On the Classes of Languages Characterized by Generalized P Colony Automata

Kristóf Kántor, György Vaszil

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

Summary. We study the computational power of generalized P colony automata and show how it is influenced by the capacity of the system (the number of objects inside the cells of the colony) and the types of programs which are allowed to be used (restricted and unrestricted com-tape and all-tape programs, or programs allowing any kinds of rules).

1 Introduction

P colonies are variants of very simple membrane systems, which are similar to so-called colonies of simple grammars, a model in the theory of grammar systems, launched by the introduction of cooperating, distributed systems of grammars in [4]. One of the grammatical models of the field is the colony of grammars, see [12], which is a collection of very simple generative grammars, but as a system, they are able to generate complicated languages. For more on grammar systems and colonies the interested reader is referred to the monograph [5].

Similarly to the grammar systems variant, P colonies also consist of a collection of very simple computing agents which interact in a shared environment, see [13, 14]. The environment and the computing agents are both described by multisets of objects which are processed by the colony members using rules which enable the transformation of the objects and the exchange of objects between the colony members and the environment. The rules are grouped into programs, which execute the rules they contain in parallel. A computation consists of a sequence of computational steps during which the colony members execute their programs in parallel, until the system reaches a halting configuration.

P colony automata, a variant of P colonies characterizing string languages instead of multiset collections were introduced in [3] where several of its variants were shown to be computationally complete in. The power of some of those left open there was further examined in [1].

Generalized P colony automata were introduced in [11] in order to make the model resemble more to the standard models of membrane computing, in particu-

lar, to the model of P automata, introduced in [7]. In this case, the computation of the colony defines an accepted multiset sequence, which is turned into an accepted string by a non-erasing mapping (as in P automata). In [11] some basic variants of the model were introduced and studied from the point of view of their computational power. Here we continue the investigations by examining generalized P colony automata of capacity one, two, and three, and also take the initial steps in the study of the relationship of their languages and the languages accepted by P automata.

2 Preliminaries and Definitions

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

A *two-counter machine*, see [9], $M = (\Sigma \cup \{Z, B\}, Q, q_0, q_F, Tr)$ is a 3-tape Turing machine where Σ is an *alphabet*, Q is a set of *internal states* with $q_0, q_F \in Q$ being the initial and the final states, and Tr is a set of *transition rules*. The machine has a read-only input tape and two semi-infinite storage tapes which are used as counters. The alphabet of the storage tapes contains only two symbols, Z and B (blank), while the alphabet of the input tape is $\Sigma \cup \{B\}$. The symbol Z is written on the first, leftmost cells of the storage tapes which are scanned initially by the tape heads. An integer t can be stored by moving a tape head t cells to the right of Z. A stored number can be incremented or decremented by moving the tape head right or left. The machine is capable of checking whether a stored value is *zero* or not by looking at the symbol scanned by the tape heads. If the scanned symbol is Z, then the value stored in the corresponding counter is *zero*.

Without the loss of generality, we assume that two-counter machines check and modify only one of their counters during any transition, thus, the rule set Tr contains transition rules of the form $(q, x, i, \alpha) \to (q', \beta)$ where $x \in \Sigma \cup \{B\} \cup \{\varepsilon\}$ corresponds to the symbol scanned on the input tape in state $q \in Q$, and $\alpha \in \{Z, B\}$, $i \in \{1, 2\}$ correspond to the symbols scanned on the i-th storage tape. By a rule of the above form, M enters state $q' \in Q$, and the i-th counter is modified according to $\beta \in \{-1, 0, +1\}$. If $x \in \Sigma \cup \{B\}$, then the machine was scanning x on the input tape, and the head moves one cell to the right; if $x = \varepsilon$, then the machine performs the transition irrespective of the scanned input symbol, and the reading head does not move.

A word $w \in \Sigma^*$ is accepted by the two-counter machine if starting in the initial state q_0, the input head reaches and reads the rightmost non-blank symbol on the input tape, and the machine is in the accepting state q_F. Two-counter machines are computationally complete; they are just as powerful as Turing machines (see [9] for more details).

We will also need the notion of a *register machine*, and we also consider a variant: a *register machine with input tape*. Such a machine consists of a given

number of registers each of which can hold an arbitrarily large non-negative integer number (we say that the register is empty if it holds the value zero), and a set of labeled instructions which specify how the numbers stored in registers can be manipulated (see [15] for more information).

Formally, a *register machine* is a construct $M = (m, H, l_0, l_h, R)$, where m is the number of registers, H is the set of instruction labels, l_0 is the start label, l_h is the halting label, and R is the set of instructions; each label from H labels only one instruction from R. There are several types of instructions which can be used. For $l_i, l_j, l_k \in H$ and $r \in \{1, \ldots, m\}$ we have

- $l_i : (\texttt{ADD}(r), l_j)$ - *add*: Add 1 to register r and then go to the instruction with label l_j.
- $l_i : (\texttt{CHECKSUB}(r), l_j, l_k)$ - *zero check and subtract*: If the value of register r is not zero, subtract one from it and go to instruction l_j, otherwise leave it unchanged and go to l_k.
- $l_h : \texttt{HALT}$ - *halt*: Stop the machine.

A register machine accepts a number m if starting the computation with the instruction labeled by l_0 while having m in the first register (and all other registers empty), it reaches the halting instruction. This way a register machine computes a set of numbers.

To be able to accept strings, we might also add an input tape to a register machine, together with a new type of instruction

- $l_i : (\texttt{READ}(a), l_j)$ for a symbol $a \in \Sigma$ of some input alphabet Σ.

Such an instruction can be applied if the reading head scans a symbol $a \in \Sigma$ on the input tape, and the head moves to the next tape cell after the application of the instruction.

It is not difficult to see that register machines with input tape characterize the class of recursively enumerable languages, as they can simulate two-counter machines. To see this, consider the following. For each transition $t : (q, x, i, \alpha) \to (q', \beta)$ of a two-counter machine M_{2c}, construct the instructions for a register machine M_R as follows.

Let M_R have a register r_q for each state $q \in Q$, and a register r_i for each counter c_i of M_{2c}. Initially the register for the initial state contains the value one, and all other registers are empty. The transition t is simulated by several instructions of M_R.

The simulation starts with $l_t : (\texttt{READ}(x), l_{t,1})$, and then continues with $l_{t,1} : (\texttt{CHECKSUB}(q), l_{t,2}, l_{trap})$ where l_{trap} is a "trap" label with a "trap" instruction $l_{trap} : (\texttt{ADD}(q), l_{trap})$. Then $l_{t,2} : (\texttt{ADD}(q'), l_{t,3})$ follows for the new state q'. Now, if $\alpha = B$, then the next instructions are $l_{t,3} : (\texttt{CHECKSUB}(i), l_{t,4}, l_{trap})$, and $l_{t,4} : (\texttt{ADD}(i), l_{t,5})$, if $\alpha = 0$ then $l_{t,3} : (\texttt{CHECKSUB}(i), l_{trap}, l_{t,5})$. These instructions check the required state of the ith register. If $\beta = 0$, then $l_{t,5}$ can be replaced by the label $l_{t'}$ for a new transition t' (starting with the state q') of M_{2c}. If $\beta = -1$, then $l_{t,5} : (\texttt{CHECKSUB}(i), l_{t'}, l_{trap})$, if $\beta = +1$, then $l_{t,5} : (\texttt{ADD}(i), l_{t'})$.

If we define the instructions of M_R in such a way that each accepting transition of M_{2c} can also lead to the halting instruction, then M_R accepts an input word if and only if M_{2c} does.

Now we define the notions related to multisets as follows. If the set of non-negative integers is denoted by $\mathbb{N}$, then a multiset over a set V is a mapping $M : V \to \mathbb{N}$ which assigns to each object $a \in V$ its multiplicity $M(a)$ in M. The support of M is the set $supp(M) = \{a \mid M(a) \geq 1\}$. If V is a finite set, then M is called a finite multiset. A multiset M is empty if its support is empty, $supp(M) = \emptyset$. We will represent a finite multiset M over V by a string w over the alphabet V with $|w|_a = M(a)$, $a \in V$, and ε will represent the empty multiset which is also denoted by $\emptyset$.

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

A P system, see [17], is a structure of hierarchically embedded membranes (a rooted tree), each having a unique label and enclosing a region containing a multiset of objects. The outmost membrane is called the skin membrane.

An antiport rule is of the form $(u, in; v, out)$, where $u, v \in V^*$ are finite multisets over V. If such a rule is applied in a region, then the objects of u enter from the parent region and, in the same step, objects of v leave to the parent region.

A *P automaton*, see [6] $\Pi = (V, \mu, w_1, \ldots, w_k, P_1, \ldots, P_k)$ is a membrane system with object alphabet V, membrane structure μ, initial contents (multisets) of the ith region $w_i \in V^*$, $1 \leq i \leq k$, and sets of antiport rules P_i, $1 \leq i \leq k$.

The configurations of the P automaton can be changed by transitions in the sequential mode (seq) or in the non-deterministic maximally parallel mode (par). In the first case one rule is applied in each region in every step, in the second case as many rules are applied simultaneously in the regions at the same step as possible. Thus, a transition in the P automaton Π is $(v_1, \ldots, v_m) \in \delta_{\Pi,X}(u_0, u_1, \ldots, u_m)$, where $\delta_{\Pi,X}$ denotes the transition relation, $X \in \{seq, par\}$, $u_1, \ldots, u_k$ are the contents of the k regions, u_0 is the multiset entering the system from the environment, and $v_1, \ldots, v_k$, respectively, are the contents of the k regions after performing the transition in the working mode.

In this way, there is a sequence of multisets which enter the system from the environment during the steps of its computations. If the computation is accepting, that is, if it halts, then this multiset sequence is called an accepted multiset sequence, and denoted by $A(\Pi)$ for a P automaton Π.

Before giving the definition of the accepted string languages of P automata, we define the notion of a generalized P colony automaton (genPCol automaton in short).

Definition 1. A *genPCol automaton* of capacity k and with n cells, $k, n \geq 1$, is a construct $\Pi = (V, e, w_E, (w_1, P_1), \ldots, (w_n, P_n), F)$ where

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

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

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

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

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

To describe the computation process formally, for any rule r we define the following multisets. Let $X \in \{T, \varepsilon\}$, and if $r = a \xrightarrow{X} b$, or $r = a \overset{X}{\leftrightarrow} b$, then let $left(r) = a$, $right(r) = b$. Let us extend this notation also for programs. For $\alpha \in \{left, right\}$ and for any program p, let $\alpha(p) = \bigcup_{r \in p} \alpha(r)$ where the union denotes multiset union (as defined above), and for a rule r and program $p = \langle r_1, \ldots, r_k \rangle$, the notation $r \in p$ denotes the fact that $r = r_j$ for some j, $1 \leq j \leq k$. Moreover, for any tape program p we also define $read(p)$ as the multiset of symbols (different from e) on the right side of rewriting tape rules and on the left side of communication tape rules, that is, $read(p) = \bigcup_{r \in p, r = a \xrightarrow{T} b, b \neq e} right(r) \cup \bigcup_{r \in p, r = a \overset{T}{\leftrightarrow} b, a \neq e} left(r)$. Thus, $left(r)$ and $right(r)$ are the multisets consisting of the symbol on the left or right side of the rule r. For a program p, $left(p)$ and $right(p)$ are the collection (multiset) of symbols on the left or right sides of the rules in the program p. Finally,

$read(p)$ is the multiset (collection) of symbols (different from e) on the right side of rewriting tape rules or the left side of communication tape rules.

We also denote by $export(p)$ and by $import(p)$ the multisets $export(p) = \bigcup_{r\in p, r=a\overset{X}{\leftrightarrow}b, a\neq e} a$ and $import(p) = \bigcup_{r\in p, r=a\overset{X}{\leftrightarrow}b} b$, and by $create(p)$ the multiset $create(p) = \bigcup_{r\in p, r=a\overset{X}{\rightarrow}b} b$. So by $export(p)$ and $import(p)$, that were defined for communication rules of a given program p, we indicate the objects that are sent out to the environment and brought inside the cell, respectively. Whereas $create(p)$ is the multiset of symbols produced by the rewriting rules of program p.

Let the programs of each P_i be labeled in a one-to-one manner by labels from the set $lab(P_i)$, $1 \leq i \leq n$, $lab(P_i) \cap lab(P_j) = \emptyset$ for $i \neq j$. In the following, for the sake of brevity, if no confusion arises, we designate programs and their labels with the same letters, thus, for a label $p \in lab(P_i)$, we also write $p \in P_i$.

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

- At most one program is selected for each cell, that is, if $p, p' \in P_c, p \neq p'$ and $p \in P_i, p' \in P_j$, then $i \neq j$;
- the selected programs are applicable in the cells (the left sides of the rules contain the same symbols that are present in the cell), that is, for each $p \in P_c$, if $p \in P_i$ then $left(p) = u_i$;
- the symbols which are brought inside the cells by the programs are present in the environment, that is, $\bigcup_{p\in P_c} import(p) \subseteq U_E$;
- P_c is maximal, that is, if any other program is added to it, then some of the above conditions are not satisfied.

A configuration $c = (u_E, u_1, \ldots, u_n)$ is *changed* to a configuration $c' = (u'_E, u'_1, \ldots, u'_n)$ and is denoted by $c \Longrightarrow c'$ by applying the set P_c of applicable programs if the following properties hold:

- If there is a $p \in P_c$ such that $p \in P_i$, then $u'_i = create(p) \cup import(p)$, otherwise $u'_i = u_i$, $1 \leq i \leq n$; and
- $U'_E = U_E - \bigcup_{p\in P_c} import(p) \cup \bigcup_{p\in P_c} export(p)$ (where U'_E again denotes $u'_E \cup \{e, e, \ldots\}$ with an infinite number of es).

We denote the reflexive and transitive closure of $\Longrightarrow$ by $\Longrightarrow^*$.

The general idea behind the above definitions is that instead of the different computational modes used in [3], we have a system with programs and we apply the programs in the maximally parallel way as usual in P colonies, that is, in each computational step, every component cell must non-deterministically choose and apply one of its applicable programs. Then we look at those rules which were tape rules (in the applied set of programs) and collect all the symbols that they "read": this multiset (of the collected symbols) is the multiset read by the system in the given computational step. A successful computation defines this way an accepted

sequence of multisets: the sequence of multisets entering the system during the steps of the computation.

Definition 2. Let $\Pi = (V, e, w_E, (w_1, P_1), \ldots, (w_n, P_n), F)$ be a genPCol automaton. The *set of input sequences accepted by* Π is defined as

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

Now we define the accepted string languages for both genPCol automata, and "ordinary" P automata.

Definition 3. Let Π be a genPCol automaton or a P automaton, and let $f : (V - \{e\})^* \to 2^{\Sigma^*}$ be a mapping, such that $f(u) = \varepsilon$ if and only if u is the empty multiset.

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

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

From now on, we are going to consider the mapping: f_{perm} defined for any multiset $x \in (V - \{e\})^*$ as

$$f(x) = \{y \in (V - \{e\})^* \mid y \in perm(x)\}$$

where $perm(x) \subseteq V^*$ denotes the set of strings representing the multiset composed of the symbols of x, or in other words, $perm(x)$ is the set of strings obtained by a permutation of the symbols of the multiset x.

Concerning the power of P automata with the mapping f_{perm}, the reader is referred to [8] and [10]. In general, they characterize a language class that is strictly included in the class of languages that can be accepted by logarithmic space bounded Turing machines that read their input tape from left to right only once.

For genPCol automata, their working modes, or in other words, the types of programs that they are allowed to use, greatly influence their computational power. Let us refine and extend the definition of the program types defined in [11] as follows.

Definition 4.

- $\mathcal{L}(\text{genPCol}, \mathcal{F}, \text{com-tape}(k))$ is the class of languages accepted by generalized PCol automata with capacity k and with mappings from the class $\mathcal{F}$ where all the communication rules are tape rules,
- $\mathcal{L}(\text{genPCol}, \mathcal{F}, \text{all-tape}(k))$ is the class of languages accepted by generalized PCol automata with capacity k and with mappings from the class $\mathcal{F}$ where all the programs must have at least one tape rule,

- $\mathcal{L}(\text{genPCol}, \mathcal{F}, *(k))$ is the class of languages accepted by generalized PCol automata with capacity k and with mappings from the class $\mathcal{F}$ where programs with any kinds of rules are allowed.

For all-tape and com-tape languages we also define their *restricted* variants, $\mathcal{L}(\text{genPCol}, \mathcal{F}, \text{restricted all-tape}(k))$ and $\mathcal{L}(\text{genPCol}, \mathcal{F}, \text{restricted com-tape}(k))$, respectively. These are accepted by systems with programs not having any rules of types

$$e \xrightarrow{T} e, \ a \xrightarrow{T} e, \text{ and } e \xleftrightarrow{T} e, \ e \xleftrightarrow{T} a,$$

for arbitrary $a \in V$, where e is the special environmental object. Note that systems which accept languages of these restricted classes must read nonempty multisets in each computational step.

In the following, we will be considering systems with the permutation mapping f_{perm} defined above. For the sake of easier readability, we denote the languages of systems with this type of mapping as

- $\mathcal{L}_{perm}(\text{genPCol}, X(k))$, where $X \in \{\text{com-tape}, \text{all-tape}, *\}$.

3 Languages Accepted by genPCol Automata

The following are immediate consequences of the definitions.

Proposition 1 *For any class of mappings $\mathcal{F}$, we have*

1. $\mathcal{L}(\text{genPCol}, \mathcal{F}, \text{com-tape}(k)) \subseteq \mathcal{L}(\text{genPCol}, \mathcal{F}, *(k))$ *and* $\mathcal{L}(\text{genPCol}, \mathcal{F}, \text{all-tape}(k)) \subseteq \mathcal{L}(\text{genPCol}, \mathcal{F}, *(k)$ *for any* $k \geq 1$*;*
2. $\mathcal{L}(\text{genPCol}, \mathcal{F}, \text{restricted } X(k)) \subseteq \mathcal{L}(\text{genPCol}, \mathcal{F}, X(k))$ *for any* $k \geq 1$ *and* $X \in \{\text{com-tape}, \text{all-tape}, *\}$*; and*
3. $\mathcal{L}(\text{genPCol}, \mathcal{F}, X(k)) \subseteq \mathcal{L}(\text{genPCol}, \mathcal{F}, X(k+1))$ *for any* $k \geq 1$ *and* $X \in \{\text{com-tape}, \text{all-tape}, *\}$.

Proof. The first inclusions hold, as com-tape systems are special cases of all-tape systems, which are both special cases of the unrestricted variant. The second inclusion holds for a similar reason, while the third inclusion can be seen to hold if we consider that adding the of object e to the initial cell contents, and a rule $e \to e$ to the programs of all cells in a system, does not change the accepted language. □

3.1 The Capacity of genPCol Automata

First we consider genPCol automata of capacity one. In the case of P colonies, all recursively enumerable sets of integers can be characterized by systems of capacity one, see [2]. This is also true for genPCol automata with languages obtained by permutation mappings, if programs with any kind of rules are allowed.

Theorem 2. $\mathcal{L}_{perm}(\text{genPCol}, *(1)) = \mathcal{L}(RE)$.

Proof. In Theorem 1 of [2] P colonies of capacity one are shown to be able to simulate register machines. The idea of the simulation is to have an object in the environment corresponding to the label of the instruction which is to be simulated next. The cells of the system "process" the instruction label in such a way that the necessary modifications of the configuration are implemented, and the label of the next instruction is sent to the environment.

Based on this construction, we can show that genPCol automata can simulate register machines with input tape (see section 2 for the definitions), and thus, characterize the class of recursively enumerable languages. In addition to the construction in Theorem 1 of [2], we need to simulate the instructions of type $l_i : (\texttt{READ}(a), l_j)$. To do this, we add one cell (e, P_{l_i}) to the system for each such instruction of the register machine with the programs

$$P_{l_i} = \{\langle e \leftrightarrow l_i \rangle,\ \langle l_i \xrightarrow{T} a \rangle,\ \langle a \rightarrow l_j \rangle, \{\langle l_j \leftrightarrow e \rangle\}.$$

These programs can be applied when l_i appears in the environment. They read an input symbol a while exchanging l_i for l_j in the environment. □

The power of systems with capacity one decreases considerably if not all kinds of programs are allowed. The next theorem examines the relationship of regular languages and languages of genPCol automata with all-tape programs.

Theorem 3. $\mathcal{L}_{perm}(\text{genPCol}, \text{all-tape}(1))$ *is incomparable with the class of regular languages.*

Proof. First we show that there is a nonregular language in $\mathcal{L}_{perm}(\text{genPCol}, *(1))$. Let $L_1 = \{\{a, \$\}^{2n}\{c, \$\}^2\{b, \$\}^{2n+2} \mid n \geq 0\}$ be the non-regular language over $\Sigma = \{a, b, c, \$\}$, where by $\{x, y\}^m$ we denote the string $w_1 w_2 \dots w_m$ with w_i being either xy or yx, $1 \leq i \leq m$.

Consider $\Pi = (\Sigma \cup \{e\}, e, w_E, (e, P_1), (e, P_2), (e, P_3), F)$, the genPCol automaton with the sets of programs as

$$\begin{aligned} P_1 &= \{\langle e \xrightarrow{T} \$ \rangle, \langle \$ \xleftrightarrow{T} e \rangle\}, \\ P_2 &= \{\langle e \xrightarrow{T} a \rangle, \langle a \xleftrightarrow{T} e \rangle, \langle e \xrightarrow{T} c \rangle, \langle c \xleftrightarrow{T} \$ \rangle\}, \\ P_3 &= \{\langle b \xleftrightarrow{T} c \rangle, \langle c \xrightarrow{T} b \rangle, \langle b \xleftrightarrow{T} a \rangle, \langle a \xrightarrow{T} b \rangle\}, \end{aligned}$$

and set of accepting configurations: $F = \{(u, e, \$, b) \mid u \in (\Sigma \setminus \{a\})^*\}$.

It is easy to see, that the first cell starts producing \$ objects indefinitely, while the second cell reads $2n$ $(n \geq 0)$ as, while sending as in the environment n times. After stopping, the third cell starts to work, eliminating every a in the environment while reading two bs.

Next, we show that $L_2 = \{bbc, c\}$ cannot be accepted by any Π genPCol automaton with capacity one, working in all-tape mode, using the f_{perm} mapping.

Let $V = \{b, c\} \cup \{e, e'\}$ be the alphabet of a genPCol automaton. Note that these are the only symbols that might appear in the programs, and e' can only serve as the initial cell contents. It is clear, that the automaton must accept *bbc* and *c*. We show that it is impossible to accept these and only these strings. Let us examine the cases:

1. There is only one cell: $\Pi = (V, e, w_E, (w_0, P_0), F)$. In this case there are three subcases:

1.(a) $w_0 = e'$. In order to decide whether to read *c* or *bbc*, Π must create two pathways. To do this, P_0 must contain the following programs: $\langle e' \overset{T}{\to} e\rangle, \langle e \overset{T}{\to} c\rangle, \langle e \overset{T}{\to} b\rangle$ or $\langle e' \overset{T}{\to} c\rangle, \langle e' \overset{T}{\to} b\rangle$. If nondeterministically Π decides to read *b*, it should be able to read one more *b*. To do this, we can either add $\langle b \overset{T}{\to} b\rangle$ or $\langle b \overset{T}{\leftrightarrow} e\rangle$ program to P_0, but it would create a nondeterminism, so that the automaton could read strings other than *bbc* or *c*.

1.(b) $w_0 = b$ or $w_0 = c$. In this case we would need to have the program $\langle b \overset{T}{\to} b\rangle$ again, or P_0 would contain $\langle b \overset{T}{\leftrightarrow} e\rangle$ or $\langle c \overset{T}{\leftrightarrow} e\rangle$. Since we have one cell, one of these rules automatically decides which string we would like to start reading, therefore it is impossible to accept both *bbc* and *c* strings.

1.(c) The only remaining option in this case is $w_0 = e$. Here P_0 must contain $\langle e \overset{T}{\to} b\rangle$ and $\langle e \overset{T}{\to} c\rangle$. If Π nondeterministically chooses $\langle e \overset{T}{\to} b\rangle$, then it would be still left to read *bc*. There are three different rules that could be used at the moment: $\langle b \overset{T}{\to} b\rangle, \langle b \overset{T}{\to} e\rangle, \langle b \overset{T}{\leftrightarrow} e\rangle$, however these cases lead to nondeterminism, where Π could read strings other than *bbc* or *c*.

2. There are $n \geq 2$ cells: $\Pi = (V, e, w_E, (w_0, P_0), \ldots, (w_n, P_n), F)$.We are able to define the ith $(0 \leq i \leq n)$ cell in three different ways. Please note that in order to decide whether to read *c* or *bbc*, Π must create two pathways. In these cases Π would create the pathways using two or more cells:

2.(a) $w_i = b$ or $w_i = c$. Hence the maximal parallelism, the ith cell would immediately read *c* or *b*, therefore restricting to accept only *bbc* or *c*.

2.(b) $w_i = e$. P_i could contain $\langle e \overset{T}{\leftrightarrow} b\rangle$ or $\langle e \overset{T}{\leftrightarrow} c\rangle$, but these cases require to have *b* or *c* in the environment, which is impossible because of the previous case. Thus P_i must contain one or more of these programs: $\langle e \overset{T}{\to} e\rangle$, $\langle e \overset{T}{\leftrightarrow} e\rangle$, $\langle e \overset{T}{\to} b\rangle$ or $\langle e \overset{T}{\to} c\rangle$. Please note that in all of these cases, Π would be able to accept strings other than *bbc* or *c*, therefore it would be impossible to accept *bbc* and *c*.

2.(c) $w_i = e'$. In this last subcase, P_i could contain $\langle e' \overset{T}{\to} e\rangle$, $\langle e' \overset{T}{\to} b\rangle$ or $\langle e' \overset{T}{\to} c\rangle$. Choosing $\langle e' \overset{T}{\to} e\rangle$ would lead to the previous case, where $w_i = e$. The two remaining programs would immediately read strings other than *bbc* or *c*.

3. The last case that is left to be examined is when there are $n \geq 2$ cells, and Π creates the nondeterministic pathways in one cell. Let the 0th cell be the one that creates the nondeterministic pathways. Hence case (1), $w_0 = e$ and P_0 must contain $\langle e \overset{T}{\to} b\rangle$ and $\langle e \overset{T}{\to} c\rangle$. If the 0th cell decides to read *b*, we must ensure that Π would read *bc* and then stop. If an other cell would continue to read, the only

logical scenario would be to have b or c in the cell at start and the cell might only use a program in the following form: $\langle k_1 \in \{b, c\} \overset{T}{\leftrightarrow} k_2 \in (\{b, c\} \cup \{e'\})\rangle$, but it is impossible to have k_2 in the environment without reading k_2. Thus the 0th cell is to continue to read b, which can happen by one of the following programs: $\langle b \overset{T}{\rightarrow} b\rangle$ or $\langle b \overset{T}{\leftrightarrow} e\rangle$, however both of them lead to unnecessary nondeterminism.

We have covered the possible ways to construct Π. It is now easy to see, that it is impossible to construct Π in any way to accept $\{bbc, c\}$. □

Now we show that for systems with capacity at least three, their all-tape and com-tape languages include any recursively enumerable language. Given a recursively enumerable language L, the idea is to take a system of capacity two which, when any kind of programs are allowed, accept L (we refer to [11] for such a system), and transform it to a system of capacity three having a communication tape rule in each program by adding "dummy" tape rules which do not interfere with the work of the rest of the system.

Proposition 4 $\mathcal{L}_{perm}(\text{genPCol}, X(3)) = \mathcal{L}(\text{RE})$ *for* $X \in \{\text{com-tape}, \text{all-tape}\}$.

Proof. The construction is based on the proof of Theorem 3 in [11] where a genPCol automaton of capacity two with no restriction on the type of programs is presented. Modifying such a system, we can easily construct a genPCol automaton of capacity three with all-tape or even com-tape type of programs by simply putting one more e object into each cell, and add the rule $e \overset{T}{\leftrightarrow} e$ to every program. □

3.2 Variants of genPCol Automata of with Capacity Two

In [11] we have started the study of genPCol automata languages that can be accepted by systems of capacity two with the mapping f_{perm}. We have shown that if they use restricted all-tape or restricted com-tape programs, then similarly to "ordinary" P automata, they characterize a language class that is strictly included in the class of languages that can be accepted by logarithmic space bounded Turing machines that read their input tape from left to right only once.

On the other hand, even genPCol automata with restricted all-tape or restricted com-tape programs are more powerful than P automata using the mapping f_{perm}.

If we denote by $\mathcal{L}_X(f_{perm}, PA)$ the class of languages characterized by P automata with $X \in \{seq, par\}$ for parallel or sequential rule application, then we have the following.

Theorem 5. $\mathcal{L}_{perm}(\text{genPCol}, \text{restricted all-tape}(2)) \setminus \mathcal{L}_X(f_{perm}, PA) \neq \emptyset$ *for* $X \in \{seq, par\}$.

Proof. Consider the language $L = \{(ab)^n(cd)^n \mid n \geq 1\}$ which, according to [10] cannot be accepted by any P automaton using the mapping f_{perm}. The following genPCol automaton accepts L with f_{perm}.

Let $\Pi = (\{a, b, c, d\}, e, \emptyset, (ee, P), F)$ with $F = \{(u, ad) \mid u \in d^*\}$, and

$$P = \{\langle e \xrightarrow{T} a, e{\leftrightarrow}e\rangle, \langle e \xrightarrow{T} b, a{\leftrightarrow}e\rangle, \langle b \xrightarrow{T} a, e{\leftrightarrow}e\rangle, \langle b \xrightarrow{T} c, e{\leftrightarrow}e\rangle,$$
$$\langle c \xrightarrow{T} d, e{\leftrightarrow}a\rangle, \langle a \xrightarrow{T} c, d{\leftrightarrow}e\rangle\}$$

In the first phase of its functioning, the system above reads a string $(ab)^n$ while sending n copies of a into the environment. Then in the second phase, as many cds are read, as the number of as that can be found in the environment. □

Next we show that if we only require that all programs contain at least one tape rule (but unlike in the restricted case, they can also use the environmental symbol e), then any recursively enumerable language can be accepted also with systems of capacity two.

Theorem 6. $\mathcal{L}(\text{genPCol}, f_{perm}, \text{all-tape}(2)) = \mathcal{L}(\text{RE})$.

Proof. Let $L \subseteq \Sigma^*$ be an arbitrary recursively enumerable language, and let $M = (\Sigma \cup \{Z, B\}, Q, q_0, q_f, Tr)$ be a two-counter machine with $L = L(M)$, as defined in section 2.

Construct the genPCol automaton $\Pi = (V, e, w_E, (w_0, P_0), \ldots, (w_n, P_n), F)$ of capacity two, where $V = \Sigma \cup Q \cup \{t, t', t'', t''' \mid t \in Tr\} \cup \{c_1, c_2, A\}$, the initial contents of the cells are $w_0 = q_0 e$, $w_i \in \{ee, te, t''e\}$, $1 \leq i \leq n$, as we will specify later, and $F = \{(u, q_f e, w_1, \ldots, w_n) \mid u \in V^*\}$.

For any $\alpha \in \{B, Z\}$, $\beta \in \{-1, 0, +1\}$, we define the disjoint sets of transitions $Tr_{\alpha,\beta} \subseteq Tr$ as follows: $t \in Tr_{\alpha,\beta}$, if and only if, $t : (q, x, i, \alpha) \to (q', \beta)$, $x \in \Sigma \cup \{\varepsilon\}$, $i \in \{1, 2\}$. Thus, $Tr = Tr_{B,-1} \cup Tr_{B,0} \cup Tr_{B,+1} \cup Tr_{Z,0} \cup Tr_{Z,+1}$.

For every $t \in Tr_{B,+1}$ the proof will need three cells each, whereas for each $t \in (Tr_{B,-1} \cup Tr_{B,0} \cup Tr_{Z,0} \cup Tr_{Z,+1})$ only two cells are required, thus $n = 3k_1 + 2k_2$, where $k_1 = |Tr_{B,+1}|$ and $k_2 = |(Tr_{B,-1} \cup Tr_{B,0} \cup Tr_{Z,0} \cup Tr_{Z,+1})|$.

As the cells in the constructed system correspond to transitions of the simulated two-counter machine, in the following we will index the cells $C_i = (w_i, P_i)$ (except C_0) with two indices: the transition and an integer (the integer will be 1, 2, or 3, depending on how many cells the simulation of the given transition requires). The sets of programs are defined as follows:

Let $w_0 = q_0 e$, and let

$$P_0 = \{\langle q_0 \leftrightarrow e; e \xrightarrow{T} e\rangle, \langle e \xrightarrow{T} e; e \leftrightarrow q_f\rangle\}.$$

For every $t \in (Tr_{B,0} \cup Tr_{B,-1})$ we will have two cells. The initial contents of the first cell is $w_{t,1} = ee$, whereas the set of programs is the following:

$$P_{t,1} = \{p_{t_1} : \langle e \leftrightarrow q; r_{t_1}\rangle, p_{t_2} : \langle q \xrightarrow{T} e; r_{t_2}\rangle, p_{t_3} : \langle e \to t'; c_i \xrightarrow{T} e\rangle,$$
$$p_{t_4} : \langle t' \leftrightarrow e; e \xrightarrow{T} e\rangle, p_{t_5} : \langle e \xrightarrow{T} e; e \leftrightarrow t'''\rangle,$$
$$p_{t_6} : \langle t''' \to q'; e \xrightarrow{T} e\rangle, p_{t_7} : \langle q' \leftrightarrow e; e \xrightarrow{T} e\rangle\},$$

where r_{t_1} and r_{t_2} are the rules $e \xrightarrow{T} a$ and $a \leftrightarrow c_i$, respectively, if the transition is such that the input symbol is $x = a \in \Sigma$, otherwise if $x = \varepsilon$, then $r_{t_1} = e \xrightarrow{T} e$ and $r_{t_2} = e \leftrightarrow c_i$.

For every $t \in Tr_{B,-1}$ the initial contents of the second cell is still $w_{t,2} = ee$, but the set of programs is different:

$$P_{t,2} = \{p_{t_8} : \langle e \leftrightarrow t'; e \xrightarrow{T} e\rangle, p_{t_9} : \langle t' \rightarrow t'''; e \xrightarrow{T} e\rangle, p_{t_{10}} : \langle t''' \leftrightarrow e; e \xrightarrow{T} e\rangle\}.$$

Next, the initial contents of the first cell for every $t \in Tr_{B,+1}$ is $w_{t,1} = te$, and the set of programs is the following:

$$P_{t,1} = \{p_{t_1} : \langle t \leftrightarrow q; r_{t_1}\rangle, p_{t_2} : \langle q \xrightarrow{T} e; r_{t_2}\rangle, p_{t_3} : \langle c_i \leftrightarrow e; e \xrightarrow{T} e\rangle,$$
$$p_{t_4} : \langle e \rightarrow t'; e \xrightarrow{T} e\rangle, p_{t_5} : \langle e \overset{T}{\leftrightarrow} t; t' \leftrightarrow e\rangle\},$$

where r_{t_1} and r_{t_2} are the rules $e \xrightarrow{T} a$ and $a \leftrightarrow c_i$, respectively, if the transition is such that the input symbol is $x = a \in \Sigma$, otherwise if $x = \varepsilon$, then $r_{t_1} = e \xrightarrow{T} e$ and $r_{t_2} = e \leftrightarrow c_i$.

For every $t \in (Tr_{B,0} \cup Tr_{B,+1})$ the initial contents of the second cell is $w_{t,2} = t''e$ and the set of programs is as follows:

$$P_{t,2} = \{p_{t_8} : \langle t'' \leftrightarrow t'; e \xrightarrow{T} e\rangle, p_{t_9} : \langle t' \rightarrow c_i; e \xrightarrow{T} e\rangle, p_{t_{10}} : \langle c_i \leftrightarrow e; e \xrightarrow{T} e\rangle,$$
$$p_{t_{11}} : \langle e \rightarrow t'''; e \xrightarrow{T} e\rangle, p_{t_{12}} : \langle e \overset{T}{\leftrightarrow} t''; t''' \leftrightarrow e\rangle\}.$$

The initial contents of the third cell for every $t \in Tr_{B,+1}$ is $w_{t,3} = ee$, and the set of programs is the following:

$$P_{t,3} = \{p_{t_{13}} : \langle e \leftrightarrow t'''; e \xrightarrow{T} e\rangle, p_{t_{14}} : \langle t''' \rightarrow q'; e \xrightarrow{T} e\rangle, p_{t_{15}} : \langle q' \leftrightarrow e; e \xrightarrow{T} e\rangle\}.$$

For every $t \in (Tr_{Z,0} \cup Tr_{Z,+1})$ the initial contents of the first cell is $w_{t,1} = ee$ and the set of programs is as follows:

$$P_{t,1} = \{p_{t_1} : \langle e \leftrightarrow q; r_{t_1}\rangle, p_{t_2} : \langle q \rightarrow t'; r_{t_2}\rangle, p_{t_3} : \langle t' \leftrightarrow e; e \xrightarrow{T} t\rangle,$$
$$p_{t_4} : \langle e \overset{T}{\leftrightarrow} c_i; t \rightarrow A\rangle, p_{t_5} : \langle e \overset{T}{\leftrightarrow} t'''; t \rightarrow q'\rangle, p_{t_6} : \langle t''' \xrightarrow{T} e; q' \leftrightarrow e\rangle\},$$

where r_{t_1} and r_{t_2} are the rules $e \xrightarrow{T} a$ and $a \xrightarrow{T} e$, respectively, if the transition is such that the input symbol is $x = a \in \Sigma$, otherwise if $x = \varepsilon$, then $r_{t_1} = e \xrightarrow{T} e$ and $r_{t_2} = e \xrightarrow{T} e$.

For every $t \in Tr_{Z,0}$ the initial contents of the second cell is $w_{t,2} = ee$ and the set of programs is as follows:

$$P_{t,2} = \{p_7 : \langle e \overset{T}{\leftrightarrow} t'; e \rightarrow t'''\rangle, p_8 : \langle t''' \leftrightarrow e; t' \xrightarrow{T} e\rangle\}.$$

Last, but not least, for every $t \in Tr_{Z,+1}$ the initial contents of the second cell is $w_{t,2} = t''e$ and the set of programs is as follows:

$$P_{t,2} = \{p_{t_8} : \langle t'' \leftrightarrow t'; e \xrightarrow{T} e\rangle, p_{t_9} : \langle t' \rightarrow t'''; e \xrightarrow{T} e\rangle, p_{t_{10}} : \langle t''' \leftrightarrow e; e \xrightarrow{T} e\rangle,$$
$$p_{t_{11}} : \langle e \rightarrow c_i; e \xrightarrow{T} e\rangle, p_{t_{12}} : \langle e \xleftrightarrow{T} t''; c_i \leftrightarrow e\rangle\}.$$

The genPCol automaton Π simulates the work of the two-counter machine M by reading the input symbols with its tape programs and keeping track of the contents of the i-th counter as the number of c_i, $i \in \{1, 2\}$ objects present in the environment.

Each transition of M is simulated separately. At the first step, the 0th cell contains and object that corresponds to $q_0 \in Q$, which is then sent to the environment. The environment keeps track of the current internal state of M. One transition rule is simulated by the interplay of programs of two or three cells from Π, if and only if M changes its state from q to q' while the counter contents are also checked and modified accordingly.

A transition $t : (q, x, i, B) \rightarrow (q', -1) \in Tr_{B,-1}$ is simulated by two cells $C_{t,1}$, $C_{t,2}$ with the sets of programs $P_{t,1}$ and $P_{t,2}$. First, q enters the first cell C_{t_1} from the environment by program p_{t_1}, activating the simulation of the transition. Then c_i enters the first cell (program p_{t_2}), and by programs p_{t_3} and p_{t_4}, object t' is sent to the environment. Now the programs of the second cell C_{t_2} are activated, t' is changed to t''' and sent to the environment by the programs p_{t_8}, p_{t_9}, and $p_{t_{10}}$. Now by p_{t_5}, p_{t_6} of the first cell, t''' is changed to q' denoting the next state of M, and it is sent to the environment by p_{t_7}.

A transition $t : (q, x, i, B) \rightarrow (q', 0) \in Tr_{B,0}$ is also simulated by two cells with the sets of programs $P_{t,1}$ and $P_{t,2}$. Now the initial contents of the second cell is $t''e$. The first four steps are identical with the previous case described above. When t' appears in the environment, the programs of the second cell exchange it with t'' and send c_i and t''' to the environment by the programs p_{t_i}, $8 \leq i \leq 12$, then t''' is exchanged with the object q' (denoting the next state of M) in the environment by p_{t_5}, p_{t_6}, and p_{t_7} of the first cell.

A transition $t : (q, x, i, B) \rightarrow (q', +1) \in Tr_{B,+1}$ is simulated by three cells with the sets of programs $P_{t,1}$, $P_{t,2}$ and $P_{t,3}$. The initial contents of the three cells are te, $t''e$, and ee, respectively. In the first five steps the programs of the first cell are active, but besides the object t', this time the cell also sends c_i to the environment. When t' appears in the environment (after the application of $P_{t,5}$, the programs of the second cell take over. They are identical to the previous case, they exchange t' with t'' and send c_i and t''' to the environment as above. Finally the third cell becomes active, and t''' is exchanged with the object q' (denoting the next state of M) in the environment by $p_{t_{13}}$, $p_{t_{14}}$, and $p_{t_{15}}$.

A transition $t : (q, x, i, Z) \rightarrow (q', 0) \in Tr_{Z,0}$ is simulated by two cells with the sets of programs $P_{t,1}$ and $P_{t,2}$. The initial contents of both cells are ee. The first three steps, the first cell exchanges q to t' in the environment while the second cell remains inactive. When t' appears in the environment, the programs of the second cell exchange it with t''' in the next computational step. During this step, the first cell is either inactive, or imports an object c_i from the environment, if there is at least one such object is present there. In this later case, the transition cannot be

applied in a simulation of the two counter machine M, as the value stored the ith counter is not zero. This is reflected by program $p_{t,4}$ which introduces a a "trap object" A. If the transition is applicable in M, that is, if there is no c_i present in the environment. Then after two inactive steps, t''' is exchanged with the object q' (denoting the next state of M) in the environment by p_{t_5} and p_{t_6} of the first cell.

Finally, a transition $t : (q, x, i, Z) \to (q', +1) \in Tr_{Z,+1}$ is simulated by two cells with the sets of programs $P_{t,1}$ and $P_{t,2}$. The initial contents of the two cells are ee and $t''e$, respectively. The first three steps is identical with those in the previous case. When t' appears in the environment, the programs of the second cell exchange it with t'', send an object t''' and then an object c_i (in exchange with t'') to the environment. These are done by programs p_{t_5} and p_{t_6}. Similarly to the previous case, during the fourth computational step, the first cell is either inactive, or imports an object c_i from the environment (by program p_{t_4}, if there is at least one such object is present there). In the later case, the transition cannot be applied in a simulation as the value stored the ith counter of M is not zero (so the "trap object" A is introduced). If there is no c_i present in the environment (the transition is applicable in a simulation), then after three steps (in which the first cell is inactive), t''' is imported into the first cell, and then the object q' (denoting the next state of M) is sent to the environment by p_{t_5} and p_{t_6}, respectively.

According to these considerations, we have seen that having the object q in the environment, the genPCol automaton Π replaces it with q', and either simulates a transition of the two-counter machine M from state q to state q' (checking and adjusting the multiplicity of the objects corresponding to the counter contents accordingly), or its computation is not successful. Thus, starting with q_0 in the environment (q_0 is sent to the environment by the first program in P_0 in the very first step of Π) the genPCol automaton produces q_{acc}, the accepting state of the two-counter machine M in the environment if and only if its computation corresponds to an accepting computation of M. Having q_{acc} in the environment, Π can reach its final configuration by importing it into the cell C_0 by using the second program of P_0. □

4 Conclusions

We have studied the effect of the capacity of generalized P colony automata on their computational power using the all-tape and com-tape variants of programs used. We have shown that even with capacity one, if we do not place additional restrictions on the types of programs allowed to be used by the system, genPCol automata characterize the class of recursively enumerable languages. On the other hand, for systems with capacity three, even the use of most restrictive program types does not result in any decrease of the computational power. The most interesting cases are the ones in between these two: the restricted variants of capacity one and capacity two. These require further study, especially interesting would be

to refine the relationship of the model with P automata, as they are very closely related, but not as similar as one might expect at the first glance.

References

1. L. Cienciala, L. Ciencialová, P Colonies and Their Extensions. In: J. Kelemen, A. Kelemenová (eds.), *Computation, Cooperation, and Life. Essays Dedicated to Gheorghe Păun on the Occasion of His 60th Birthday.* LNCS 6610, Springer Berlin Heidelberg, 2011, 158–169.
2. L. Ciencialová, E. Csuhaj-Varjú, A. Kelemenová, Gy. Vaszil, Variants of P colonies with very simple cell structure. *International Journal of Computers Communication and Control*, **4** (2009), 224–233.
3. L. Cienciala, L. Ciencialová, E. Csuhaj-Varjú, Gy. Vaszil, *PCol automata: Recognizing strings with P colonies.* In: M. A. Martínez del Amor, Gh. Păun, I. Pérez Hurtado, A. Riscos Nuñez (eds.), Eighth Brainstorming Week on Membrane Computing, Sevilla, February 1-5, 2010, Fénix Editora, 2010, 65–76.
4. E. Csuhaj-Varjú, J. Dassow, On cooperating/distributed grammar systems. *Journal of Information Processing and Cybernetics* EIK, **26** (1990), 49–63.
5. E. Csuhaj-Varjú, J. Dassow, J. Kelemen, Gh. Păun, *Grammar Systems – A Grammatical Approach to Distribution and Cooperation.* Gordon and Breach, London, 1994.
6. E. Csuhaj-Varjú, M. Oswald, Gy. Vaszil, P automata. In [17], chapter 6, 144–167.
7. E. Csuhaj-Varjú, Gy. Vaszil, P automata or purely communicating accepting P systems. In: Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron (eds.), *Membrane Computing. International Worskhop, WMC-CdeA 2002, Curtea de Arges, Romania, August 19-23, 2002, Revised Papers.* LNCS 2597, Springer Berlin Heidelberg, 2003, 219–233.
8. E. Csuhaj-Varjú, Gy. Vaszil, P automata with restricted power. *International Journal of Foundations of Computer Science*, **25** (2014), 391–408.
9. P.C. Fischer, Turing machines with restricted memory access. *Information and Control*, **9** (1966), 364–379.
10. R. Freund, M. Kogler, Gh. Păun, M.J. Pérez-Jiménez, On the power of P and dP automata. *Annals of Bucharest University, Mathematics-Informatics Series* **63** (2009), 5-22.
11. K. Kántor and Gy. Vaszil Generalized P Colony Automata *Journal of Automata, Languages and Combinatorics* **19** (2014), 145–156.
12. J. Kelemen, A. Kelemenová, A grammar-theoretic treatment of multiagent systems. *Cybernetics and Systems*, **23** (1992), 621–633.
13. J. Kelemen, A. Kelemenová, Gh. Păun, Preview of P colonies: A biochemically inspired computing model. In: M. Bedau et al. (eds.), *Workshop and Tutorial Proceedings. Ninth International Conference on the Simulation and Synthesis of Living Systems (Alife IX).* Boston Mass., 2004, 82–86.
14. A. Kelemenová, P Colonies. In [17], chapter 23.1, 584–593.
15. M.L. Minsky, *Computation: Finite and Infinite Machines.* Prentice Hall, 1967.
16. A. Păun, Gh. Păun, The power of communication: P systems with symport/antiport. *New Generation Computing* **20**(3) (2002), 295-306.
17. Gh. Păun, G. Rozenberg, A. Salomaa, editors, *The Oxford Handbook of Membrane Computing.* Oxford University Press, 2010.