Systems

Artiom Alhazov[1] and 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, TU Wien
Favoritenstraße 9-11, 1040 Vienna, Austria
E-mail: rudi@emcc.at

Abstract. We present a new variant of how catalytic P systems can simulate register machines, thus reducing again the number of rules needed for simulating register machines. Moreover, we show that only 20 rules are needed to generate a non-semilinear set of natural numbers by a catalytic P system with two catalysts. Finally, we establish improved versions of universal catalytic P systems.

1 Introduction

Membrane systems were introduced by Gheorghe Păun in [10] and therefore called *P systems* since then. P systems are motivated by the biological functioning of molecules in cells. From a mathematical point of view a P system can be viewed as a parallel multiset rewriting system. When using non-cooperative rules without any additional control, it has the behavior of an $E0L$ system; yet when only taking the results when the system halts means that the objects evolve in a context-free manner, generating a set in $PsCF$, which is known (by Parikh's theorem) to coincide with $PsREG$, i.e., with the family of semilinear sets.

We work out a slightly refined method to simulate register machines with m decrementable registers by P systems with m catalysts, thereby improving not only the result established in [14], but even the improved version based on observations just found recently which allowed for reducing the number of rules again in a considerable way, as done by Petr Sosík in 2015, see [16].

We also recall the concept of *toxic objects* which allows us to "kill" a computation branch if we cannot find a multiset of rules covering all occurrences of toxic objects which then somehow become "lethal" by killing such a computation. For all the proof techniques using a trap symbol # to "kill" a computation by introducing the trap symbol # with a non-cooperative rule $a \to \#$, the concept of toxic objects allows us to save most of the trap rules or even all of them, thus improving the descriptional complexity of the underlying P systems.

The rest of the paper is organized as follows: We first recall the basic definitions from formal language theory as well as the definitions for (purely) catalytic

P systems. Then we improve some general results for catalytic P systems with respect to the number of rules needed for simulating register machines and give an example of a catalytic P system with two catalysts generating a non-semilinear set of natural numbers with only 20 rules, thus improving previous results established in [14] and just recently obtained by Petr Sosík in 2015, see [16]. Finally, we apply the new construction for simulating register machines by catalytic P systems to the universal register machine URM_{22} of Korec (see [8]).

2 Definitions

In this section we first recall the basic notions from formal language theory needed in this paper and then the definitions of the basic variants of P systems considered in the following sections. For more details in formal language theory we refer the reader to the standard monographs and textbooks as [13] and for the area of regulated rewriting to [4]. All the main definitions and results for P systems can be found in [11] and [12]; the model of P systems with toxic objects was introduced in [2]. For actual informations and new developments in the area of membrane computing we refer to the P systems webpage [17].

2.1 Prerequisites

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

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

The family of regular, context-free, and recursively enumerable string languages is denoted by REG, CF, and RE, respectively.

2.2 Register machines

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

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

A *configuration* of a register machine is described by the contents of each register and by the value of the current label, which indicates the next instruction to be executed. Computations start by executing the first instruction of P (labeled with l_0), and terminate with reaching the $HALT$-instruction.

Register machines provide a simple universal computational model, for example, see [9]. In the following, we shall call a specific model of P systems *computationally complete* or *universal* if and only if for any register machine M we can effectively construct an equivalent P system Π of that type simulating M and yielding the same results.

Non-semilinear sets of numbers and vectors of numbers In most of the examples established in the literature, variants of the set of natural numbers

$$\{2^n \mid n \geq 0\} = N\left(\left\{a^{2^n} \mid n \geq 0\right\}\right)$$

are considered as the typical non-semilinear sets of natural numbers.

2.3 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 $v = (b_1, tar_1) \dots (b_k, tar_k)$ with $b_i \in O$ and $tar_i \in \{here, out, in\}$ or $tar_i \in \{here, out\} \cup \{in_j \mid j \in Lab\}$, $1 \leq i \leq k$. Using such a rule means "consuming" the objects of u and "producing" the objects $b_1, \dots, b_k$ of v; the *target indications here*, *out*, and *in* mean that an object with the target *here* remains in the same region where the rule is applied, an object with the target *out* is sent out of the respective membrane (in this way, objects can also be sent to the environment, when the rule is applied in the skin region), while an object with the target *in* is sent to one of the immediately inner membranes, non-deterministically chosen, whereas with in_j this inner membrane can be specified directly. In general, we may omit the target indication *here*.

Due to the possibility of flattening, see [7], in the following we will mostly restrict ourselves to P systems with only one membrane.

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

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

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

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

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 $R_1, \dots, R_m$ 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 $1, \dots, m$.

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 region f 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 $Ps_{gen,\delta}(\Pi)$.

The families of sets $Y_{gen,\delta}(\Pi)$, $Y \in \{N, Ps\}$, computed by P systems with at most m membranes working in the derivation mode δ and with rules of type X are denoted by $Y_{gen,\delta}OP_m(X)$.

It is well known (for example, see [10]) 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 any of the families of (vectors of) natural numbers $Y_{gen,\delta}OP_m(X)$ we will add subscript l at the end to indicate that only systems with at most l rules are considered, i.e., we write $Y_{gen,\delta}OP_m(X)_l$. If any of the finite parameters like m and l is unbounded, we replace it by $*$ or even omit it.

2.4 P Systems with Catalysts

P systems with catalysts were already considered in the originating papers for membrane systems, see [10]. In [5], two catalysts (three catalysts) were shown to be sufficient for getting computational completeness with catalytic (purely catalytic) P systems. Whether or not one catalyst (respectively two catalysts) might already be enough to obtain computational completeness, is still one of the most challenging open problems in the area of P systems. We only know that purely catalytic P systems (working in the maximally parallel mode) with only one catalyst simply correspond with sequential P systems with only one membrane, hence, to multiset rewriting systems with context-free rules, and therefore can only generate linear sets.

Using additional control mechanisms as, for example, priorities or promoters/inhibitors, P systems with only one catalyst can be shown to be computationally complete, e.g., see Chapter 4 of [12]. On the other hand, additional features for the catalyst may be taken into account; for example, we may use bi-stable catalysts (catalysts switching between two different states).

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

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

The task of generating a non-semilinear set by catalytic P systems is rather complicated. Although catalytic P systems are known to be universal, a direct translation of a register machine generating powers of 2 yields a rather big number of rules. Starting with the first example established in [14] using 54 rules, the number of rules for a catalytic P system generating a non-semilinear set of natural numbers was reduced to 32 in [15] and to 29 in [2]; finally, a construction for a catalytic P system generating a non-semilinear set of natural numbers needing only 24 rules and a purely catalytic P system needing only 26 rules was elaborated in [16]. At least for catalytic P systems, in this paper we again are able to reduce the number of rules to 20.

2.5 P Systems with Toxic Objects

In many variants of (catalytic) P systems, for proving computational completeness it is common to introduce a trap symbol # for the case that the derivation goes the wrong way as well as the rule $\# \to \#$ (or $c\# \to c\#$ with a catalyst c) guaranteeing that the derivation will never halt. Yet most of these rules can be avoided if we use toxic objects as introduced in [2]:

We specify a specific subset of *toxic* objects O_{tox}; the P system is only allowed to continue a computation from a configuration C by using an applicable multiset of rules covering all copies of objects from O_{tox} occurring in C; moreover, if there exists no multiset of applicable rules covering all toxic objects, the whole computation having yielded the configuration 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 by P systems with toxic objects using rules of type X we add the subscript tox to O, thus obtaining the families $Y_{gen,\delta}O_{tox}P_m\,(X)$, for any $\delta \in \{sequ, asyn, maxpar\}$, $Y \in \{N, Ps\}$, and $m \geq 1$.

Looking closer into the computational completeness proofs for catalytic P systems given in [5], we see that the only non-cooperative rules used in the proofs given there are rules involving the trap symbol. When going to purely catalytic P systems, we realize that all rules involving the trap symbol are assigned to the additional catalyst; hence, to generate any recursively enumerable set of natural numbers we only need two catalysts for both catalytic P systems and purely catalytic P systems:

$$PsRE = Ps_{gen,maxpar}O_{tox}P_1(cat_2) = Ps_{gen,maxpar}O_{tox}P_1(pcat_2).$$

For more details concerning P systems with toxic objects we refer the reader to [2].

3 Small Catalytic P Systems

We now establish a new construction for simulating a register machine $M = (d, B, l_0, l_h, R)$ by a catalytic P system Π, with $m \leq d$ being the number of decrementable registers.

For all d registers, n_i copies of the symbol o_i are used to represent the value n_i in register i, $1 \leq i \leq d$. For each of the m decrementable registers, we take a catalyst c_i and two specific symbols d_i, e_i, $1 \leq i \leq m$, for simulating SUB-instructions on these registers. For every $l \in B$, we use p_l, and also its variants $\bar{p}_l, \hat{p}_l, \tilde{p}_l$ for $l \in B_{\texttt{SUB}}$, where $B_{\texttt{SUB}}$ denotes the set of labels of SUB-instructions.

$$
\begin{aligned}
&\Pi = (O, C, \mu = [\ ]_1, w_1 = c_1 \dots c_m d_1 \dots d_m p_1 w_0, R_1, f = 1) \text{ where}\\
&O = C \cup D \cup E \cup \Sigma \cup \{\#\} \cup \{p_l \mid l \in B\} \cup \{\bar{p}_l, \hat{p}_l, \tilde{p}_l \mid l \in B_{\texttt{SUB}}\},\\
&C = \{c_i \mid 1 \leq i \leq m\},\\
&D = \{d_i \mid 1 \leq i \leq m\},\\
&E = \{e_i \mid 1 \leq i \leq m\},\\
&\Sigma = \{o_i \mid 1 \leq i \leq d\},\\
&R_1 = \{p_j \to o_r p_k D_m, p_j \to o_r p_l D_m \mid j : (\texttt{ADD}(r), k, l) \in R\}\\
&\quad \cup \{p_j \to \hat{p}_j e_r D_{m,r}, p_j \to \bar{p}_j D_{m,r},\\
&\qquad \hat{p}_j \to \tilde{p}_j D'_{m,r}, \bar{p}_j \to p_k D_m, \tilde{p}_j \to p_k D_m \mid j : (\texttt{SUB}(r), k, l) \in R\}\\
&\quad \cup \{c_r o_r \to c_r d_r, c_r d_r \to c_r, c_{r \oplus_m 1} e_r \to c_{r \oplus_m 1} \mid 1 \leq r \leq m\},\\
&\quad \cup \{d_r \to \#, c_r e_r \to c_r \# \mid 1 \leq r \leq m\}\\
&\quad \cup \{\# \to \#\}.
\end{aligned}
$$

Here $r \oplus_m 1$ for $r < m$ simply is $r+1$, whereas for $r = m$ we define $m \oplus_m 1 = 1$; w_0 stands for additional input present at the beginning, for example, for the given input in case of accepting systems.

Usually, every catalyst c_i, $i \in \{1, \dots, m\}$, is kept busy with the symbol d_i using the rule $c_i d_i \to c_i$, as otherwise the symbols d_i would have to be trapped by the rule $d_i \to \#$, and the trap rule $\# \to \#$ then enforces an infinite non-halting computation. Only during the simulation of SUB-instructions on register r the corresponding catalyst c_r is left free for decrementing or for zero-checking in the second step of the simulation, and in the decrement case both c_r and its "coupled" catalyst $c_{r \oplus_m 1}$ are needed to be free for specific actions in the third step of the simulation.

For the simulation of instructions, we use the following shortcuts:

$$
\begin{aligned}
D_m &= \textstyle\prod_{i \in [1..m]} d_i,\\
D_{m,r} &= \textstyle\prod_{i \in [1..m] \setminus \{r\}} d_i,\\
D'_{m,r} &= \textstyle\prod_{i \in [1..m] \setminus \{r, r \oplus_m 1\}} d_i.
\end{aligned}
$$

The HALT-instruction labeled l_h is simply simulated by not introducing the corresponding state symbol p_{l_h}, i.e., replacing it by λ, in all rules defined in R_1.

Each ADD-instruction $j : (\texttt{ADD}(r), k, l)$, for $r \in \{1, \dots, d\}$, can easily be simulated by the rules $p_j \to o_r p_k D_m$ and $p_j \to o_r p_l D_m$; in parallel, the rules $c_i d_i \to c_i$, $1 \leq i \leq m$, have to be carried out, as otherwise the symbols d_i would have to be trapped by the rules $d_i \to \#$.

Each SUB-instruction $j : (\texttt{SUB}(r), k, l)$, is simulated as shown in the table listed below (the rules in brackets [and] are those to be carried out in case of a wrong choice):

Simulation of the SUB-instruction $j : (\texttt{SUB}(r), k, l)$ if	
register r is not empty	register r is empty
$p_j \to \hat{p}_j e_r D_{m,r}$	$p_j \to \bar{p}_j D_{m,r}$
$c_r o_r \to c_r d_r$ $[c_r e_r \to c_r \#]$	c_r should stay idle
$\hat{p}_j \to \tilde{p}_j D'_{m,r}$	$\bar{p}_j \to p_k D_m$
$c_r d_r \to c_r$ $[d_r \to \#]$	$[d_r \to \#]$
$\tilde{p}_j \to p_k D_m$	
$c_{r \oplus_m 1} e_r \to c_{r \oplus_m 1}$	

In the first step of the simulation of each instruction (ADD-instruction, SUB-instruction, and even HALT-instruction) due to the introduction of D_m in the previous step (we also start with that in the initial configuration) every catalyst c_r is kept busy by the corresponding symbol d_r, $1 \leq r \leq m$. Hence, this also guarantees that the zero-check on register r works correctly enforcing $d_r \to \#$ to be applied, as in the case of a wrong choice two symbols d_r are present.

In sum we have obtained the following result:

Theorem 1. *For any register machine $M = (d, B, L_0, l_h, R)$, with $m \leq d$ being the number of decrementable registers, we can construct a catalytic P system*

$$\Pi = (O, C, \mu = [\ \ ]_1, w_1, R_1, f = 1)$$

simulating the computations of M such that

$$|R_1| \leq \texttt{ADD}^1(R) + 2 \times \texttt{ADD}^2(R) + 5 \times \texttt{SUB}(R) + 5 \times m + 1,$$

where $\texttt{ADD}^1(R)$ denotes the number of deterministic ADD*-instructions in R, $\texttt{ADD}^2(R)$ denotes the number of non-deterministic* ADD*-instructions in R, and $\texttt{SUB}(R)$ denotes the number of* SUB*-instructions in R.*

For the purely catalytic case, one additional catalyst c_{m+1} is needed to be used with all the non-cooperative rules. Unfortunately, in this case a slightly more complicated simulation of SUB-instructions is needed, see Sosík, 2015 ([16]), where for catalytic P systems

$$|R_1| \leq 2 \times \texttt{ADD}^1(R) + 3 \times \texttt{ADD}^2(R) + 6 \times \texttt{SUB}(R) + 5 \times m + 1,$$

and for purely for catalytic P systems

$$|R_1| \leq 2 \times \texttt{ADD}^1(R) + 3 \times \texttt{ADD}^2(R) + 6 \times \texttt{SUB}(R) + 6 \times m + 1,$$

is shown.

Remark 1. On the other hand, exactly the same construction as elaborated above can be used when allowing for $m + 2$ catalysts, with catalyst c_{m+1} being used with the state symbols and catalyst c_{m+2} being used with the trap rules.

Finally we mention that the simulation results established above hold true for register machines and their corresponding (purely) catalytic P systems in the case of generating or accepting systems and even for systems computing functions or relation on natural numbers.

4 Small Catalytic P Systems – an Example

For constructing specific examples, the construction elaborated above can even be refined a little bit in order to more reduce the number of rules needed. We will now show this by constructing a P system generating the set of natural numbers $\{2^n \mid n \geq 1\}$.

In fact, we are going to simulate a *generalized register machine* (a variant of the model of *generalized counter automata* as described in [3]): a *generalized* SUB*-instruction* in the generalized register machine $M = (d, B, l_0, l_h, P)$ is of the form $j : (\texttt{SUB}(r), \{X_1, \cdots, X_k\}, \{Y_1, \cdots, Y_h\})$, where each X_i, $1 \leq i \leq k$, and $Y_{i'}$, $1 \leq i' \leq h$, is of the form $\{\texttt{ADD}(r_1)^{n_1}, \cdots, \texttt{ADD}(r_p)^{n_p}\}l$ with $l \in B$ and $r_q \in [1..d]$, $n_q \geq 1$, $1 \leq q \leq p$, $p \geq 0$. For sake of conciseness, we omit the empty set in these notations and write $\texttt{ADD}(r_1)^{n_1} \cdots \texttt{ADD}(r_p)^{n_p}$ instead of $\{\texttt{ADD}(r_1)^{n_1}, \cdots, \texttt{ADD}(r_p)^{n_p}\}$.

Starting with 1 in register 2, we use the following program (using the notions of generalized SUB-instructions):

$$1 : (\texttt{SUB}(2), \{\texttt{ADD}(1)^2 ADD(3)1\}, \{2, \texttt{ADD}(3)3\})$$
$$2 : (\texttt{SUB}(1), \{\texttt{ADD}(2)2\}, \{1\})$$
$$3 : HALT$$

With using the generalized SUB-instruction 1, the contents of register 1 is doubled in register 1 and copied to register 3, after which one may go to 3 and halt after having added 1 to register 3 in this final step, or copy back the contents of register 1 into register 2 using the generalized SUB-instruction 2. Then the cycle starts again with using the generalized SUB-instruction 1. We observe that this generalized register machine computes exactly the set of natural numbers $\{2^n \mid n \geq 1\}$; its computations can be simulated by the following P system Π.

$$\Pi = (O, C, \mu = [\ \ ]_1, w_1 = c_1 c_2 d_1 \hat{p}_2 e_2 o_2, R_1, f = 1) \text{ where}$$
$$O = C \cup \Gamma \cup \{\#\} \cup \{p_l, \bar{p}_l, \hat{p}_l, \tilde{p}_l \mid 1 \leq l \leq 2\},$$
$$C = \{c_i \mid 1 \leq i \leq m\},$$
$$\Gamma = \{d_i, e_i \mid 1 \leq i \leq 2\} \cup \{o_i \mid 1 \leq i \leq 3\},$$

and R_1, besides the trap rule $\# \to \#$, contains the rules depicted in the following two tables:

Simulation of the SUB-instruction $(\texttt{SUB}(2), \{\texttt{ADD}(1)^2\texttt{ADD}(3)1\}, \{2, \texttt{ADD}(3)3\})$ if

register 2 is not empty	register 2 is empty
$c_2 o_2 \to c_2 d_2$ $[c_2 e_2 \to c_2 \#]$	c_2 should stay idle
$\hat{p}_2 \to \tilde{p}_2 {o_1}^2 o_3$	$\bar{p}_2 \to \hat{p}_1 d_2 e_1$, $\bar{p}_2 \to d_1 d_2 o_3$
$c_2 d_2 \to c_2$ $[d_2 \to \#]$	$[d_2 \to \#]$
$c_1 e_2 \to c_1$	
$\tilde{p}_2 \to \hat{p}_2 d_1 e_2$, $\tilde{p}_2 \to \bar{p}_2 d_1$	

Simulation of the SUB-instruction $(\texttt{SUB}(1), \{ADD(2)2\}, \{1\})$ if

register 1 is not empty	register 1 is empty
$c_1 o_1 \to c_1 d_1$ $[c_1 e_1 \to c_1 \#]$	c_1 should stay idle
$\hat{p}_1 \to \tilde{p}_1 o_2$	$\bar{p}_1 \to \hat{p}_2 d_1 e_2$
$c_1 d_1 \to c_1$ $[d_1 \to \#]$	$[d_1 \to \#]$
$c_2 e_1 \to c_2$	
$\tilde{p}_1 \to \hat{p}_1 d_2 e_1$, $\tilde{p}_1 \to \bar{p}_1 d_2$	

In contrast to the general construction, we here omit the first line of the table of SUB-instructions, already generating the situation of the second line in the final step of the simulation of the preceding instruction. It is important to mention that at the beginning we know that we can start with decrementing register 2, and moreover, each of the two registers is going to be emptied completely as soon as we start to decrement it. Finally, we have to remark that after the execution of the rule $\bar{p}_2 \to d_1 d_2 o_3$ the simulation of the generalized register machine has ended, but the P system still has to continue with applying the rules $c_2 o_2 \to c_2 d_2$ and $c_2 d_2 \to c_2$ in a cycle, thus emptying register 2, in order to halt correctly.

In sum, the total number of rules in R_1 is 20, i.e.,

$$\{2^n \mid n \geq 1\} \in N_{gen,maxpar}OP_m\,(cat_2)_{20} \cap N_{gen,maxpar}O_{tox}P_m\,(cat_2)_{17}\,,$$

as in the toxic case, only the objects $d_1, d_2, \#$ are toxic, hence, only three rules can be saved, again yielding 17 rules, the same number as already obtained with the construction given in [2].

5 Small Catalytic Universal P Systems

In this section we establish various results for universal catalytic P systems, thereby improving several results from [3].

5.1 Generalized Counter Automata

For the descriptional complexity results established in the following, we define and use the concept of *generalized counter automata* (similar to the ones from [1] and [3]).

We now consider an extended variant of register machines called *generalized counter automaton*, written $M = (d, B, Q, q_i, q_h, P)$, where B is the set of labels, Q is a set of states, q_i is the initial state, q_h is the final state, and P contains the more general type of instructions $j : (q, M_-, N, M_+, q')$. Let R denote the set of registers $1, \cdots, d$; then in the instruction $j : (q, M_-, N, M_+, q')$ $q, q' \in Q$ are states, $N \subseteq R$ is a set of registers, and M_+, M_- are multisets over R. A *generalized counter automaton* now applies such an instruction j as follows: first, for all $r \in R$ with $M_-(r) > 0$, $M_-(r)$ is subtracted from register r (if at least one resulting value would be negative, the machine is blocked without producing any result); second, every r in the subset N of registers is checked to be zero

(if at least one of them is found to be non-zero, the machine is blocked without producing any result); third, for each $r \in R$, $M_+(r)$ is added to the contents of register r; finally the state changes to q'.

Example 1. Consider the instruction $j : (q, \langle r \rangle, \{r\}, \langle r \rangle, q')$, which performs a 1-test on register r, i.e., a transition from q to q' if the value of register r is exactly 1, but leaving it unchanged: it decrements register r, then tests it for zero, and finally increments it again.

In any derivation step, the *generalized counter automaton*,

$$M = (d, B, Q, q_i, q_h, P)$$

applies one instruction associated with the current state, chosen in a non-deterministic way. The computation starts in the initial state q_i, and we say that it halts if the final state q_h has been reached (which replaces the condition of reaching the final HALT-instruction labeled by h).

In [6], Theorem 4, a universal sequential P system using 16 antiport rules with forbidden context was described based on the universal register machine U_{32} from [8], and based on this universal sequential P system with 16 antiport rules with forbidden context a strongly universal generalized counter automaton was given in [1]. In a similar way, we present the rules of a strongly universal generalized counter automaton which is based on the universal register machine U_{22} from [8] (see Figure 1) in Table 5.1. The inputs for U_{22} are given in register 1 for the machine to be simulated and in register 0 for the input to this machine.

However, for technical reasons, we have to make the following additional assumptions, and we will call the corresponding systems *weakly generalized counter automata* (or *wGCA* for short). Consider a coupling function f_c, a bijective mapping from the set of registers to the same set. For each instruction, we require that M_- does not contain multiple copies of any register, and, moreover, the sets $supp(M_-)$, $f_c(supp(M_-))$ and N are all disjoint. This requirement comes from the fact that for the zerotest, we need the corresponding catalyst, while for decrementing a register we need the corresponding catalyst and its coupled catalyst, and our aim is to perform these simulations in parallel, which lets us considerably reduce the number of rules. However, if this requirement is not satisfied, then the generalized instruction can be split into simpler instructions satisfying the requirement.

After having carefully inspected the Korec machines and the resulting GCA from [3], we decided to use the following coupling function f_c:

r :	0	1	2	3	4	5	6	7
$f_c(r)$	6	5	7	4	3	1	0	2

Remark 2. For technical reasons, we have to produce the output in an additional register 8 that only has increment instructions associated to it, but at the end we need not worry about all other registers to be emptied in the end. For this

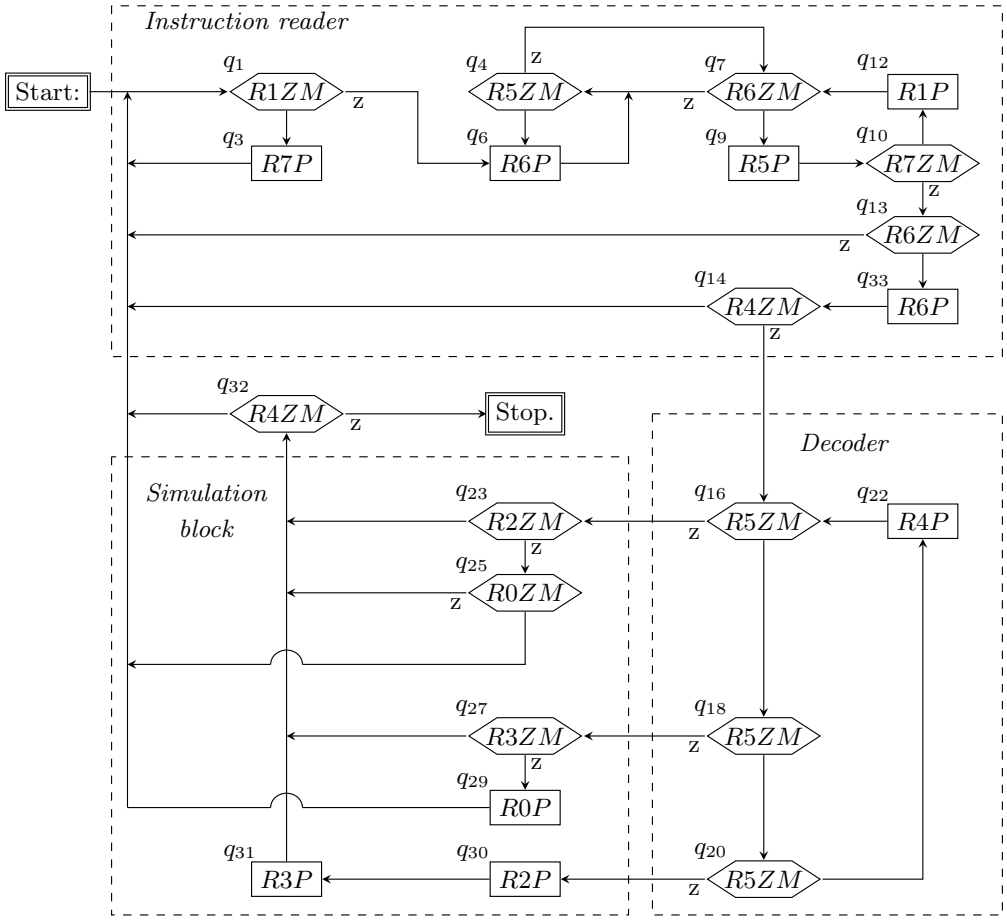


Fig. 1. The strongly universal register machine U_{22}.

purpose, the additional instructions $18'$ and $18''$ are used, and λ is the new halting state.

Yet the rules 18, $18'$, and $18''$ can even be replaced by the following rules 18 and $18'$ to reduce the complexity of the final output procedure:

$$18 : (q_{32}, \langle 0 \rangle, \{4\}, \langle 8 \rangle, q_{32}),$$
$$18' : (q_{32}, \langle \rangle, \{0, 4\}, \langle \rangle, \lambda)$$

This finally yields a total of 21 wGCA instructions. Notice, moreover, that the instructions 2, 7, 11, 14, 16, and $18'$ are not decrementing; we will use this to further decrease the number of rules. The goal of wGCA is to serve as a model of easy and efficient straightforward (i.e., without register encoding) simulation of the strongly universal register machine URM_{22} established by Korec, also taking advantage of parallel operations, but using the only one catalyst per register and no additional catalysts. The approach is pretty similar to how a very small strongly universal system was constructed in [3] using 21 catalysts, yet there two catalysts were used per register (and three copies of these catalyst for the fifth register); hence, here we are saving catalysts at the expense of having more rules.

Table 1. Universal generalized counter automaton simulating URM_{22} from [8].

$1 : (q_1, \langle 1\rangle, \{\}, \langle 7\rangle, q_1)$,
$2 : (q_1, \langle\rangle, \{1\}, \langle 6\rangle, q_4)$,
$3 : (q_4, \langle 5\rangle, \{\}, \langle 6\rangle, q_4)$,
$4 : (q_4, \langle 6\rangle, \{5\}, \langle 5\rangle, q_{10})$,
$5 : (q_{10}, \langle 6, 7\rangle, \{\}, \langle 1, 5\rangle, q_{10})$,
$6 : (q_{10}, \langle 7\rangle, \{6\}, \langle 1\rangle, q_4)$,
$7 : (q_{10}, \langle\rangle, \{6, 7\}, \langle\rangle, q_1)$,
$8 : (q_{10}, \langle 4, 6\rangle, \{7\}, \langle\rangle, q_1)$,
$9 : (q_{10}, \langle 5, 6\rangle, \{4, 7\}, \langle\rangle, q_{18})$,
$10 : (q_{18}, \langle 5\rangle, \{\}, \langle\rangle, q_{20})$,
$10' : (q_{20}, \langle 5\rangle, \{\}, \langle 4\rangle, q_{16})$,
$10'' : (q_{16}, \langle 5\rangle, \{\}, \langle\rangle, q_{18})$.
$11 : (q_{18}, \langle\rangle, \{3, 5\}, \langle 0\rangle, q_1)$,
$12 : (q_{16}, \langle 0\rangle, \{2, 5\}, \langle\rangle, q_1)$,
$13 : (q_{16}, \langle 2\rangle, \{5\}, \langle\rangle, q_{32})$,
$14 : (q_{16}, \langle\rangle, \{0, 2, 5\}, \langle\rangle, q_{32})$
$15 : (q_{18}, \langle 3\rangle, \{5\}, \langle\rangle, q_{32})$,
$16 : (q_{20}, \langle\rangle, \{5\}, \langle 2, 3\rangle, q_{32})$,
$17 : (q_{32}, \langle 4\rangle, \{\}, \langle\rangle, q_1)$,
$18 : (q_{32}, \langle\rangle, \{4\}, \langle\rangle, q_h)$,
$18' : (q_h, \langle 0\rangle, \{\}, \langle 8\rangle, q_h)$,
$18'' : (q_h, \langle\rangle, \{0\}, \langle\rangle, \lambda)$

5.2 Computational Completeness for Catalytic P Systems with Multiple Catalysts

As it was already described above, $5m + 1$ rules are associated with m decrementable registers: $\# \to \#$ and the following 5 rules for every register r:

$$c_r o_r \to c_r d_r,\ c_r d_r \to c_r,\ c_r e_r \to c_r \#,\ c_{f_c(r)} e_r \to c_{f_c(r)},\ d_r \to \#.$$

The rest of this section is dedicated to the discussion of the rule complexity of simulating one wGCA instruction.

For an instruction $j : (q_i, M_-, N, M_+, q_k)$ of a wGCA, we define

$$\begin{aligned} D_{m,M_-,N} &= \textstyle\prod_{i\in[1..m]\setminus(supp(M_-)\cup N)} d_i, \\ D'_{m,M_-} &= \textstyle\prod_{i\in[1..m]\setminus\{r,c(r)|r\in M_-\}} d_i, \text{ and} \\ E_{M_-} &= \textstyle\prod_{r\in M_-} e_r. \end{aligned}$$

We first consider the case when the instruction j of a wGCA is non-decrementing, i.e., if M_- is empty. An instruction $j : (q_i, \langle\rangle, N, M_+, q_k)$ then is simulated by the following two rules:

$$q_i \to p_j\ D_{m,\emptyset,N},\ p_j \to q_k D_m O_{M_+},$$

where $D_{m,\emptyset,N} = \prod_{i\in\{1,\cdots,m\}\setminus N} d_i$, and O_{M_+} is the multiset of objects o_i for all copies of i in M_+.

For a general instruction j of a wGCA, $j : (q_i, M_-, N, M_+, q_k)$, it suffices to have the following three rules:

$$q_i \to p_j E_{M_-} D_{m,M_-,N},\ p_j \to p_j D'_{m,M_-},\ p_j \to q_k D_m O_{M_+}.$$

Remark 3. Finally, for eventually saving some more rules it might remain to check if we could skip the first step of the next instruction, for example by producing $p_{j'} D_{m,M'_-,N'} O_{M_+}$ instead of $q_k D_m O_{M_+}$, where j', M'_- and N' stand for the label, decrementing multiset and zerotesting set of the next generalized instruction, and if this skip actually could save instructions.

Remark 4. In case of toxic objects, only the $m+1$ rules $d_r \to \#$ and $\# \to \#$ can be saved.

5.3 A Universal Catalytic P System with 8 Catalysts

We take the wGCA presented above, having in total 15 decrementing instructions and 6 non-decrementing ones. Consider the simulation from Subsection 5.2: it uses 3 rules per decrementing wGCA instruction, 2 rules per non-decrementing wGCA instruction, plus 5 rules for each of the 8 working registers 0 to 7 (register 8 is not counted here) plus one rule. This yields a strongly universal catalytic P system with 8 catalysts and $3 \times 15 + 2 \times 6 + 5 \times 8 + 1 = \mathbf{98}$ rules.

For conciseness, we now will denote the multiset of objects d_r, $0 \le r \le 7$, $r \notin M$ by $d(M)$, and omit the braces denoting the set M:

$$
\begin{aligned}
&\Pi = (O, \Sigma, C = \{c_r \mid 0 \le r \le 7\}, \mu = [\ ]_1, w_1, R_1, f = 1),\\
&O = \{o_r, d_r, e_r \mid 0 \le r \le 7\} \cup \{\#, p_{10'}, p_{10''}, p_{18'}, o_8\} \cup \{p_j \mid 1 \le j \le 18\}\\
&\quad \cup \{p'_j \mid j \in \{1,3,4,5,6,8,9,10,10',10'',12,13,15,17,18'\}\}\\
&\quad \cup \{q_1, q_4, q_{10}, q_{16}, q_{18}, q_{20}, q_{32}\},\\
&R_1 = R \cup \{\# \to \#\} \cup \{c_r o_r \to c_r d_r,\ c_r d_r \to c_r,\ c_r e_r \to c_r \#,\\
&\qquad c_{f_c(r)} e_r \to c_{f_c(r)},\ d_r \to \# \mid 0 \le r \le 7\},\\
&w_1 = q_1 d(),\ \text{and the rules from the set } R \text{ are listed below:}
\end{aligned}
$$

$$
\begin{array}{lll}
q_1 \to p_1 e_1 d(1), & p_1 \to p'_1 d(1,5), & p'_1 \to q_1 d() o_7,\\
q_1 \to p_2 d(1), & p_2 \to q_4 d() o_6, & \\
q_4 \to p_3 e_5 d(5), & p_3 \to p'_3 d(1,5), & p'_3 \to q_4 d() o_6,\\
q_4 \to p_4 e_6 d(5,6), & p_4 \to p'_4 d(0,6), & p'_4 \to q_{10} d() o_5,\\
q_{10} \to p_5 e_6 e_7 d(6,7), & p_5 \to p'_5 d(0,2,6,7), & p'_5 \to q_{10} d() o_1 o_5,\\
q_{10} \to p_6 e_7 d(6,7), & p_6 \to p'_6 d(2,7), & p'_6 \to q_4 d() o_1,\\
q_{10} \to p_7 d_{6,7}, & p_7 \to q_1 d(), & \\
q_{10} \to p_8 e_4 e_6 d(4,6,7), & p_8 \to p'_8 d(0,3,4,6), & p'_8 \to q_1 d(),\\
q_{10} \to p_9 e_5 e_6 d(4,5,6,7), & p_9 \to p'_9 d(0,1,5,6), & p'_9 \to q_{18} d(),\\
q_{18} \to p_{10} e_5 d(5), & p_{10} \to p'_{10} d(1,5), & p'_{10} \to q_{20} d(),\\
q_{20} \to p_{10'} e_5 d(5), & p_{10'} \to p'_{10'} d(1,5), & p'_{10'} \to q_{16} d(),\\
q_{16} \to p_{10''} e_5 d(5), & p_{10''} \to p'_{10''} d(1,5), & p'_{10''} \to q_{18} d(),\\
q_{18} \to p_{11} d(3,5), & p_{11} \to q_1 d() o_0, & \\
q_{16} \to p_{12} e_0 d(0,2,5), & p_{12} \to p'_{12} d(0,6), & p'_{12} \to q_1 d(),\\
q_{16} \to p_{13} e_2 d(2,5), & p_{13} \to p'_{13} d(2,7), & p'_{13} \to q_{32} d(),\\
q_{16} \to p_{14} d(0,2,5), & p_{14} \to q_{32} d(), & \\
q_{18} \to p_{15} e_3 d(3,5), & p_{15} \to p'_{15} d(3,4), & p'_{15} \to q_{32} d(),\\
q_{20} \to p_{16} d(5), & p_{16} \to q_{32} d() o_2 o_3, & \\
q_{32} \to p_{17} e_4 d(4), & p_{17} \to p'_{17} d(3,4), & p'_{17} \to q_1 d(),\\
q_{32} \to p_{18} e_0 d(0,4), & p_{18} \to p'_{18} d(0,6), & p'_{18} \to q_{32} d() o_8,\\
q_{32} \to p_{18'} d(0,4), & p_{18'} \to d(). &
\end{array}
$$

In addition to w_1, to the initial configuration we add the number of symbols o_1 corresponding with the code of the machine to be simulated and the number of symbols o_0 corresponding with the input number to this machine; the result of the simulation is represented by the number of symbols o_8 in the final configuration.

This finally yields a catalytic P system with 8 catalysts and only **98** rules, thus improving the previously known best result of 185 rules from [3].

According to Remark 4, when using toxic objects, this number of 98 rules can be reduced by 9 to **89** rules, thus improving the previously known best result of 120 rules from [3].

5.4 Purely Catalytic P Systems

As already explained in Remark 1 for the general case. it is not difficult to see from the construction in Subsection 5.2 that the rule complexity of the constructions obtained there for cat_m also holds for $pcat_{m+2}$, with catalyst c_{m+1} being used with the state symbols and catalyst c_{m+2} being used with the trap rules. Hence, any generalized register machine with m decrementable registers and s generalized SUB-instructions can be simulated by a *purely* catalytic P system with $m + 2$ catalysts and $5s + 5m + 1$ rules.

It is possible to save one catalyst by using more rules, as it follows from the construction given in [16] that any generalized register machine with m decrementable registers and s generalized SUB-instructions can be simulated by a *purely* catalytic P system with $m+1$ catalysts and $6s+6m+1$ rules. In that way, one catalyst can be saved at the cost of having more rules, i.e, we get a universal purely catalytic P system with 9 catalysts and with $6 \times 16 + 6 \times 8 + 1 = \mathbf{145}$ rules (thus improving the result of 185 rules from [3]).

6 Summary

In this paper we have again illustrated that a rather small number of rules is needed for obtaining computational completeness and to get some specific non-semilinear sets of natural numbers with catalytic P systems; as an example, we have shown that only **20** rules are needed to generate the set of natural numbers $\{2^n \mid n \geq 1\}$, thus again improving the result from [16] where 24 rules were shown to be sufficient for generating a similar non-semilinear set of natural numbers.

We also have improved the number of rules needed for a universal catalytic P system simulating the universal register machine URM_{22} of Korec to **98** rules and to **89** rules for the case of using toxic objects.

Acknowledgements

Both authors are very grateful to Petr Sosík for useful discussions.

References

1. A. Alhazov, B. Aman, R. Freund, Gh. Păun: Matter and anti-matter in membrane systems. In: H. Jürgensen, J. Karhumäki, A. Okhotin (Eds.): *16th International Workshop on Descriptional Complexity of Formal Systems, DCFS 2014*. Lecture Notes in Computer Science **8614**, 2014, 65–76.
2. A. Alhazov, R. Freund: P systems with toxic objects. In: M. Gheorghe, G. Rozenberg, A. Salomaa, P. Sosík, C. Zandron (Eds.): *Membrane Computing - 15th International Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers*. Lecture Notes in Computer Science **8961**, Springer, 2014, 99–125.
3. A. Alhazov, R. Freund: Variants of small universal P systems with catalysts. *Fundamenta Informaticae* **138** (1-2), 227–250 (2015).
4. J. Dassow, Gh. Păun: *Regulated Rewriting in Formal Language Theory*. Springer, 1989.
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, M. Oswald: Small universal antiport P systems and universal multiset grammars. In: M.Á. Gutiérrez-Naranjo, Gh. Păun, A. Riscos-Núñez, F.J. Romero-Campero (Eds.): *Fourth Brainstorming Week on Membrane Computing*, Sevilla, 2006, vol. II, RGNC Report **03/2006**, Fénix Editora, Sevilla, 2006, 51–64.
7. 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.
8. I. Korec: Small universal register machines. *Theoretical Computer Science* **168**, 1996, 267–301.
9. M.L. Minsky: *Computation: Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1967.
10. Gh. Păun: Computing with membranes. *J. Comput. Syst. Sci.* **61**, 108–143 (2000); also see TUCS Report 208, 1998, `www.tucs.fi`.
11. Gh. Păun: *Membrane Computing. An Introduction*. Springer, 2002.
12. Gh. Păun, G. Rozenberg, A. Salomaa (Eds.): *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010, 118–143.
13. G. Rozenberg, A. Salomaa (Eds.): *Handbook of Formal Languages*, 3 volumes. Springer, 1997.
14. P. Sosík: A catalytic P system with two catalysts generating a non-semilinear set. *Romanian Journal of Information Science and Technology* **16** (1), 3–9 (2013).
15. P. Sosík, M. Langer: Improved universality proof for catalytic P systems and a relation to non-semilinear sets. In: S. Bensch, R. Freund, F. Otto (Eds.): *Sixth Workshop on Non-Classical Models of Automata and Applications (NCMA 2014)*, books@ocg.at, BAND 304, 2014, 223–233.
16. P. Sosík, M. Langer: Small catalytic P systems simulating register machines, *to appear in TCS*.
17. The P systems webpage. `http://ppage.psystems.eu`

Going Beyond Turing with 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, TU Wien
Favoritenstraße 9-11, 1040 Vienna, Austria
E-mail: {rudi,marion}@emcc.at

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

Abstract. 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 prove that this extension of the original model can easily simulate register machines. Based on this proof, we then define red-green variants of these extended spiking neural P systems with white hole rules and show how to go beyond Turing with these red-green 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 [18], [19], and [11]. In the area of P systems, the model of *spiking neural P systems* was introduced in [16]. Whereas the basic model of membrane systems, see [22], 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 [24] and then further elaborated, for example, in [10] and [20]. 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 neuron allow us to send information to other neurons in the form of electrical impulses (also called spikes) which are summed up at the target neuron; 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 neuron 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 [30], spiking neural P systems with weights on the axons and firing threshold were investigated, where the values of these weights and firing thresholds as well as the potential consumed by each rule could be natural numbers, integer numbers, rational numbers, and even (computable) real numbers.

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 neurons, 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 [16] the output of a spiking neural P system was considered to be the time between two spikes in a designated output neuron. 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 [15]. In [6], 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 the 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 [29] and [28]. 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 [2] are based on the extended model as introduced in [1]. In this paper we now use this new model of spiking neural P systems with white hole rules together the idea of considering infinite computations on finite inputs, which will allow us to "go beyond Turing".

Variants of how to "go beyond Turing" are discussed in [17], for example, the definitions and results for red-green Turing machines can be found there. In [3] 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 [5] and [27]. Computations with infinite words by P automata were investigated in [8].

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. Then we define red-green Turing machines and red-green register machines and recall some results from [3]. In Section 4 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.

In Section 5, we define the model of extended spiking neural P systems extended by the use of white hole rules as introduced in [2]. 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, i.e., 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. Based on this result, we define the *red-green* variant of spiking neural P systems with white hole rules and show that their computational power is similar to the

computational power of red-green register machines. A short summary of the results we obtained concludes the paper.

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] and [2].

For the basic elements of formal language theory needed in the following, we refer to any monograph in this area, in particular, to [26]. 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 [21] for original definitions, and to [7] for the definitions we use in this paper:

An *n-register machine* is a tuple $M = (n, B, l_0, l_h, P)$, where n 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 $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq n$.
  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 $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq n$.
  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.

– $l_h : halt$ (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 [10]).

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 [21], the results proved in [7] and [9] 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 $M = (m, B, l_0, l_h, P, T)$.

The following result is folklore, too (e.g., see [21]):

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.*

2.2 The Arithmetical Hierarchy

The Arithmetical Hierarchy (e.g., see [4]) 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 Red-Green Automata

The exposition of this section mainly follows the corresponding section in [2].

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.

3.1 Red-Green Turing Machines

Red-green Turing machines, see [17], 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 [8]), 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 [17]:

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.*

3.2 Red–Green Register Machines

In [3], 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 [21], 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 [3]; 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 $\Sigma_2 \cap \Pi_2$ by red-green register machines as Theorem 2 does for red-green Turing machines, see [3]:

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.*

4 Extended Spiking Neural P Systems

The reader is supposed to be familiar with basic elements of membrane computing, e.g., from [23] and [25]; comprehensive information can be found on the P systems web page [31]. Moreover, for the motivation and the biological background of spiking neural P systems we refer the reader to [16]. 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.

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

Definition 1. *An* extended spiking neural P system *(of degree $m \geq 1$) (an* ESNP system *for short) 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 neuron 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 neuron), $w \in \{a\}^*$ 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 $\left(i, E/a^k \to P; d' - 1\right)$ 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 [16], in the productions (l, w, t) of a rule $\left(i, E/a^k \to \{(l, w, t)\}; d\right)$, 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 $\left(i, E/a^k \to P; d\right)$ 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 $\left(i, E/a^k \to P; d\right)$ is activated, the packages induced by the productions (l, w, t) in the set P of a rule $\left(i, E/a^k \to P; d\right)$ 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 [16], [24], and [6]. 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 [16], 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 [6], 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 $(2, \{i\}/i \to \{(1, 1), (2, j)\})$.

4.1 ESNP Systems as Generating Devices

The following results were already proved in [1]:

Lemma 3. *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 4. *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 5. *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.*

5 ESNP Systems with White Hole Rules

In this section, we recall the definition of extended spiking neural P systems with white hole rules as introduced in [2]. 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 neuron 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 neuron), $w \in \mathbb{N}$ 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 ESNP system also is an EESNP system, but without white hole rules, and a finite EESNP system also is a finite ESNP system, as in this

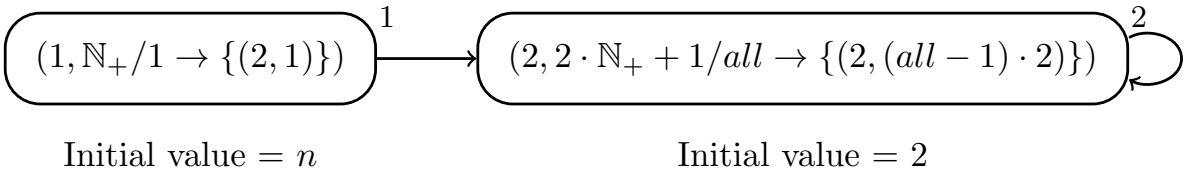
case the effect of white hole rules is also bounded, i.e., even with allowing the use of white hole rules, the following lemma as a counterpart of Lemma 3 is still valid:

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 EESNP systems which really make use of the whole power of white hole rules.

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



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.

5.1 Computational Completeness of EESNP Systems

The following main result was already established in [2].

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

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

Then we can effectively construct an EESNP system $\Pi = (m, S, R)$ 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
  $\{(1, q_{l_1} \cdot \mathbb{N}_+ / all \rightarrow \{(1, all \cdot q_{l_i} p_r / q_{l_1})\}) \mid 2 \leq i \leq 3\}$ in neuron 1.
- $l_1 : (SUB(r), l_2, l_3)$ (SUB instruction)
  This instruction can be simulated by the rules
  $(1, q_{l_1} p_r \cdot \mathbb{N}_+ / all \rightarrow \{(1, all \cdot q_{l_2} / (q_{l_1} p_r))\})$ and
  $(1, q_{l_1} \cdot \mathbb{N}_+ \setminus q_{l_1} p_r \cdot \mathbb{N}_+ / all \rightarrow \{(1, all \cdot q_{l_2} / q_{l_1})\})$ 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
  $(1, q_{l_h} \cdot \mathbb{N}_+ / all \rightarrow \{(1, all \cdot 1 / q_{l_h})\})$ 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 EESNP system constructed in the preceding proof even again yield integer numbers, we could also interpret this EESNP system as an ESNP system with exhaustive use of rules:

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

$(1, q \cdot \mathbb{N}_+ / all \rightarrow \{(1, all \cdot p / q)\})$

with p and q being natural numbers. Each of these rules can be simulated in a one-to-one manner by the rule

$(1, q \cdot \mathbb{N}_+ / q \rightarrow p)$

used in an ESNP system with one neuron in the exhaustive way.

Based on the preceding main result, i.e., Lemma 5, the following theorems were proved in [2].

Theorem 6. *Any recursively enumerable set of n-dimensional vectors can be generated by an EESNP system with $n + 1$ neurons.*

Theorem 7. *Any recursively enumerable language L with $L \subseteq T^*$ for a terminal alphabet T with $card(T) = n$ can be generated by an EESNP system with $n + 1$ neurons.*

6 Red-Green EESNP Systems

For defining a suitable model of red-green EESNP systems we have to consider several constraints.

First of all, the computations should be deterministic, i.e., for any configuration of the EESNP system Π there should be at most one rule applicable in each

neuron. This condition can be fulfilled syntactically by requiring the checking sets of all the rules in each neuron to be disjoint.

Whereas in the generating case we had one output neuron for each possible input symbol, these specific neurons now have to act as input neurons. As we only want deterministic behavior to be considered now, we assume that in every derivation step at most one input neuron spikes until the whole input is "read", but this input has to be made "on demand", i.e., we imagine that the EESNP system Π sends out an input request to the environment which is answered in the next step by the spiking of exactly one input neuron as long as the string has not been "read" completely.

"Reading" the spiking of an input neuron into the single actor neuron means that we have to be able to distinguish the signals coming from different input neurons. Hence, the simplest variant to do this is to multiply the spike coming from input neuron number k by k. Yet then we have to take into account that the minimum value in the actor neuron must be bigger than the maximal number k, i.e., the smallest prime number used for the prime number encoding must fulfill this condition, and our encoding of the number n_i now is chosen to be ${p_i}^{n_i+1}$.

Finally, we have to define red and green "states" of the red-green EESNP system; yet as we only have a finite number of rules in each neuron, each of the possible vectors of rules represents a color; hence, the color of the current configuration, i.e., its "state", can be defined via the (unique) vector of rules to be applied.

Based on the proof Lemma 5, we now can easily establish the following results, similar to the results obtained for red-green register machines, see Lemmas 1 and 2 as well as Theorem 3:

Lemma 6. *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 EESNP system Π 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 EESNP system Π in states of color c and only in the last step of the simulation changes to a state of color c'.*

Lemma 7. *The computations of a red-green EESNP system Π 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 derivation step of Π 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'.*

As an immediate consequence, the preceding two lemmas yield the characterization of Σ_2 and $\Sigma_2 \cap \Pi_2$ by red-green EESNP systems:

Theorem 8. *(Computational power of red-green EESNP systems)*

(i) *A set of strings L is recognized by a red-green EESNP systems with one mind change if and only if $L \in \Sigma_1$, i.e., if L is recursively enumerable.*

(ii) *Red-green EESNP systems recognize exactly the Σ_2-sets of the Arithmetical Hierarchy.*

(iii) *Red-green EESNP systems accept exactly those sets which simultaneously are Σ_2- and Π_2-sets of the Arithmetical Hierarchy.*

7 Conclusion

In this paper, we have shown that the model of extended spiking neural P systems with white hole rules as introduced in [2] is computationally complete, but also allows for a red-green variant and thus to "go beyond Turing". Computational completeness can already be obtained with only one actor neuron, and with the red-green variant of extended spiking neural P systems with white hole rules exactly the Σ_2-sets of the Arithmetical Hierarchy can be recognized.

References

1. A. Alhazov, R. Freund, M. Oswald, M. Slavkovik: Extended spiking neural P systems. In: [13], 123–134.
2. A. Alhazov, R. Freund, S. Ivanov, M. Oswald, S. Verlan: Extended spiking neural P systems with white holes. In: *Thirteenth Brainstorming Week on Membrane Computing*, to appear.
3. B. Aman, E. Csuhaj-Varjú, R. Freund: Red-green P automata. In: [12], 139–157.
4. P. Budnik: *What Is and What Will Be.* Mountain Math Software, 2006.
5. C.S. Calude, Gh. Păun: Bio-steps beyond Turing. *Biosystems* **77**, 175–194 (2004).
6. H. Chen, R. Freund, M. Ionescu, Gh. Păun, M.J. Pérez-Jiménez: On string languages generated by spiking neural P systems. In: [13], 169–194.
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, 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.
9. R. Freund, Gh. Păun: From regulated rewriting to computing with membranes: collapsing hierarchies. *Theoretical Computer Science* **312** (2-3), 143–188 (2004).
10. R. Freund, Gh. Păun, M.J. Pérez-Jiménez: Tissue-like P systems with channel states. *Theoretical Computer Science* **330**, 101–116 (2004).
11. W. Gerstner, W. Kistler: *Spiking Neuron Models. Single Neurons, Populations, Plasticity.* Cambridge Univ. Press, 2002.
12. M. Gheorghe, G. Rozenberg, A. Salomaa, P. Sosík, C. Zandron (Eds.): *15th International Conference, CMC 2014*, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers. Lecture Notes in Computer Science **8961**, Springer, 2014.
13. 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.

14. 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.
15. O.H. Ibarra, A. Păun, Gh. Păun, A. Rodríguez-Patón, P. Sosík, S. Woodworth: Normal forms for spiking neural P systems. In: [14], 105–136, 2006.
16. M. Ionescu, Gh. Păun, T. Yokomori: Spiking neural P systems. *Fundamenta Informaticae* **71** (2 –3), 279–308 (2006).
17. J. van Leeuwen, J. Wiedermann: Computation as an unbounded process. *Theoretical Computer Science* **429**, 202–212 (2012).
18. W. Maass: Computing with spikes. *Special Issue on Foundations of Information Processing of TELEMATIK* **8** (1), 32–36 (2002).
19. W. Maass, C. Bishop (Eds.): *Pulsed Neural Networks*. MIT Press, Cambridge, 1999.
20. C. Martín-Vide, J. Pazos, Gh. Păun, A. Rodríguez-Patón: A new class of symbolic abstract neural nets: tissue P systems. In: *Proceedings of COCOON 2002*, Singapore. Lecture Notes in Computer Science **2387**, Springer, Berlin, 2002, 290–299.
21. M. L. Minsky: *Computation: Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1967.
22. 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`).
23. Gh. Păun: *Membrane Computing. An Introduction.* Springer, 2002.
24. Gh. Păun, Y. Sakakibara, T. Yokomori: P systems on graphs of restricted forms. *Publicationes Mathematicae Debrecen* **60**, 635–660 (2006).
25. Gh. Păun, G. Rozenberg, A. Salomaa (Eds.): *The Oxford Handbook of Membrane Computing.* Oxford University Press, 2010.
26. G. Rozenberg, A. Salomaa (Eds.): *Handbook of Formal Languages,* 3 volumes. Springer, 1997.
27. 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.
28. X. Zhang, B. Luo, X. Fang, L. Pan: Sequential spiking neural P systems with exhaustive use of rules. *BioSystems* **108**, 52–62 (2012).
29. X. Zhang, X. Zeng, L. Pan: On string languages generated by spiking neural P systems with exhaustive use of rules. *Natural Computing* **7** (4), 535–549 (2008).
30. J. Wang, H.J. Hoogeboom, L. Pan, Gh. Păun, M.J. Pérez-Jiménez: Spiking neural P systems with weights. *Neural Computation* **22** (10), 2615–2646 (2010).
31. The P Systems Website: `www.ppage.psystems.eu`.

About Bounded Parameters in P Colonies

Luděk Cienciala and Lucie Ciencialová

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

Abstract. P colonies were introduced in 2004 as an abstract computing device evolved from membrane systems – a biologically motivated computational massive parallel model. P colony is composed of independent single membrane agents, reactively acting and evolving in a shared environment. The computational power of such computing devices is the theme of numerous papers. In this paper we summarize the results obtained for P colonies with bounded number of agents and programs, we improve and add new results for P colonies with capacity two.

1 Introduction

P colonies were introduced in [10] as formal models of a computing device inspired by membrane systems ([13]) and by grammar systems called colonies ([8]). This model intends to structure and functioning of a community of living organisms in a shared environment. The independent organisms living in a P colony are called agents. Each agent is given by a collection of objects embedded in a membrane. The number of objects inside the agent is the same for each one of them. The environment contains several copies of a basic environmental object denoted by e. The number of the copies of e placed in the environment is sufficient for every computation.

A set of programs is associated with each agent. The program determines the activity of the agent by rules. In every moment of computation all the objects inside of the agent are being either evolved (by an evolution rule) or transported (by a communication rule). Two such rules can also be combined into checking rule which specifies two possible actions: if the first rule is not applicable then the second one should be applied. So it sets the priority between two rules.

The computation starts in the initial configuration. Using their programs the agents can change their objects and possibly objects in the environment. This gives possibility to affect the behaviour of the other agents in next steps of computation. In each step of the computation, each agent with at least one applicable program non-deterministically chooses one of them and executes it. The computation halts when no agent can apply any of its programs. The result of the computation is given by the number of some specific objects present at the environment at the end of the computation.

There are several different ways used how to define the initial state of the computation. (1) At the beginning of computation the environment and all agents contain only copies of object e. (2) All the agents can contain various objects at the beginning of computation - the agents are in different initial states. The environment contains only copies of object e. (3) Only environment can contain objects different from the object e.

The P colonies were studied in conjunction with three parameters: (1) the number of objects inside the agent – the capacity of the P colony – (2) the number of agents in P colony – the degree of the P colony – (3) the maximal number of programs associated with one agent – the height of the P colony. In following results the number of necessary agents or the necessary programs stays unbounded to reach computational completeness.

In [7, 9, 10] the authors study P colonies with two objects inside the agents. In this case programs consist of two rules, one for each object. If the former of these rules is an evolution and the latter is a communication or checking, we speak about restricted P colonies. If also another combination of the types of the rules is used, we obtain non-restricted P colonies. The restricted P colonies with the checking rules are computationally complete [7].

In the paper [6] the authors use P colonies with the third type of initial configuration to simulate small universal register machines introduced in [11] to bound all parameters in computationally complete classes of P colonies. This rises a question if the P colonies introduced in results with one "unbounded" parameter cannot be optimized to bounded parameters.

We start with definitions in Section 2. In Section 3 we will deal with P colonies using checking programs with two objects inside each agent. Homogeneous P colonies with capacity two without use of checking programs are studied in Section 4.

2 Definitions

Throughout the paper we assume the reader to be familiar with the basics of the formal language theory. For more information on membrane computing, we recommended [14]. We briefly summarize notations used in the present paper.

We use NRE to denote the family of the recursively enumerable sets of non-negative integers and N to denote the set of non-negative integers.

Let Σ be the alphabet. Let Σ^* be the set of all words over Σ (including the empty word ε). We denote the length of the word $w \in \Sigma^*$ by $|w|$ and the number of occurrences of the symbol $a \in \Sigma$ in w by $|w|_a$.

A multiset of objects M is a pair $M = (V, f)$, where V is an arbitrary (not necessarily finite) set of objects and f is a mapping $f : V \to N$; f assigns to each object in V its multiplicity in M. The set of all finite multisets over the finite set V is denoted by V^*. Any finite multiset M over V can be represented as a string w over alphabet V with $|w|_a = f_M(a)$ for all $a \in V$. Obviously, all words obtained from w by permuting the letters can also represent the same M, and ε represents the empty multiset.

2.1 P colonies

We briefly recall the notion of P colonies introduced in [10]. A P colony consists of agents and environment. Both the agents and the environment contain objects. With every agent the set of programs is associated. There are two types of rules in the programs. The first type, called the evolution, 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, is in the form $c \leftrightarrow d$. When this rule is performed, the object c inside the agent and the object d outside of the agent change their positions, so, after execution of the rule object d appears inside the agent and c is placed outside in the environment.

In [9] the ability of agents was extended by checking rule. This rule gives to the agents an opportunity to opt between two possibilities. It has form r_1/r_2. If the checking rule is performed, the rule r_1 has higher priority to be executed as the rule r_2. It means that the agent checks the possibility to use rule r_1. If it can be executed, the agent has to use it. If the rule r_1 cannot be applied, the agent uses the rule r_2.

Definition 1. *The P colony of the capacity c is a construct*
$$\Pi = (A, e, f, v_E, B_1, \ldots, B_n), \text{ where}$$

- A *is an alphabet of the colony, its elements are called objects,*
- e *is the basic object of the colony,* $e \in A$,
- f *is the final object of the colony,* $f \in A$,
- v_E *is a multiset over* $A - \{e\}$,
- B_i, $1 \leq i \leq n$, *are agents, each agent is a construct* $B_i = (o_i, P_i)$, *where*
  - o_i *is a multiset over A, it determines the initial state (content) of agent* B_i *and* $|o_i| = c$,
  - $P_i = \{p_{i,1}, \ldots, p_{i,k_i}\}$ *is a finite set of programs, where each program contains exactly c rules, which are in one of the following forms each: (1)* $a \rightarrow b$, *called an evolution rule, (2)* $c \leftrightarrow d$, *called a communication rule and (3)* r_1/r_2, *called a checking rule; where* r_1, r_2 *are an evolution or a communication rules.*

The initial configuration of a P colony is an $(n+1)$-tuple of strings of objects present in the P colony at the beginning of the computation. It is given by the multiset O_i for $1 \leq i \leq n$ and by the set V_E. Formally, the configuration of the P colony Π is given by $(w_1, \ldots, w_n, w_E)$, where $|w_i| = k$, $1 \leq i \leq n$, w_i represents all the objects placed inside the i-th agent, and $w_E \in (A - \{e\})^*$ represents all the objects in the environment different from the object e.

At each step of the computation, the contents of the environment and of the agents change in the following manner: In the maximally parallel derivation mode, each agent which can use any of its programs should use one (non-deterministically chosen), whereas in the sequential derivation mode, one agent (non-deterministically chosen from the set of agents with at least one applicable program) uses one of its programs at a time. If the number of applicable programs for chosen agent is higher than one, then the agent non-deterministically chooses one of the programs.

A sequence of transitions is called a computation. A computation is said to be halting, if a configuration is reached where no program can be applied any more. With a halting computation we associate a result which is given as the number of copies of the objects f present in the environment in the halting configuration.

Because of the non-determinism in choosing the programs, starting from the initial configuration we obtain several computations, hence, with a P colony we can associate a set of numbers, denoted by $N(\Pi)$, computed by all possible halting computations of given P colony.

Given a P colony $\Pi = (A,\ e,\ f,\ v_E,\ B_1,\ \ldots,\ B_n)$ the maximal number of programs associated with the agents in P colony Π is called the height of P colony Π. The degree of P colony Π is the number of agents in P colony Π. The third parameter characterizing a P colony is the capacity of P colony Π describing the number of the objects inside each of the agents.

If the programs are composed of one rewriting and one communication (or checking) rule in the case of P colony with capacity two, we call such P colony restricted. Restricted program is in one of following forms: $< a \to b, c \leftrightarrow d >$ and $< a \to b, c \leftrightarrow d / f \leftrightarrow g >$.

Let us use the following notations:
$NPCOL_{par}(c, n, h)$ for the family of all sets of numbers computed by these P colonies working in parallel, using no checking rules and with: the capacity at most c, the degree at most n and the height at most h. If the checking rules are allowed the family of all sets of numbers computed by P colonies is denoted by $NPCOL_{par}K$. If the P colonies are restricted, we use notation $NPCOL_{par}R$ and $NPCOL_{par}KR$, respectively.

2.2 Register machine

In the following we compare the families $NPCOL_{par}(c, n, h)$ with the recursively enumerable sets of numbers. To achieve this aim we use the notion of a register machine.

Definition 2. [12, 11] *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 labeled 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 (labeled with) l_2 or l_3.
- $l_1 : (SUB(r), l_2)$ – If the register r stores the value different from zero, then subtract 1 from its content, otherwise leave it unchanged and go to the instruction labelled l_2.

- $l_1 : (CHECK(r), l_2, l_3)$ – If the value stored in register r is zero, go to the instruction labelled l_2, otherwise go to instruction labelled l_3.
- $l_1 : (CHECKSUB(r), l_2, l_3)$ – If register r is non-empty, then subtract 1 from its content and go to the instruction labelled l_2, otherwise go to instruction labelled l_3.
- l_h : *HALT*– Halt the machine. The final label l_h is only assigned to this instruction.

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 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.[12]) that in this way register machines using *ADD*, *CHECKSUB* and *HALT* instructions compute all Turing computable sets.

In [11] the several results on small universal register machines are presented. In this framework the register machines are used to compute result of the function of non-negative integers by having this argument of the function stored in one of the registers at the beginning of computation and the result can be found in other register after halting computation. The universal machines have eight registers and they can simulate computation of register machine M with the information stored as a natural number $code(M)$ coding the particular machine M. The $code(M)$ is placed in the second register.

Theorem 1. *[11] Let $\mathbb{M}$ be the set of register machines. Then, there are register machines U_1, U_2, U_3 with eight registers and a recursive function $g : \mathbb{M} \to N$ such that for each $M \in \mathbb{M}, N(M) = N(U_i(g(M)))$, where $N(U_i(g(M)))$ denotes the set of numbers computed by $U_i, 1 \leq i \leq 3$, with initially containing $g(M)$ in the second register. All these machines have one HALT instruction labelled by l_h, one instruction of the type ADD labelled l_0, and:*

- *U_1 has $8 + 11 + 13 = 32$ instructions of the type ADD, SUB and CHECK, respectively,*
- *U_2 has $9 + 13 = 22$ instructions of the type ADD and CHECKSUB, respectively,*
- *U_3 has $8 + 1 + 12 = 21$ instructions of the type ADD, CHECK and CHECKSUB, respectively.*

Moreover, these machines either halt using HALT instruction and having the result of the computation in the first register, or their computation on infinitely.

3 Using checking rules in P colonies with capacity two

We open this section with list of results for classes of P colonies with capacity two that the reader can find in literature cited below.

1. $NPCOL_{par}KR(2,*,5) = NRE$ in [5, 10],
2. $NPCOL_{par}R(2,*,5) = NRE$ in [7],
3. $NPCOL_{par}KR(2,1,*) = NRE$ in [7],
4. $NPCOL_{par}R(2,2,*) = NRE$ in [3],
5. $NPCOL_{par}KR(2,23,5) = NPCOL_{par}KR(2,22,6) = NRE$ in [6],
6. $NPCOL_{par}K(2,22,5) = NRE$ in [6],
7. $NPCOL_{par}(2,35,8) = NPCOL_{par}R(2,57,8) = NRE$ in [6].

If we sum the programs associated with one agent in the P colony defined in the proof of the result 3. (we can omit the programs for initialization of simulation generating label l_0) we obtain: $NPCOL_{par}KR(2,1,93) = NRE$ The next theorem determines computational power of P colonies working with checking rules.

Theorem 2. $NPCOL_{par}K(2,1,66) = NRE$.

Proof. Let us consider a register machine M with 8 registers. We construct a P colony $\Pi = (A, e, f, v_E, B)$ simulating the computations of register machine M with:

- $A = \{l_i, l_i' \mid l_i \in H\} \cup \{a_m \mid 1 \leq m \leq 8\}$,
- $v_E = a_2^{g(M)} l_0$
- $f = a_1$,
- $B = (ee, P)$

At the beginning of the computation the agent consumes the object l_0 (the label of starting instruction of M) and generates a_r because the first instruction is of the type *ADD*.

Then it starts to simulate instruction labelled l_0 and it generates the label of the next instruction. The set of programs is as follows:

(1) For the simulation of the initial instruction $l_0 = (ADD(r), l_j, l_k)$ there are programs in P:

$$1 : \langle e \leftrightarrow l_0; e \rightarrow a_r \rangle, \quad 2 : \langle l_0 \rightarrow l_j; a_r \leftrightarrow e \rangle, \quad 3 : \langle l_0 \rightarrow l_k; a_r \leftrightarrow e \rangle$$

The initial configuration of Π is $(ee, ee, l_0 a_2^m), m = g(M)$. After the first step of computation (only the program 1 is applicable) the system enters configuration $(l_0 a_r, ee, a_2^m)$. Now the second or the third program is applicable and agent uses one of them. After the second step the P colony is in the configuration $(ie, ee, a_r a_2^m), i \in \{l_j, l_k\}$.

(2) For every *ADD*-instruction $l_i : (ADD(r), l_j, l_k)$ we add to P the programs:

$$4 : \langle l_i \rightarrow l_i'; e \rightarrow a_r \rangle, \quad 5 : \langle l_i' \rightarrow l_j; a_r \leftrightarrow e \rangle, \quad 6 : \langle l_i' \rightarrow l_k; a_r \leftrightarrow e \rangle$$

When there is object l_i inside the agent, it generates one copy of a_r, puts it to the environment and generates the label of the next instruction (it non-deterministically chooses one of the last two programs 5 and 6)

	B	Env	P
1.	$l_i e$	$a_r^x w$	4
2.	$l_i' a_r$	$a_r^x w$	**5** *or* 6
3.	$l_j e$	$a_r^{x+1} w$	

(3) For every *CHECKSUB*-instruction $l_i : (CHECKSUB(r), l_j, l_k)$, the next programs are added to set P:

$$7 : \langle l_i \to l_i'; e \leftrightarrow a_r / e \leftrightarrow e \rangle, \quad 8 : \langle l_i' \to e; a_r \to l_j \rangle \quad 9 : \langle l_i' \to e; e \to l_k \rangle$$

The simulation of the *CHECKSUB* instruction is done in two steps. In the first step agent uses program no. 7 to check whether there is any copy of object a_r in the environment. In positive case it consumes one a_r. The second step is done in accordance to the content (state) of agent. If it contains a_r agent generates object - label l_j, if there is no a_r inside the agent it generate object - label l_k. Instruction $l_i : (CHECKSUB(r), l_j, l_k)$ is simulated by the following sequence of steps.

If the register r stores nonzero value:

	B	Env	P
1.	$l_i e$	$a_r^x w$	7
2.	$l_i' a_r$	$a_r^{x-1} w$	8
3.	$l_j e$	$a_r^{x-1} w$	

If the register r stores value zero:

	B	Env	P
1.	$l_i e$	w	7
2.	$l_i' e$	w	9
3.	$l_j e$	w	

(4) For *CHECK* instruction we construct three programs similar to previous programs.

$$10 : \langle l_i \to l_i'; e \leftrightarrow a_r / e \leftrightarrow e \rangle, \quad 11 : \langle l_i' \to l_j; a_r \leftrightarrow e \rangle \quad 12 : \langle l_i' \to l_k; e \to e \rangle$$

Instruction $l_i : (CHECK(r), l_j, l_k)$ is simulated by the following sequence of steps.

If the register r stores nonzero value:

	B	Env	P
1.	$l_i e$	$a_r^x w$	10
2.	$l_i' a_r$	$a_r^{x-1} w$	11
3.	$l_j e$	$a_r^x w$	

If the register r stores value zero:

	B	Env	P
1.	$l_i e$	w	10
2.	$l_i' e$	w	12
3.	$l_k e$	w	

(5) For halting instruction l_h no program is added to the set P.

P colony Π correctly simulates all computations of the register machine M and the number contained on the first register of M corresponds to the number of copies of the object a_1 present in the environment of Π. If we count the programs used for simulation of function of register machine we obtain:

$$h = \overbrace{3}^{l_0(ADD)} + \overbrace{8 \cdot 3}^{ADD} + \overbrace{12 \cdot 3}^{CHECKSUB} + \overbrace{1 \cdot 3}^{CHECK} = 66$$

and the proof is complete. □

Now we add result for restricted P colonies with checking programs.

Theorem 3. $NPCOL_{par}KR(2, 1, 74) = NRE$.

Proof. Let us consider a register machine M with 8 registers. We construct a P colony $\Pi = (A, e, f, v_E, B)$ simulating the computations of register machine M with:

- $A = \{e\} \cup \{l_i, l_i' \mid l_i \in H\} \cup \{a_m \mid 1 \leq m \leq 8\}$,
- $v_E = a_2^{g(M)} l_0$; $f = a_1$,
- $B = (ee, P)$

The beginning of simulation is very similar to this one in previous theorem.
(1) For the simulation of the initial instruction $l_0 = (ADD(r), l_j, l_k)$ there are programs in P:

$$1 : \langle e \to a_r; e \leftrightarrow l_0 \rangle, \quad 2 : \langle l_0 \to l_j; a_r \leftrightarrow e \rangle, \quad 3 : \langle l_0 \to l_k; a_r \leftrightarrow e \rangle$$

At the beginning of the computation the agent consumes object l_0 (the label of starting instruction of M) and generates a_r because the first instruction is of the type *ADD*. Then it generates the label of the next instruction.

The initial configuration of Π is $(ee, ee, l_0 a_2^m), m = g(M)$. After the first step of computation (only the program 1 is applicable) the system enters configuration $(l_0 a_r, ee, a_2^m)$. Now the second or the third program is applicable and agent uses one of them. After the second step the P colony is in the configuration $(ie, ee, a_r a_2^m), i \in \{l_j, l_k\}$.

(2) For every *ADD*-instruction $l_i : (ADD(r), l_j, l_k)$ we add to P the programs:

$$4 : \langle e \to e; l_i \leftrightarrow e \rangle, \qquad 5 : \langle e \to a_r; e \leftrightarrow l_i \rangle,$$
$$6 : \langle l_i \to l_j; a_r \leftrightarrow e \rangle \qquad 7 : \langle l_i \to l_k; a_r \leftrightarrow e \rangle$$

When there is object l_i inside the agent, it generates one copy of a_r, puts it to the environment and generates the label of the next instruction (it non-deterministically chooses one of the last two programs 6 and 7)

	B	Env	P
1.	$l_i e$	$a_r^x w$	4
2.	ee	$l_i a_r^x w$	5
3.	$a_r l_i$	$a_r^x w$	**6** *or* 7
4.	$l_j e$	$a_r^{x+1} w$	

(3) For every *CHECKSUB*-instruction $l_i : (CHECKSUB(r), l_j, l_j)$, the next programs are added to set P:

$$8 : \langle l_i \to l_i'; e \leftrightarrow a_r / e \leftrightarrow e \rangle, \quad 9 : \langle a_r \to l_j; l_i' \leftrightarrow e \rangle \quad 10 : \langle l_i' \to l_k; e \leftrightarrow e \rangle$$

The simulation of the *CHECKSUB* instruction is done in two steps. In the first step agent uses program no. 8 to check whether there is any copy of object a_r in the environment. In positive case it consumes one a_r. The second step is done in accordance to the content (state) of agent. If it contains a_r agent generate object - label l_2, if there is no a_r inside the agent it generate object - label l_3. Instruction $l_i : (CHECKSUB(r), l_j, l_k)$ is simulated by the following sequence of steps. $w \in A^*$

If the register r stores nonzero value:

	B	Env	P
1.	$l_i e$	$a_r^x w$	8
2.	$l_i' a_r$	$a_r^{x-1} w$	9
3.	$l_j e$	$a_r^{x-1} l_i' w$	

If the register r stores value zero:

	B	Env	P
1.	$l_i e$	w	8
2.	$l_i' e$	w	10
3.	$l_k e$	w	

(4) For *CHECK* instruction $l_i : (CHECK(r), l_j, l_k)$ we construct there programs similar to previous programs.

$10 : \langle l_i \to l_i'; e \leftrightarrow a_r / e \leftrightarrow e \rangle$, $\quad 11 : \langle l_i' \to l_j; a_r \leftrightarrow e \rangle \quad 12 : \langle l_i' \to l_k; e \leftrightarrow e \rangle$

Instruction $l_i : (CHECK(r), l_j, l_k)$ is simulated by the following sequence of steps.

If the register r stores nonzero value:

	B	Env	P
1.	$l_i e$	$a_r^x w$	10
2.	$l_i' a_r$	$a_r^{x-1} w$	11
3.	$l_j e$	$a_r^x w$	

If the register r stores value zero:

	B	Env	P
1.	$l_i e$	w	10
2.	$l_i' e$	w	12
3.	$l_k e$	w	

(5) For halting instruction l_h no program is added to the set P.

P colony Π correctly simulates all computations of the register machine M and the number contained on the first register of M corresponds to the number of copies of the object a_1 present in the environment of Π. If we count the programs used for simulation of function of register machine we obtain:

$$h = \overbrace{3}^{l_0(ADD)} + \overbrace{8 \cdot 4}^{ADD} + \overbrace{12 \cdot 3}^{CHECKSUB} + \overbrace{1 \cdot 3}^{CHECK} = 74$$

and the proof is complete. □

4 Bounded classes of homogeneous P colonies

The program is said to be homogeneous if it is composed of rules of the same type. P colony having only homogeneous programs is called homogeneous. Each P colony with capacity one that does not use checking rules is homogeneous. Let us summarize results found in papers cited below and add to them the third parameter that we can count from proofs of the theorems:

- $NPCOL_{par}KH(1, *, 6) = NPCOL_{par}KH(1, 26, 6) = NRE$ in [4]
- $NPCOL_{par}KH(2, *, 4) = NPCOL_{par}KH(2, 25, 4) = NRE$ in [4]
- $NPCOL_{par}KH(2, 1, *) = NPCOL_{par}KH(2, 1, 176) = NRE$ in [4]
- $NPCOL_{par}KH(3, 2, *) = NPCOL_{par}KH(3, 2, 236) = NRE$ in [2]

It seems that no result is published related to homogeneous P colonies with capacity two that do not use checking rules.

Theorem 4. $NPCOL_{par}H(2, 2, 163) = NRE$.

Proof. Let us consider a register machine M with 8 registers. We construct a P colony $\Pi = (A, e, f, v_E, B_1, B_2)$ simulating the computations of register machine M with:

- $A = \{e, e'\} \cup \{l_i, l_i', l_i''', \overline{l_i} \mid l_i \in H\} \cup \{a_m \mid 1 \leq m \leq 8\}$,
- $v_E = a_2^{g(M)} l_0$, $f = a_1$,
- $B_i = (ee, P_n), n = \{1, 2\}$

At the beginning of the computation the agent B_1 consumes the object l_0 (the label of starting instruction of M).
(1) For the simulation of the initial instruction $l_0 = (ADD(r), l_j, l_k)$ there are programs in P_1:

$$1 : \langle e \leftrightarrow l_0; e \leftrightarrow e \rangle, \quad 2 : \langle l_0 \rightarrow l_0'; e \rightarrow a_r \rangle, \quad 3 : \langle l_0' \leftrightarrow e; a_r \leftrightarrow e \rangle$$
$$4 : \langle e \leftrightarrow l_0'; e \leftrightarrow e \rangle, \quad 5 : \langle l_0' \rightarrow l_j; e \rightarrow e \rangle, \quad 6 : \langle l_0' \rightarrow l_k; e \rightarrow e \rangle$$

The initial configuration of Π is $(ee, ee, l_0 a_2^m), m = g(M)$. Agent B_1 consumes object L_0 and then it starts to simulate instruction labelled l_0. It generates the label of the next instruction. Because each program is homogeneous the agent can only rewrite all its content or exchange both objects inside it for another two objects from the environment. If the content of agent B_1 is ee only programs with communication rules are applicable.
(2) For every ADD-instruction $l_i : (ADD(r), l_j, l_k)$ we add to P_1 the programs:

$$7 : \langle l_i \rightarrow l_i'; e \rightarrow a_r \rangle, \quad 8 : \langle l_i' \leftrightarrow e; a_r \leftrightarrow e \rangle, \quad 9 : \langle e \leftrightarrow l_i'; e \leftrightarrow e \rangle$$
$$10 : \langle l_i' \rightarrow l_j; e \rightarrow e \rangle, \quad 11 : \langle l_i' \rightarrow l_j; e \rightarrow e \rangle,$$

When there is object l_i inside the agent, it generates one copy of a_r, puts it to the environment and generates the label of the next instruction (it non-deterministically chooses one of the last two programs 10 and 11).

	B_1	B_2	Env	P_1	P_2
1.	$l_i e$	ee	w	7	–
2.	$l_i' a_r$	ee	w	8	–
3.	ee	ee	$l_i' a_r w$	9	–
4.	$l_i' e$	ee	$a_r w$	**10** *or* 11	–
6.	$l_j e$	ee	$a_r w$	–	–

(3) For every $CHECKSUB$-instruction $l_i : (CHECKSUB(r), l_j, l_k)$, the next programs are added to sets P_1 and P_2:

P_1
$$12 : \langle l_i \rightarrow l_i'; e \rightarrow l_i'' \rangle, \quad 13 : \langle l_i' \leftrightarrow e; l_i'' \leftrightarrow e \rangle, \quad 14 : \langle e \leftrightarrow l_i'; e \leftrightarrow a_r \rangle,$$
$$15 : \langle l_i'' \rightarrow \overline{l_i}; a_r \rightarrow e' \rangle, \quad 16 : \langle \overline{l_i} \leftrightarrow e; e' \leftrightarrow e \rangle, \quad 17 : \langle e \leftrightarrow \overline{l_i}; e \leftrightarrow e \rangle,$$
$$18 : \langle \overline{l_i} \rightarrow l_j; e \rightarrow e \rangle, \quad 19 : \langle e \leftrightarrow l_i''; e \leftrightarrow l_i''' \rangle, \quad 20 : \langle l_i'' \rightarrow l_k; l_i''' \rightarrow e \rangle$$

P_2
$$21 : \langle e \leftrightarrow l_i'; e \leftrightarrow e \rangle, \quad 22 : \langle l_i' \rightarrow l_i'''; e \rightarrow e \rangle, \quad 23 : \langle l_i''' \leftrightarrow e; e \leftrightarrow e \rangle,$$
$$24 : \langle e \leftrightarrow l_i'''; e \leftrightarrow e' \rangle, \quad 25 : \langle l_i''' \rightarrow e; e' \rightarrow e \rangle$$

The simulation of the $CHECKSUB$ instruction is following: Agent B_1 puts to object (l_i', l_i'') corresponding to given instruction to the environment; object l_i' is consumed by agent B_2; in the next step object l_i'' can be consumed only together with object a_r. If there is no a_r in the environment, agent B_1 has to wait until agent B_2 puts object l_i''' to the environment. Now program 19 is applicable. The next step is done in accordance to the content (state) of agent. If it contents a_r agent generate object - label l_2 and put object l_i' to the environment, if there is no a_r inside the agent it generate object - label l_3. Instruction $l_i : (CHECKSUB(r), l_j, l_k)$ is simulated by the following sequence of steps. Multiset $w \in \{a_m \mid 1 \leq m \leq 8\}^*$ is placed in the environment.

If the register r stores non-zero value:

	B_1	B_2	Env	P_1	P_2
1.	$l_i e$	ee	$a_r^x w$	12	–
2.	$l_i' l_i''$	ee	$a_r^x w$	13	–
3.	ee	ee	$l_i' l_i'' a_r^x w$	14	21
4.	$l_i'' a_r$	$l_i' e$	$a_r^{x-1} w$	15	22
5.	$\overline{l_i} e'$	$l_i''' e$	$a_r^{x-1} w$	16	23
6.	ee	ee	$\overline{l_i} e' l_i''' a_r^{x-1} w$	17	24
7.	$\overline{l_i} e$	$l_i''' e'$	$a_r^{x-1} w$	18	25
8.	$l_j e$	ee	$a_r^{x-1} w$	–	–

If the register r stores value zero:

	B_i	B_2	Env	P_1	P_2
1.	$l_i e$	ee	w	12	–
2.	$l_i' l_i''$	ee	w	13	–
3.	ee	ee	$l_i' l_i'' w$	–	21
4.	ee	$l_i' e$	$l_i'' w$	–	22
5.	ee	$l_i''' e$	$l_i'' w$	–	23
6.	ee	ee	$l_i''' l_i'' w$	19	–
7.	$l_i''' l_i''$	ee	w	20	–
8.	$l_k e$	ee	w	–	–

(4) For *CHECK* instruction we construct there programs similar to previous programs for *CHECKSUB* instruction.

P_1 $26 : \langle l_i \to l_i'; e \to l_i'' \rangle$, $27 : \langle l_i' \leftrightarrow e; l_i'' \leftrightarrow e \rangle$, $28 : \langle e \leftrightarrow l_i''; e \leftrightarrow a_r \rangle$,
$29 : \langle l_i'' \to \overline{l_i}; a_r \to a_r \rangle$, $30 : \langle \overline{l_i} \leftrightarrow e; a_r \leftrightarrow e \rangle$, $31 : \langle e \leftrightarrow \overline{l_i}; e \leftrightarrow l_i''' \rangle$,
$32 : \langle \overline{l_i} \to l_j; l_i''' \to e \rangle$, $33 : \langle e \leftrightarrow l_i''; e \leftrightarrow l_i''' \rangle$, $34 : \langle l_i'' \to l_k; l_i''' \to e \rangle$

P_2 $35 : \langle e \leftrightarrow l_i'; e \leftrightarrow e \rangle$, $36 : \langle l_i' \to l_i'''; e \to e \rangle$, $37 : \langle l_i''' \leftrightarrow e; e \leftrightarrow e \rangle$

Instruction $l_i : (CHECK(r), l_2, l_3)$ is simulated by the following sequence of steps.

If the register r stores non-zero value:

	B_1	B_2	Env	P_1	P_2
1.	$l_i e$	ee	$a_r^x w$	26	–
2.	$l_i' l_i''$	ee	$a_r^x w$	27	–
3.	ee	ee	$l_i' l_i'' a_r^x w$	28	35
4.	$l_i'' a_r$	$l_i' e$	$a_r^{x-1} w$	29	36
5.	$\overline{l_2} a_r$	$l_i''' e$	$a_r^{x-1} w$	30	37
6.	ee	ee	$\overline{l_2} l_i''' a_r^x w$	31	–
7.	$\overline{l_2} l_i'''$	ee	$a_r^x w$	32	–
8.	$l_j e$	ee	$a_r^x w$	–	–

If the register r stores value zero:

	B_1	B_2	Env	P_1	P_2
1.	$l_i e$	ee	w	26	–
2.	$l_i' l_i''$	ee	w	27	–
3.	ee	ee	$l_i' l_i'' w$	–	35
4.	ee	$l_i' e$	$l_i'' w$	–	36
5.	ee	$l_i''' e$	$l_i'' w$	–	37
6.	ee	ee	$l_i''' l_i'' w$	33	–
7.	$l_i''' l_i''$	ee	w	34	–
8.	$l_k e$	ee	w	–	–

(5) For halting instruction l_h no program is added to the set P.

P colony Π correctly simulates all computations of the register machine M and the number contained on the first register of M corresponds to the number of copies of the object a_1 present in the environment of Π. If we count the programs used for simulation of function of register machine we obtain:

$$h = \max \left\{ \begin{array}{lccccccc} & l_0(ADD) & & ADD & & CHECKSUB & & CHECK \\ h_1 = & \overbrace{6} & + & \overbrace{8 \cdot 5} & + & \overbrace{12 \cdot 9} & + & \overbrace{1 \cdot 9} \\ & l_0(ADD) & & ADD & & CHECKSUB & & CHECK \\ h_2 = & \overbrace{0} & + & \overbrace{8 \cdot 0} & + & \overbrace{12 \cdot 5} & + & \overbrace{1 \cdot 3} \end{array} \right\} = 163$$

and the proof is complete. □

In next result we minimize number of programs associated with agent.

Theorem 5. $NPCOL_{par}H(2, 92, 3) = NRE$.

Proof. Let us consider a register machine M with 8 registers. We construct a P colony $\Pi = (A, e, f, v_E, B_1, B_2)$ simulating the computations of register machine M with:

- $A = \{e, e'\} \cup \{l_i, l_i', l_i''', \overline{l_i} \mid l_i \in H\} \cup \{a_m \mid 1 \leq m \leq 8\}$,
- $v_E = a_2^{g(M)} l_0$, $f = a_1$,
- $B_i = (ee, P_i), i = \{1, \ldots, 92\}$

Because we want to minimize the number of programs associated with each agent we have to divide simulation of each instruction among more agents. To set order among agents we use the following labelling: $B_{l_i,j}$ implies that this is j-th agent associated with instruction l_i.

At the beginning of the computation the agent B_1 consumes the object l_0 (the label of starting instruction of M).

(1) For the simulation of the initial instruction $l_0 = (ADD(r), l_j, l_k)$ and every ADD instruction $l_i = (ADD(r), l_j, l_k)$ there are programs in $P_{l_i,p}, p = \{1, 2, 3\}$. To simulate the ADD instruction we need three agents : one agent to generate object a_r and two agents to generate object – label of the next instruction.

$P_{l_i,1}$ $1 : \langle e \leftrightarrow l_i; e \leftrightarrow e \rangle$, $2 : \langle l_i \rightarrow l_i'; e \rightarrow a_r \rangle$, $3 : \langle l_i' \leftrightarrow e; a_r \leftrightarrow e \rangle$

$P_{i,2}$ $4 : \langle e \leftrightarrow l_i'; e \leftrightarrow e \rangle$, $5 : \langle l_i' \rightarrow l_2; e \rightarrow e \rangle$, $6 : \langle l_2 \leftrightarrow e; e \leftrightarrow e \rangle$

$P_{i,3}$ $7 : \langle e \leftrightarrow l_i'; e \leftrightarrow e \rangle$, $8 : \langle l_i' \rightarrow l_3; e \rightarrow e \rangle$, $9 : \langle l_3 \leftrightarrow e; e \leftrightarrow e \rangle$

In the following table the reader can find a part of computation – simulation of execution of initial instruction l_0 – sequence of configurations and the labels of used programs.

	$B_{i,1}$	$B_{i,2}$	$B_{i,3}$	Env	$P_{1,1}$	$P_{i,2}$	$P_{i,3}$
1.	ee	ee	ee	wl_i	1	–	–
2.	l_ie	ee	ee	w	2	–	
3.	$l_i'a_r$	ee	ee	w	3	–	
4.	ee	ee	ee	$l_i'a_rw$	–	**4**	or 7
5.	ee	$l_i'e$	ee	a_rw	–	5	–
6.	ee	l_je	ee	a_rw	–	6	–
7.	ee	ee	ee	l_ja_rw	–	–	–

If the agent $B_{i,3}$ uses the program 7 in the configuration 4, the label l_3 is generated instead of l_2.

(2) For every $CHECKSUB$ instruction $l_i : (CHECKSUB(r), l_j, l_k)$, the next programs are added to sets $P_{l_i,p}, p \in \{1, \ldots, 5\}$:

$P_{l_i,1}$ $10 : \langle e \leftrightarrow l_i; e \leftrightarrow e \rangle$, $11 : \langle l_i \rightarrow l_i'; e \rightarrow l_i'' \rangle$, $12 : \langle l_i' \leftrightarrow e; l_i'' \leftrightarrow e \rangle$

$P_{i,2}$ $13 : \langle e \leftrightarrow l_i'; e \leftrightarrow a_r \rangle$, $14 : \langle l_i' \rightarrow l_j; a_r \rightarrow \overline{l_i} \rangle$, $15 : \langle l_j \leftrightarrow e; \overline{l_i} \leftrightarrow e \rangle$

$P_{i,3}$ $16 : \langle e \leftrightarrow l_i''; e \leftrightarrow e \rangle$, $17 : \langle l_i' \rightarrow l_i'''; e \rightarrow e \rangle$, $18 : \langle l_i''' \leftrightarrow e; e \leftrightarrow e \rangle$

$P_{i,4}$ $19 : \langle e \leftrightarrow l_i'; e \leftrightarrow l_i''' \rangle$, $20 : \langle l_i' \rightarrow l_k; l_i''' \rightarrow e \rangle$, $21 : \langle l_k \leftrightarrow e; e \leftrightarrow e \rangle$

$P_{i,5}$ $22 : \langle e \leftrightarrow l_i'''; e \leftrightarrow \overline{l_i} \rangle$, $23 : \langle l_i'''' \rightarrow e; \overline{l_i} \rightarrow e \rangle$,

The simulation of the *CHECKSUB* instruction is following: Agent $B_{l_i,1}$ puts objects (l'_i, l''_i) corresponding to given instruction to the environment; object l'_i is consumed by agent $B_{l_i,2}$ iff there is at least one copy of a_r in the environment; in positive case agent rewrites these objects to object corresponding to label of the next instruction and one more object $(\overline{l_i})$ – the message for agent $B_{l_i,5}$ that the unused object l'''_i must be erased from the environment. The agent $B_{l_i,3}$ consumes object l''_i and in the next step agent rewrites it to object l'''_i and in the following step agent puts this object to the environment. In the case that register r stores value zero, agent $B_{l_1,4}$ consumes objects l'_i and l'''_i and finally it generates object - label l_k. Instruction $l_i : (CHECKSUB(r), l_j, l_k)$ is simulated by the following sequence of steps. Multiset $w \in \{a_m \mid 1 \leq m \leq 8\}^*$ is placed in the environment.

If the register r stores non-zero value:

	$B_{l_i,1}$	$B_{l_i,2}$	$B_{l_i,3}$	$B_{l_i,4}$	$B_{l_i,5}$	Env	$P_{l_i,1}$	$P_{l_i,2}$	$P_{l_i,3}$	$P_{l_i,4}$	$P_{l_i,5}$
1.	ee	ee	ee	ee	ee	$l_i a_r^x w$	10	–	–	–	–
2.	$l_i e$	ee	ee	ee	ee	$a_r^x w$	11	–	–	–	–
3.	$l'_i l''_i$	ee	ee	ee	ee	$a_r^x w$	12	–	–	–	–
4.	ee	ee	ee	ee	ee	$l'_i l''_i a_r^x w$	–	13	16	–	–
5.	ee	$l'_i a_r$	$l''_i e$	ee	ee	$a_r^{x-1} w$	–	14	17	–	–
6.	ee	$l_j \overline{l_i}$	$l'''_i e$	ee	ee	$a_r^{x-1} w$	–	15	18	–	–
7.	ee	ee	ee	ee	ee	$l_j \overline{l_i} l'''_i a_r^{x-1} w$	–	–	–	–	22
8.	ee	ee	ee	ee	$\overline{l_i} l'''_i$	$a_r^{x-1} w$	–	–	–	–	23
9.	ee	ee	ee	ee	ee	$a_r^{x-1} w$	–	–	–	–	–

If the register r stores value zero:

	$B_{l_i,1}$	$B_{l_i,2}$	$B_{l_i,3}$	$B_{l_i,4}$	$B_{l_i,5}$	Env	$P_{l_i,1}$	$P_{l_i,2}$	$P_{l_i,3}$	$P_{l_i,4}$	$P_{l_i,5}$
1.	ee	ee	ee	ee	ee	$l_i w$	10	–	–	–	–
2.	$l_i e$	ee	ee	ee	ee	w	11	–	–	–	–
3.	$l'_i l''_i$	ee	ee	ee	ee	w	12	–	–	–	–
4.	ee	ee	ee	ee	ee	$l'_i l''_i w$	–	–	16	–	–
5.	ee	ee	$l''_i e$	ee	ee	$l'_i w$	–	–	17	–	–
6.	ee	ee	$l'''_i e$	ee	ee	$l'_i w$	–	–	18	–	–
7.	ee	ee	ee	ee	ee	$l'_i l'''_i w$	–	–	–	19	–
8.	ee	ee	ee	$l'_i l'''_i$	ee	w	–	–	–	20	–
9.	ee	ee	ee	$l_k e$	ee	w	–	–	–	21	–
10.	ee	ee	ee	ee	ee	$l_k w$	–	–	–	–	–

(3) For *CHECK* instruction we construct three programs similar to programs in previous paragraph. The only change is in programs associated with agent $B_{l_i,5}$.

$$P_{i,5}\ 22 : \left\langle e \leftrightarrow l'''_i; e \leftrightarrow \overline{l_i} \right\rangle, \quad 23 : \left\langle l'''_i \rightarrow e; \overline{l_i} \rightarrow a_r \right\rangle, \quad 24 : \langle a_r \leftrightarrow e; e \leftrightarrow e \rangle$$

(4) For halting instruction l_h no program is added to the set P.

P colony Π correctly simulates all computations of the register machine M and the number contained on the first register of M corresponds to the number of copies of the object a_1 present in the environment of Π. If we count the

programs used for simulation of function of register machine we obtain:

$$n = \overbrace{9 \cdot 3}^{ADD(+l_0)} + \overbrace{12 \cdot 5}^{CHECKSUB} + \overbrace{1 \cdot 5}^{CHECK} = 92$$

and the proof is complete. □

Theorem 6. $NPCOL_{par}H(2, 70, 5) = NRE$.

It is very easy to see that we obtain the result by union of agents $B_{l_i,2}$ and $B_{l_i,3}$ constructed in previous proof for ADD-instructions.

5 Conclusions

In this paper we focused on P colonies with all bounded parameters with capacity two. The first we improve results for P colonies with checking rules. In the second part we focus on homogeneous P colonies without use of checking programs. We can summarize our results in following list:

- $NPCOL_{par}K(2, 1, 66) = NRE$
- $NPCOL_{par}KR(2, 1, 74) = NRE$
- $NPCOL_{par}H(2, 2, 163) = NRE$
- $NPCOL_{par}H(2, 92, 3) = NRE$
- $NPCOL_{par}H(2, 70, 5) = NRE$

For more information on membrane computing, see [14]; for more on computational machines and colonies in particular, see [12] and [8–10], respectively. Activities carried out in the field of membrane computing are currently numerous and they are available also at [15].

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 SGS/6/2014.

References

1. Ciencialová, L., Cienciala, L.: *Variations on the theme: P colonies*, In Kolář, D., Meduna, A. (Eds.): Proceedings of the 1^{st} International workshop WFM'06, Ostrava, Czech Republic, 2006, pp. 27–34
2. Cienciala, L., Ciencialová, L.: *P colonies and their extensions* In Kelemen, J., Kelemenová, A. (Eds.): Computation, cooperation, and life. Springer-Verlag, Berlin, Heidelberg (2011) 158–169.
3. Ciencialová, L. Cienciala, L., Kelemenová, A.: *On the number of agents in P colonies*, In G. Eleftherakis, P. Kefalas, and G. Paun (eds.), *Proceedings of the 8th Workshop on Membrane Computing (WMC'07)*, June 25-28, Thessaloniki, Greece, 2007, pp. 227–242.
4. L. Cienciala, L. Ciencialová, A. Kelemenová, Homogeneous P colonies, *Computing and informatics*, Vol. **27**, 481–496, (2008).

5. Csuhaj-Varjú, E., Kelemen, J., Kelemenová, A., Păun, Gh., Vaszil, G.: *Cells in environment: P colonies*, Journal of Multiple-valued Logic and Soft Computing **12**, 3-4 (2006) 201–215
6. Csuhaj-Varjú, E., Margenstern, M., Vaszil, G.: *P colonies with a bounded number of cells and programs.* Pre-Proceedings of the 7^{th} Workshop on Membrane Computing (H. J. Hoogeboom, Gh. Păun, G. Rozenberg, eds.), Leiden, The Netherlands (2006) 311–322
7. Freund, R., Oswald, M.: *P colonies working in the maximally parallel and in the sequential mode.* Pre-Proceedings of the 1^{st} International Workshop on Theory and Application of P Systems (G. Ciobanu, Gh. Păun, eds.), Timisoara, Romania (2005) 49–56
8. Kelemen, J., Kelemenová, A.: *A grammar-theoretic treatment of multi-agent systems.* Cybernetics and Systems **23** (1992) 621–633
9. Kelemen, J., Kelemenová, A.: *On P colonies, a biochemically inspired model of computation.* Proc. of the 6^{th} International Symposium of Hungarian Researchers on Computational Intelligence, Budapest TECH, Hungary (2005) 40–56
10. Kelemen, J., Kelemenová, A., Păun, Gh.: *Preview of P colonies: A biochemically inspired computing model.* Workshop and Tutorial Proceedings, Ninth International Conference on the Simulation and Synthesis of Living Systems, ALIFE IX (M. Bedau at al., eds.) Boston, Mass. (2004) 82–86
11. Korec, I.: *Small universal register machines.* Theoretical Computer Science **168** (1996) 267–301
12. Minsky, M. L.: *Computation: Finite and Infinite Machines.* Prentice Hall, Englewood Cliffs, NJ (1967)
13. Păun, Gh.: *Computing with membranes.* Journal of Computer and System Sciences **61** (2000) 108–143
14. Păun, Gh.: *Membrane computing: An introduction.* Springer-Verlag, Berlin (2002)
15. *P systems web page.* 15 Jan. 2001. 7 Sept. 2007 <http://psystems.disco.unimib.it>

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, TU Wien
Favoritenstraße 9-11, 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

Abstract. The set of **NP**-complete problems is divided 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 systems model.

1 Introduction

In [10], 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 this P systems class without antimatter and matter/antimatter annihilation rules is exactly the complexity class **P** (see [13]).

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 [11]. 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 [7], and it has been widely used in membrane computing to prove the ability of a P system model to solve **NP**-complete problems (e.g. [12, 14, 15, 17, 20, 21]).

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 systems 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 speak about the results in P systems found in the literature on the power and the limitations of antimatter. In Section 4, 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 5, some basics on recognizer P systems are recalled, and in Section 6 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 [3], attracted the attention of researchers. 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 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, see [4], membrane creation rules, see [19], (or string replication rules, but string-objects lie outside of the scope of the current paper); in tissue P systems, one could apply a 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 [6]. 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 [2] is matter/antimatter annihilation. What are the strengths and the weaknesses of these possible ingredients (the weaker is a combination of ingredients, the stronger is the result, while sometimes weaker ingredients do not let us do what stronger ones 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 $\mathbf{P^{PP}}$=**PSPACE**, see [16]), 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.

In the the present paper we focus on using matter/antimatter annihilation rules.

3 Antimatter Overview

The idea of matter/antimatter annihilation rules in P systems initially appeared in [2] as an adaptation of the idea of anti-spikes in spiking neural P systems, see [**?**], to the model of transitional P systems and later to the model of P systems with active membranes. It turned out that combining annihilation rules, which are a specific form of cooperative erasing, with non-cooperative rules yields an elegant computationally complete model. Note that immediate annihilation precisely corresponds to *weak priority* of annihilation. It has been shown that this priority may be removed at the price of adding *one* catalyst. Then, it has also been shown that P systems with non-cooperative rules and matter/antimatter annihilation are computationally complete even in the deterministic case. A variant with annihilation generating energy was also considered in [2].

The work of [2] has been continued in [1]. In particular, the computational completeness results were generalized to computing vectors over $\mathbb{Z}$ instead of $\mathbb{N}$, as well as to computing languages, or even subsets of groups (as languages over symbols and anti-symbols).

A number of small universality results was obtained in [3], in particular, a universal accepting P system with 53 rules, simulating a model called generalized counter automata introduced there for that purpose.

Besides being studied for computational completeness and small universalities, matter/antimatter annihilation rules have been considered in the model of P systems with active membranes. While it has been recently shown in [9] that without the weak priority of annihilation, only the complexity class **P** is characterized within the framework of recognizer P systems, under the basic settings (i.e., with this weak priority), uniform families of recognizer P systems with active membranes solve *Subset-Sum*, a known **NP**-complete problem, and in the current paper we present a solution to **SAT**, a known *strongly* **NP**-complete problem.

4 The P Systems Model

In this paper, we use the usual rules of evolution, communication and division of elementary membranes which are common 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 [18, 22, 26, 27]. 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 is an active research area [1, 3, 10].

Inspired by physics, we consider the annihilation of two objects a and b from the alphabet O 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 (weak) priority over all rules of the other types of rules (see [10]).

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 [5], we use subscript 0 for the rule type to represent a restriction that such a rule does not depend on the polarization and is not allowed to change it; if all rules have this subscript, this is equivalent to saying that the P system is without polarizations):

(a_0) $[\ a \to u\ ]_h$ for $h \in H$, $a \in O$, $u \in O^*$. This is an object evolution rule, associated with a membrane labeled by h: an object $a \in O$ belonging to that membrane evolves to a multiset represented by the string $u \in O^*$.

(b_0) $a[\ \ ]_h \to [\ b\ ]_h$ for $h \in H$, $a, b \in O$. 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) $[\ a\ ]_h \to b[\ \ ]_h$ for $h \in H$, $a, b \in O$. An object is sent out from a membrane labeled by h to the region immediately outside, possibly being transformed into another object.

(e_0) $[\ a\ ]_h \to [\ b\ ]_h[\ c\ ]_h$ for $h \in H$, $a, b, c \in O$. 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) $[\ a\overline{a} \to \lambda\ ]_h$ 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. 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.

5 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 see, for example, [23, 24]. 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.

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 $\theta_X(w) = 1$, and $\mathbf{\Pi}$ is *complete* with regard to X if for each instance of the problem $w \in I_X$, provided that $\theta_X(w) = 1$, then every computation of $\Pi(w)$ is an accepting computation.

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).

6 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 uniform 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 [8] and [25]):

- *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 system sends out the correct answer to the environment.

Let us consider the pairing 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 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:

$$\begin{aligned}
&\Pi = (O, \Sigma, H = \{1,2\}, \mu = [\ [\ \ ]_2\ ]_1, w_1, w_2, R, i_{in} = 2), \text{ where}\\
&\Sigma = \{x_{i,j}, y_{i,j} \mid 1 \le i \le m,\ 1 \le j \le n\},\\
&O = \{d, t, f, F, \overline{F}, T, \overline{no}_{n+5}, \overline{F}_{n+5}, \overline{yes}_{n+6}, yes_{n+6}, no_{n+6}, yes, no\}\\
&\quad \cup \{x_{i,j}, y_{i,j} \mid 1 \le i \le m,\ -1 \le j \le n\} \cup \{\overline{x}_{i,-1}, \overline{y}_{i,-1} \mid 1 \le i \le m\}\\
&\quad \cup \{c_i, \overline{c}_i \mid 1 \le i \le m\} \cup \{e_j \mid 1 \le j \le n+3\}\\
&\quad \cup \{yes_j, no_j, F_j \mid 0 \le j \le n+5\},\\
&w_1 = no_0\ yes_0\ F_0,\ \ w_2 = d^n\ e_1,
\end{aligned}$$

and the rules of the set R are given below, presented in the groups Generation, Input Processing, Checking, and Output, together with explanations about 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. $[\ \overline{x}_{i,-1} \to \lambda\ ]_2$, $1 \leq i \leq m$;
G5. $[\ \overline{y}_{i,-1} \to \lambda\ ]_2$, $1 \leq i \leq m$.

In each step j, $1 \leq j \leq n$, every elementary membrane is divided, one new 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. $[\ x_{i,j} \to x_{i,j-1}\ ]_2$, $1 \leq i \leq m$, $0 \leq j \leq n$;
I2. $[\ y_{i,j} \to y_{i,j-1}\ ]_2$, $1 \leq i \leq m$, $0 \leq j \leq n$;
I3. $[\ x_{i,-1}\ \overline{x}_{i,-1} \to \lambda\ ]_2$, $1 \leq i \leq m$;
I4. $[\ y_{i,-1}\ \overline{y}_{i,-1} \to \lambda\ ]_2$, $1 \leq i \leq m$;
I5. $[\ x_{i,-1} \to c_i\ ]_2$, $1 \leq i \leq m$;
I6. $[\ y_{i,-1} \to c_i\ ]_2$, $1 \leq i \leq m$.

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 takes $n+2$ 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. $[\ yes_j \to yes_{j+1}\ ]_1$, $0 \leq j \leq n+5$;
O2. $[\ no_j \to no_{j+1}\ ]_1$, $0 \leq j \leq n+5$;
O3. $[\ F_j \to F_{j+1}\ ]_1$, $0 \leq j \leq n+4$;
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 send out the negative answer *no*, while the initial object F counts up to $n+5$, generates the antimatter object for the *yes*-object at stage $n+6$ and annihilates with the corresponding object *yes* at stage $n+6$. On the other hand, if (at least one) object T arrives in the skin, then the object *no* is annihilated at stage $n+5$ before it would be sent out in the next step, and the object F is annihilated before it could annihilate with the object *yes*, 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.

7 Conclusions

Although the ability for solving **NP**-complete problems with this kind of P systems was proved in [10], to the best of our knowledge this is the first solution for 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 further investigations under these specific 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. A. Alhazov, B. Aman, R. Freund: P systems with anti-matter. In: M. Gheorghe, G. Rozenberg, A. Salomaa, P. Sosík, C. Zandron (Eds.): *Membrane Computing - 15th International Conference, CMC 2014, Prague, 2014, Revised Selected Papers*. Lecture Notes in Computer Science **8961**, Springer, 2014, 66–85.
2. A. Alhazov, B. Aman, R. Freund, Gh. Păun: Matter and anti-matter in membrane systems. In: L.F. Macías-Ramos, M.A. Martínez-del-Amor, Gh. Păun, A. Riscos-Núñez, L. Valencia-Cabrera (Eds.): *Twelfth Brainstorming Week on Membrane Computing*. RGNC report 1/2014, University of Seville, Fénix Editora, Sevilla, 2014, 1–26.
3. A. Alhazov, B. Aman, R. Freund, Gh. Păun: Matter and anti-matter in membrane systems. In: H. Jürgensen, J. Karhumäki, A. Okhotin (Eds.): *Descriptional Complexity of Formal Systems 2014*. Lecture Notes in Computer Science **8614**, Springer, 2014, 65–76.
4. A. Alhazov, Ts.-O. Ishdorj: Membrane operations in P systems with active membranes. In: Gh. Păun, A. Riscos-Núñez, A. Romero-Jimenez, F. Sancho-Caparrini (Eds.): *Second Brainstorming Week on Membrane Computing*. RGNC report 01/2004, University of Seville, Sevilla, 2004, 37–44.
5. A. Alhazov, L. Pan, Gh. Păun: Trading polarizations for labels in P systems with active membranes. *Acta Informatica* **41** (2-3), 111–144 (2004).
6. A. Alhazov, M.J. Pérez-Jiménez: Uniform solution of QSAT using polarizationless active membranes. In: J. Durand-Lose, M. Margenstern (Eds.): *5th International Conference, MCU 2007, Orléans, France, September 10-13, 2007. Proceedings*. Lecture Notes in Computer Science **4664**, Springer, 2007, 122–133.
7. S.A. Cook: The complexity of theorem-proving procedures. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC '71*. ACM, New York, NY, 1971, 151–158.
8. A. Cordón-Franco, M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, F. Sancho-Caparrini: A Prolog simulator for deterministic P systems with active membranes. *New Generation Computing* **22** (4), 349–363 (2004).
9. D. Díaz-Pernil, R. Freund, M.A. Gutiérrez-Naranjo, A. Leporati: On the semantics of annihilation rules in membrane computing. *16th International Conference on Membrane Computing*, accepted paper, 2015.
10. D. Díaz-Pernil, F. Peña-Cantillana, A. Alhazov, R. Freund, M.A. Gutiérrez-Naranjo: Antimatter as a frontier of tractability in membrane computing. *Fundamenta Informaticae* **134** (1-2), 83–96 (2014).
11. M.R. Garey, D.S. Johnson: *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.
12. Zs. Gazdag, G. Kolonits: A new approach for solving SAT by P systems with active membranes. In: E. Csuhaj-Varjú, M. Gheorghe, G. Rozenberg, A. Salomaa, Gy. Vaszil (Eds.): *International Conference on Membrane Computing. 13th International Conference, CMC 2012, Budapest, Hungary, August 28-31, 2012, Revised Selected Papers*. Lecture Notes in Computer Science **7762**, Springer, 2012, 195–207.
13. M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez, F.J. Romero-Campero: On the power of dissolution in P systems with active membranes. In: R. Freund, Gh. Păun, G. Rozenberg, A. Salomaa (Eds.): *Workshop on Membrane Computing. 6th International Workshop, WMC 2005, Vienna, Austria, July 18-21, 2005, Revised Selected and Invited Papers*. Lecture Notes in Computer Science **3850**, Springer, 2005, 224–240.

14. M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, F.J. Romero-Campero: A uniform solution to SAT using membrane creation. *Theoretical Computer Science* **371** (1-2), 54–61 (2007).
15. Ts.-O. Ishdorj, A. Leporati: Uniform solutions to SAT and 3-SAT by spiking neural P systems with pre-computed resources. *Natural Computing* **7** (4), 519–534 (2008).
16. A. Leporati, L. Manzoni, G. Mauri, A.E. Porreca, C. Zandron: Simulating elementary active membranes - with an application to the P conjecture. In: M. Gheorghe, G. Rozenberg, A. Salomaa, P. Sosík, C. Zandron (Eds.): *Conference on Membrane Computing*. Lecture Notes in Computer Science **8961**, Springer, 2014, 284–299.
17. A. Leporati, G. Mauri, C. Zandron, Gh. Păun, M.J. Pérez-Jiménez: Uniform solutions to SAT and subset sum by spiking neural P systems. *Natural Computing* **8** (4), 681–702 (2009).
18. V.P. Metta, K. Krithivasan, D. Garg: Computability of spiking neural P systems with anti-spikes. *New Mathematics and Natural Computation* **8** (3), 283–295 (2012).
19. M. Mutyam, K. Krithivasan: P systems with membrane creation: universality and efficiency. In: M. Margenstern, Yu. Rogozhin (Eds.): Proceedings of *Machines, Computations, and Universality, MCU 2001, Chişinău, 2001*. Lecture Notes in Computer Science **2055**, Springer, 2001, 276–287.
20. A. Obtułowicz: Deterministic P systems for solving SAT problem. *Romanian Journal of Information Science and Technology* **4** (1-2), 195–201 (2001).
21. L. Pan, A. Alhazov: Solving HPP and SAT by P systems with active membranes and separation rules. *Acta Informatica* **43** (2), 131–145 (2006).
22. L. Pan, Gh. Păun: Spiking neural P systems with anti-spikes. *International Journal of Computers, Communications & Control* **IV** (3), 273–282 (2009).
23. M.J. Pérez-Jiménez: An approach to computational complexity in membrane computing. In: G. Mauri, Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg, A. Salomaa (Eds.): *Workshop on Membrane Computing. 5th International Workshop, WMC 2004, Milan, Italy, June 14-16, 2004, Revised Selected and Invited Papers*. Lecture Notes in Computer Science **3365**, Springer, 2004, 85–109.
24. M.J. Pérez-Jiménez, A. Riscos-Núñez, Á. Romero-Jiménez, D. Woods: Complexity - membrane division, membrane creation. In: Gh. Păun, G. Rozenberg, A. Salomaa (Eds.): *The Oxford Handbook of Membrane Computing*, Oxford University Press, 2010, 302–336.
25. M.J. Pérez-Jiménez, Á. Romero-Jiménez, F. Sancho-Caparrini: Complexity classes in models of cellular computing with membranes. *Natural Computing* **2** (3), 265–285 (2003).
26. T. Song, Y. Jiang, Xi. Shi, Xi. Zeng: Small universal spiking neural P systems with anti-spikes. *Journal of Computational and Theoretical Nanoscience* **10** (4), 999–1006 (2013).
27. G. Tan, T. Song, Zh. Chen, Xi. 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 (2013).

Using Membrane Systems to Solve the Bounded Fanout Broadcast Problem

Michael J. Dinneen and Yun-Bum Kim

Department of Computer Science, University of Auckland,
Private Bag 92019, Auckland, New Zealand
mjd@cs.auckland.ac.nz, tkim021@aucklanduni.ac.nz

Abstract. Broadcasting is the information distribution process in a communication network, which aims to inform all network nodes with a unique message, initially held by a subset of nodes called originators. This paper considers a decision problem that asks if it is possible to inform all nodes within t time units. This paper presents a non-deterministic solution, implemented with a bio-inspired distributed and parallel computational model called membrane systems, which decides in $t+1$ steps.

Keywords: P systems, broadcast, fanout, communication network.

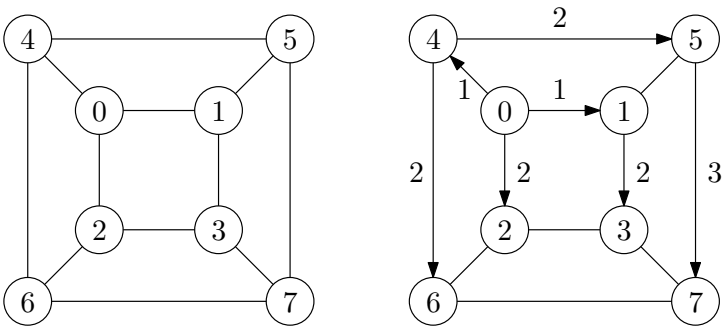
1 Introduction

For a given communication network $G = (V, E)$, broadcasting from node $v \in V$ is the process of distributing information from v to every other node, under the following constraints: (i) messages are exchanged between neighboring nodes, (ii) each message exchange takes one time unit, (iii) each node can exchange messages with up to $f \geq 1$ neighbors in one time unit. The problem is to design a messaging protocol that informs all network nodes from a starting set of vertices with the unique message within a deadline. This problem, a variant to the *Minimum Broadcast Time Problem* [3, 2], is formulated next in Problem 1.

Problem 1. The Bounded Fanout Broadcast Problem
Instance: graph $G = (V, E)$, subset $V_0 \subseteq V$ called *originators*, a positive integer f called *fanout*, a positive integer t called *deadline*.
Question: Is there a sequence of sets $V_0, E_1, V_1, \ldots, E_t, V_t$, such that each $V_i \subseteq V$, each $E_i \subseteq E$, $V_t = V$, and, for $1 \leq i \leq t$,

1. $V_i = V_{i-1} \cup \{v \mid (u, v) \in E_i\}$,
2. each edge in E_i has an endpoint in both V_{i-1} and $V_i \setminus V_{i-1}$,
3. each vertex in V_{i-1} is incident to at most f edges in E_i,
4. each vertex in $V_i \setminus V_{i-1}$ is incident to at most 1 edge in E_i.

The set of edges E_i, $1 \leq i \leq t$, satisying the constraints of Problem 1 is considered to be a *broadcast tree (protocol)* of time t for a graph G. Usually the



$V_0 = \{0\}$
$E_1 = \{(0,1),(0,4)\}$
$V_1 = \{0,1,4\}$
$E_2 = \{(0,2),(1,3),(4,5),(4,6)\}$
$V_2 = \{0,1,2,3,4,5,6\}$
$E_3 = \{(5,7)\}$
$V_3 = \{0,1,2,3,4,5,6,7\}$

Fig. 1. Left: A connected graph. **Center:** A graph that shows the time step in which nodes have been informed (indicated with edge labels) during broadcasting from node 0 with fanout $f = 2$. **Right:** The sequence of sets $V_0, E_1, V_1, E_2, V_2, E_3, V_3$ corresponding to the graph shown in the center.

set of originators is a single source vertex $v \in V$. We say the *fanout f broadcast time* of G originating at v, denoted $BT_f(G, v)$, is the smallest value t such that there is corresponding broadcast tree of time t.

The main contribution in this paper is to present a non-deterministic[1] solution to the bounded fanout broadcast problem using a computing model called membrane system. Membrane systems [6, 7] (also known as P systems) are distributed and parallel computing model, inspired by the structure and function of living cells. A membrane system consists of a network of (multiset processing) computing units called membranes. Each membrane contains a multiset of symbols and is associated with a set of multiset processing rules.

This paper is organized as follows. Section 2 recalls several key mathematical concepts that are used in this paper. Section 3 presents the definition of a membrane system used in this paper. Section 4 presents the details of constructing a membrane system that solves the bounded fanout broadcast problem for a given instance. Finally, Section 5 summarizes this paper and provides some open problems.

2 Preliminaries

This section covers several key mathematical concepts that are used in this paper, such as sets, strings, multisets and graphs.

An *alphabet* is a finite non-empty set with elements called *symbols*. A *string over* alphabet O is a finite sequence of symbols from O. The set of all strings over O is denoted by O^*. The length of a string $x \in O^*$, denoted by $|x|$, is the number of symbols in x. The number of occurrences of a symbol $o \in O$ in a string x over O is denoted by $|x|_o$. The *empty string* is denoted by λ.

A *multiset* is a set with multiplicities associated with its elements. A set that contains the distinct elements of a multiset v is denoted by $\texttt{distinct}(v)$. The empty string or multiset is represented by λ. The *size* of a multiset v is

[1] each informed node non-deterministically selects uninformed neighbors

denoted by $|v|$. The *multiplicity* of an element x in a multiset v is denoted by $|v|_x$. We say that a multiset v is *included* in a multiset w, denoted by $w \subseteq v$, if, for all $o \in O$, $|w|_o \leq |v|_o$. The *union* of multisets v and w, denoted by $v \cup w$, is a multiset x, such that, for all $o \in O$, $|x|_o = |v|_o + |w|_o$. The *difference* of multisets v and w, denoted by $v - w$, is a multiset x, such that, for all $o \in O$, $|x|_o = \max(|v|_o - |w|_o, 0)$.

A (binary) *relation* R over two sets X and Y is a subset of their Cartesian product, $R \subseteq X \times Y$. For $A \subseteq X$ and $B \subseteq Y$, we set $R(A) = \{y \in Y \mid \exists x \in A, (x, y) \in R\}$, $R^{-1}(B) = \{x \in X \mid \exists y \in B, (x, y) \in R\}$.

A *graph* is an ordered pair (V, E), where V is a finite set of elements called nodes and E is a set of unordered pairs of V called edges. A *path* of length $n-1$ is a sequence of n nodes, $v_1, v_2, \ldots, v_n$, such that $\{(v_1, v_2), \ldots, (v_{n-1}, v_n)\} \subseteq E$. The *diameter* of G, denoted by $\texttt{dia}(G)$, is the maximum of the lengths of shortest paths between every pair of nodes of G.

A *directed graph* (digraph) is a pair (V, A), where V is a finite set of elements called nodes and A is a set of an ordered pair of V called *arcs*. Given a digraph $D = (V, A)$, for $v \in V$, the *parents* of v are $A^{-1}(v) = A^{-1}(\{v\})$ and the *children* of v are $A(v) = A(\{v\})$.

3 Membrane systems

Membrane systems (also known as P systems) are distributed and parallel computing models. A membrane system consists of a network of (multiset processing) computing units called membranes. Each membrane contains a multiset of symbols and is associated with a set of multiset processing rules. Several P system models [5, 4, 1] have been introduced, inspired from various features of living cells, that provide new ways to process information and solve the computational problems of interest. A membrane system model used in this paper has the form $\Pi = (O, Q, K, R, \Delta)$, where

1. O is a finite non-empty alphabet of *symbols*.
2. Q is a finite set of *states*.
3. $K = \{\mu_1, \mu_2, \ldots, \mu_n \mid n \in \mathbb{N}^+\}$ is a finite set of *membranes*. Each membrane $\mu_i \in K$ is of the form $\mu_i = (s_i, w_i)$, where
   - $s_i \in Q$ denotes the *current state* of μ_i,
   - $w_i \in O^*$ denotes the *current content* of μ_i.
4. R is a set of *evolution rules*, where an evolution rule $r \in R$ has the form:

$$j\ s\ u \rightarrow_\alpha s'\ v\ w\ x$$

   - $\alpha \in \{\texttt{min}, \texttt{max}\}$ is a *rewriting* operator of r,
   - $j \in \mathbb{N}$ is the *priority* of r, where the lower value j indicates higher priority,
   - $s, s' \in Q$, where $s = \texttt{source}(r)$ is the *start state* and $s' = \texttt{target}(r)$ is the *target state* of r,
   - $u \in O^+$ are the rule symbols on the "left hand side", denoted $\texttt{LHS}(r)$,

- $v \in (O \times \tau)^*$, where $\tau \in \{\odot, \uparrow, \downarrow, \updownarrow\}$ is a *target indicator*. Note that, $(o, \odot) \in v$, $o \in O$, is abbreviated to o,

5. Δ is an irreflexive and asymmetric relation on K, representing a set of arcs between membranes with bidirectional communication capabilities.

A *configuration* of system Π of order n is $(s_1, w_1, s_2, w_2, \ldots, s_n, w_n)$, where, for $1 \leq i \leq n$, s_i and w_i correspond to the current state and content of membrane σ_i, respectively. Consider two configurations of system Π, C' and C''. A *transition* in system Π is a transformation from C' to C'' in one time unit, denoted by $C' \Rightarrow C''$, such that C'' is obtained from C'. A transition $C' \Rightarrow C''$ consists of two substeps (substep 1 and substep 2). All membranes simultaneously perform substep 2, after every membrane has finished substep 1.

- **Substep 1:** Each membrane μ_i, $1 \leq i \leq n$, finds a maximal multiset of evolution rules, M_i, as described in Definitions 2 and 3.
- **Substep 2:** Each membrane μ_i, $1 \leq i \leq n$, executes a multiset of evolution rules found in substep 1, M_i, as described in Definition 4.

System Π *halts*, if it reaches a configuration (called the halting configuration), where no evolution rule can be applied to the existing symbols inside all membranes. The *computational results* of a halted system are the multiplicities of symbols present in the membranes of the system.

Definition 2. Given a multiset $w \in O^*$ and an evolution rule $r \in R$, where $\texttt{LHS}(r) \subseteq w$, the number of applications of r over w is

$$\texttt{apply}(r, w) = \begin{cases} 1 & \text{if } \texttt{rewrite}(r) = \texttt{min}, \\ |w|_{\texttt{LHS}(r)} & \text{if } \texttt{rewrite}(r) = \texttt{max}. \end{cases}$$

Definition 3. For membrane μ_i, in state s_i with content w_i and a set of evolution rules R_i, a maximal multiset of rules, M_i, is obtained by the procedure below.

Input: a set of evolution rules R_i and a multiset $w := w_i$.
Output: a maximal multiset M_i.
$M_i := \emptyset$
for each $r_j \in R_i$ with $\texttt{source}(r_j) = s_i$, $1 \leq j \leq |R_i|$ (by priority order)
  if $(M_i = \emptyset \;||\; \forall r_k \in M_i \; (\texttt{target}(r_j) = \texttt{target}(r_k)))$ **then**
    if $(\texttt{LHS}(r_j) \subseteq w$ **then**
      $m := \texttt{apply}(r_j, w)$
      $M_i := M_i \cup \{r_j^m\}$
      $w := w - \texttt{LHS}(r_j)^m$
    endif
  endif
endfor

Definition 4. For each membrane μ_i, $1 \leq i \leq n$, consider a maximal multiset of evolution rules, M_i, found according to Definition 3. For membrane μ_i with the

current content w_i, multisets U_i, V_i, $V_i^{\downarrow}$, $V_i^{\uparrow}$ and $V_i^{\updownarrow}$, for each $\mu_k \in \Delta(i) \cup \Delta^{-1}(i)$, are defined as follow:

- $U_i = \bigcup_{r_j \in M_i} \texttt{LHS}(r_j)$, denotes the multiset that will be consumed from w_i.
- $V_i = \bigcup_{r_j \in M_i} \bigcup_{(o,\odot) \in \texttt{RHS}(r_j)} \{o\}$, denotes the multiset that will be produced and added to w_i.
- $V_i^{\downarrow} = \bigcup_{r_j \in M_i} \bigcup_{(o,\downarrow) \in \texttt{RHS}(r_j)} \{o\}$, denotes the multiset that will be sent to each $\mu_k \in \Delta(i)$.
- $V_i^{\uparrow} = \bigcup_{r_j \in M_i} \bigcup_{(o,\uparrow) \in \texttt{RHS}(r_j)} \{o\}$, denotes the multiset that will be sent to each $\mu_k \in \Delta^{-1}(i)$.
- $V_i^{\updownarrow} = \bigcup_{r_j \in M_i} \bigcup_{(o,\updownarrow) \in \texttt{RHS}(r_j)} \{o\}$, denotes the multiset that will be sent to each $\mu_k \in \Delta(i) \cup \Delta^{-1}(i)$.

For each membrane μ_i in state s_i with content w_i:

- If $M_i = \emptyset$, then μ_i remains in state s_i with content w_i.
- Otherwise, μ_i transforms:
  - its current state to $s_i = \texttt{target}(r_f)$, where $r_f \in M_i$.
  - its current content w_i to w_i', where

$$w_i' = w_i - U_i \ \cup \ V_i \ \cup \bigcup_{f \in \Delta^{-1}(i)} V_f^{\downarrow} \ \cup \bigcup_{g \in \Delta(i)} V_g^{\uparrow} \ \cup \bigcup_{h \in \Delta(i) \cup \Delta^{-1}(i)} V_h^{\updownarrow}$$

4 Non-deterministic P systems solutions

This section presents P system Π that correspond to a non-deterministic solution to the bounded fanout broadcast problem of Problem 1. A trace of system Π for the example of Figure 1 is given in Section 4.4.

4.1 Overview of system Π

System Π consists of one membrane, labeled μ, that determines if every node can be informed within t steps from nodes of V_0, using the procedure illustrated in Figure 2. Activities and decisions indicated inside boxes of the procedure are accompanied by the corresponding evolution rules specified in Section 4.2.

As illustrated in Figure 2, μ produces one copy of symbol o if every node can be informed within t steps. The final configuration of a halted system Π can be interpreted, with respect to Problem 1, as follows:

- If μ ends with one copy of symbol o, then the answer is "Yes".
- Otherwise, the answer is "No".

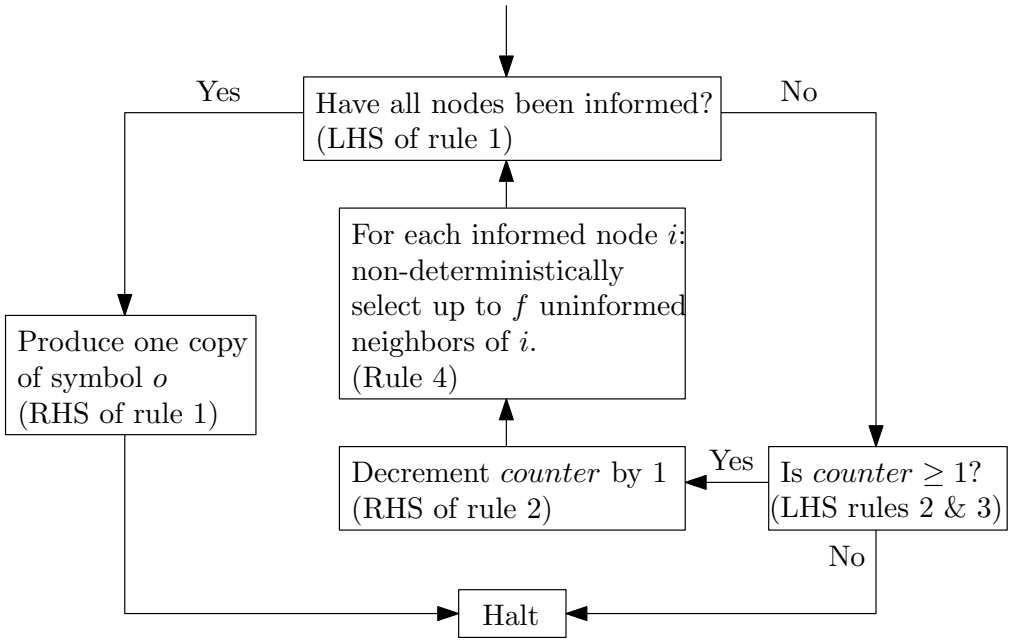


Fig. 2. Procedure for μ to determine if all nodes can be informed within t steps from nodes of V_0. Initially, nodes of V_0 are marked as "informed" and every other node is marked as "uninformed". Variable *counter* has an initial value of input parameter t.

4.2 Specification of system Π

Specification of system Π described earlier is (O, Q, R, K, Δ), where

1. $O = \{v_i, u_i, e_{i,j}, h, o \mid i, j \in \{1, 2, \ldots, n\}\}$.
   - Symbols $e_{i,j}$ and $e_{j,i}$ represent edge $(i, j) \in E$.
   - Symbols v_i and u_i represent the "informed" and "uninformed" status of node $i \in V$, respectively.
   - Multiplicity of symbol v_i represents the fanout parameter f.
   - Recall variable *counter* of Figure 2, which has an initial value of input parameter t. Multiplicity of symbol h corresponds to value *counter* $+ 1$.
   - Symbol o represents "Yes-output", i.e. every node can be informed within in t steps.
2. $Q = \{s_0, s_1, s_2\}$, where
   - s_0 represents an active state where informed nodes non-deterministically select up to f uninformed nodes.
   - s_1 represents a halt state where all nodes could not be informed within t steps.
   - s_2 represents a halt state where every node is informed within t steps.
3. R corresponds to the following rules. The task each rule undertakes is indicated in Figure 2.
   1. $s_0\ v_1^f\ v_2^f\ \ldots\ v_n^f \rightarrow_{\texttt{min}} s_2\ o$
   2. $s_0\ h\ h \rightarrow_{\texttt{min}} s_0\ h$
   3. $s_0\ h \rightarrow_{\texttt{min}} s_1\ h$
   4. $s_0\ v_i\ e_{i,j}\ e_{j,i}\ u_j \rightarrow_{\texttt{min}} s_0\ v_i\ v_j^f$
4. $K = \{\mu\}$, where μ has the initial form of $(s_0, V_K \cup U_K \cup E_K \cup h^{t+1})$, where

- $V_K = \{{v_j}^f \mid j \in \{V_0\}\}$,
- $U_K = \{u_j \mid j \in \{1, 2, \ldots, n\} \setminus \{V_0\}\}$,
- $E_K = \{e_{i,j}, e_{j,i} \mid (i,j) \in E\}$.

5. $\Delta = \emptyset$.

4.3 Analysis of system Π

Propositions 5 and 6 demonstrate the correctness of construction of system Π for solving the Problem 1. The run-time complexity of system Π is indicated in Proposition 7.

Proposition 5. Using rule 4, each informed node non-deterministically selects f uninformed neighbors repeatedly, if any, and marks them as "informed".

Proof. Each copy of symbol v_i is used to find one uninformed neighbor, if any, as follows. If symbols v_i, u_j, $e_{i,j}$ and $e_{j,i}$ are available (i.e. node i is visited, node j is unvisited and nodes i and j are neighbors), then rule 4 rewrites symbol u_j into f copies of symbol v_j (i.e. transforms the status of node j from "uninformed" to "informed"). Every copy of symbol v_i is preserved, such that node i can select up to f uninformed neighbors in the future repeatedly, if necessary. □

Proposition 6. Membrane μ replicates the the procedure of Figure 2.

Proof. We show that the evolution rules of R, which govern the behavior of μ resemble the procedure of Figure 2. Membrane μ starts from state s_0. Membrane μ in state s_0 finds and executes rules in each step as follows:

- Due to the rule priority, rule 1 is the first rule checked by μ. Rule 1 inspects whether every node is informed by requiring multiset $\{v_i^f \mid 1 \leq i \leq n\}$. If μ meets this requirement, rule 1 is executed, which prompts μ to produce one copy of symbol o and halt by entering state s_2.
- Rule 2 is the next rule checked by μ, given that μ does not contain multiset $\{v_i^f \mid 1 \leq i \leq n\}$ (i.e. not every node is informed). Rule 2 inspects the condition "$counter \geq 1$?" by requiring multiset $\{hh\}$. If μ contains $\{hh\}$, rule 2 is executed, which prompts μ to consume one copy of symbol h (i.e. decrement *counter* by 1) and remain in state s_0 such that μ can check through rules of R in the next step.
- Rule 3 is the rule executed by μ, given that μ does not satisfy the requirements of rules 1 and 2, i.e. not every node is informed and $counter = 0$. Executing rule 3 prompts μ to halt by entering state s_1.
- Rule 4 can be executed in parallel with rule 2 in one step, since these rules have the same target state of s_0. As described in Proposition 5, rule 4 enables each informed node to non-deterministically select up to f uninformed neighbors.

The manner in which rules 1, 2, 3 and 4 are selected, and the results these rules produce resemble the procedure of Figure 2. Thus, μ replicates the procedure of Figure 2. □

Proposition 7. System Π takes at most $t + 1$ steps.

Proof. In each step, μ executes (i) rule 1, (ii) rules 2 and 4, or (iii) rule 3. The maximum number of steps rules 2 and 4 can be executed is t. If all nodes have been informed in $t' \leq t$ steps, then μ halts at step $t' + 1$ by executing rule 1. Otherwise, μ halts at step $t' + 1$ by executing rule 3. □

4.4 Example - an evolution trace of system Π

The table below illustrates an evolution trace of system Π for the instance: G is the graph shown in Figure 1 (Left), initiators $V_0 = \{0\}$, fanout $f = 2$ and deadline $t = 3$. The order in which nodes are informed in the trace below corresponds to the sequence given in Figure 1 (Right). The table indicates the state and content of membrane μ in each step. The content column is divided into five sub-columns that respectively indicate (i) "edge" symbols, (ii) "counter" symbol, (iii) "unvisited node" symbols, (iv) "visited node" symbols and (v) "Yes-output" symbol.

Step	State	Content				
0	s_0	$e_{0,1}\ e_{0,2}\ e_{0,4}\ e_{1,0}\ e_{1,3}\ e_{1,5}\ e_{2,0}\ e_{2,3}$ $e_{2,6}\ e_{3,1}\ e_{3,2}\ e_{3,7}\ e_{4,0}\ e_{4,5}\ e_{4,6}\ e_{5,1}$ $e_{5,4}\ e_{5,7}\ e_{6,2}\ e_{6,4}\ e_{6,7}\ e_{7,3}\ e_{7,5}\ e_{7,6}$	h^4	$u_1\ u_2\ u_3\ u_4$ $u_5\ u_6\ u_7$	v_0^2	
1	s_0	$e_{0,2}\ e_{1,3}\ e_{1,5}\ e_{2,0}\ e_{2,3}\ e_{2,6}\ e_{3,1}\ e_{3,2}$ $e_{3,7}\ e_{4,5}\ e_{4,6}\ e_{5,1}\ e_{5,4}\ e_{5,7}\ e_{6,2}\ e_{6,4}$ $e_{6,7}\ e_{7,3}\ e_{7,5}\ e_{7,6}$	h^3	$u_2\ u_3\ u_5\ u_6$ u_7	$v_0^2\ v_1^2\ v_4^2$	
2	s_0	$e_{1,5}\ e_{2,3}\ e_{2,6}\ e_{3,2}\ e_{3,7}\ e_{5,1}\ e_{5,7}\ e_{6,2}$ $e_{6,7}\ e_{7,3}\ e_{7,5}\ e_{7,6}$	h^2	u_7	$v_0^2\ v_1^2\ v_2^2\ v_3^2$ $v_4^2\ v_5^2\ v_6^2$	
3	s_0	$e_{1,5}\ e_{2,3}\ e_{2,6}\ e_{3,2}\ e_{3,7}\ e_{5,1}\ e_{6,2}\ e_{6,7}$ $e_{7,3}\ e_{7,6}$	h		$v_0^2\ v_1^2\ v_2^2\ v_3^2$ $v_4^2\ v_5^2\ v_6^2\ v_7^2$	
4	s_2	$e_{1,5}\ e_{2,3}\ e_{2,6}\ e_{3,2}\ e_{3,7}\ e_{5,1}\ e_{6,2}\ e_{6,7}$ $e_{7,3}\ e_{7,6}$	h			o

4.5 Remark

There are several variants to this bounded fanout broadcast problem. One of the variants is to compute the fanout f broadcast time of a graph $G = (V, E)$, defined $BT_f(G) = \max_{v \in V} BT_f(G, v)$, where the broadcast time of an originator, $BT(G, f, v)$, was defined just after Problem 1.

An overview of P system Π' that can solve this global broadcast problem is as follows. Assume that for the input graph G, $V = \{v_1, v_2, \ldots, v_n\}$. System Π' consists of $n+1$ membranes, labeled $\mu_{skin}, \mu_{v_1}, \mu_{v_2}, \ldots, \mu_{v_n}$, which are arranged in a rooted tree structure of Figure 3.

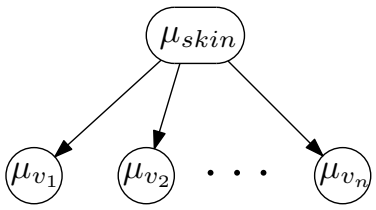


Fig. 3. The membrane structure of system Π'.

Membrane μ_{v_i}, $1 \leq i \leq n$, covers the instance $V_0 = \{v_i\}$ by determining if node v_i can inform every node within t steps. Membrane μ_{v_i} uses the procedure illustrated in Figure 2 with the following difference: instead of producing one copy of symbol o locally, μ_{v_i} sends up one copy of symbol o to membrane μ_{skin}, i.e. replace rule $s_0\ v_1^f\ v_2^f\ \ldots\ v_n^f \rightarrow_{\texttt{min}} s_2\ o$ with $s_0\ v_1^f\ v_2^f\ \ldots\ v_n^f \rightarrow_{\texttt{min}} s_2\ (o, \uparrow)$. The final configuration of a halted system Π' can be interpreted as follows:

- If μ_{skin} ends with n copies of symbol o, then the answer is "Yes".
- Otherwise, the answer is "No".

5 Conclusions

In this paper, we studied a communication networks problem, called the bounded fanout broadcast problem, that asks: is it possible to inform all network nodes within a specified deadline, under a communication constraint that limits the number of neighbors each node can communicate simultaneously?

We designed our solution to this decision problem using membrane systems that decides within $t+1$ steps, where t denotes the deadline. Future work include two natural optimization problems: (i) find smallest fanout f when deadline t is fixed, and (ii) find smallest t when f is fixed.

References

1. M. J. Dinneen, Y.-B. Kim, and R. Nicolescu. A faster P solution for the Byzantine agreement problem. In M. Gheorghe, T. Hinze, and G. Păun, editors, *Conference on Membrane Computing*, volume 6501 of *Lecture Notes in Computer Science*, pages 175–197. Springer-Verlag, Berlin Heidelberg, 2010.
2. M. J. Dinneen, G. Pritchard, and M. C. Wilson. Degree- and time- constrained broadcast networks. *Networks*, 39(3):121–129, Mar. 2002.
3. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

4. M. Ionescu, G. Păun, and T. Yokomori. Spiking neural P systems. *Fundam. Inform.*, 71(2-3):279–308, 2006.
5. C. Martín-Vide, G. Păun, J. Pazos, and A. Rodríguez-Patón. Tissue P systems. *Theor. Comput. Sci.*, 296(2):295–326, 2003.
6. G. Păun. *Membrane Computing: An Introduction.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.
7. G. Păun, G. Rozenberg, and A. Salomaa. *The Oxford Handbook of Membrane Computing.* Oxford University Press, Inc., New York, NY, USA, 2010.

On Simulating Cooperative Transition P systems in Evolution-Communication P systems with Energy

Richelle Ann B. Juayong, Henry N. Adorna

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

Abstract. In this paper, we investigate simulations of Transition P systems (TP systems) in Evolution-Communication P systems with Energy (ECPe systems). We only focus on TP systems where an object that triggers a cooperative rule also triggers a non-cooperative rule. In this way, the presence of a rule trigger always implies that a rule will be applied. In our constructed ECPe systems, a transition in the TP system is simulated by a k-step computation where k is a factor of the cardinality of the alphabet in the original system. Also, the maximum energy needed for communication rules are dependent on the number of copies of a trigger in a cooperative rule.

Keywords: Membrane computing, Evolution-Communication P systems with energy, Transition P systems

1 Introduction

One of the models proposed for analyzing communication complexity in membrane computing [6, 8] is the so-called Evolution-Communication P system with energy (ECPe system) introduced in [2]. ECPe system is an extension of Evolution-Communication P system (ECP system)[3]. Both models use separate forms of rules for evolution and communication. However, in ECPe system, communication requires a cost. The cost comes in the form of special objects called 'energy', produced during evolution and required during communication.

In this work, we contribute to the study of ECPe systems by investigating how this model can be used to simulate a basic membrane computing model called Transition P system [5]. Although both are cell-like models, TP and ECPe systems differ in how rules are formed. (These distinctions on the rules used becomes more apparent when we look at their definitions in Section 2). Through this work, we also contribute to a research topic given in [7] on simulating a class of P systems with another class. Simulations of TP systems may also give us an

idea on how other class of cell-like P systems (e.g. ECP systems which is a more similar variant) can be simulated in ECPe system.

In a previous work [4], we are able to show an ECPe system simulator for non-cooperative TP systems. A transition can be simulated by a 3-step computation in the constructed system. This work is a continuation of the effort in [4]; this time, we focus on extending the constructed simulator in [4] to handle cooperative rules. One of the difficulties in handling cooperative rules is validating that the required multiset in the left-hand side of a rewriting rule exists in the region where the rule is defined. In this work, we only focus on a restricted cooperative TP system. In such model, a cooperative rule trigger can also be consumed by a non-cooperative rule trigger. Thus, in the presence of triggers, we are sure that the system continues to move to a next configuration. The resulting Transition P system is called Transition P systems with independent triggers (or TP-ind systems).

2 Preliminaries

Let V be an alphabet, V^* is the free monoid over V with respect to concatenation and the identity element λ (empty string). The set of all non-empty strings over V is denoted as V^+ so $V^+ = V^* - \{\lambda\}$. The length of a string $w \in V^*$ is denoted by $|w|$. For $a \in V$, $|w|_a$ represents the number of a in string w. Let U be an arbitrary set. A multiset (over U) is a mapping $M : U \to \mathbf{N}$. The value $M(a)$, for $a \in U$, is the *multiplicity of a in the multiset M*. The *support* of a multiset M is the set $supp(M) = \{a \in U \mid M(a) > 0\}$. A multiset M can also be represented by a string: $w = a_1^{M(a_1)} a_2^{M(a_2)} \dots a_n^{M(a_n)}$ where $a_i \in supp(M)$, $1 \leq i \leq n$. In string w and all its permutation, $|w|_{a_i} = M(a_i)$. Thus, string w and all its permutations precisely identify and refer to the same multiset M. We use the phrase "multiset w", where w is a string, to refer to the multiset represented by the string w.

We define a Transition P (TP) system without dissolution, similar to [1] as follows:

Definition 1. (TP systems) *A Transition P (TP) system without dissolution is a construct of the form* $\Pi = (O, \mu, w_1, \dots, w_m, R_1, \dots, R_m, h_{output})$ *where* m *is the total number of membranes;* O *is the alphabet of objects;* μ *is a hierarchical membrane structure (a rooted tree) of degree* m*, bijectively labelled from* 1 *to* m*, and the interior of each membrane defines a region; the environment is referred as region* 0*;* w_h *is the initial multiset in region* h $(1 \leq h \leq m)$*;* R_h *is the set of rules in region* h $(1 \leq h \leq m)$*; Each rule has the form* $u \to v$ *where multiset* $u \in O^+$*,* $v \in (O \times Tar_h)^*$ *and* $Tar_h = \{here, out\} \cup \{in_j \mid j \in children(h)\}$*,* $1 \leq h \leq m$*;* $h_{output} \in \{0, 1, \dots, m\}$ *is the output region. When the system is a language generator,* $h_{output} = 0$*;*

As in usual P systems, μ is a hierarchical membrane structure denoted by a string of matching square brackets with labels. If membrane j is immediately contained in membrane h, i.e. $[\dots[\,]_j]_h$, h is referred as parent of j (denoted by $parent(j)$).

Consequently, j is a *child membrane* of h. This is denoted by $j \in children(h)$ where $children(h)$ is the set of all child membranes of h. Aside from possibly containing other membranes, a membrane has a multiset of objects. We use the term 'copy of a' to refer to an instance of object a present in a multiset w. The multiset in a region h is evolved and transported through the set R_h of rules. To describe how each rule is executed, we refer to Definition 1 to recall the form followed by each rule in R_h. When rule r is applied, the multiset u is removed from region h and multisets in v is produced in the next time step. Symbols *here*, *out* and in_j indicate the destination of the objects produced (target *here* is typically omitted). A rule $u \rightarrow v$ labelled r is denoted by $r : u \rightarrow v$. The left-hand side of rule r (that is, u) is denoted by $LHS(r)$. A rule with $|u| = 1$ is said to be non-cooperative; otherwise, the rule is cooperative.

Starting from the initial multiset in each region, a TP system computes by applying rules in a non-deterministic and maximally parallel manner. Non-determinism implies that at a certain step, if there are more than two rules that can be applied to a copy of an object, the system non-deterministically chooses the rule to be applied for each copy. Maximally parallel means that there are no further rules applicable to copies that are not used in any rule. A configuration of a P system describes the membrane structure and the content of regions at a certain time. The process of applying all applicable rules in a current configuration, thus obtaining a new configuration is called a transition. A computation 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 then the last term is a halting configuration (a configuration where no rule of the system is applicable to it). Computation succeeds when the system halts. If the computation doesn't halt, computation fails because the system did not produce any output.

We only consider models that generate languages. In this case, we follow [1] wherein the result of a successful computation is the sequence of objects sent to the environment, i.e. $h_{output} = 0$. The order of how copies of objects are sent to the environment dictates their position in the output string. When multiple objects are sent at a given time, the output string is formed from any of their permutations. The language generated by TP system Π is denoted by $L(\Pi)$.

In our study, we only focus on a specific type of TP system called TP-ind system. First, we introduce the notion of rule triggers. Given a TP system, a trigger corresponds to an object that exists on the left-hand side of a rule.

Definition 2. (Trigger, Independent Trigger) *Given a TP system Π, an object $o \in O$ is a trigger in a region h if there exists a rule $r \in R_h$ where $o \in supp(LHS(r))$. Object o is an* **independent trigger in region** *h if there exists a rule $r' \in R_h$ that is a non-cooperative rule and $o \in supp(LHS(r'))$. When the region is clear, we simply say that a trigger is independent.*

Definition 3. (coop-ind rule) *Let Π be a TP system. A cooperative rule having independent triggers only (coop-ind rule) is a cooperative rule $r \in R_h$ $(1 \leq h \leq m)$ where for all trigger $o \in supp(LHS(r))$, o is independent.*

Definition 4. (TP-ind system) *A TP system with independent triggers only (TP-ind system) is a TP system Π where all triggers in all regions are independent triggers. This implies that if Π contains cooperative rules, then these rules are all coop-ind rules.*

Notice that independent triggers are always consumed (in a computation step) when some copies of these triggers occur in a configuration; this means, TP-ind system continues evolving as long as a trigger exists since all triggers are independent. Also, the halting configuration of a TP-ind system doesn't have triggers.

We use the definition for Evolution-Communication P system with energy (ECPe system) from [2].

Definition 5. (ECPe system) *An Evolution-Communication P system with energy (ECPe system) is a construct of the form $\Pi^\dagger = (O^\dagger, e, \mu^\dagger, w_1^\dagger, \ldots, w_{\bar{m}}^\dagger, R_1^\dagger, R'^\dagger_1, \ldots, R_{\bar{m}}^\dagger, R'^\dagger_{\bar{m}}, h^\dagger_{output})$ where $\bar{m}$ pertains to the total number of membranes; $O^\dagger$ is the alphabet of objects; $e \notin O^\dagger$ is a special object; $\mu^\dagger$ is a hierarchical membrane structure; $w_h^\dagger$ is the initial multiset over $(O^\dagger)^*$ in region h ($1 \leq h \leq \bar{m}$); $R_h^\dagger$ is set of evolution rules in region h; Each rule has the form $a \to v$ where $a \in O^\dagger$, $v \in (O^\dagger \cup \{e\})^*$. $R'^\dagger_h$ are sets of communication rules in membrane h; There are two types of communication rules: symport and antiport. A symport rule takes one of the following form: (ae^i, in) or (ae^i, out), where $a \in O^\dagger$, $i \geq 1$. An antiport rule takes the form $(ae^i, out; be^j, in)$ where $a, b \in O^\dagger$ and $i, j \geq 1$. $h^\dagger_{output} \in \{0, 1, \ldots, \bar{m}\}$ is the output region. As in TP systems, $h^\dagger_{output} = 0$ when the system is a language generator.*

Notice that since e is not part of $O^\dagger$, any copy of e cannot be in the initial configuration. A set of evolution rules R_h is associated with each region h. To describe how each evolution rule is executed, we refer to Definition 5 to recall the form followed by each rule in R_h. When applying a rule of this type, a copy of a transforms into a multiset v in the next time step. This is similar to the multiset-rewriting rule for TP systems. However, evolution rules are non-cooperative; also, the multiset produced stays in the same region.

Each membrane h ($1 \leq h \leq \bar{m}$) has a set of communication rules. A communication rule can either be a symport or an antiport rule. A symport rule can be of the form (ae^i, in) or (ae^i, out), where $a \in O^\dagger$, $i \geq 1$. By using this rule, i copies of e are consumed to transport a copy of a inside (denoted by in) or outside (denoted by out) the membrane where the rule is defined. The copies of e cannot pass through membranes, thus, they disappear (or said to be lost) after the communication. An antiport rule is of the form $(ae^i, out; be^j, in)$ where $a, b \in O^\dagger$ and $i, j \geq 1$. By using this rule, a copy of a in the region immediately outside the membrane where the rule is declared, and a copy of b inside the region bounded by the membrane should exist. When such rule is applied, copy of a and copy of b are swapped using i and j copies of e in the different regions, respectively. As in symport rules, the copies of e disappears after the swap. The language generated by ECPe system $\Pi^\dagger$ is denoted by $L(\Pi^\dagger)$.

We are interested in constructing simulations of different TP system variants in ECPe system where for every transition in the TP system, there is a corresponding computation in ECPe system:

Definition 6. *Given a TP system Π and an ECPe system $\Pi^\dagger$, we say that Π simulates $\Pi^\dagger$ when we can establish a correspondence between configurations in Π and $\Pi^\dagger$. Suppose a configuration C_1 in Π corresponds to a configuration $C_1^\dagger$ in $\Pi^\dagger$ and a configuration C_2 in Π corresponds to a configuration $C_2^\dagger$ in $\Pi^\dagger$: There is a transition from C_1 to C_2 in Π if and only if there is a computation in $\Pi^\dagger$ that reaches $C_2^\dagger$ from $C_1^\dagger$.*

3 Main Results

Definition 7. (Categories for coop-ind rules) *Given a TP-ind system Π, a coop-ind rule in a region h ($1 \leq h \leq m$) can be classified as one of the following:*

cat 1: coop-dis rule *A cooperative rule $r \in R_h$ has $|LHS(r)|_a = 1$ for all $a \in supp(LHS(r))$. This type requires only one copy of each distinct trigger.*

cat 2: coop-one rule *A cooperative rule $r \in R_h$ has $|u| > 1$ and $|supp(u)| = 1$ where $LHS(r) = u$. This type requires many copies of only one trigger.*

cat 3: coop-mul rule *A cooperative rule $r \in R_h$ has $|supp(u)| > 1$ and at least one trigger a has $|u|_a > 1$ where $LHS(r) = u$. This type requires more than one trigger, and at least one trigger requires more than one copy.*

We give additional notations for TP systems before we give our main result. These notations will be used in the suceeding subsections. Given a TP system $\Pi = (O, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, h_{output})$, we impose a total order on alphabet O so that we can label every element in O as o_k ($1 \leq k \leq |O|$). Let rule $r \in R_h$ where $r : u \to v$ (u and v are as specified for multiset-rewriting rules in TP systems in Section 2). Multisets in v are divided according to their receiving region. We define x_r, $y_{r,j}$ ($j \in children(h)$) and z_r as part of multiset v where:

- x_r is the multiset of objects produced by r and stays in the region h.
- $y_{r,j}$ is the multiset of objects produced by r and is communicated inside an inner membrane j.
- z_r is the multiset of objects produced by r and is communicated outside h.

We use the total mapping on alphabet O to fix an order on the left-hand side of a TP-ind system rule. Specifically, given a rule $r : u \to v$:

- $first(r) = o_k$ where $o_k \in supp(u)$ and $\forall o_{k'}$, $k' < k$, $o_{k'} \notin supp(u)$.
- $last(r) = o_k$ where $o_k \in supp(u)$ and $\forall o_{k'}$, $k' > k$, $o_{k'} \notin supp(u)$.
- $prev(r, o_{k'}) = o_k$ where $o_k, o_{k'} \in supp(u)$, $k' > k$ and $\forall o_{k''}$, $k < k'' < k'$, $o_{k''} \notin supp(u)$. The object $prev(r, o_{k'})$ is the object in the sequence $\mathcal{O}$ that triggers rule r before object $o_{k'}$.
- $next(r, o_{k'}) = o_k$ where $o_k, o_{k'} \in supp(u)$, $k' < k$ and $\forall o_{k''}$, $k' < k'' < k$, $o_{k''} \notin supp(u)$. The object $next(r, o_{k'})$ is the object in the sequence $\mathcal{O}$ that triggers rule r after object $o_{k'}$

Suppose a TP system Π has an alphabet $O = \{a, b, c, d, e\}$ and a rule $r : a^2c^3 \rightarrow ab(c, in_2)(a, out)$.We can impose a total order on O based on the position of the symbols in the set (i.e. a is labelled o_1 and e is labelled o_5). Based on this labelling, $r : o_1^2 o_3^3 \rightarrow o_1 o_2(o_3, in_2)(o_1, out)$. Thus, $x_r = o_1 o_2$, $y_{r,2} = o_3$, and $z_r = o_1$. Also, $first(r) = o_1$, $last(r) = o_3$, $next(r, o_1) = o_3$ and $prev(r, o_3) = o_1$.

3.1 Simulating TP-ind system where cooperative rules are coop-dis

Consider a TP-ind system of degree $m \geq 1$, $\Pi_1 = (O, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, 0)$ where all cooperative rules are coop-dis.

We construct the simulator ECPe system of degree $\bar{m} \geq 2$, ${\Pi_1}^\dagger = (O^\dagger, e, \mu^\dagger, w_1^\dagger, \ldots, w_{\bar{m}}^\dagger, R_1^\dagger, {R'}_1^\dagger, \ldots, R_{\bar{m}}^\dagger, {R'}_{\bar{m}}^\dagger, 0)$. The number $\bar{m} = m + n + 1$ where n is the total number of coop-dis rules in Π_1. Membrane structure $\mu^\dagger$ is obtained in the following way: 1) $\mu^\dagger$ initially adapts membrane structure μ. To avoid confusion, we let every membrane h of Π_1 $(1 \leq h \leq m)$ be represented by a membrane labelled (h) in $\Pi_1^\dagger$. 2) For every coop-dis rule $r \in R_h$, a membrane labelled (h, r) is introduced as a child membrane of (h). 3) An additional membrane labelled (0) is introduced as the parent membrane of (1).

For every copy of object $o_k \in O$, consider new objects $o_{k,p}$ where the second index of the subscript functions as a timer. Also, for every rule trigger o_k, consider new objects $o_{k,p,r}$ where rule r is triggered by o_k in Π_1. The third index of the subscript indicates that a copy of object o_k will be consumed by applying rule r. For every rule $r \in R_h$, we also consider new objects $\varkappa_r, \varkappa_r^\bullet, \varsigma_r, \varphi_{r,j}$ $(j \in children(h))$. These objects are used when producing the multisets in the right-hand side of rule r. Taking these into account, alphabet $O^\dagger$ is defined by:

$$\begin{aligned} O^\dagger \quad = \quad & \{o_{k,p}, o_{k,p,r} \mid 1 \leq k \leq |O|, 0 \leq p \leq 2|O| + 2, o_k \in supp(LHS(r))\} \\ & \cup \{\varkappa_r, \varkappa_r^\bullet, \varsigma_r, \varphi_{r,j} \mid 1 \leq j \leq m, r \in R_h, 1 \leq h \leq m\} \end{aligned}$$

The initial multiset $w_{(h)}^\dagger$ is obtained by replacing every copy o_k in w_h by $o_{k,0}$. All other membranes in $\Pi_1^\dagger$ is initially empty. The sets of rules for the ECPe system are obtained in the following way:

For every trigger o_k that is present in the LHS of a rule $r \in R_h$ (i.e. $o_k \in supp(LHS(r))$), we add the following rules:

[a] $o_{k,p} \rightarrow o_{k,p+1} \in R_{(h)}^\dagger$ for $1 \leq k \leq |O|$, $0 \leq p < 2k - 2$

[b] $o_{k,2k-2} \rightarrow o_{k,2k-1,r} \in R_{(h)}^\dagger$ for a non-cooperative rule r

[c] $o_{k,2k-2} \rightarrow o_{k,2k,r} e \in R_{(h)}^\dagger$ for a cooperative rule r

Rule [a] increments the timer of a copy $o_{k,p}$ until p reaches a value of $2k - 2$. In the next step, one of rules [b] or [c] will be applied on a copy $o_{k,2k-2}$. When rule [b] is used, copy $o_{k,2k-2}$ evolves; incrementing the value of the timer by one and appending another index in the subscript to remember the rule which will consume o_k. When rule [c] is used, copy $o_{k,2k-2}$ becomes $o_{k,2k,r}$ while also producing a copy of the special object e. Through rule [c], the involved timer is incremented by two. The 'plus one' in such increment is used to account the communication in the next step, wherein the produced copy e will be utilized.

Rules [a], [b] and [c] indicates that in the simulator $\Pi_1^\dagger$, the timer is used to impose a specific time by which a copy of a trigger o_k decides the rule that consumes it. If the said rule is non-cooperative, then simulating the rule only involves production of its RHS in the succeeding steps. However, if the involved rule is cooperative, there is a need to make sure that the LHS of the rule is satisfied. This implies making sure that other triggers exist and are consumed via the same rule. We shall call this LHS validation.

The following rules are added to simulate a coop-dis rule $r \in R_h$:

[d] $(o_{k,2k,r}e, in) \in {R'}^\dagger_{(h,r)}$ for $first(r) = o_k$

[e] $(o_{k_2,2k_2,r}e, in; o_{k_1,2k_2,r}e, out) \in {R'}^\dagger_{(h,r)}$
for $o_{k_1}, o_{k_2} \in supp(LHS(r)), last(r) \neq o_{k_1}, next(r, o_{k_1}) = o_{k_2}$

[f] $(o_{k,2k+2,r}e, out) \in {R'}^\dagger_{(h,r)}$ for $last(r) = o_k$

Rule [d] involves transferring a copy $o_{k,2k,r}$ in region (h, r) where $first(r) = o_k$. This rule is used to signal the start of the LHS validation for a single application of a cooperative rule r in the simulated Π_1. Afterwards, LHS validation is mainly executed by application of rule [e] to consecutive triggers of rule r (recall that determining consecutive triggers depends on a total order imposed on O). Suppose copies o_{k_1}, o_{k_2} both trigger rule r and $next(r, o_{k_1}) = o_{k_2}$. The copy in region (h, r), i.e. representing o_{k_1}, acts as a validator that a copy of the next trigger (o_{k_2}) exists in region (h) and shall be consumed via rule r. Rule [e] requires that a copy of both $o_{k_1,2k_2,r}$ and e be present in region (h, r). Similarly, a copy of both $o_{k_2,2k_2,r}$ and e must be present in region (h). Upon application of rule r, the involved triggers swap places so that a representation of o_{k_2} is present in region (h, r). This trigger will then be used for the next pairwise validation. Notice that both triggers must have the same timer value (equal to $2k_2$). The following rules in region (h, r) are employed to synchronize the timer of the involved triggers and to dissolve the validator once rule [e] is used, respectively. Rules [g],[h] and [i] below are added for all $o_{k_2}, o_{k_1} \in supp(LHS(r)), last(r) \neq o_{k_1}, next(r, o_{k_1}) = o_{k_2}$:

[g] $o_{k_1,p,r} \to o_{k_1,p+1,r} \in R^\dagger_{(h,r)}$ where $2k_1 \leq p < 2k_2 - 2$

[h] $o_{k_1,2k_2-2,r} \to o_{k_1,2k_2,r}e \in R^\dagger_{(h,r)}$ where $2k_1 \leq p < 2k_2 - 2$

[i] $o_{k_1,2k_2,r} \to \lambda \in R^\dagger_{(h)}$

In case the pairwise validation fails, i.e. there is no copy $o_{k_2,2k_2,r}$ in region (h), we use the next rules to produce a trap symbol and lead the simulator to a non-halting state so that no output can be produced. Rules [j] and [k] below are added for all $o_{k_2}, o_{k_1} \in supp(LHS(r)), last(r) \neq o_{k_1}, next(r, o_{k_1}) = o_{k_2}$:

[j] $o_{k_1,2k_2,r} \to \# \in R^\dagger_{(h,r)}$ [k] $\# \to \# \in R^\dagger_{(h,r)}$

To signal completion of the LHS validation for an application of rule r, rule [f] must be utilized. Rule [f] transfers last trigger from region (h, r) to region (h). The next rule is an additional rule in region (h, r) needed to produce a special object e and update the timer for rule [f]:

[l] $o_{k,2k,r} \to o_{k,2k+2,r}e \in R^{\dagger}_{(h,r)}$ where $last(r) = o_k$

(An example illustrating LHS validation for an application of a coop-dis rule is given in Appendix A.)

When a coop-dis rule in Π_1 involves an object $o_{|O|}$, then the LHS validation in $\Pi_1^{\dagger}$ for the said rule takes $2|O|+2$ steps. LHS validation for this rule takes the longest time. The next phase in simulating a rule is simulating the production of multisets in the right-hand side of the rules applied in a transition in Π_1. We shall call this phase RHS production. Before we proceed with RHS production, we make sure that all rule applications in a transition in Π_1 have accomplished LHS validation in $\Pi_1^{\dagger}$. For this purpose, the following rules are added in simulating a coop-dis rule $r \in R_h$:

[m] $o_{k,p,r} \to o_{k,p+1,r} \in R^{\dagger}_{(h)}$ for $last(r) = o_k$, $2k+2 \leq p < 2|O|+2$

Similarly, in simulating a non-cooperative rule $r \in R_h$, the following rules (for updating timers after applying rule [b]) are added:

[n] $o_{k,p,r} \to o_{k,p+1,r} \in R^{\dagger}_{(h)}$ for $last(r) = o_k$, $2k-1 \leq p < 2|O|+2$

When the timer of all remaining copies in a region (h) reaches $2|O|+2$, we proceed with RHS production. For this purpose, the next set of rules are adapted and slightly modified from [4]. For the next set of rules, we need to recall that for every rule $r \in R_h$, we declared multisets x_r, y_r and $z_{r,j}$ for every $j \in children(h)$ to represent the multisets produced by rule r in region h and its neighboring regions. The rules below are added in simulating a rule $r \in R_h$:

[o] $o_{k,2|O|+2,r} \to \tilde{v} \in R^{\dagger}_{(h)}$
for $last(r) = o_k$ and $\tilde{v}$ is a multiset formed from adding $\varkappa_r$, adding ς_r and e, and for every $j \in children(h)$, adding both $\varphi_{r,j}$ and e.

[p] $\varkappa_r \to \varkappa_r^{\bullet} \in R^{\dagger}_{(h)}$

[q] $\varkappa_r^{\bullet} \to \tilde{v} \in R^{\dagger}_{(h)}$
where $\tilde{v}$ is formed from adding $o_{k,0}$ for every o_k in the multiset x_r.

[r] $(\varphi_{r,j}e, in) \in {R'}^{\dagger}_{(j)}$ for $j \in children(h)$

[s] $\varphi_{r,j} \to \tilde{v} \in R^{\dagger}_{(j)}$
where $j \in children(h)$ and $\tilde{v}$ is formed from adding $o_{k,0}$ for every o_k in the multiset $z_{r,j}$.

[t] $(\varsigma_r e, out) \in {R'}^{\dagger}_{(h_1)}$ for $parent(h) = h_1$

[u] $\varsigma_r \to \tilde{v} \in R^{\dagger}_{(h_1)}$
where $h_1 \in parent(h)$, $h_1 \neq (0)$ and $\tilde{v}$ is formed from adding $o_{k,0}$ for every o_k in the multiset y_r.

[v] $\varsigma_r \to \tilde{v} \in R^{\dagger}_{(h_1)}$
where $h_1 \in parent(h)$, $h_1 = (0)$ and $\tilde{v}$ is formed from adding both $o_{k,0}$ and e for every o_k in the multiset y_r.

[w] $(o_{k,0}e, out) \in {R'}^{\dagger}_{(0)}$ for every $o_{k,0} \in O^{\dagger}$.

As in [4], using the rules above, it takes three steps to accomplish RHS production for a rule $r \in R_h$. The first step uses rule [o] and produces the symbols $\varkappa_r$, ε_r and $\varphi_{r,j}$. The next step involves the use of rules [p], [r] and [t]. Rule [p] will be used to evolve $\varkappa_r$ to $\varkappa_r^\bullet$ while rules [r] and [t] are used to transfer symbols ε_r and $\varphi_{r,j}$ to their respective regions, respectively. Finally, rules [q], [s], [u] and [v] are used to produce the multisets representing the multisets in the RHS of rule r. The timer in the subscript of the produced multiset will be reset to 0 indicating that the next transition in Π_1 is ready to be simulated.

Note that rule [v] is used in the case where region h is the skin (i.e. $h = 0$). In this case, the multiset z_r for rule r is communicated to the environment. In $\Pi_1^\dagger$, this is simulated by applying rule [v] and then, applying rule [w]. The extra step of applying rule [w] can cause an overlap on the simulation of the current and the next transition. However, since the first step in the simulation of the next transition doesn't involve region (0), the extra step in the simulation of the current transition doesn't effect the simulation of the next transition.

Lemma 1. *For each TP-ind system Π_1 where cooperative rules are coop-dis, we can construct an ECPe system $\Pi_1^\dagger$ such that each computation of length τ in Π_1 is simulated by an equivalent computation of length at most $((2|O| + 5)\tau) + 1$ in $\Pi_1^\dagger$ where O is the alphabet of Π_1 and $L(\Pi_1) = L(\Pi_1^\dagger)$.*

Proof. The ECPe system simulator $\Pi_1^\dagger$ is as constructed above. We note that the initial multiset $w_h^\dagger$ in $\Pi_1^\dagger$ is an exact representation of the initial multiset w_h in Π_1. The choice of applicable rules by trigger o_k in Π_1 is also exactly represented by rules [b] and [c] for the corresponding object $o_{k,2k-2}$ in $\Pi_1^\dagger$. From our construction, we have shown that applicable rules are simulated in at most $2|O| + 6$ steps. These can be broken down to the following phases:

(a) Exactly $2|O| + 2$ steps are needed to assign applicable rules to every copy of a trigger (via any one of rule [b] or [c]) and perform LHS validation (via rules [d] to [n]).
(b) Three steps are needed to perform RHS production on the applicable rules
(c) If a rule applied in Π_1 sends a multiset to the environment, an extra step is performed in $\Pi_1^\dagger$ to send the corresponding multiset to the environment.

The effect of simulating all applicable rules will be reflected in at most $2|O| + 6$ steps. Note that the optional last step (c) overlaps the simulation of the next transition. The description above shows that the non-deterministic and maximal parallelism property of a transition in Π_1 is respected in the corresponding computation of the transition in $\Pi_1^\dagger$. Also, all strings generated by a computation τ in Π_1 is also generated in $\Pi_1^\dagger$ in at most $((2|O| + 5)\tau) + 1$ steps.

While all rules in Π_1 are represented in $\Pi_1^\dagger$, there are additional computation paths in $\Pi_1^\dagger$ due to wrong guesses for item (a) above. However, rules [j] and [k] employed for LHS validation of a candidate applicable rule makes sure that the computation leads to a non-halting state and thus, produces no extra strings. This implies that all strings in $\Pi_1^\dagger$ is also generated in Π_1.

3.2 Simulating TP-ind system where cooperative rules are either coop-dis or coop-one only

Lemma 2. *Given a TP-ind system Π_2 where cooperative rules are either coop-dis or coop-one, we can construct an ECPe system $\Pi_2^\star$ such that each computation of length τ in Π_2 is simulated by an equivalent computation of length at most $((2|O|+5)\tau)+1$ in $\Pi_2^\star$ where O is the alphabet of Π_2 and $L(\Pi_2) = L(\Pi_2^\star)$.*

Proof. We now consider a TP-ind system Π_2 where cooperative rules are restricted to either coop-dis or coop-one. Our ECPe system simulator $\Pi_2^\dagger$ to simulate Π_2 will be an extension of the simulator $\Pi_1^\dagger$ defined in Section 3.1. Specifically, we construct our ECPe system simulator $\Pi_2^\dagger$ by starting with copying the definition of our simulator in Section 3.1. This means, simulation of non-cooperative rules and coop-dis rules are the same as in the previous subsection.

In order to simulate coop-one rules, there are several elements we add in our current $\Pi_2^\dagger$ simulator. First, as in simulating coop-dis rules, we add additional membrane (h, r) for every coop-one rule $r \in R_h$. The membrane (h, r) is also a child membrane of membrane (h). The set of symbols $\{\#_{(h)} \mid 1 \leq h \leq m\}$ are also added to the alphabet $O^\dagger$. We add the symbol $\#_{(h)}$ to the initial multiset of region (h). The role of these additional symbols will be discussed when we give the rules to simulate a coop-one rule.

Rules $[\mathrm{c}_1']$ and $[\mathrm{c}_2']$ below are added for every trigger $o_k \in supp(LHS(r))$ where $r \in R_h$ is a coop-one rule:

$[\mathrm{c}_1']$ $o_{k,2k-2} \to o_{k,2k,r}e \in R^\dagger_{(h)}$ $\qquad$ $[\mathrm{c}_2']$ $o_{k,2k-2} \to e \in R^\dagger_{(h)}$

The simulation of a coop-one rule is similar to that of a coop-dis rule. Specifically, when a copy $o_{k,0}$ (for a coop-one trigger o_k) reaches $o_{k,2k-2}$, the timer is added by two while producing a copy of the special object e as well (i.e. rule $[\mathrm{c}_1']$ and rule [c] in Section 3.1 are the same). The added one step accounts for transferring the produced $o_{k,2k,r}$ to the region (h, r). The object $o_{k,2k,r}$ symbolizes one application of rule r. However, since several copies of o_k is needed to trigger a coop-one rule r, we create another rule $[\mathrm{c}_2']$ which only produces a copy e upon consuming a copy $o_{k,2k-2}$. Suppose we simulate a transition in Π_2. If there are f copies of o_k in a configuration, then in $\Pi_2^\dagger$, there are also f copies of $o_{k,2k-2}$. If all copies of $o_{k,2k-2}$ are evolved to any of rules $[\mathrm{c}_1']$ and $[\mathrm{c}_2']$, then the total number of e's is equal to f. The next set of rules are added in the rules of $\Pi_2^\dagger$ for simulating a coop-one rule $r \in R_h$: Rules $[\mathrm{d}']$, $[\mathrm{l}']$ and $[\mathrm{f}']$ are added for every trigger $o_k \in supp(LHS(r))$ where $r \in R_h$ is a coop-one rule:

$[\mathrm{d}']$ $(o_{k,2k,r}e^f, in) \in {R'}^\dagger_{(h,r)}$ where $|LHS(r)|_{o_k} = f$.
$[\mathrm{l}'\ ]$ $o_{k,2k,r} \to o_{k,2k+2,r}e$
$[\mathrm{f}'\ ]$ $(o_{k,2k+2,r}e, out) \in {R'}^\dagger_{(h,r)}$

These set of rules are used to carry out the LHS validation of one application of rule r. Rule $[\mathrm{d}']$ implies that there are f copies of e for copy $o_{k,2k,r}$ to move to region (h, r). If there is enough o_k's in a configuration in Π_2, rule $[\mathrm{d}']$ can be

applied in $\Pi_2^\dagger$. The next two rules have the same effect as rule [l] and [f] in Section 3.1 since for a coop-one rule r having a trigger o_k, $first(r) = last(r) = o_k$.

When the timer associated for an object o_k is $2k-2$, it can be observed that the copies of e in the next step is solely dependent on the number of copies of $o_{k,2k-2}$. (This is caused only by the application of rules [c] (for coop-dis rules), [c$_1'$] or [c$_2'$] (for coop-one rules) for rules triggered by o_k). We now look at the scenario where there are two coop-one rules that require the same object o_k. Let these be rules r_1 and r_2 where $|LHS(r_1)|_{o_k} = f_1$ and $|LHS(r_2)|_{o_k} = f_2$. The $(f_1 + f_2)$ copies of o_k in Π_2 implies that there are $(f_1 + f_2)$ copies of $o_{k,2k-2}$ in $\Pi_2^\dagger$. Suppose all these copies use rules of the form [c$_1'$] and [c$_2'$], there will be $(f_1 + f_2)$ copies of e that can be used to transfer a copy of both $o_{k,2k,r_1}$ and $o_{k,2k,r_2}$ in their respective regions. However, in the case where both $o_{k,2k,r_1}$ and $o_{k,2k,r_2}$ are produced and the number of copies of e is not equal to $f_1 + f_2$, then either of the following cases (indicating failure of LHS validation for a coop-one rule) holds: (i) at least one of the objects $o_{k,2k,r_1}$ and $o_{k,2k,r_2}$ will remain in region h (ii)some extra copies of e occur in region h since they cannot be used to apply a rule [d$'$]. The same can be said when more than two rules require the same trigger. The following rules are used to force a non-halting computational path when LHS validation fails: (Recall that in the initial multiset of region (h), $1 \leq h \leq m$, in $\Pi_2^\dagger$, we added a copy of the symbol $\#_{(h)}$.) For every region h and $parent((h)) = (j)$, rules [x] to [ź] are added:

[x] $(\#_{(h)}e, out) \in {R'}^\dagger_{(j)}$ [y] $\#_{(h)} \to \#_{(h)} \in R^\dagger_{(j)}$

[z] $o_{k,2k,r} \to \# \in R^\dagger_{(h)}$ [ź] $\# \to \# \in R^\dagger_{(h)}$

In order to sync with how other type of rules are simulated, the next phase in simulating a coop-one rule r involves evolving the copy $o_{k,2k+2,r}$ in region (h) to copy $o_{k,2|O|+2,r}$ and then, carrying out RHS production phase. The rules are then the same as the rules in Section 3.1. Thus, for each coop-one rule $r \in R_h$, we further add rules [m], and rule [o] to rule [w].

In simulating coop-one rules, there are also cases where the system may guess incorrectly, e.g. although rule [d$'$] can be used, any of rules [x], [y], [z] and [ź] are chosen by the system instead. This results to additional non-halting computational paths. However, as in Lemma 1, since no extra strings are produced in the additional branches, the language of Π_2 and $\Pi_2^\dagger$ are the same.

3.3 Simulating TP-ind system where cooperative rules are either coop-dis, coop-one or coop-mul

The technique used for handling coop-dis and coop-one rules can both be used to handle coop-mul rules. Specifically:

Lemma 3. *Given a TP-ind system Π_3 where cooperative rules are either coop-dis, coop-one, or coop-mul, we can construct an ECPe system $\Pi_3^\dagger$ such that each computation of length τ in Π_3 is simulated by an equivalent computation of length at most $((2|O| + 5)\tau) + 1$ in $\Pi_3^\dagger$ where O is the alphabet of Π_3 and $L(\Pi_3) = L(\Pi_3^\dagger)$.*

Proof. The ECPe system simulator $\Pi_3^\dagger$ for a TP-ind system Π_3 is an extension of $\Pi_2^\dagger$. Coop-dis and coop-mul rules are handled the same way as the previously constructed simulators. We only describe the additional membranes and rules for handling coop-mul rules. The same reasoning as in the previous lemmas can be said about the constructed $\Pi_3^\dagger$.

For coop-mul rules, as in handling other types of coop-ind rules, we also allocate a membrane for a coop-mul rule $r \in R_h$. We label it as (h, r) and let $(h, r) \in children(h)$. The following rules are added to simulate a coop-mul rule:

- For every trigger $o_k \in supp(LHS(r))$ where $r \in R_h$ is a coop-mul rule:
  [c$''_1$] $o_{k,2k-2} \to o_{k,2k,r}e \in R^\dagger_{(h)}$ [c$''_2$] $o_{k,2k-2} \to e \in R^\dagger_{(h)}$
- For every coop-mul rule $r \in R_h$:
  [d$''$] $(o_{k,2k,r}e^f, in) \in {R'}^\dagger_{(h,r)}$ for $first(r) = o_k, |LHS(r)|_{o_k} = f$
  [e$''$] $(o_{k_2,2k_2,r}e^f, in; o_{k_1,2k_2,r}e, out) \in {R'}^\dagger_{(h,r)}$
  for $o_{k_1}, o_{k_2} \in supp(LHS(r)), last(r) \neq o_{k_1}$,
  $next(r, o_{k_1}) = o_{k_2}, |LHS(r)|_{o_{k_2}} = f$
  [f$''$] $(o_{k,2k+2,r}e, out) \in {R'}^\dagger_{(h,r)}$ for $last(r) = o_k$
- Similarly, we also add rules of the form [g] to [ź] to complete the simulation of one application of a coop-mul rule, considering both LHS validation and RHS production.

From the lemmas we have provided, the following theorem can be derived:

Theorem 1. *For each TP-ind system Π, we can construct an ECPe system $\Pi^\dagger$ such that each computation of length τ in Π is simulated by an equivalent computation of length at most $((2|O| + 5)\tau) + 1$ in $\Pi^\dagger$ where O is the alphabet of Π and $L(\Pi) = L(\Pi^\dagger)$.*

4 Conclusion

We are able to provide a simulation of TP-ind systems in ECPe systems. The cooperative rules in such systems are first categorized into three forms: coop-dis, coop-one and coop-mul. In our simulators, we maintain the hierarchical relations of membranes in the simulated system and added membranes for every cooperative rule. We also take note of the role of energy in our simulations. For coop-dis rule, the maximum energy needed for any communication rule is minimal (one for symport rules and two, one for each involved region, for antiport rules). However, for both coop-one and coop-mul rules, the maximum energy depends on the number of copies of a trigger that a certain rule requires. The number of communication steps depends on several factors: first, each rule application requires communication steps (during RHS production) dependent on the number of neighbors of a certain membrane. Also, for every application of a cooperative rule, there are additional communication steps (during LHS validation) dependent on the number of triggers required in the left-hand side of a cooperative rule. Finally, in our simulation, a TP-system that computes a string in τ steps is simulated by a computation with $((2|O| + 5)\tau) + 1$ steps.

We end our conclusion with several open problems for future work. First, it can be observed that in the design of our rules, we let the system decide the rule to be used when a copy of a trigger exists. Afterwards, the system validates whether the chosen rule can actually be applied. This manner of simulation leads to additional (non-halting) computational paths in the event of wrong guesses. We leave as an open problem the construction of simulators where such additional (non-halting) paths are eliminated. Also, we ask the following questions: can we construct ECPe system simulators for TP systems where antiports are eliminated? What about simulators for TP systems with dependent triggers?

References

1. A. Alhazov, C. Ciubotaru, S. Ivanov, Y. Rogozhin. The Family of Languages Generated by Non-cooperative Membrane Systems. LNCS 6501, 65–80, 2011.
2. H. Adorna, Gh. Păun, M. Pérez-Jiménez. On Communication Complexity in Evolution-Communication P systems. ROMJIST 13, 2, 113–130, 2010.
3. M. Cavaliere. Evolution-Communication P systems. LNCS 2597, 134–145, 2003.
4. R. Juayong, H. Adorna. Relating Computations in Non-cooperative Transition P systems and Evolution-Communication P systems with Energy. FI 136, 3, 209–217, 2015.
5. Gh. Păun: Computing with membranes. J. Computer and System Sciences 61, 108–143, 2000.
6. Păun, G.: Membrane Computing. Springer-Verlag Berlin Heidelberg (2002)
7. Păun, G.: Further twenty six open problems in membrane computing. Proc. of the Third BWMC, Sevilla, RGNC Report 01/2005, 249–262, 2005.
8. The P system website: `www.ppage.psystems.eu`

APPENDIX A

Example 1. Let Π_1 be a TP-ind system having a coop-dis rule r in region h. Let the multiset required to trigger rule r be multiset acd, i.e. $LHS(r) = acd$. Also, suppose Π_1 has the alphabet $O = \{a, b, c, d, e\}$. We can let a total order on O be based on how the symbols are positioned in the set (i.e. a is labelled o_1 and e is labelled o_5). Based on this labelling, the left-hand side of rule r becomes the multiset $o_1o_3o_4$. The next rules in $\Pi_1^\dagger$ simulates LHS validation for rule r:

$R^\dagger_{(h)}$

$a.1 : o_{3,0} \to o_{3,1}$	$c.1 : o_{1,0} \to o_{1,2,r}e$
$a.2 : o_{3,1} \to o_{3,2}$	$c.2 : o_{3,2} \to o_{3,4,r}e$
$a.3 : o_{3,2} \to o_{3,3}$	$c.3 : o_{4,6} \to o_{4,8,r}e$
$a.4 : o_{3,3} \to o_{3,4}$	$i.1 : o_{1,6,r} \to \lambda$
$a.5 : o_{4,0} \to o_{4,1}$	$i.2 : o_{3,8,r} \to \lambda$
$a.6 : o_{4,1} \to o_{4,2}$	$a.9 : o_{4,4} \to o_{4,5}$
$a.7 : o_{4,2} \to o_{4,3}$	$a.10 : o_{4,5} \to o_{4,6}$
$a.8 : o_{4,3} \to o_{4,4}$	

$R'^\dagger_{(h,r)}$	$R^\dagger_{(h,r)}$
$d.1 : (o_{1,2,r}e, in)$	$g.1 : o_{1,2,r} \to o_{1,3,r}$
$e.1 : (o_{3,6,r}e, in; o_{1,6,r}e, out)$	$g.2 : o_{1,3,r} \to o_{1,4,r}$
$e.2 : (o_{4,8,r}e, in; o_{3,8,r}e, out)$	$h.1 : o_{1,4,r} \to o_{1,6,r}e$
$f.1 : (o_{4,10,r}e, out)$	$h.2 : o_{3,6,r} \to o_{3,8,r}e$
	$j.1 : o_{1,4,r} \to \#$
	$j.2 : o_{3,8,r} \to \#$
	$k.1 : \# \to \#$
	$l.1 : o_{4,8,r} \to o_{4,10,r}e$

Suppose the multiset $o_1o_3o_4$ exists in region h of Π_1. Then, the corresponding multiset in region (h) of $\Pi_1^\dagger$ is $o_{1,0}o_{3,0}o_{4,0}$. The sequence of transitions showing LHS validation for a single application of rule r are as follows:
$[o_{1,0}\ o_{3,0}\ o_{4,0}[\ ]_{(h,r)}]_{(h)} \Rightarrow [o_{1,2,r}\ e\ o_{3,1}\ o_{4,1}[\ ]_{(h,r)}]_{(h)}$ via rules $c.1$, $a.1$ and $a.5$;
$[o_{1,2,r}\ e\ o_{3,1}\ o_{4,1}[\ ]_{(h,r)}]_{(h)} \Rightarrow [o_{3,2}\ o_{4,2}[o_{1,2,r}]_{(h,r)}]_{(h)}$ via rules $d.1$, $a.2$ and $a.6$;
$[o_{3,2}\ o_{4,2}[o_{1,2,r}]_{(h,r)}]_{(h)} \Rightarrow [o_{3,3}\ o_{4,3}[o_{1,3,r}]_{(h,r)}]_{(h)}$ via rules $g.1$, $a.3$ and $a.7$;
$[o_{3,3}\ o_{4,3}[o_{1,3,r}]_{(h,r)}]_{(h)} \Rightarrow [o_{3,4}\ o_{4,4}[o_{1,4,r}]_{(h,r)}]_{(h)}$ via rules $g.2$, $a.4$ and $a.8$;
$[o_{3,4}\ o_{4,4}[o_{1,4,r}]_{(h,r)}]_{(h)} \Rightarrow [o_{3,6,r}\ e\ o_{4,5}[o_{1,6,r}\ e]_{(h,r)}]_{(h)}$ via rules $c.2$, $h.1$ and $a.9$;
$[o_{3,6,r}\ e\ o_{4,5}[\ o_{1,6,r}\ e]_{(h,r)}]_{(h)} \Rightarrow [o_{1,6,r}\ o_{4,6}[\ o_{3,6,r}]_{(h,r)}]_{(h)}$ via rules $e.1$ and $a.10$;
$[o_{1,6,r}\ o_{4,6}[o_{3,6,r}]_{(h,r)}]_{(h)} \Rightarrow [o_{4,8,r}\ e[o_{3,8,r}\ e]_{(h,r)}]_{(h)}$ via rules $c.3$, $h.2$ and $i.1$;
$[o_{4,8,r}\ e[o_{3,8,r}\ e]_{(h,r)}]_{(h)} \Rightarrow [o_{3,8,r}[o_{4,8,r}]_{(h,r)}]_{(h)}$ via rule $e.3$;
$[o_{3,8,r}[o_{4,8,r}]_{(h,r)}]_{(h)} \Rightarrow [[o_{4,10,r}\ e]_{(h,r)}]_{(h)}$ via rules $i.2$ and $l.1$;
$[[o_{4,10,r}\ e]_{(h,r)}]_{(h)} \Rightarrow [o_{4,10,r}[\ ]_{(h,r)}]_{(h)}$ via rule $f.1$.
The computation above doesn't make use of rules $j.1$ and $j.2$ although they may be used in the presence of copies $o_{1,6,r}$ and $o_{3,8,r}$. Upon use of any of these rules, a trap symbol # will be produced. A LHS validation that makes use of any of rules $j.1$ and $j.2$ is not successful since a trap symbol # enables rule $k.1$ leading the system to a non-halting state.

Suppose only the multiset $o_{1,0}o_{3,0}$ exists in region (h), then rule $j.1$ is inevitably used. Similarly, rule $j.2$ will be used when the multiset in region h is $o_{1,0}o_{4,0}$. Both indicates that LHS validation for a single rule application of rule r fails.

Deterministic Transition P Systems Modeled as Register Machines

Yun-Bum Kim and Michael J. Dinneen

Department of Computer Science, University of Auckland,
Private Bag 92019, Auckland, New Zealand
tkim021@aucklanduni.ac.nz, mjd@cs.auckland.ac.nz

Abstract. This paper presents the details for constructing register machines that simulate deterministic transition P systems with rule priorities. Our conversion preserves the run-time complexity (the number of machine instruction steps compared to the number of rewriting steps of P systems) within a small constant factor. We illustrate our conversion with a non-trivial example.

Keywords: P systems, register machines, modeling, simulation.

1 Introduction

In theoretical computer science, there are many computational models that are equivalent to the computation power of a Turing machine. Two such models are register machines and, more recently, membrane systems. Register machines have been proposed in many flavors (mainly by theoreticians). These machines have a common theme of having a finite set of registers that can represent arbitrarily large non-negative integers. Also these machines are presented as a finite sequence taken from a small set of basic instructions (e.g. to do arithmetic, data handling and flow control). P systems [10, 12] (also called membrane systems) are distributed and parallel computing models, inspired by the structure and function of living cells. Several variants of P systems [9, 8] have been introduced, inspired from various features of living cells, that provide new ways to process information and solve computational problems of interest. Essentially, all P system models have a structure consisting of connected cells and a set of evolution rules that govern their evolution over time.

Several studies have investigated the relationship between P systems and register machines [7, 4] and presented universality results by proving that P systems can simulate a universal register machine [5]. Previously, in a companion paper [5], we presented the details for constructing an efficient P system from an arbitrary register machine [2]. In this paper we address the opposite direction of mapping deterministic P systems to register machines. Our motivation is based on the fact that it is easier to design parallel algorithms using P systems instead of sequential-based classical (i.e. von Neumann architecture) computer models. Thus, we want to automate the conversion of a P system framework

to the existing computers, which today have multi-core CPUs and many-core GPUs available. We want to make the argument that today's computers are closely modeled as "parallel" register machines (but with memory constraints of size and communication latency).

This paper is organized as follows. Section 2 recalls several key mathematical concepts that are used in this paper. Section 3 provides the definition of a transition P system model. Section 4 presents the definition of a register machine model. Section 5 presents the construction details for building a register machine that simulates a transition P system. Section 6 gives a non-trivial concrete mapping of a P system to a register machine. Finally, Section 7 summarizes this paper and provides some future areas of study.

2 Preliminaries

This section covers several key mathematical concepts that are used in this paper, such as sets, strings, multisets and graphs.

An *alphabet* is a finite non-empty set with elements called *symbols*. A *string* over alphabet O is a finite sequence of symbols from O. The set of all strings over O is denoted by O^*. The length of a string $x \in O^*$, denoted by $|x|$, is the number of symbols in x. The number of occurrences of a symbol $o \in O$ in a string x over O is denoted by $|x|_o$. The *empty string* is denoted by λ.

A multiset over an alphabet O is represented as strings over O, such as $o_1^{n_1} \dots o_k^{n_k}$, where $o_i \in O$ and $n_i \geq 0$, for $1 \leq i \leq k$. The *multiplicity* of an element x in a multiset v is denoted by $|v|_x$. We say that a multiset v is *included* in a multiset w, denoted by $w \subseteq v$, if, for all $o \in O$, $|w|_o \leq |v|_o$. The *union* of multisets v and w, denoted by $v \cup w$, is a multiset x, such that, for all $o \in O$, $|x|_o = |v|_o + |w|_o$. The *difference* of multisets v and w, denoted by $v - w$, is a multiset x, such that, for all $o \in O$, $|x|_o = \max(|v|_o - |w|_o, 0)$. The empty multiset is represented by λ. A set that contains the distinct elements of a multiset v is denoted by $\texttt{distinct}(v)$.

A (binary) *relation* R over two sets X and Y is a subset of their Cartesian product, $R \subseteq X \times Y$. For $A \subseteq X$ and $B \subseteq Y$, we set $R(A) = \{y \in Y \mid \exists x \in A, (x, y) \in R\}$, $R^{-1}(B) = \{x \in X \mid \exists y \in B, (x, y) \in R\}$.

A *directed graph* (digraph) is a pair (V, A), where V is a finite set of elements called nodes (or vertices), and A is a set of ordered pairs of V called *arcs*. Given a digraph $D = (V, A)$, for $v \in V$, the *parents* of v are $A^{-1}(v) = A^{-1}(\{v\})$ and the *children* of v are $A(v) = A(\{v\})$.

3 Transition P Systems

Definition 1 (Transition P systems). A deterministic transition P system (of order $n \geq 1$) is a construct of the form:

$$\Pi = (O, K, \Delta, W, R)$$

where:

1. O is the finite and non-empty alphabet of symbols.
2. $K = \{\sigma_i \mid 1 \leq i \leq n\}$ is the set of cells, where i represents the *cell ID* of σ_i.
3. Δ is an irreflexive and asymmetric relation on K, representing a set of arcs between cells with *bidirectional* communication capabilities, (i.e. parents can communicate to their children and vice versa).
4. $W = \{w_i \mid 1 \leq i \leq n\}$, where $w_i \in O^*$ is a multiset of symbols, called the *content*, currently present in cell σ_i.
5. $R = \{R_i \mid 1 \leq i \leq n\}$, where R_i is a finite set of evolution rules that are associated with cell σ_i. An evolution rule $r \in R_i$ is a *linearly ordered* transition multiset rewriting rule of the form:

$$r : \; j \; u \to v$$

where:

- $j \in \{1, 2, \ldots, |R_i|\}$ indicates the *priority order* of r, where the lower value j indicates higher priority.
- $u \in O^+$.
- $v \in (O \times \tau)^*$, where $\tau \in \{\odot, \uparrow, \downarrow\}$ is a set of *target indicators*. Note that $(o, \odot) \in v$, $o \in O$, is abbreviated to o. Moreover, we denote:
  - multiset $v_\odot = \{o \mid (o, \odot) \in v\}$,
  - multiset $v_\uparrow = \{o \mid (o, \uparrow) \in v\}$ and
  - multiset $v_\downarrow = \{o \mid (o, \downarrow) \in v\}$.

Thanks to the unique priority assigned to each rule in the item 5 above, this transition P system is a deterministic model.

A cell *evolves* by applying one or more rules, which can change its content and can send multisets to its parent and child cells. For a cell $\sigma_i \in K$, a rule $j \; u \to v \in R_i$ is *applicable*, if $u \subseteq w_i$. The rules are applied in the *weak priority* order [11], i.e. higher priority applicable rules are applied, as many times as possible, before lower priority applicable rules. All applicable rules of all cells are applied simultaneously in one step. A computation *halts*, if none of the cells can evolve. The *output* of a halted transition P system computation is defined by the multiset of symbols present inside each cell $\sigma_i \in K$.

Applying an applicable rule $j \; u \to v$ in cell σ_i at step $k \geq 1$: (i) consumes multiset u at step k, i.e. $w_i = w_i - u$, (ii) produces multiset $v_\odot$, which will become available to σ_i at step $k + 1$, (iii) sends multiset $v_\uparrow$ to every parent cell $\sigma_p \in \Delta^{-1}(i)$ and sends multiset $v_\downarrow$ to every child cell $\sigma_c \in \Delta(i)$, which will

become available to σ_p and σ_c at step $k+1$. We denote the multiset produced in cell σ_i in the current step by $\overline{w_i}$, i.e. $\overline{w_i} = \{v_\odot \mid j\ u \to v \in R_i\} + \{v_\uparrow \mid j\ u \to v \in R_c, \sigma_c \in \Delta(i)\} + \{v_\downarrow \mid j\ u \to v \in R_p, \sigma_p \in \Delta^{-1}(i)\}$. At the end of step k, σ_i updates its content as $w_i = w_i + \overline{w_i}$. The following pseudocode describes the behavior of the transition P system Π of Definition 1 at each step $k \geq 1$. This pseudocode terminates when it reaches line 19.

```
1  bool evolve := false
2  for σ_i, i = 1, 2, ... |K|
3     for j = 1, 2, ..., |R_i|
4        r := (j u → v) ∈ R_i
5        while (u ⊆ w_i)
6           evolve := true
7           w_i := w_i − u
8           w̄_i := w̄_i + v_⊙
9           foreach σ_p ∈ Δ^{-1}(i)
10             w̄_p := w̄_p + v_↑
11          endfor
12          foreach σ_c ∈ Δ(i)
13             w̄_c := w̄_c + v_↓
14          endfor
15       endwhile
16    endfor
17 endfor
18 if (evolve = false) then
19    system Π halts
20 endif
20 foreach σ_i ∈ K
21    w_i := w_i + w̄_i
22    w̄_i := ∅
23 endfor
24 goto line 1
```

4 Register Machines

The register machine model used in this paper extends the register machine of [2] by adding an instruction that performs subtraction. A register machine has $n > 1$ instructions and $m > 0$ registers, where each register may hold an arbitrarily large non-negative integer.

A register machine program consists of a finite list of instructions, `EQ`, `SET`, `ADD`, `SUB`, `READ` and `HALT`, followed by an optional *input data*, denoted as a sequence of bits, with the restriction that the `HALT` instruction appears only once as the last instruction of the list. The first instruction of a program is indexed at address (i.e. line number) 0, and any value greater than or equal to n denotes the illegal branch error. In general, a register machine program is

presented in: (i) *symbolic instruction* form or (ii) *machine instruction* (i.e. raw binary) form. In this paper we adopt the symbolic instruction form, where labels of the form "L_x:" are added to make the presentation more readable.

A set of instructions of a register machine M, denoted in Chaitin's style [3], is described below. In the instructions below, variables z_1, z_2 and z_3 denote registers and k denotes a non-negative binary integer constant. The content of register z_i, $1 \leq i \leq 3$, is denoted by $\texttt{value}(z_i)$.

Instruction	**Description**
(EQ, z_1, z_2, z_3) or (EQ, z_1, k, z_3)	If $\texttt{value}(z_1) = \texttt{value}(z_2)$ or $\texttt{value}(z_1) = k$, then the execution of M continues at the $\texttt{value}(z_3)$-th instruction in the sequence. Otherwise, the execution of M continues at the next instruction.
(SET, z_1, z_2) or (SET, z_1, k)	$\texttt{value}(z_2)$ or the constant k is assigned to register z_1.
(ADD, z_1, z_2) or (ADD, z_1, k)	$\texttt{value}(z_1)+\texttt{value}(z_2)$ or $\texttt{value}(z_1)+k$ is assigned to register z_1.
(SUB, z_1, z_2) or (SUB, z_1, k)	$max\{\texttt{value}(z_1)-\texttt{value}(z_2), 0\}$ or $max\{\texttt{value}(z_1)-k, 0\}$ is assigned to register z_1.
(READ, z_1)	One bit is read into r_1, so the numerical value of z_1 becomes either 0 or 1. Any attempt to read past the last data-bit results in a run-time error.
(HALT)	This is the last instruction of the register machine program.

In the following, we will not use the READ instruction in our translation from P systems to register machines.

5 Translating P Systems into Register Machines

This section presents the details for building a register machine program I_{M_Π} for a register machine M_Π that simulates a deterministic transition P system $\Pi = (O, K, \Delta, W, R)$ of Definition 1. The instructions of I_{M_Π} are in the symbolic instruction form, separated by white space.

We present two pseudocodes, side by side, with corresponding lines, where:

- **Left:** describes evolution of a transition P system Π (according to Section 3).
- **Right:** gives the details for building I_{M_Π}. The methods used in this pseudocode, such as INITIALIZE, CONSUME, PRODUCE, APPLICABLE and EXECUTE, are described in Sections 5.1, 5.2, 5.3, 5.4 and 5.5, respectively.

In a translated register machine, multisets are represented as follows. Register o_i stores the multiplicity of symbol $o \in O$ in cell $\sigma_i \in K$, i.e. multiset $\{o^{o_i} \mid o \in O\}$ equals w_i. Register $\overline{o_i}$ gives the multiplicity of symbol $o \in O$ to be stored into cell $\sigma_i \in K$ in the next step, i.e. multiset $\{o^{\overline{o_i}} \mid o \in O\}$ equals $\overline{w_i}$.

```
1
2   bool evolve := false
3   for σ_i, i = 1, 2, ... |K|
4       for j = 1, 2, ..., |R_i|
5           r := (j  u → v) ∈ R_i
6           while (u ⊆ w_i)
7               evolve := true
8               w_i := w_i − u
9               w̄_i := w̄_i + v_⊙
10              foreach σ_p ∈ Δ^{-1}(i)
11                  w̄_p := w̄_p + v_↑
12              endfor
13              foreach σ_c ∈ Δ(i)
14                  w̄_c := w̄_c + v_↓
15              endfor
16          endwhile
17      endfor
18  endfor
19  if (evolve = false) then
20      HALT
21  endif
22  foreach σ_i ∈ K
23      w_i := w_i + w̄_i
24      w̄_i := ∅
25  endfor
26  goto line 2
```

```
1   INITIALIZE()
2   append L_STEP: (SET, evolve, 0)
3   for σ_i, i = 1, 2, ..., |K|
4       for j = 1, 2, ..., |R_i|
5           r := (j  u → v) ∈ R_i
6           APPLICABLE(i, r, |R_i|)
7           append (SET, evolve, 1)
8           CONSUME(i, r)
9           PRODUCE(i, r, ⊙)
10          foreach σ_p ∈ Δ^{-1}(i)
11              PRODUCE(p, r, ↑)
12          endfor
13          foreach σ_c ∈ Δ(i)
14              PRODUCE(c, r, ↓)
15          endfor
16          append (EQ, a, a, L_R(i,j))
17      endfor
18  endfor
19
20  append L_R(|K|+1,1): (EQ, evolve, 0, L_HALT)
21
22  foreach σ_i ∈ K
23      EXECUTE(i)
24  endfor
25
26  append (EQ, a, a, L_STEP)
27  append L_HALT:(HALT)
```

5.1 INITIALIZE method

This method sets register o_i with the multiplicity of symbol $o \in O$ in cell $\sigma_i \in K$. For example, for cell σ_i with content $w_i = aabc \in O^*$, the values of registers a_i, b_i and c_i are 2, 1 and 1, respectively.

```
1   INITIALIZE()
2       foreach σ_i ∈ K
3           foreach o ∈ O
4               append (SET,  o_i,  |w_i|_o)
5           endfor
6       endfor
```

Proposition 2. `INITIALIZE` *appends* $|K| \cdot |O|$ *instructions.*

5.2 CONSUME method

This method implements $w_i := w_i - u$ of line 8, which corresponds to a cell consuming the multiset u.

```
1  CONSUME(cell_ID i, rule r = j u → v)
2      foreach o ∈ distinct(u)
3          append (SUB, o_i, |u|_o)
4      endfor
```

A difference of multisets operation $w_i := w_i - u$ transforms w_i, such that $|w_i|_o = |w_i|_o - |u|_o$ for each $o \in O$. An instruction (SUB, o_i, $|u|_o$), appended for each $o \in O$, subtracts the value $|u|_o$ to register o_i.

Proposition 3. *For a rule $r = j\ u \to v$,* CONSUME *appends* $|\texttt{distinct}(u)|$ *instructions.*

5.3 PRODUCE method

This method implements $\overline{w_i} := \overline{w_i} + v_\tau$, $\tau \in \{\odot, \uparrow, \downarrow\}$, of lines 9, 11 and 14, which determines a multiset to be produced and stored in $\sigma_i \in K$.

```
1  PRODUCE(cell_ID i, rule r = j u → v, target τ)
2      foreach o ∈ distinct(v_τ)
3          append (ADD, o_i-bar, |v_τ|_o)
4      endfor
```

A union of multisets operation $\overline{w_i} := \overline{w_i} + v_\tau$ transforms $\overline{w_i}$, such that $|\overline{w_i}|_o = |\overline{w_i}|_o + |v_\tau|_o$ for each $o \in O$. An instruction (ADD, $\overline{o_i}$, $|v_\tau|_o$), appended for each $o \in O$, adds the value $|v_\tau|_o$ to register $\overline{o_i}$.

Proposition 4. *For a rule $r = u \to v$, with target indicator $\tau \in \{\odot, \uparrow, \downarrow\}$,* PRODUCE *appends* $|\texttt{distinct}(v_\tau)|$ *instructions.*

5.4 APPLICABLE method

This method, together with "append (EQ, $a, a, L_{\mathtt{R}(i,j)}$)" of line 16, implements the `while` statement of line 6, which involves a cell to check if its content contains the multiset specified on the left-hand side of a rule.

```
1  APPLICABLE(cell_ID i, rule r = j u → v, rulesetSize n)
2      append L_R(i,j):
3      foreach o ∈ distinct(u)
4          for m = 0, 1, ..., |u|_o − 1
5              if (j < n) then
6                  append (EQ, o_i, m, L_R(i,j+1))
7              else
8                  append (EQ, o_i, m, L_R(i+1,1))
10             endif
11         endfor
12     endfor
```

Condition $u \subseteq w_i$ is false, if there is a $o \in O$, such that $|u|_o > |w_i|_o$. For each $o \in \texttt{distinct}(u)$, APPLICABLE generates $|u|_o$ instructions below:

$$\begin{aligned} L_{\mathtt{R}(i,j)}\colon\ & (\texttt{EQ},\ o_i,\ 0,\ L) \\ & (\texttt{EQ},\ o_i,\ 1,\ L) \\ & (\texttt{EQ},\ o_i,\ 2,\ L) \\ & \vdots \\ & (\texttt{EQ},\ o_i,\ |u|_o - 1,\ L') \end{aligned}$$

which check the condition $\texttt{value}(o_i) \geq |u|_o$. If $\texttt{value}(o_i) \leq |u|_o - 1$, then, by one of these instructions, the execution continues to the line specified by the label L or L', which indicates the line number $k+1$, where line k contains instruction $(\texttt{EQ}, a, a, L_{\mathtt{R}(i,j)})$. If $\texttt{value}(o_i) \geq |u|_o - 1$ for all $o \in \texttt{distinct}(u)$, then the execution continues to the next instruction, and eventually, reaches instruction $(\texttt{EQ}, a, a, L_{\mathtt{R}(i,j)})$ that prompts an unconditional jump back to the line with the label $L_{\mathtt{R}(i,j)}$.

We note that a slight optimization in number of steps is possible if $|u|_o > 5$, where we can replace the sequence of $(\texttt{EQ},\ o_i, \dots)$ with a direct test of register machine instructions that check $\texttt{value}(o_i) < |u|_o$. However, in practice we believe rules have small $|u|_o$.

```
1  APPLICABLE(cell_ID i, rule r = j u → v, rulesetSize n)
2      append L_R(i,j): (SET, t1, |u|_o)
3      append (SUB, t1, 1)
4      append (SET, t2, |w_i|_o)
5      append (SUB, t2, t1)
6      if (j < n) then
7          append (EQ, t2, 0, L_R(i,j+1))
8      else
9          append (EQ, t2, 0, L_R(i+1,1))
10     endif
```

Proposition 5. *For a rule $r = j\ u \to v$,* APPLICABLE *will append at most* $\min(|u|, 5 \cdot |\texttt{distinct}(u)|)$ *instructions.*

5.5 EXECUTE method

This method implements $w_i := w_i + \overline{w_i}$ of line 23 and $\overline{w_i} := \emptyset$ of line 24, which represent cells updating their current content with the multiset produced from the execution of the rules.

```
1  EXECUTE(cell_ID i)
2      foreach o ∈ O
3          append (ADD, o_i, ō_i)
4          append (SET, ō_i, 0)
5      endfor
```

A union of multiset operation $w_i := w_i + \overline{w_i}$ transforms w_i, such that $|w_i|_o = |w_i|_o + |\overline{w_i}|_o$ for each $o \in O$. An instruction (ADD, o_i, $\overline{o_i}$), appended for each $o \in O$, adds the value of register $\overline{o_i}$ to register o_i. A multiset assignment operation $\overline{w_i} := \emptyset$ transforms w_i, such that $|\overline{w_i}|_o = 0$ for each $o \in O$. An instruction (SET, $\overline{o_i}$, 0), appended for each $o \in O$, sets the value of register $\overline{o_i}$ to 0.

Proposition 6. EXECUTE *appends* $2 \cdot |O|$ *instructions.*

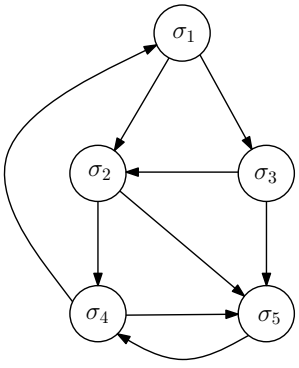
6 Translation Example

We illustrate a non-trivial example for the following (deterministic) transition P system $\Pi_{\text{BFS}} = (O, K, \Delta, W, R)$. The system Π_{BFS}, starting with cell $\sigma_1 \in K$, visits all cells in breadth-first search (BFS) manner.

- $O = \{a, b\}$.
- $K = \{\sigma_1, \sigma_2, \ldots, \sigma_5\}$, where σ_1 represents the initiator.
- $\Delta = \{(\sigma_1, \sigma_2), (\sigma_1, \sigma_3), (\sigma_2, \sigma_4), (\sigma_2, \sigma_5), (\sigma_3, \sigma_2), (\sigma_3, \sigma_5), (\sigma_4, \sigma_1), (\sigma_4, \sigma_5), (\sigma_5, \sigma_4)\}$. Figure 1 (left) shows the membrane structure of the system Π_{BFS}.
- $w_1 = \{aab\}$ and $w_j = \{aa\}$, for $2 \leq j \leq 5$.
- Each R_i, $1 \leq i \leq 5$, consists of the following two evolutions rules.
  1 $a\ a\ b \rightarrow (b, \downarrow)$
  2 $b \rightarrow \lambda$

Initially, only σ_1 contains one copy of symbol b. By rule 1, when a cell contains symbol b, it sends one copy of symbol b to all its children. By rule 2, cells consume any additional copies of symbol b received from their parents. Note that these algorithmic rules work for alternative Δ structures.

Figure 1 (right) shows evolution trace, i.e. content of each cell at each step, of the system Π_{BFS}. Starting from cell σ_1, at each step $k \geq 0$, cells in level k with respect to σ_1 are visited (i.e. receive symbol b), e.g. cells σ_2 and σ_3 receive symbol b at step 1.



Step	σ_1	σ_2	σ_3	σ_4	σ_5
0	a^2b	a^2	a^2	a^2	a^2
1		a^2b	a^2b	a^2	a^2
2		b		a^2b	a^2b^2
3	b			b	b
4					

Fig. 1. Left: the membrane structure of the system Π_{BFS}. Right: evolution traces of the system Π_{BFS}.

The following table contains the register machine program $I_{\Pi_{\text{BFS}}}$, which simulates the transition P system Π_{BFS}. $I_{\Pi_{\text{BFS}}}$ is generated by translating Π_{BFS} according to the pseudocode given in Section 5.

Line	Instruction
0	(SET, a_1, 2)
1	(SET, b_1, 1)
2	(SET, a_2, 2)
3	(SET, b_2, 0)
4	(SET, a_3, 2)
5	(SET, b_3, 0)
6	(SET, a_4, 2)
7	(SET, b_4, 0)
8	(SET, a_5, 2)
9	(SET, b_5, 0)
10	$L_{\texttt{STEP}}$: (SET, evolve, 0)
11	$L_{\texttt{R(1,1)}}$: (EQ, a_1, 0, $L_{\texttt{R(1,2)}}$)
12	(EQ, a_1, 1, $L_{\texttt{R(1,2)}}$)
13	(EQ, b_1, 0, $L_{\texttt{R(1,2)}}$)
14	(SET, evolve, 1)
15	(SUB, a_1, 2)
16	(SUB, b_1, 1)
17	(ADD, $\overline{b_2}$, 1)
18	(ADD, $\overline{b_3}$, 1)
19	(EQ, a, a, $L_{\texttt{R(1,1)}}$)
20	$L_{\texttt{R(1,2)}}$: (EQ, b_1, 0, $L_{\texttt{R(2,1)}}$)
21	(SET, evolve, 1)
22	(SUB, b_1, 1)
23	(EQ, a, a, $L_{\texttt{R(1,2)}}$)
24	$L_{\texttt{R(2,1)}}$: (EQ, a_2, 0, $L_{\texttt{R(2,2)}}$)
25	(EQ, a_2, 1, $L_{\texttt{R(2,2)}}$)
26	(EQ, b_2, 0, $L_{\texttt{R(2,2)}}$)
27	(SET, evolve, 1)
28	(SUB, a_2, 2)
29	(SUB, b_2, 1)
30	(ADD, $\overline{b_4}$, 1)
31	(ADD, $\overline{b_5}$, 1)
32	(EQ, a, a, $L_{\texttt{R(2,1)}}$)
33	$L_{\texttt{R(2,2)}}$: (EQ, b_2, 0, $L_{\texttt{R(3,1)}}$)
34	(SET, evolve, 1)
35	(SUB, b_2, 1)
36	(EQ, a, a, $L_{\texttt{R(2,2)}}$)
37	$L_{\texttt{R(3,1)}}$: (EQ, a_3, 0, $L_{\texttt{R(3,2)}}$)
38	(EQ, a_3, 1, $L_{\texttt{R(3,2)}}$)
39	(EQ, b_3, 0, $L_{\texttt{R(3,2)}}$)
40	(SET, evolve, 1)
41	(SUB, a_3, 2)
42	(SUB, b_3, 1)
43	(ADD, $\overline{b_2}$, 1)
44	(ADD, $\overline{b_5}$, 1)
45	(EQ, a, a, $L_{\texttt{R(3,1)}}$)
46	$L_{\texttt{R(3,2)}}$: (EQ, b_3, 0, $L_{\texttt{R(4,1)}}$)
47	(SET, evolve, 1)
48	(SUB, b_3, 1)
49	(EQ, a, a, $L_{\texttt{R(3,2)}}$)

Line	Instruction
50	$L_{\texttt{R(4,1)}}$: (EQ, a_4, 0, $L_{\texttt{R(4,2)}}$)
51	(EQ, a_4, 1, $L_{\texttt{R(4,2)}}$)
52	(EQ, b_4, 0, $L_{\texttt{R(4,2)}}$)
53	(SET, evolve, 1)
54	(SUB, a_4, 2)
55	(SUB, b_4, 1)
56	(ADD, $\overline{b_1}$, 1)
57	(ADD, $\overline{b_5}$, 1)
58	(EQ, a, a, $L_{\texttt{R(4,1)}}$)
59	$L_{\texttt{R(4,2)}}$: (EQ, b_4, 0, $L_{\texttt{R(5,1)}}$)
60	(SET, evolve, 1)
61	(SUB, b_4, 1)
62	(EQ, a, a, $L_{\texttt{R(4,2)}}$)
63	$L_{\texttt{R(5,1)}}$: (EQ, a_5, 0, $L_{\texttt{R(5,2)}}$)
64	(EQ, a_5, 1, $L_{\texttt{R(5,2)}}$)
65	(EQ, b_5, 0, $L_{\texttt{R(5,2)}}$)
66	(SET, evolve, 1)
67	(SUB, a_5, 2)
68	(SUB, b_5, 1)
69	(ADD, $\overline{b_4}$, 1)
70	(EQ, a, a, $L_{\texttt{R(5,1)}}$)
71	$L_{\texttt{R(5,2)}}$: (EQ, b_5, 0, $L_{\texttt{R(6,1)}}$)
72	(SET, evolve, 1)
73	(SUB, b_5, 1)
74	(EQ, a, a, $L_{\texttt{R(5,2)}}$)
75	$L_{\texttt{R(6,1)}}$: (EQ, evolve, 0, $L_{\texttt{HALT}}$)
76	(ADD, a_1, $\overline{a_1}$)
77	(SET, $\overline{a_1}$, 0)
78	(ADD, b_1, $\overline{b_1}$)
79	(SET, $\overline{b_1}$, 0)
80	(ADD, a_2, $\overline{a_2}$)
81	(SET, $\overline{a_2}$, 0)
82	(ADD, b_2, $\overline{b_2}$)
83	(SET, $\overline{b_2}$, 0)
84	(ADD, a_3, $\overline{a_3}$)
85	(SET, $\overline{a_3}$, 0)
86	(ADD, b_3, $\overline{b_3}$)
87	(SET, $\overline{b_3}$, 0)
88	(ADD, a_4, $\overline{a_4}$)
89	(SET, $\overline{a_4}$, 0)
90	(ADD, b_4, $\overline{b_4}$)
91	(SET, $\overline{b_4}$, 0)
92	(ADD, a_5, $\overline{a_5}$)
93	(SET, $\overline{a_5}$, 0)
94	(ADD, b_5, $\overline{b_5}$)
95	(SET, $\overline{b_5}$, 0)
96	(EQ, a, a, $L_{\texttt{STEP}}$)
97	$L_{\texttt{HALT}}$: (HALT)

7 Conclusions

The main result of this paper is a procedure that takes a transition P system and converts it to an equivalent register machine with the same run-time complexity. We hope to exploit the register machine model on conventional parallel computers, where registers are mapped to dynamic memory and few synchronization issues are needed.

As possible future work, we are interested in extending our preliminary results of [5] and the results of this paper by using more practical register machines and P systems, e.g. [9, 6, 1]. P systems with active membranes [11] extends transition P systems by incorporating membrane handling rules that support *membrane creation* operation (which adds new cells to the system) and *membrane dissolution* operation (which removes existing cells from the system).

References

1. C. S. Calude and D. Desfontaines. Anytime algorithms for non-ending computations. In preparation, The University of Auckland, 2014.
2. C. S. Calude and M. J. Dinneen. Exact approximations of Omega numbers. *Intl. J. of Bifurcation and Chaos*, 17(6):1937–1954, July 2007.
3. G. J. Chaitin. *Algorithmic Information Theory*. Cambridge University Press, Cambridge, UK, 1987.
4. E. Csuhaj-Varjú, M. Margenstern, G. Vaszil, and S. Verlan. On small universal antiport P systems. *Theor. Comput. Sci.*, 372(2-3):152–164, 2007.
5. M. J. Dinneen and Y.-B. Kim. A new universality result on P systems. Report CDMTCS-423, Centre for Discrete Mathematics and Theoretical Computer Science, University of Auckland, Auckland, New Zealand, July 2012. `http://www.cs.auckland.ac.nz/CDMTCS/researchreports/423mjdkim.pdf`.
6. M. J. Dinneen, Y.-B. Kim, and R. Nicolescu. A faster P solution for the Byzantine agreement problem. In M. Gheorghe, T. Hinze, and G. Păun, editors, *Conference on Membrane Computing*, pages 167–192. Verlag ProBusiness, Berlin, 2010.
7. R. Freund, L. Kari, M. Oswald, and P. Sosík. Computationally universal P systems without priorities: two catalysts are sufficient. *Theor. Comput. Sci.*, 330(2):251–266, 2005.
8. M. Ionescu, G. Păun, and T. Yokomori. Spiking neural P systems. *Fundam. Inform.*, 71(2-3):279–308, 2006.
9. C. Martín-Vide, G. Păun, J. Pazos, and A. Rodríguez-Patón. Tissue P systems. *Theor. Comput. Sci.*, 296(2):295–326, 2003.
10. G. Păun. *Membrane Computing: An Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.
11. G. Păun. Introduction to membrane computing. In G. Ciobanu, M. J. Pérez-Jiménez, and G. Păun, editors, *Applications of Membrane Computing*, Natural Computing Series, pages 1–42. Springer-Verlag, 2006.
12. G. Păun, G. Rozenberg, and A. Salomaa (Editors). *The Oxford Handbook of Membrane Computing*. Oxford University Press, Inc., New York, NY, USA, 2010.

Accelerated Simulation of Membrane Computing Approach on the Graphics Processing Unit to Solve the N-queens Problem

Ali Maroosi[1,2] and Ravie Chandren Muniyandi[1]

[1]Research Center for Software Technology and Management, Faculty of Information Science and Technology, National University of Malaysia, 43600 Bangi, Selangor, Malaysia, Malaysia
[2]Department of Computer Engineering and IT, University of Torbat-e-Heydarieh, Torbat-e-Heydarieh, Khorasan, Iran
ali.maroosi@gmail.com,ravie@ukm.edu.my

Abstract. The N-queens problem has attracted growing attention because of its potential application in different areas, including parallel memory storage, image processing, and chemical studies. Previous approaches using active membrane systems to solve the N-queens problem defined many membranes with just one rule inside of each membrane and many communication rules between membranes. Execution of communication rules between cores and threads is time consuming and decreases processing speed. The proposed approach reduces unnecessary membrane and communication rules by defining two membranes with many objects and rules inside of each membrane. With this structure, objects and rules can evolve in parallel, making the model suitable for implementation on a graphics processing unit (GPU). This study uses the GPU to exploit the parallelism of membrane systems for the N-queens problem. Tiling techniques and shared memory are used to accelerate the GPU for the proposed membrane computing model to solve the N-queens problem. The improved simulation on the GPU is 33-fold faster relative to the sequential approach.

Keywords: membrane computing, graphics processing unit, N-queens problem, parallel processing.

1 Introduction

Membrane computing, whose models are called membrane systems or P systems, is a branch of molecular computing inspired from cell biology [1]. It is a general computing architecture wherein various types of objects can be processed by various operations. A membrane system includes a membrane structure where each membrane surrounds a region that includes objects, rules, or possibly other membranes. Rules govern the processing of objects and membranes [2]. There are variants of membrane systems, such as cell-like, tissue-like, and spike-like

[3–5]. Several simulators have been proposed for implementing membrane computing [6–8]. Software applications for membrane computing normally implement sequential algorithm simulations adapted to common central processing unit (CPU) architectures [7, 8]. These kinds of algorithms do not perform well when the problem size increases. To take advantage of the parallelism available in membrane computing, efforts have been undertaken to implement membrane computing on parallel tools. For example, membrane computing has been implemented on computer clusters [9], reconfigurable hardware, e.g., field programming gateways, [10], and graphics processing units (GPUs) [11–13]. Membrane computing is used to simulate biological processes [14–16]. For example, it has been applied to simulate molecular interactions [17, 18] and predict the evolution of the bearded vulture [19]. Although it is biologically inspired, membrane computing has also been applied to problems outside biology, including modelling, economics, databases, networks, and other complex problems. Membrane systems display high levels of parallelism [20–22] and are used for solving optimization and combination problems [23–25]. Further information about active membrane systems has been provided by Paun [26]. The N-queens problem is encountered in various fields of study, including parallel memory storage approaches, image processing, physical and chemical studies, and networks [27]. The N-queens problem is classified as a nondeterministic polynomial problem, which is intractable for large N values. The goal when solving the N-queens problem is to place N queens on an $N \times N$ board so that no queen threatens other queens using standard chess queen moves and no more than one queen sits in the same column, row, ascending diagonal, or descending diagonal. The N-queens problem has been modelled into the membrane system framework using active membranes. The first study of the N-queens problem using membrane computing was published by Gutierrez-Naranjo et al., who applied it to a 4-queens problem that included 65,536 elementary membranes [28]. Depth-first search was later introduced into membrane computing by Gutierrez-Naranjo et al. [29], who used it to solve the N-queens problem. Gutierrez-Naranjo et al. [30] improved the speed of solving the N-queens problem using membrane computing as a local search strategy. Previous membrane systems using active membrane models to address the N-queens problem involved several membranes, but with few objects within each membrane. These membranes needed to communicate with each other, which reduced execution speed. Here, a proposed membrane system with active membranes from [31] is used for solving the N-queens problem. The proposed active membrane system improves upon previous approaches by decreasing the number of unnecessary communication rules and membranes. The number of rules that can be evolved simultaneously during each step is also increased in the proposed model, making this active membrane model suitable for parallel implementation. Previous studies simulated membrane systems for solving the N-queens problem using a sequential approach; however, this research uses a GPU to exploit membrane system parallelism. Techniques, such as tiling and shared memory, have been used to improve GPU performance.

2 Complete and partial solutions of the N-queens problem

2.1 Partial solutions of the N-queens problem by active membrane systems and their applications

There are many solutions for the N-queens problem that qualify as complete. In some applications, partial solutions are desirable for large N, e.g., $N = 256$ in [32]. In addition to *trivial solutions* [33] for the N-queens problem, other subsets (partial solutions) of the N-queens problem may also be desirable. Partial solutions of the N-queens problem obtained by placing one of the queens in a special position (x, y) are used in [34–36]. Such solutions can be used to construct maximum partial spreads of many sizes in the three-dimensional projective space over the finite field, $F_q(PG(3; q))$. Because this solution should include the special position, a *non-trivial solution* of N-queens may be needed. Using [32, 37], partial solutions of N-queens are needed to construct the sparse parity-check matrices and to generate low-density parity-check codes. As another example, consider a narrow-band directional communication system. To achieve high communication bandwidth, an array of N transmitters/receivers must be placed to freely communicate with the outside world in eight directions, i.e., two horizontal directions, two vertical directions, and four diagonal directions, without being obscured by other transmitters/receivers. Assuming that the positions of one or more of the transmitters/receivers is known and predetermined, finding the location of other transmitters/receivers constitutes a solution to the N-queens problem. Because the positions of some queens have been predetermined, a trivial solution of N-queens may not constitute a solution for this problem and, therefore, a nontrivial solution of the N-queens problem is needed. In Fig. 1, the position of one of the transmitters/receivers is known and the position of another six transmitters/receivers should be determined so that they are not obscured by other transmitters/receivers. This involves solving the 7-queens problem. In the proposed active membrane from [31], in the initial state, membrane 2 ($[]_2$) will be initialized by string $R_6.C_2$ (see Fig. 1; this means that one queen is allocated to row 6 and column 2), which will remain unchanged until the end of processing. Therefore, the proposed active membrane model can efficiently find one or more solutions for the N-queens problem.

2.2 Complete solutions by the proposed active membrane system on a GPU

GPU architecture Single-instruction multiple-data architectures enable GPUs to process and run several threads simultaneously [38]. The smallest parts of a GPU are cores, with a group of cores referred to as a streaming multiprocessor (SMP). Cores inside of each SMP are synchronized to execute the same instructions and each SMP works asynchronously with other SMPs. Each core has a small amount of memory, referred to as local memory, and each thread has access to a certain number of 32-bit registers. A small amount of shared

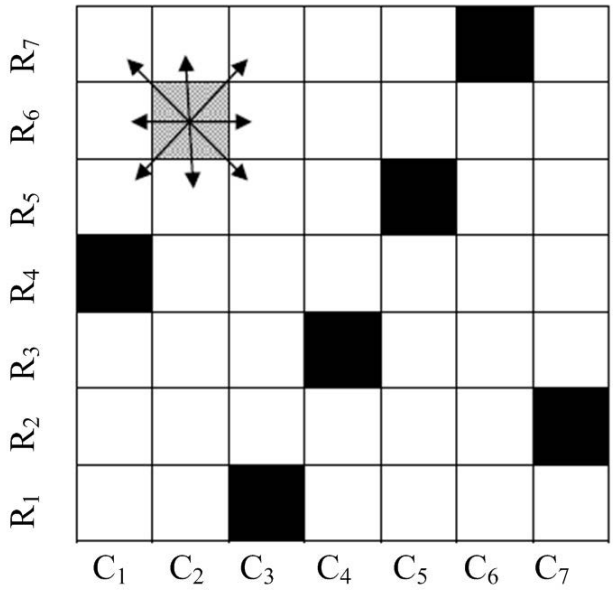


Fig. 1. Solution using the proposed active membrane model for seven transmitters/receivers, given that the position of one of the transmitters/receivers is predetermined (shaded square).

memory is dedicated to each SMP and all SMPs can access a large amount of global memory (Fig. 2). Access to register memory is faster relative to shared memory and access to shared memory is faster relative to global memory [38, 39]. Instead of cores, SMPs, and groups of SMPs, a programmer uses threads, blocks, and the kernel. A program contains one or several kernels, with each containing one or more blocks. Each block is run on a single SMP and all threads within a block can use the same shared memory, as well as barrier synchronization. Synchronization and shared-memory sharing are impossible across blocks. The programmer creates a program called a kernel that includes one or several blocks. Execution of blocks in the kernel maps to SMPs in the GPU.

Complete solutions of N-queens using an active membrane model on a GPU Active membrane models are naturally nondeterministic and initialized randomly. Therefore, it is possible to find more or all solutions by running the proposed model in [31] independently a number of times or by running it concurrently on different cores or computer clusters. Consequently, the proposed active membrane model in [31] can run on several cores or computer clusters without the need for synchronization or communication between cores or computer clusters. Subset solutions generated by copies of one active membrane model running independently on different cores will be collected to produce a complete list of solutions after removing repeated solutions.

To generate partial or complete solutions on a GPU, first assign m copies of the active membrane model to m thread blocks. Each model on each thread block then runs L times. $L \times m$ possible solutions from the m thread blocks

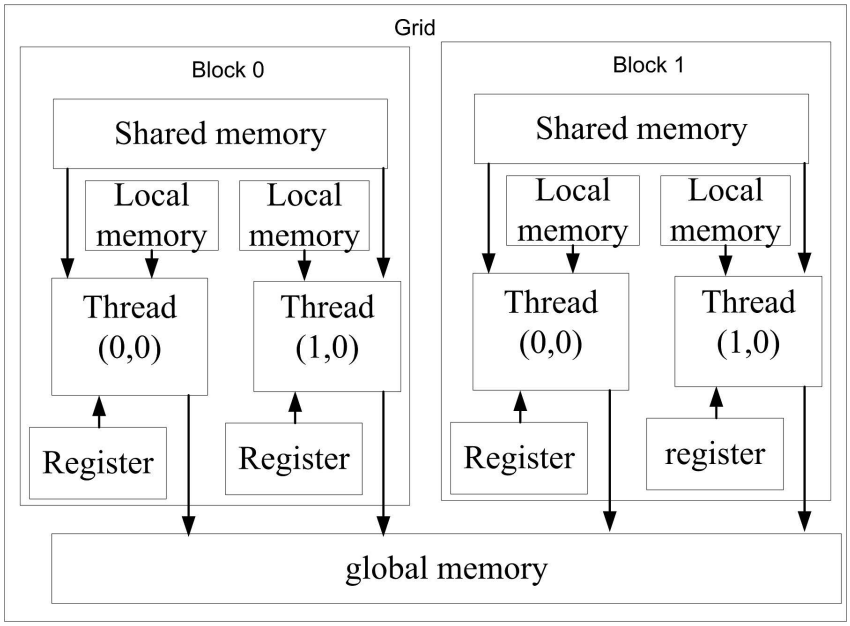


Fig. 2. Memory and CUDA architecture for GPU.

are gathered and sent to another kernel where repeat solutions are removed (see Fig. 3). The steps of this procedure can be described as follows:

Step1: Assign an active membrane system model to each thread block.

Step2: (Random_Ini_Kernel) Initialize a random number generator with different seeds to generate a different random number for each active membrane system. Because this step occurs only once, using a separate kernel enables the release of used registers and shared memory for the rest of the simulation. Note that the proposed active membrane system in [31] finds solutions using a non-deterministic approach, and, therefore, needs an independent random number generator for each thread to find different solutions when run independently on different thread blocks.

Step3: (Run_ActiveMem_Kernel) Run each active membrane model L times on each thread block to find at most L solutions. Note that some runs do not lead to a solution and, according to rule (f) of the proposed active membrane model in [31], the model should then be restarted. In this kernel, shared memory is also used to improve GPU performance. In each step, $thread\ i-1; i=1,\ldots,N$ is responsible for computations related to the i^{th} row on the board (object R_i) in each thread block (see Fig. 3). For example, in rule (a) from [31], $thread\ i-1$ should choose one object, C_j, randomly to react with object R_i in order to generate multiset $R_i.C_j$. In rule (c) in [31], each $thread\ i-1$ and $thread\ k-1$ is responsible for checking and changing the number of objects related to the i^{th} and k^{th} rows, e.g., $R_i.C_j$, u_{i-j}, d_{i+j}, $R_k.C_s$, $,u_{k-s}$, and d_{k+s}. Variables and arrays for objects are stored in shared memory for fast access. For fast random number generation, initial seeds and states produced previously (Random_Ini_Kernel) are loaded from global memory to shared memory. After the random number is generated, updated states are stored in global memory in order to release shared memory. When a solution ($R_i.C_j$, having no conflicts on the board) has been

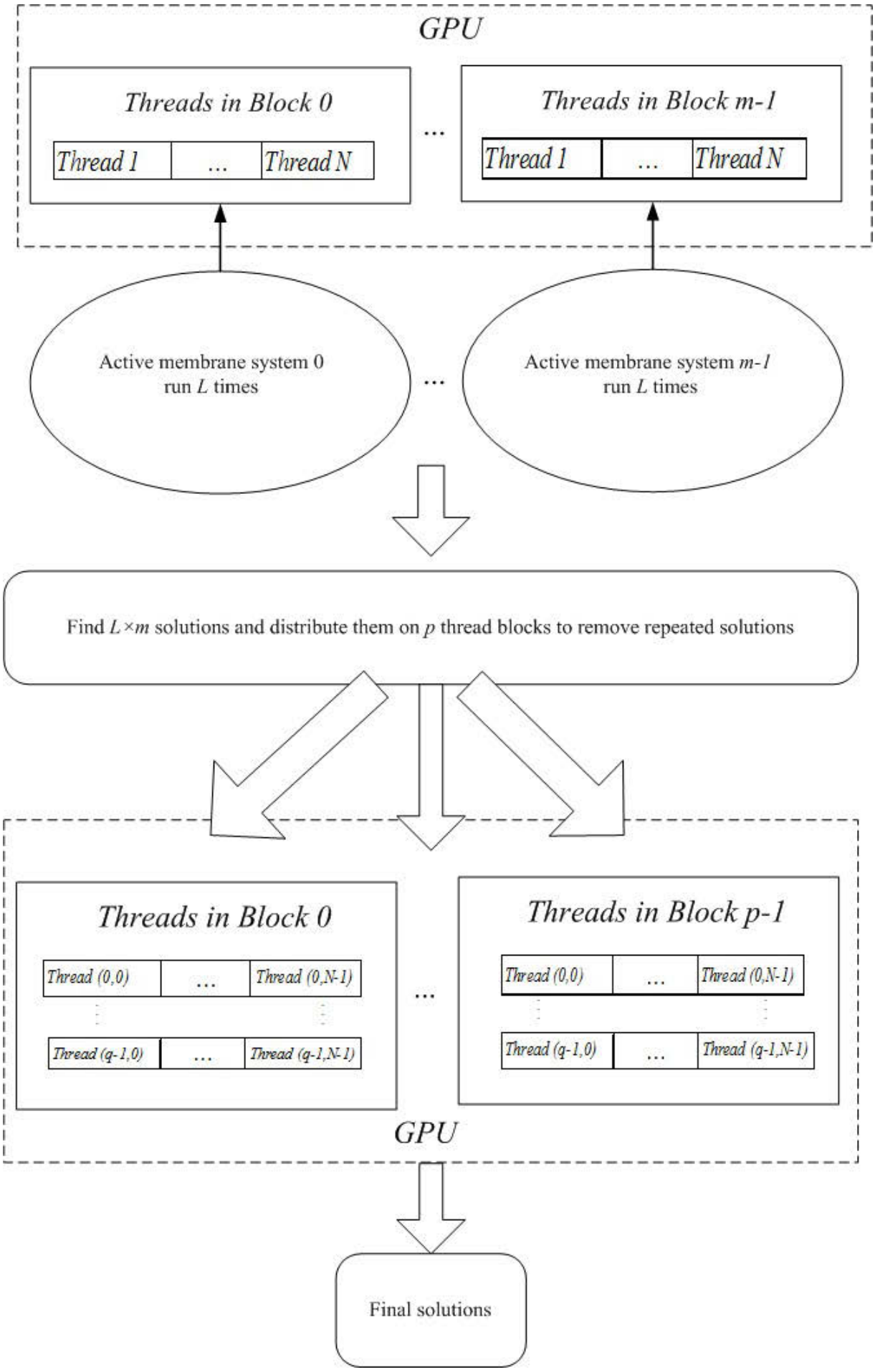


Fig. 3. Procedure for generating solutions and removing repeated solutions in order to find all or partial solutions for the N-queens problem using an active membrane model.

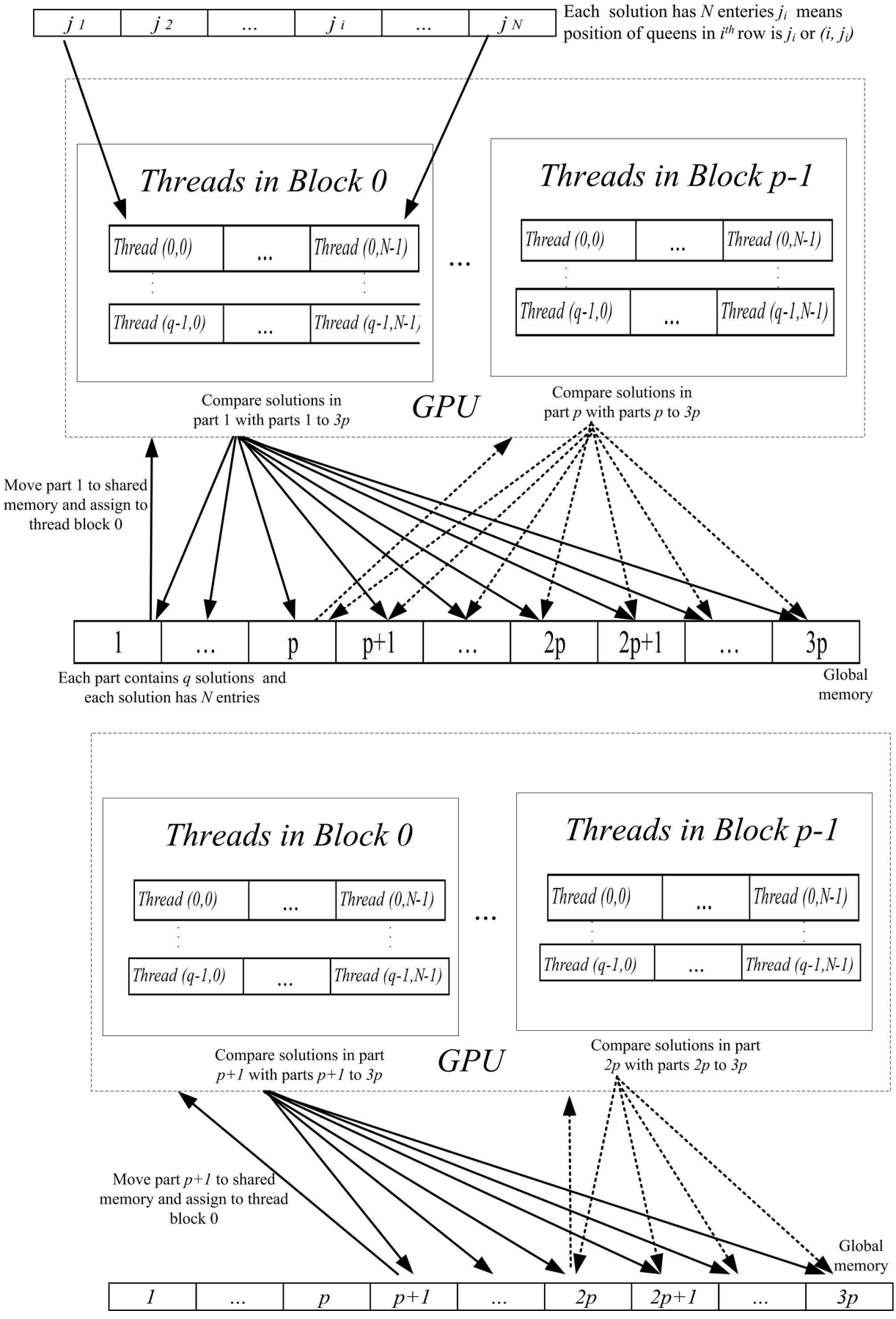


Fig. 4. Using tiling (subdivision) and shared memory to remove repeated solutions from a GPU.

found and checked against rule (d) in [31], it is stored in an array in global memory. This array is allocated in global memory to store at most L solutions from each thread block and to have at most $L \times m$ solutions from all thread blocks. Solutions collected (at most $L \times m$) from all thread blocks will go to the next kernel for removal of repeat solutions.

Step4: (Remove_Repeated_Kernel) At most, $L \times m$ solutions were generated in the previous step. These solutions were generated using the nondeterministic rules of the active membrane system and, therefore, some solutions may be repeated. In the proposed approach, repeat solutions are discarded in the GPU.

Global memory is slow and, therefore, using shared memory instead of global memory improves performance, given that the latency associated with accessing global memory is 400-800 cycles, while that for shared memory is 8-22 cycles [38]. A limited amount of shared memory can be accessed by thread blocks within each SMP. Therefore, tiling and shared memory can be used to improve performance. Solutions stored in global memory are divided into different parts, with each part having q solutions. First, p parts are assigned to p thread blocks, with p and q determined according to available shared memory and other GPU resources. Each solution is then compared with other solutions stored in adjacent memory locations. Each solution is considered unique when it is not repeated in memory locations subsequent to its own (see Fig. 4).

3 Simulations and results

Simulations of active membrane systems on a GPU in order to find complete or partial solutions to the N-queens problem were executed using an NVIDIA GeForce GTX680 graphics card with the specifications listed in Table 1. Shared memory and data tiling (subdivision) were used to improve simulation of the active membrane system on a GPU. Several active membrane systems with different initializations using random number generators were assigned to different thread blocks in order to generate subsections of solutions. These subsections were collected from different thread blocks and repeat solutions removed in order to form a list of complete solutions. Algorithm details are provided in Section 4.2.

Simulations for various sizes of N-queens problems on a CPU and a GPU were performed. As the size of the N-queens problem increases, occupancy of the GPU increases, leading to increased GPU performance. The increased processing speed associated with using the tiling and shared-memory approaches on a GPU relative to the sequential approach on a CPU was 15-fold for $N = 5$ and 33-fold for $N = 9$ (Table 2). This study used tiling and shared memory instead of global memory to improve GPU performance. Additionally, the speed increase from using tiling and shared memory on the GPU was better relative to a normal GPU implementation. Relative to the sequential approach using a CPU, the speed increased 10.6-fold using a normal GPU implementation as compared to 33-fold using the tiling and shared memory-based GPU implementation (Table 2).

Table 1. Technical specifications of an NVIDIA GeForce GTX680 graphics card with computing capability 3.

Number of SMPs	8
Maximum number of resident warps per SMP	64
Maximum number of resident thread blocks per SMP	16
Maximum number of resident threads per SMP	2048
Maximum number of resident threads per warp	32
Maximum number of resident threads per thread Block	1024
Maximum shared memory per SMP	48k
Maximum resident 32-bit registers per SMP	64k

Table 2. Finding complete solutions with active membrane systems on the GPU.

size of board (N)	No. of all sol.	success rate percentage in finding all sol.	No. thread blocks in GPU	No. of iter in each thread block	Execution time on the CPU (sec)	Execution time usual way on the GPU (sec)	Execution time by shared and tilling on a GPU (sec)	speed up usual way on the GPU vs. CPU	speed up tilling an shared on the GPU vs. CPU
5	10	99	32	4	0.152	0.030	0.010	5.06	15
6	4	99	32	4	0.283	0.046	0.015	6.15	19
7	40	99	32	8	2.13	0.264	0.085	8.06	25
8	92	99	32	32	3.40	0.376	0.121	9.04	28
9	352	99	64	64	7.62	0.714	0.231	10.6	33
10	724	99	64	128	30.2	2.83	0.913	10.6	33

When the size of the problem (N) increases, use of available computational resources on the GPU also increases as a result of increased processing speed. However, GPU computational resources are limited and for larger problem sizes, increases in processing speed remain constant (results indicate similar speeds associated with N=9 and N=10) (Table. 2).

4 Conclusions and future work

In this paper, a GPU was used to increase the speed of membrane system applications toward finding solutions to the N-queens problem. Tiling and shared memory were used to improve GPU performance. The increased speed associated with implementing a GPU using global memory for N=10 was 10.6-fold, while using tiling and shared memory resulted in a 33-fold increase. This study used many GPU cores to extract parallelism in the membrane system model. Shared memory can be accessed substantially faster relative to global memory and was, therefore, used to enhance improvement. Given that the amount of GPU shared memory is small, a tiling strategy was considered in order to improve shared-memory utilization. Our future work will use the isomorphic characteristics of the N-queens problem to improve the speed of the N-queens membrane model, given that some solutions can be obtained from rotation or reflection of other solutions on a GPU. Additionally, we want to increase implementation speed by extracting instruction-level parallelism within the GPU and applying it to our proposed model.

Acknowledgments. This work has been supported by the Science Fund of the Ministry of Science, Technology, and Innovation (MOSTI) - (Malaysia; Grant code: 01-01-02-SF1104).

References

1. Paun, G.: A quick introduction to membrane computing. Journal of Logic and Algebraic Programming 79, 291-294 (2010)
2. Paun, G.: Tracing some open problems in membrane computing. Romanian Journal of Information Science and Technology 10, 303-314 (2007)
3. Ishdorj, T., Leporati, A., Pan, L., Zeng, X., Zhang, X.: Deterministic solutions to QSAT and Q3SAT by spiking neural P systems with pre-computed resources. Theoretical Computer Science 411, 2345-2358 (2010)
4. Linqiang, P., Alhazov, A.: Solving HPP and SAT by P systems with active membranes and separation rules. Acta Informatica 43, 131-145 (2006)
5. Linqiang, P., Zeng, X., Zhang, X., Jiang, Y.: Spiking neural P systems with weighted synapses. Neural Processing Letters 35, 13-27 (2012)
6. Paun, G., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing, pp. 437-454, Oxford University Press (2010)
7. Garcia-Quismondo, M., Gutierrez-Escudero, R., Perez-Hurtado, I., Perez-Jimenez, M.J., Riscos-Nunez, A.: An overview of P-lingua 2.0. Membrane Computing, vol. 5957, pp. 264-288, Springer, Berlin, Heidelberg (2010)

8. Gutierrez-Naranjo, M.A., Perez-Jimenez, M.J., Riscos-Nunez, A.: Available membrane computing software. Applications of Membrane Computing, Natural Computing Series, pp. 411-436, Springer, Berlin, Heidelberg (2006)
9. Ciobanu, G., Wenyuan, G.: P Systems Running on a Cluster of Computers, Lecture Notes in Computer Science 2933, 123-139 (2004)
10. Nguyen, V., Kearney, D., Gioiosa, G.: A Region-Oriented Hardware Implementation for Membrane Computing Applications and Its Integration into Reconfig-P, Lecture Notes in Computer Science 5957, 385-409 (2010)
11. Cecilia, J.M., Garca, J.M., Guerrero, G.D., Martinezdel- Amor, M.A., Perez-Hurtado, I., Perez-Jimenez, M.J.: Simulating a P system based efficient solution to SAT by using GPUs, Journal of Logic and Algebraic Programming 79, 317-325 (2010)
12. Cecilia, J.M., Garcia, J.M., Guerrero, G.D., Martinezdel-Amor, M.A., Perez-Hurtado, I., Perez-Jimenez, M.J.: Simulation of P systems with active membranes on CUDA, Briefings in Bioinformatics 11, 313-322 (2010)
13. Cecilia, J.M., Garcia, J.M., Guerrero, G.D., Martinezdel-Amor, M.A., Perez-Jimenez, M.J., Ujaldon, M.: The GPU on the simulation of cellular computing models, journal of Soft Comput 16, 231246 (2012)
14. Muniyandi, R., Abdullah, M.Z.: Modeling hormone-induced calcium oscillations in liver cell with membrane computing. Romanian Journal of Information Science and Technology 15, 63-76 (2012)
15. Muniyandi, R., Abdullah, M.Z.: Membrane computing as a modeling tool for discrete systems. Journal of Computer Science 7, 1667-1673 (2011)
16. Muniyandi, R., Abdullah, M.Z.: Experimenting with the simulation strategy of membrane computing with Gillespie algorithm by using two biological case studies. Journal of Computer Science 6, 525-535 (2010)
17. Romero-Campero, F.J., Perez-Jimenez, M.J.: A model of the quorum sensing system in Vibrio fischeri using P systems. Artificial Life 14, 95-109 (2008)
18. Krasnogor, N., Gheorghe, M., Terrazas, G., Diggle, S., Williams, P., Camara, M.: An appealing computational mechanism drawn from bacterial quorum sensing. Bulletin of the European Association of Theoretical Computer Science 85, 135-148 (2005)
19. Cardona, M., Colomer, M.A., Perez-Jimenez, M.J., Sanuy, D., Margalida, A.: Modelling ecosystems using P systems: the bearded vulture, a case study. LNCS, vol. 5391, pp. 95-116, Springer, Berlin, Heidelberg (2009)
20. Maroosi, A., Muniyandi, R. C.: Enhancement of membrane computing model implementation on GPU by introducing matrix representation for balancing occupancy and reducing inter-block communications. Journal of Computational Science 5(6), 861-871 (2014)
21. Maroosi, A., Muniyandi, R. C., Sundararajan, E., Zin, A. M.: Parallel and distributed computing models on a graphics processing unit to accelerate simulation of membrane systems. Simulation Modelling Practice and Theory 47, 60-78 (2014)
22. Maroosi, A., Muniyandi, R. C., Sundararajan, E. A., Zin, A. M.: Improved Implementation of Simulation for Membrane Computing on the Graphic Processing Unit. Procedia Technology 11, 184-190 (2013)
23. Maroosi, A., Muniyandi, R. C., Membrane Computing Inspired Genetic Algorithm On Multi-Core Processors. Journal of Computer Science 9, 264-270 (2013)
24. Ali M., Muniyandi, R.C. A hybrid membrane computing and honey bee mating algorithm as an intelligent algorithm for channel assignment problem. Proceedings, Eighth International Conference on Bio-Inspired Computing: Theories and Applications, Springer, Berlin, Heidelberg, 1021-1028 (2013)

25. Maroosi, A., Muniyandi, R. C.: Accelerated simulation of membrane computing to solve the n-queens problem on multi-core. Swarm, Evolutionary, and Memetic Computing, Lecture Notes in Computer Science 8298, 257-267 (2013)
26. Paun, G.: Membrane Computing: An Introduction. Springer, Berlin (2002)
27. Bell, J., Stevens, B.: A survey of known results and research areas for N-queens. Journal of Discrete Mathematics 309, 1-31 (2009)
28. Gutierrez-Naranjo, M.A., Martinez-del-Amor, M.A., Perez-Hurtado, I., Perez-Jimenez, M.J.: Solving the N-queens puzzle with P systems. Seventh Brainstorming Week on Membrane Computing, Fenix Editora, vol.1, pp. 199-210 Sevilla, Spain (2009)
29. Gutierrez-Naranjo, M.A., Perez-Jimenez, M.J.: Depth-first search with P systems. LNCS, vol. 6501, pp. 257-264, Springer, Berlin, Heidelberg (2010)
30. Gutierrez-Naranjo, M.A., Perez-Jimenez, M.J.: Local search with P systems: a case study. International Journal of Natural Computing Research 2, 47-55 (2011)
31. Maroosi, A., Muniyandi, R. C.: Accelerated execution of P systems with active membranes to solve the N-queens problem. Theoretical Computer Science 551, 39-54 (2014).
32. Li, P., Guangxi, Z., Xiao, L.: Low-density parity-check codes based on the N-queens problem. Proceedings, 2004 ACM Workshop on Next-Generation Residential Broadband Challenges, pp. 37-41, ACM (2004)
33. Bernhardsson, B.: Explicit solutions to the N-queens problem for all N. ACM SIGART Bulletin, 2(2), 7 (1991)
34. Heden, O.: Maximal partial spreads and the modular N-queens problem. Discrete Math. 120, 75-91 (1993)
35. Heden, O.: Maximal partial spreads and the modular N-queens problem. II. Discrete Math. 142, 97-106 (1995)
36. Heden, O.: Maximal partial spreads and the modular N-queens problem. III. Discrete Math. 243, 135-150 (2002)
37. Xiling, L., Zhang, J., Zhang, J.: LDPC block-coding schedule based on queen matrix. Proceedings, 4th International IEEE Conference on Wireless Communications, Networking, and Mobile Computing (WiCOM08), pp. 1-4, IEEE (2008)
38. NVIDIA Corporation. NVIDIA CUDA C Programming Guide, Version 4.2, 2012, `http://docs.nvidia.com/cuda/index.html`
39. Takizawa, H., Koyama, K., Sato, K., Komatsu, K., Kobayashi, H.: CheCL: transparent checkpointing and process migration of OpenCL applications. International Parallel and Distributed Processing Symposium, pp. 864-876, Anchorage, Alaska, USA (2011)

Decision Tree Models Induced by Membrane Systems

Hong Peng[1], Jun Wang[2] ⋆, Mario J. Pérez-Jiménez[3]
, Agustín Riscos-Núñez[3], Jian Xiao[4], and Guixiang Zhang[5]

[1] School of Computer and Software Engineering,
Xihua University, Chengdu, 610039, China
[2] School of Electrical and Information Engineering,
Xihua University, Chengdu, 610039, China
[3] Research Group of Natural Computing,
Department of Computer Science and Artificial Intelligence,
University of Seville, Sevilla, 41012, Spain
[4] School of Electrical Engineering,
Southwest Jiaotong University, Chengdu, 610031, China
[5] College of Mechanical and Vehicle Engineering,
Hunan University, Changsha, 410082, China

Abstract. This paper focuses on application of membrane systems to solve classification problems. Decision tree technique has been widely used to construct classification models because such models can closely resemble human reasoning and are easy to understand. In this paper, an extended tissue membrane system with tree-like objects is developed as the computing framework of the presented decision tree induction algorithm. Each object in cells expresses a feasible decision tree and the transformation-communication mechanism is used to tackle the tree-like objects. The proposed decision tree induction algorithm is evaluated on some data sets and compared with two classical methods.

Keywords: Membrane computing; Tissue membrane systems; Data classification; Decision tree

1 Introduction

Membrane computing, as a class of distributed parallel computing models, is inspired from the structure and functioning of living cells as well as the cooperation of cell populations in tissues and organs [1, 2], also known as membrane systems and P systems. Over the past years, a variety of membrane systems and variants have been proposed [3], and most of them have been proven to be universal and effective [4–7]. Usually, a membrane system can be characterized by several components: membrane structure, objects, operations with objects, ways to control the operations. In recent years, membrane systems have been used to solve a lot of real-world problems, for example, optimization problems [8], fuzzy

⋆ Corresponding author (J. Wang).

reasoning [9, 10], fault diagnosis [11], image processing [12, 13], robot control [14] and ecology [15]. Particularly, the object's transformation-communication mechanism has been developed to process different real-world problems.

Machine learning algorithms have two main categories: unsupervised learning (clustering) and supervised learning (classification). In recent years, application of membrane computing in data clustering has received a lot of attention. Clustering is such a process that partitions a data set into several clusters such that patterns within the same cluster are more similar than those from different clusters [16]. K-means algorithm is one of most popular clustering algorithms. However, there are some shortcomings: it easily falls into local minima and severely depends on the initial solutions [17]. To overcome the shortcomings, the object's transformation-communication mechanism in membrane systems has been developed to determine the global optimal cluster centers for data clustering problem. Huang et al. [18] proposed a clustering algorithm based on membrane computing to solve the clustering problem, called PSO-MC, which introduced the velocity-position model in particle swarm optimization (PSO) as the object's transformation mechanism. In Jiang [19], genetic operations and simulated annealing were combined into the object's transformation mechanism of the presented clustering algorithm. Similarly, a transformation mechanism based on genetic operations was developed according to the used membrane structure for data clustering [20]. Combined with differential evolution (DE) and the object's communication mechanism, a clustering algorithm has been present, called DE-MC [21]. Peng et al. [22] used an evolution-communication membrane system to solve fuzzy clustering problem. In addition, a clustering algorithm with hybrid evolutionary mechanisms has been reported in Peng [23].

This paper focuses on another machine learning problem, that is, classification problem. Decision tree technique has been widely used to build the classification models. In comparison to "block-box" model such as artificial neural network, decision tree has high comprehensibility. In each node, a test that uses a or more variables is finished. Thus, the tree can be traversed from left subtree to right subtree according to the test results. In the past, a lot of decision tree algorithms have been proposed, for example, ID3, CRAT and C4.5 [24]. The algorithms are greedy local search algorithms, which construct decision trees in a top-down way. The motivation behind this work is to apply membrane systems to generate a decision tree for a data set. For this, classical membrane system is extended with tree-like objects, and transformation rules with the complex objects are developed to find the global optimal decision tree.

The rest of this paper is arranged as follows. An extended tissue membrane system that can tackle tree-like objects is discussed in detail in Section 2. Section 3 describes the proposed decision tree induction algorithm. In Section 4, experimental results carried out on some real-life data sets are presented. Finally, conclusions are drawn in Section 5.

2 Tissue Membrane Systems with Tree-like Objects

The goal of this paper is to apply membrane systems to generate a decision tree from a data set, It is well-known that classical decision tree algorithms, such as ID3, CRAT and C4.5, use the top-down approach to build the decision tree by test variables on each node. Different from these methods, our idea is that a tissue membrane system is used to search the optimal decision tree form feasible solution space. Thus, this requires that the tissue membrane system can express and process the data with tree-like structure. However, the existing tissue membrane systems are based on multisets of strings, so they are not able to express and process the tree-like objects. Therefore, the classical tissue membrane systems will be extended to propose a tissue membrane system with tree-like objects.

The tissue membrane system with tree-like objects is defined as a construct

$$\Pi = (w_1, \ldots, w_q, R_1, \ldots, R_q, R', i_0) \tag{1}$$

where

(1) w_i is finite set of tree-like objects in cell i, $1 \leq i \leq q$;
(2) R_i is finite set of transformation rules of tree-like objects in cell i, $1 \leq i \leq q$;
(3) R' is finite set of communication rules of the q cells;
(4) i_0 indicates the output region of the system.

The extended tissue membrane system consists of q cells labeled by $1, 2, \ldots, q$ respectively. Figure 1 shows the membrane structure of the tissue membrane system, in which the region labeled by 0 is the environment. Each cell contains a or more objects, and each object expresses a tree. The tree-like objects in cells will be changed by transformation rules during computation. Moreover, communication rules provides a mechanism to achieve the sharing of objects between the q cells. As usual in membrane systems, the q cells as computing units work in parallel. When the system halts, the final result is stored in the output region.

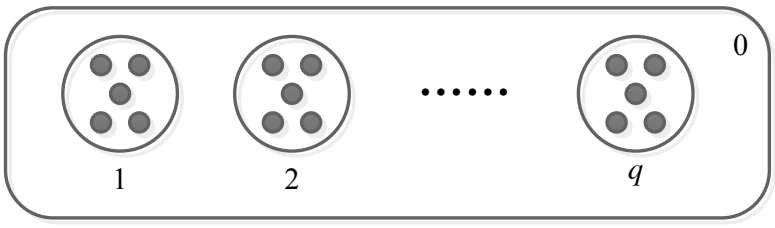


Fig. 1. The membrane structure of the used tissue membrane system.

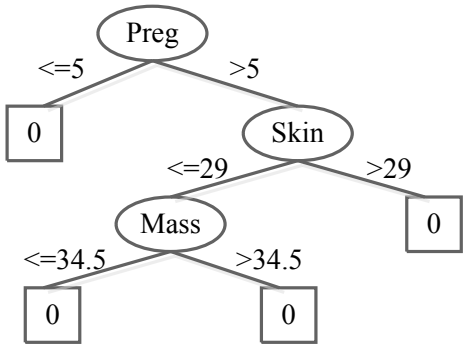


Fig. 2. An example of tree-like objects.

2.1 Tree-like objects

The tissue membrane system is designed to generate a decision tree, so each object in the system is used to express a candidate tree. Figure 2 illustrates an example, which represents a tree in Pima data set.

Initially, the membrane system will randomly generate some initial objects, that is, some initial trees. When an object (tree) is generated, a subset is selected randomly from a data set, and then a subtree is generated by C4.5 as the object. It is important that objects in the cells should have enough diversity.

2.2 Transformation rules

In the tissue membrane system, three classical genetic operations are introduced as transformation rules of objects, including selection, crossover and mutation operations. However, the three genetic operations are extended in this work in order to make them suitable to process the tree-like objects.

The selection operation reflects the principle of the survival of the fittest. In the membrane system, classical roulette method is used to select the objects (trees) that can be processed by crossover and mutation operations. To apply the roulette method, a criterion is required to evaluate each object in the cells, so it is regarded as the object's fitness function. The object's evaluation criterion will be discussed below. The crossover and mutation operations of objects are used to achieve the improvement of objects (trees) in cells. To process the tree-like objects, however, classical crossover and mutation operations need to be extended.

Figure 3 illustrates the crossover operation of two tree-like objects. The crossover operation is similar to classical crossover operation, but it is achieved based on subtree exchange rather than string. Parent 1 and parent 2 are two objects and two cross points are chosen in the two tree respectively, and then two subtrees that are associated with the two cross points are exchanged.

The extended mutation operation based on tree-like objects are shown in Figure 4. Different from classical mutation operation, the extended mutation operation is also achieved by subtree exchange: two subtrees are chosen randomly in the parent object (tree), and then the two subtrees are exchanged.

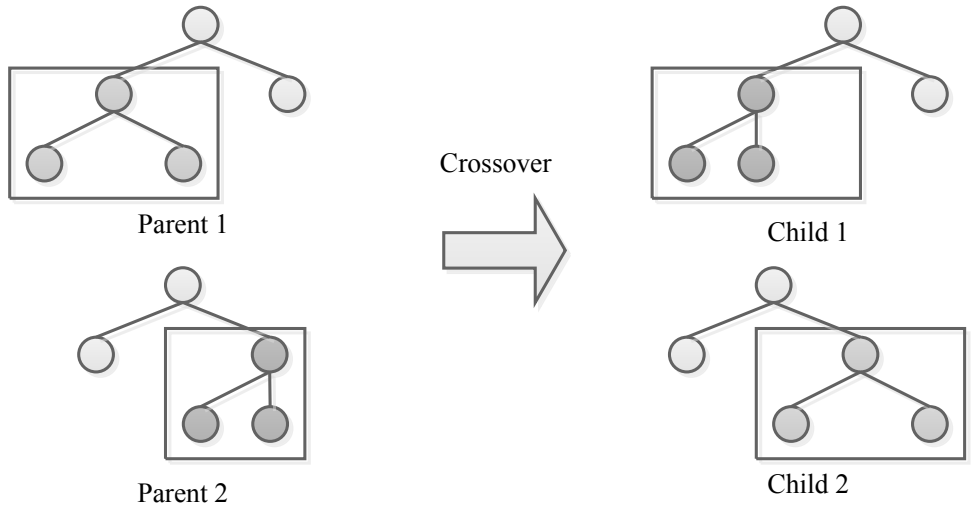


Fig. 3. An example of the crossover operation for two tree-like objects.

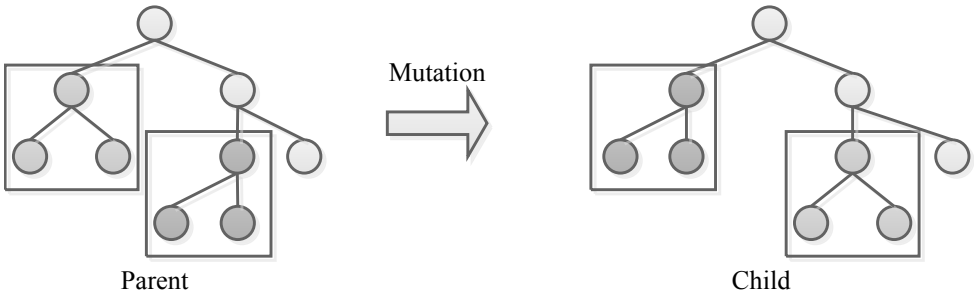


Fig. 4. An example of mutation operation for a tree-like object.

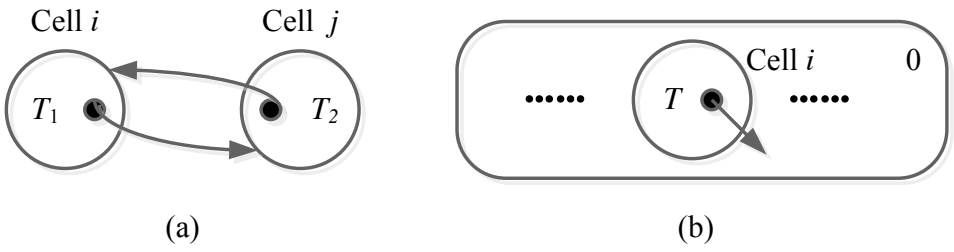


Fig. 5. The object's communiction mechanisms (a) between two cells and (b) between a cell and the environment.

2.3 Communication rules

The communication rules are used to achieve the sharing of objects. As usual, the tissue membrane system has the communication rules of two types:

- Rule $(i, T_1/T_2, j)$, where T_1 and T_2 are the objets in cell i and cell j respectively, $i, j = 1, 2, \ldots\ldots, q$.
  The rule indicates communication rule between cell i and cell j, shown in Figure 5(a). Object T_1 in cell i is transmitted into cell j, and at the same time object T_2 in cell j is transmitted into cell i.
- Rule $(i, T/\lambda, 0)$, where T is the objet in cell i and λ is the empty object, $i = 1, 2, \ldots\ldots, q$.
  The rule indicates communication rule between cell i and the environment, shown in Figure 5(b). Object T in cell i is transmitted into the environment.

3 Proposed Decision Tree Induction Algorithm

Decision tree induction algorithm is designed to generate a decision tree from a data set. In this paper, only single variable is considered on each node. Different from classical top-down approaches such as ID3, CRAT and C4.5, the proposed method will use a tissue membrane system to search a global optimal decision tree in solution space. Therefore, the tissue membrane system described above is used as its computing framework, in which each object in cells expresses a candidate decision tree. Starting from initial objects (trees), the system constantly uses the transformation-communication mechanism to improve the objects in the cells until it halts.

During computation, objects (trees) in cells are improved constantly. The object's improvement mechanism usually requires a criterion to evaluate each object in the system. In this work, classification accuracy and tree's complexity are combined together as the object's evaluation criterion. which can be defined by

$$J(T) = C(T) - v \cdot (S(T) - 1) \tag{2}$$

where T is an object (tree) in cells; $C(T)$ is classification accuracy of the object (tree), and $S(T)$ is the size of the tree; v is a factor to control the tree's complexity (default value is 0.001).

Based on the tissue membrane system, the proposed decision tree induction algorithm can be described as follows.

```
program Decision_tree_induced_by_membrane_systems
   input
     Data set, D;
     the number of cells, q;
     the number of objects in each cell, n;
     crossover and mutation probabilities, Pc and Pm;
     the factor, v;
   output
```

```
    the optimal decision tree, T;
  begin
    Initialize objects in cells;
    Iter := 1;
    repeat
      Transform objects in cells by transformation rules;
      Communicate objects by communication rules;
      Evaluate objects in cells by the criterion (2)
      Update T in the output region;
      Iter := Iter + 1;
    until Iter > MaxIter
    Export the optimal decision tree, T;
  end
end.
```

4 Experimental Results and Analysis

In order to evaluate the availability of the proposed decision tree induction algorithm, ten real-life data sets from UCI repository [26] have been selected in the experiment: Blance-Scale, Bupa, Cars, German, Glass, Heart, Pima, Sat, Vehicle and Vote. The input parameters of the proposed algorithm are chosen: the number of cells is $q = 5$, the number of objects in each cell is $n = 20$, crossover and mutation probabilities are $p_c = 0.8$ and $p_m = 0.01$, and control factor is $v = 0.001$. The computing step number in the tissue membrane system is set to 1000.

The proposed algorithm was compared with two existing decision tree induction algorithms: a classical decision tree algorithm C4.5 [25] and an evolutionary technique-based decision tree induction algorithm GDT-MA [27]. The comparison includes two metrics: classification accuracy and tree size. Classification accuracy is often used to indicate the quality of a classifier: usually, the higher the accuracy, the better the quality. On the other hand, it is hoped that the complexity of decision tree should be as small as possible when the classification performance cab be guaranteed. Considered some random factors in these algorithms, the average values obtained by them on 10 runs are computed in terms of classification accuracy and tree size.

Table 1 provides the comparison results of the three algorithms over ten data sets, which are average accuracies and sizes of the 10 runs. The comparison results are illustrated as follows:

- Blance-Scale. The proposed algorithm has the best classification accuracy and the smallest size, 79.9 and 19.5. C4.5 has the worst classification performance. GDT-MA is close to membrane systems in term of accuracy, but its size is greater than that of C4.5.
- Bupa. The proposed algorithm attains the highest classification accuracy and the smallest size, 64.8 and 31.7. So it is the best in the three algorithms

- Cars. The proposed algorithm and GDT-MA have the same accuracy and size, 97.9 and 3, while the accuracy and size of C4.5 are 97.7 and 31 respectively.
- German. GDT-MA has the best accuracy and the smallest size. The proposed algorithm is close to GDT-MA. C4.5 is worse than other two algorithms.
- Glass. The accuracy and size of the proposed algorithm are 66.5 and 34.9 respectively, so it is the best in the three algorithms.
- Heart. The accuracy and size of C4.5 are 77.1 and 22 respectively, so it attains the best classification performance. The accuracy of the proposed algorithm is slightly better than than of GDT-MA, but the size of the proposed algorithm is smaller than that of GDT-MA.
- Pima. The accuracy of the proposed algorithm is slightly better than than of GDT-MA and C4.5, but GDT-MA has the smallest size.
- Sat. The accuracy of the proposed algorithm is 86.2, so it is the best in the three algorithms. However, GDT-MA attains the smallest size, 18.9.
- Vehicle. C4.5 has the best classification accuracy because of its accuracy 72.7, but it has the worst size, 138.6. The accuracy of the proposed algorithm is close to that of C4.5. GDT-MA has the smallest size, 43.2.
- Vote. C4.5 attains the best classification accuracy and smallest size. The accuracy of the proposed algorithm is better than that of GDT-MA, and the size of the proposed algorithm is smaller than that of GDT-MA.

Table 1. Comparison results of the proposed algorithm with two decision tree induction algorithms.

Data sets	C4.5		GDT-MA		Membrane systems	
	Accuracy	Size	Accuracy	Size	Accuracy	Size
Blance-Scale	77.5	57	79.8	20.8	**79.9**	**19.5**
Bupa	64.7	44.6	63.7	33.6	**64.8**	**31.7**
Cars	97.7	31	**97.9**	**3**	**97.9**	**3**
German	73.7	77	**74.2**	**18.4**	74.1	18.6
Glass	62.5	39	66.2	35.3	**66.5**	**34.9**
Heart	**77.1**	**22**	76.5	29	76.9	24.8
Pima	74.6	40.6	74.2	**14.8**	**74.8**	17.3
Sat	85.5	435	83.8	**18.9**	**86.2**	22.5
Vehicle	**72.7**	138.6	71.1	**43.2**	72.5	45.9
Vote	**97**	**5**	96.2	10.9	96.8	7.4

5 Conclusions

This paper discussed an application of membrane systems in classification problem: tissue membrane system was considered to induce a decision tree for data set. A tissue membrane system with tree-like objects was developed, where three genetic operations were extended as transformation mechanism of the tree-like objects. Based on tissue membrane system with tree-like objects, a decision tree induction algorithm has been proposed to generate the optimal decision tree from data set. The proposed algorithm was tested on ten real-life data sets and compared with two existing algorithms. The comparison results demonstrate the availability of the proposed algorithm.

Acknowledgements

This work was partially supported by the National Natural Science Foundation of China (No. 61170030, No. 61372187), and Research Fund of Sichuan Science and Technology Project (No. 2015HH0057), China.

References

1. Păun, Gh.: Computing with membranes. Journal of Computer System Sciences 61(1), 108–143 (2000)
2. Păun, Gh., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrance Computing. Oxford University Press, New York (2010)
3. Păun, Gh., Pérez-Jiménez, M.J.: Membrane computing: brief introduction, recent results and applications. BioSystem 85, 11–22 (2006)
4. Pan, L., Zeng, X.: Small universal spiking neural P systems working in exhaustive mode. IEEE Transactions on Nanbioscience 10(2) 99–105 (2011)
5. Zhang, X., Liu, Y., Luo, B., Pan, L.: Computational power of tissue P systems for generating control languages. Information Sciences 278(10), 285-C297 (2014)
6. Zeng, X., Pan, L., Pérez-Jiménez, M.J.: Small universal simple spiking Nneural P systems with weights. Science China. Information Sciences 57(9), 1–11 (2014)
7. Song, T., Macías-Ramos L., Pan L., Pérez-Jiménez, M.J.: Time-free solution to SAT problem using P systems with active membranes. Theoretical Computer Science 529, 61–68 (2014)
8. Zhang, G., Cheng, J., Gheorghe, M., Meng, Q.: A hybrid approach based on differential evolution and tissue membrane systems for solving constrained manufacturing parameter optimization problems. Applied Soft Computing 13(3), 1528–1542 (2013)
9. Wang, J., Zhou, L., Peng, H., Zhang, G.X.: An extended spiking neural P system for fuzzy knowledge representation. International Journal of Innovative Computing, Information and Control 7(7A) 3709–3724 (2011)
10. Wang, J., Shi, P., Peng, H., Pérez-Jiménez, M.J., Wang, T.: Weighted fuzzy spiking neural P systems. IEEE Transactions on Fuzzy Systems 21(2), 209–220 (2013)
11. Peng, H., Wang, J., Pérez-Jiménez, M.J., Wang, H., Shao, J., Wang, T.: Fuzzy reasoning spiking neural P system for fault diagnosis. Information Sciences 235 106–116 (2013)

12. Peng, H., Wang, J., Pérez-Jiménez, M.J., Shi, P.: A novel image thresholding method based on membrane computing and fuzzy entropy. Journal of Intelligent & Fuzzy Systems 24(2) 229–237 (2013)
13. Peng, H., Wang, J., Pérez-Jiménez, M.J.: Optimal multi-level thresholding with membrane computing. Digital Signal Processing 37, 53C-64 (2015)
14. Pavel, A., Buiu, C.: Using enzymatic numerical P systems for modeling mobile robot controllers. Natural Computing 11(3), 387–393 (2012)
15. Colomer, A. M., Margalida A., Pérez-Jiménez, M.J.: Population Dynamics P System (PDP) models: a standarized protocol for describing and applying novel bio-inspired computing tools. Plos One 4, 1–13 (2013)
16. Everitt, B., Landau, S., Leese, M.: Cluster Analysis. Arnold, London (2001)
17. Jain, A.K.: Data clustering: 50 years beyond k-means. Pattern Recognition Letters 31, 651–666 (2010)
18. Huang, X., Peng, H., Jiang, Y., Zhang, J., Wang, J.: PSO-MC: A novel PSO-based membrane clustering algorithm. ICIC Express Letters 8(2), 497–503 (2014)
19. Jiang, Y., Peng, H., Huang, X., Zhang, J., Shi, P.: A novel clustering algorithm based on P systems. International Journal of Innovative Computing, Information and Control 10(2), 753–765 (2014)
20. Peng, H.,Luo, X., Gao, Z., Wang, J., Pei, Z.: A novel clustering algorithm inspired by membrane computing. The Scientific World Journal, Article ID 929471, 1–9 (2015)
21. Peng, H., Zhang, J., Jiang, Y., Huang, X., Wang, J.: DE-MC: A membrane clustering algorithm based on differential evolution mechanism. Romanian Journal of Information Science and Technilogy 17(1), 2014, 76–88 (2014)
22. Peng, H., Wang, J., Pérez-Jiménez, M.J., Riscos-Núñez, A.: An unsupervised learning algorithm for membrane computing. Information Sciences 304, 80–91 (2015)
23. Peng, H., Jiang, Y., Wang, J., Pérez-Jiménez, M.J.: Membrne clustering algorithm with hybrid evolutionary mechanisms. Journal of Software 26(5), 1001–1012 (2015)
24. Quinlan, J.: Induction of decision trees. Machine Learning 1(1), 81–106 (1986)
25. Quinlan, J.: C4.5: Progrms for Machine Learning. Morgan Kaufmann, San Francisco (1993)
26. UCI datasets. http://www.ics.uci.edu/ mlearn/MLRepository.html.
27. Kretowski, M.: A memetic algorithm for global induction of decision trees. LNCS 4910, 531–540 (2008)

Fault Section Estimation of Power Systems with Optimization Spiking Neural P Systems

Tao Wang[1], Sikui Zeng[1], Gexiang Zhang[1⋆], Mario J. Pérez-Jiménez[2] and Jun Wang[3]

[1]School of Electrical Engineering, Southwest Jiaotong University,
Chengdu, 610031, P.R. China
email: wangatao2005@163.com, xiaohao0722@126.com, zhgxdylan@126.com
[2]Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla, Sevilla, 41012, Spain
email: marper@us.es
[3]School of Electrical and Information Engineering, Xihua University,
Chengdu, 610039, P.R. China
email: wangjun@mail.xhu.edu.cn

Abstract. An optimization spiking neural P system (OSNPS) provides a novel way to directly use a P system to solve optimization problems. This paper discusses the practical application of OSNPS for the first time and uses it to solve the power system fault section estimation problem formulated by an optimization problem. When the status information of protective relays and circuit breakers read from a supervisory control and data acquisition system is input, OSNPS can automatically search and output fault sections. Case studies show that OSNPS is effective in fault sections estimation of power systems in different types of fault cases, including single fault, multiple faults and multiple faults with incomplete and uncertain information.

Keywords: Membrane computing, optimization spiking neural P system, fault section estimation, power systems, fault diagnosis

1 Introduction

Membrane computing is an attractive branch of natural computing, initiated by Gh. Păun in [1], aiming at abstracting innovative computing models or computing ideas from functioning and structures of living cells, as well as from the way the cells are organized in tissues or other higher order structures. The obtained models, called membrane systems or P systems, are distributed and parallel computing models. Currently, there are three basic types of P systems: cell-like P systems, tissue-like P systems and neural-like P systems.

In recent years, the research on neural-like P systems mainly focused on spiking neural P systems (SN P systems), which were introduced in [2]. An SN P system is a class of distributed and parallel computing devices which are inspired by the way neurons communicate by means of electrical impulses (spikes). Since then, SN P systems

⋆ Corresponding author.

have become a hot topic in membrane computing [3]-[13] and an overview of this field can be found in [14], with up-to-date information available at the membrane computing website (http://ppage.psystems.eu).

In [12], an extended spiking neural P system (ESNPS) was proposed by introducing the probabilistic selection of evolution rules and multi-neurons output and correspondingly a novel way to design a P system for directly obtaining the approximate solutions of combinatorial optimization problems without the aid of evolutionary operators was introduced. Besides, a family of ESNPS, called optimization spiking neural P system (OSNPS), were further designed by introducing a guider to adaptively adjust rule probabilities to approximately solve combinatorial optimization problems. This is the first time that a strategy to design SN P systems capable of solving optimization problems is proposed. Experimental results on knapsack problems in [12] proved the viability and effectiveness of OSNPS. Moreover, the future work in [12] pointed out that OSNPS can be used to solve various application problems, such as fault diagnosis of electric power systems.

Strictly speaking, fault diagnosis of power systems includes fault detection, fault section estimation, fault type identification, failure isolation and recovery [13], [23]. Among the five processes, fault section estimation is especially important [13], [18]. Fault section estimation (FSE) identifies the fault section in power systems by using the status information of protective relays and circuit breakers (CBs) obtained from supervisory control and data acquisition (SCADA) systems [16]. So far, various approaches have been proposed to solve this problem, such as expert systems (ES) [17], fuzzy logic (FL) [15], fuzzy Petri nets (FPN)[18], artificial neural networks (ANN) [19], multi agent systems (MAS) [20], optimization methods (OM) [16], [21]-[23]. Each method has its own pros and cons [13]. Therefore, improving the aforementioned methods and developing new ones to solve fault section estimation problems is a hot topic in the research field of electrical power systems.

The power system fault section estimation problem can be effectively solved by formulating it into a 0-1 integer programming problem. In [12], only the widely used benchmark problems, knapsack problems, were applied to verify the OSNPS effectiveness and the authors pointed out that OSNPS can be used to solve various application problems. However, until now there is not any work about the real application of OSNPS. This paper discusses the application of OSNPS to fault section estimation of power systems. This is the first time to use OSNPS to solve real application problems. When the status information of protective relays and circuit breakers read from a supervisory control and data acquisition system is input, OSNPS can automatically search and output fault sections. Case studies show that OSNPS is effective in fault sections estimation of power systems in different types of fault cases including single fault, multiple faults and multiple faults with incomplete and uncertain information.

This paper is structured as follows. Section 2 states the problem to solve. Section 3 presents the fault section estimation method based on OSNPS. Subsequently, three case studies are provided in Section 4. Conclusions are finally drawn in section 5.

2 Problem Description

The aim of *fault section estimation* (FSE, for short) problem in power systems based on optimization methods (OM) is to obtain a fault hypothesis which can explain warning signals (status information) in the maximum degree. Specifically, fault section estimation can be abstracted as a 0-1 programming problem with an objective function (error function), as shown in (1), which is obtained according to the causality between a fault and the statuses of protection devices including protective relays and circuit breakers (CBs) [21]. Then, one optimization method is used to find the fault hypothesis, i.e. the minimal value of $E(S)$ in (1).

$$E(S) = \sum_{j=1}^{n_c} \left| c_j - c_j^*(S,R) \right| + \sum_{k=1}^{n_r} |r_k - r_k^*(S)|, \tag{1}$$

where:

(1) n_c represents the number of circuit breakers (CBs), n_r represents the number of protective relays;
(2) $E(S)$ represents a status function of all the sections in a power system;
(3) S is an n-vector representing the status of sections in a power system and n represents the number of sections: if section i is faulty, then $S_i = 1$, otherwise, $S_i = 0$, $i = 1, \ldots, n$;
(4) c is an n_c-vector representing the real status of CBs in a protection system: if CB j trips, then $c_j = 1$, otherwise, $c_j = 0$, $j = 1, \ldots, n_c$;
(5) $c^*(S,R)$ is an n_c-vector representing the expected status of CBs in a protection system and n_c represents the number of CBs: if CB j should trip, then $c_j^* = 1$, otherwise, $c_j^* = 0$, $j = 1, \ldots, n_c$;
(6) r is an n_r-vector representing the real status of protective relays in a protection system and n_r represents the number of protective relays: if a protective relays operates, then $r_k = 1$, otherwise, $r_k = 0$, $k = 1, \ldots, n_r$;
(7) $r^*(S)$ is an n_r-vector representing the expected status of protective relays in a protection system: if protective relay k should operate, then $r_k^* = 1$, otherwise, $r_k^* = 0$, $k = 1, \ldots, n_r$.

In this study, OSNPS is used to fulfill fault section estimation in power systems by minimizing $E(S)$ in (1). Specifically, the expected status of protective relays and CBs can be obtained according to their operation principles and the protection structure of a power system. The real status of protective relays and CBs are normally read from a power SCADA system. When all the expected status and real status of protections are obtained, we can use an OSNPS to find the minimal value of $E(S)$ in (1). The aim of fault section estimation is to obtain vector elements of S corresponding to the the minimum value of (1).

3 Fault Section Estimation Based on OSNPS

3.1 Optimization Spiking Neural P System

First, let us recall the concept of extended spiking neural P systems introduced in [12] (it is depicted in Fig. 1).

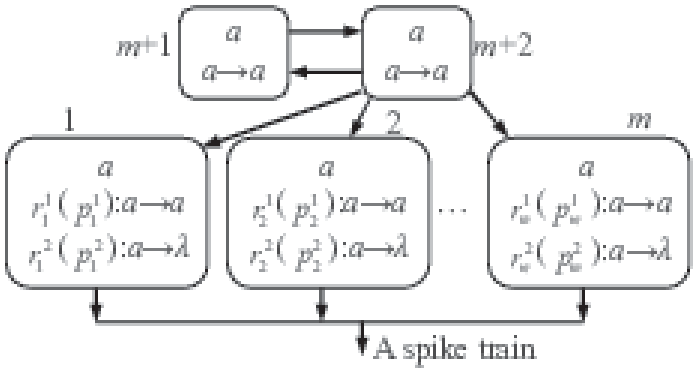


Fig. 1. The ESNPS structure

Definition 1. *An extended spiking neural P system (ESNPS, for short) of degree* $m \geq 1$*, is a tuple* $\Pi = (O, \sigma_1, \ldots, \sigma_{m+2}, syn, I_0)$*, where:*

(1) $O = \{a\}$ *is the singleton alphabet (*a *is called* spike*);*

(2) $\sigma_i, 1 \leq i \leq m$*, are neurons* $\sigma_i = (1, R_i, P_i)$*, where* $R_i = \{r_i^1, r_i^2\}$ *being* $r_i^1 = \{a \to a\}$*,* $r_i^2 = \{a \to \lambda\}$*, and* $P_i = \{p_i^1, p_i^2\}$ *is a finite set of probabilities (*p_i^j *is associated with rule* r_i^j*,* $1 \leq j \leq 2$*) such that* $p_i^1 + p_i^2 = 1$*;*

(3) $\sigma_{m+1} = \sigma_{m+2} = (1, \{a \to a\})$*;*

(4) $syn = \{(i, j) \mid (i = m + 2 \wedge 1 \leq j \leq m + 1) \vee (i = m + 1 \wedge j = m + 2)\}$*;*

(5) $I_0 = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ *is a finite set of* output *neurons, i.e., the output is a spike train formed by concatenating the outputs of* σ_1*,* σ_2*, ...,* σ_m*.*

This system contains the subsystem consisting of neurons σ_{m+1} and σ_{m+2}, and this subsystem is used as a as a step by step supplier of spikes to neurons $\sigma_1, \ldots, \sigma_m$. In the subsystem, there are two identical neurons, each of which fires at each moment of time and sends a spike to each of neurons $\sigma_1, \ldots, \sigma_m$, and reloads each other continuously. At each time unit, each of neurons $\sigma_1, \ldots, \sigma_m$ performs the firing rule r_i^1 by probability p_i^1 and the forgetting rule r_i^2 by probability p_i^2, $i = 1, 2, \ldots, m$. If the ith neuron spikes, we obtain its output 1, i.e., we obtain 1 by probability p_i^1, otherwise, we obtain its output 0, i.e., we obtain 0 by probability p_i^2, $i = 1, 2, \ldots, m$. Thus, this system outputs a spike train consisting of 0 and 1 at each moment of time. If we can adjust the probabilities p_1^1, $\ldots$, p_m^1, we can control the output spike train. So, a method to adjust the probabilities $p_i^1, \ldots, p_m^1$ by introducing a family of ESNPS is presented and described as follows.

A certain number of ESNPS can be organized into a family of ESNPS (called OSNPS) by introducing a guider to adjust the selection probabilities of rules inside each neuron of each ESNPS. The structure of OSNPS is shown in Fig. 2, where OSNPS consists of H ESNPS, $\text{ESNPS}_1, \text{ESNPS}_2, \ldots, \text{ESNPS}_H$. Each ESNPS is identical with the one in Fig. 1 and the pseudocode algorithm of the guider algorithm is illustrated in Fig. 3. For details about the guider and more information about ESNPS and OSNPS, please see [12].

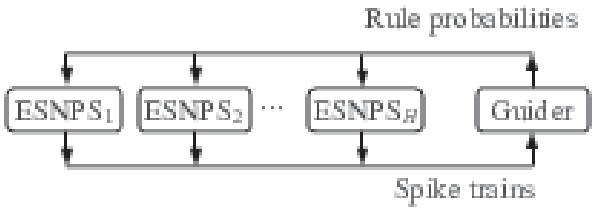


Fig. 2. OSNPS

3.2 Fault Section Estimation Based on OSNPS

The process of OSNPS applied to the FSE problem can be illustrated by the sketch map in Fig. 4, which depicts how to estimate fault sections using OSNPS. To clearly present the process in Fig. 4, a detailed description is given as follows.

Step 1: Input data

To start the method, SCADA data, parameters of OSNPS and initial value of the fitness function are required. Thus, the input data block/process consist of three parts which are described as follows.

1) Read SCADA data. The status information including the status of protective relays and CBs, the topological connection of a given power system and its protection system structure information are read from an SCADA system;

2) Set parameters of OSNPS. The parameters refer to the number of ESNPS (H), the dimension of each ESNPS (m), the learning probabilities, the learning rate, the rule probability matrix, maximum iterations and so on;

3) Initial fitness function. Above mentioned data are used to initial fitness function of the FSE problem according to (1).

Step 2: Fault section estimation with OSNPS

Perform OSNPS to produce and update spike trains to find the minimum value of (1). As mentioned in Subsection 3.1, each ESNPS can produce a spike train, which stores the needed result in binary encoding. H ESNPS are organized into an OSNPS by a guider to adjust the selection probabilities of rules inside each neuron of each ESNPS. The guider algorithm, as shown in Fig. 3 and described in [12] in detail, is used to help OSNPS getting the spike train which brings the minimum value of (1).

Step 3: Stopping condition

The optimization process is terminated when either reaching the maximum iterations or concluding that no better solution would appear in the following iterations.

Step 4: Output fault section estimation results

The spike train corresponding to the minimum value of (1) is output in an n-vector S and $S_i = 1$ is the ith faulty section, $i = 1, \ldots, n$.

4 Case studies

Fig. 5 shows a typical 4-substation system including 28 system sections, 40 CBs and 84 protective relays [13], [23]. Normally, the protective relays consist of main protective relays (MPRs), first backup protective relays (FBPRs) and second backup protective

Input: Spike train T_s, p_j^a, Δ, H and m
Rearrange T_s as matrix P_R
$i = 1$
while $(i \leq H)$ **do**
  j=1
  while $(j \leq m)$ **do**
    if $(rand < p_j^a)$ **then**
      $k_1, k_2 = ceil(rand * H), k_1 \neq k_2 \neq i$
      if $(f(C_{k_1}) > f(C_{k_2}))$ **then**
        $b_j = b_{k_1}$
      else
        $b_j = b_{k_2}$
      end if
      if $(b_j > 0.5)$ **then**
        $p_{ij}^1 = p_{ij}^1 + \Delta$
      else
        $p_{ij}^1 = p_{ij}^1 - \Delta$
      end if
    else
      if $(b_j^{max} > 0.5)$ **then**
        $p_{ij}^1 = p_{ij}^1 + \Delta$
      else
        $p_{ij}^1 = p_{ij}^1 - \Delta$
      end if
    end if
    if $(p_{ij}^1 > 1)$ **then**
      $p_{ij}^1 = p_{ij}^1 - \Delta$
    else
      if $(p_{ij}^1 < 0)$ **then**
        $p_{ij}^1 = p_{ij}^1 + \Delta$
      end if
    end if
    $j = j + 1$
  end while
  $i = i + 1$
end while
Output: Rule probability matrix P_R

Fig. 3. Guider Algorithm

relays (SBPRs) in power systems. The detailed operational rules of protective relays for main sections in a power system can be found in [13], [23].

To test the effectiveness and superiority of OSNPS in fault section estimation, three cases of the local power system in Fig. 5 are considered. The status information about protective relays and CBs of these cases is shown in Table 1, where *Case 1* has a single fault, *Case 2* has multiple faults and *Case 3* has multiple faults with incompleteness and uncertainty. OSNPS is used to estimate fault sections for the three cases, the estimation results are shown in Table 2 with a comparison with three other fault section estimation

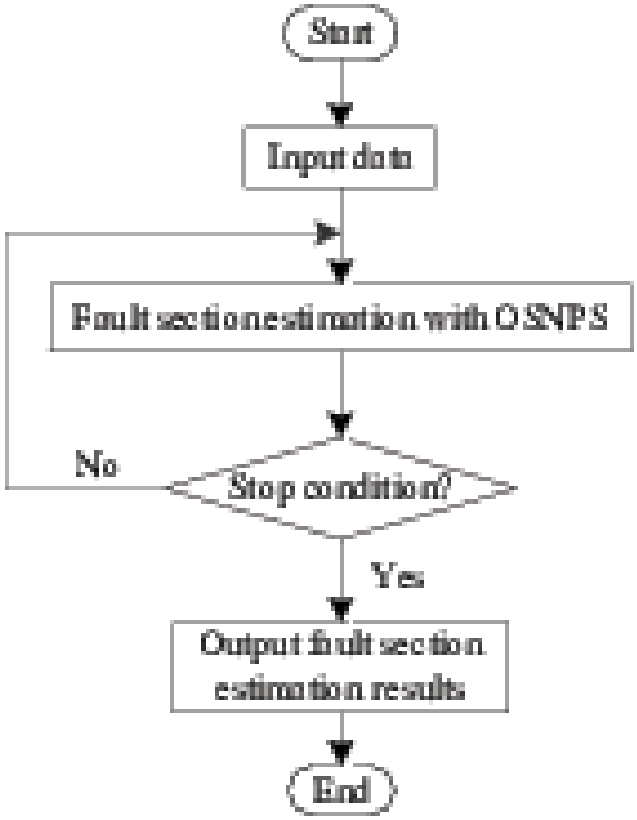


Fig. 4. The sketch map of fault section estimation based on OSNPS

methods, where " – " means that this case was not considered in the corresponding reference.

From Table 2, we can see that the estimation results of OSNPS, in *Cases 1-2*, are the same as those of fuzzy logic [FL], genetic algorithm (GA) and FDSNP in [15], [23] and [13], respectively. In other words, OSNPS is effective in fault section estimation of power systems for single and multiple faults. In *Case 3*, the estimation result of OSNPS is different from those in [15] and [23]. According to the results in [13] and [21], we know that the result of OSNPS is correct. Therefore, from the three typical cases, OSNPS is effective in fault section estimation of power systems for single fault, multiple faults and multiple faults with incomplete and uncertain alarm information.

5 Conclusions

In this study, an optimization spiking neural P system (OSNPS) is applied to fault section estimation of power systems. When status information of protection devices (protective relays and CBs) are obtained from the SCADA system, OSNPS can automatically get the minimal value of the objective function of the FSE problem and accordingly determine fault sections. Three typical case studies show that OSNPS is effective in fault section estimation of power systems. On the one hand, this study provides an alternative method for solving the fault section estimation problem in power systems. On

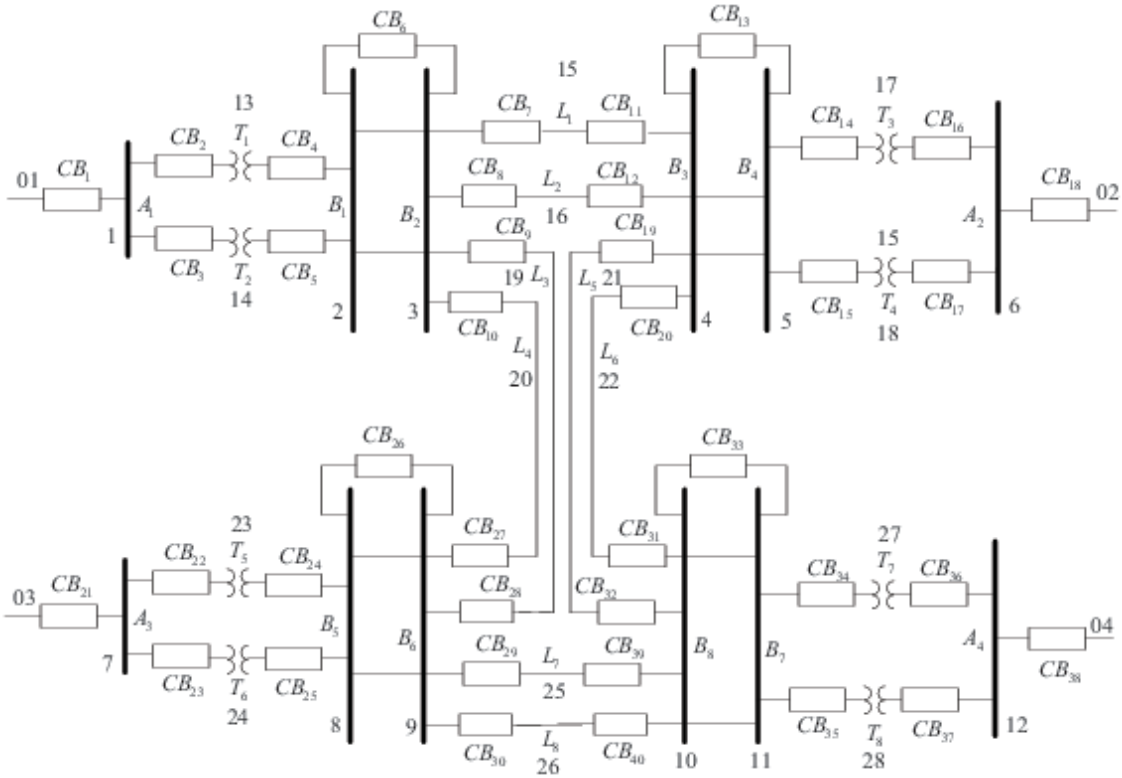


Fig. 5. A local sketch map of the protection system of an EPS.

Table 1. Status information about protective relays and CBs

Cases	Status information	
	Operated relays	Tripped CBs
1	B_{1m}, L_{2Rs}, L_{4Rs}	CB_4, CB_5, CB_7 CB_9, CB_{12}, CB_{27}
2	B_{1m}, L_{1Sm}, L_{1Rp} B_{2m}, L_{2Sp}, L_{2Rm}	CB_4, CB_5, CB_6 CB_7, CB_8, CB_9 $CB_{10}, CB_{11}, CB_{12}$
3	T_{7m}, T_{8P}, B_{7m} B_{8m}, L_{5Sm}, L_{5Rp} $L_{6Ss}, L_{7Sp}, L_{7Rm}, L_{8Ss}$	$CB_{19}, CB_{20}, CB_{29}, CB_{30}$ $CB_{32}, CB_{33}, CB_{34}, CB_{35}$ $CB_{36}, CB_{37}, CB_{39}$

the other hand, this study advances the work in [12] forward and is of great significance in extending the application of P systems and variant SN P systems.

This works focuses on the effectiveness of OSNPS in fault section estimation of power systems. In the future, we will pay attention to explore superiority of OSNPS in fault diagnosis of power systems and its availability in large-scale power grid and complex power systems.

Table 2. Comparisons between OSNPS and three fault diagnosis methods

Cases	Diagnosis results				
	OSNPS	FL [15]	GA [23]	FDSNP [13]	GATS [21]
1	B_1	B_1	B_1	B_1	-
2	B_1, B_2 L_1, L_2	B_1, B_2 L_1, L_2	B_1, B_2 L_1, L_2	B_1, B_2 L_1, L_2	-
3	L_5, L_7 B_7, B_8 T_7, T_8	L_5, L_7 B_8, T_7 T_8	(1)L_5, L_7, B_7, B_8 T_7, T_8 (2)L_5, L_7, T_7, B_8	L_5, L_7 B_7, B_8 T_7, T_8	L_5, L_7 B_7, B_8 T_7, T_8

Acknowledgment

This work is supported by the National Natural Science Foundation of China (61170016, 61373047, 61472328). The work of M.J. Pérez-Jiménez is supported by Project TIN2012-37434 of the Ministerio de Economía y Competitividad of Spain.

References

1. Gh. Păun, "Computing with Membranes". J. Comput. Syst. Sci., 61(1), 108-143 (2000)
2. M. Ionescu, Gh. Păun and T. Yokomori, "Spiking neural P systems," *Fund. Inform.*, 71(2-3), 279-308 (2006)
3. Gh. Păun, M. J. Pérez-Jiménez and G. Rozenberg, "Spike train in spiking neural P systems," *Int. J. Found. Comput. Sci.*, 17(4), 975-1002 (2006)
4. R. Freund, M. Ionescu and M. Oswald, "Extended spiking neural P systems with decaying spikes and/or total spiking," *Int. J. Found. Comput. Sci.*, 19(5), 1223-1234 (2008)
5. M. Cavaliere, O.H. Ibarra, Gh. Păun, O. Egecioglu, M. Ionescu and S. Woodworth, "Asynchronous spiking neural P systems," *Theor. Comput. Sci.*, 410(24-25), 2352-2364 (2009)
6. F. George, C. Cabarle, H. N. Adorna, M.A. Martínez-del-Amor and M.J. Pérez-Jiménez, "Improving GPU simulations of spiking neural P systems," *Rom. J. Inf. Sci. Tech.*, 15(1), 5-20 (2012)
7. T. Song, L. Q. Pan and Gh. Păun, "Asynchronous spiking neural P systems with local synchronization," *Inform. Sciences*, 219, 197-207 (2013)
8. J. Wang, P. Shi, H. Peng, Mario J. Pérez-Jiménez and T. Wang, "Weighted fuzzy spiking neural P system," *IEEE Trans. Fuzzy Syst.*, 21(2), 209-220 (2013)
9. H. Peng, J. Wang, M. J. Pérez-Jiménez, H. Wang, J. Shao and T. Wang, "Fuzzy reasoning spiking neural P system for fault diagnosis," *Inform. Sciences*, 235, 106-116 (2013)
10. M. Tu, J. Wang, H. Peng and P. Shi, "Application of adaptive fuzzy spiking neural P systems in fault diagnosis of power systems," *Chinese J. Electron*, 23(1), 87-92 (2014)
11. T. Wang, G. X. Zhang, H. N. Rong and M. J. Pérez-Jiménez, "Application of fuzzy reasoning spiking neural P systems to fault diagnosis," *Int. J Comput. Commun. Control*, 9(6), 786-799 (2014)
12. G. X. Zhang, H. N. Rong, F. Neri and Mario J. Pérez-Jiménez, "An optimization spiking neural P system for approximately solving combinatorial optimization problems," *Int. J. Neural Syst.*, 24(5), 1440006 (16 pages) (2014)

13. T. Wang, G. X. Zhang, J. B. Zhao, Z. Y. He, J. wang and M. J. Pérez-Jiménez, "Fault diagnosis of electric power systems based on fuzzy reasoning spiking neural P systems", *IEEE Trans. Power Syst.*, 30(3), 1182-1194 (2015)
14. Gh. Păun, G. Rozenberg and A. Salomaa, The Oxford Handbook of Membrane Computing, Oxford Unversity Press, New York (2010)
15. C. S. Chang, J. M. Chen, D. Srinivasan, F. S. Wen and A. C. Liew, "Fuzzy logic approach in power system fault section identification," *IEE Proc. of Gener. Transm. Distrib.*, 144(5), 406-414 (1997)
16. S. J. Huang and X. Z. Liu, "Application of artificial bee colony-based optimization for fault section estimation in power systems," *Int. J. Elec. Power*, 44(2013), 210-218 (2013)
17. H. J. Lee, B. S. Ahn and Y. M. Park, "A fault diagnosis expert system for distribution substations," *IEEE Trans. Power Del.*, 15(1), 92-97 (2000)
18. J. Sun, S. Y. Qin and Y. H. Song, "Fault diagnosis of electric power systems based on fuzzy Petri nets," *IEEE Trans. Power Syst.*, 19(4), 2053-2059 (2004)
19. G. Cardoso, J. G. Rolim and H. H. Zurn, "Identifying the primary fault section after contingencies in bulk power systems," *IEEE Trans. Power Del.*, 23(3), 1335-1342 (2008)
20. Y. L. Zhu, L. M. Huo and J. L. Liu "Bayesian networks based approach for Power Systems Fault Diagnosis," *IEEE Trans. Power Del.*, 21(2) 634-639 (2006)
21. X. N. Lin, S. H. Ke, Z. T. Li, H. L. Weng and X. H. Han, "A fault diagnosis method of power systems based on improved objective function and genetic algorithm-tabu search," *IEEE Trans. Power Del.*, 25(3), 1268-1274 (2010)
22. Z. Y. He, H. D. Chiang, C. W. Li and Q. F. Zeng, "Fault-section estimation in power systems based on improved optimization model and binary particle swarm optimization," *Proc. of IEEE PESGM*, pp. 1-8, IEEE Press, Canada (2009)
23. F. S. Wen and Z. X. Han, "Fault section estimation in power systems using a genetic algorithm," *Electr. Power Syst. Res.*, 34(3), 165-172 (1995)