# Automaton-like P Colonies

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

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

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

**Key words:** String processing; P Colonies; computational power

## 1 Introduction

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

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

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

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

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

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

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

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

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

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

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

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

## 2 Definitions

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

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

A multiset of objects $M$ is a pair $M = (V, f)$, where $V$ is an arbitrary (not necessarily finite) set of objects and $f$ is a mapping $f : V \to N$; $f$ assigns to each object in $V$ its multiplicity in $M$. The set of all multisets with the set of objects $V$ is denoted by $V^\circ$. The set $V'$ is called the support of $M$ and denoted by $supp(M)$. The cardinality of $M$, denoted by $|M|$, is defined by $|M| = \sum_{a \in V} f(a)$. Any multiset of objects $M$ with the set of objects $V' = \{a_1, \ldots a_n\}$ can be represented as a string $w$ over alphabet $V'$ with $|w|_{a_i} = f(a_i);\ 1 \leq i \leq n$. Obviously, all words obtained from $w$ by permuting the letters can also represent the same multiset $M$, and $\varepsilon$ represents the empty multiset.

### 2.1 Register machines and matrix grammars

**Definition 1.** [19] *A register machine is the construct $M = (m, H, l_0, l_h, P)$ where:*

- *$m$ is the number of registers,*
- *$H$ is the set of instruction labels,*
- *$l_0$ is the start label,*
- *$l_h$ is the final label,*
- *$P$ is a finite set of instructions injectively labelled with the elements from the set $H$.*

The instructions of the register machine are of the following forms:

| | |
|---|---|
| $l_1 : (ADD(r), l_2, l_3)$ | Add 1 to the content of the register $r$ and proceed to the instruction (labelled with) $l_2$ or $l_3$. |
| $l_1 : (SUB(r), l_2, l_3)$ | If the register $r$ stores a value different from zero, then subtract 1 from its content and go to instruction $l_2$, otherwise proceed to instruction $l_3$. |
| $l_h : HALT$ | Halt the machine. The final label $l_h$ is only assigned to this instruction. |

Without loss of generality, we may assume that in each *ADD*-instruction $l_1 : (ADD(r), l_2, l_3)$ and in each *SUB*-instruction $l_1 : (SUB(r), l_2, l_3)$ the labels $l_1, l_2, l_3$ are pairwise different.

The register machine $M$ computes a set $N(M)$ of numbers in the following way: it starts with all registers empty (hence storing the number zero) with the instruction labelled by $l_0$ and it proceeds to apply the instructions as indicated by the labels (and made possible by the contents of registers). If it reaches the halt instruction, then the number stored at that time in the register 1 is said to be computed by $M$ and hence it is introduced in $N(M)$. (Because of the non-determinism in choosing the continuation of the computation in the case of *ADD*-instructions, $N(M)$ can be an infinite set.) It is known (see e.g.[19]) that in this way we compute all Turing computable sets.

Moreover, we call a register machine partially blind [13], if we interpret a subtract instruction in the following way: $l_1 : (SUB(r); l_2; l_3)$ - if in register $r$ there is value different from zero, then subtract one from its contents and go to instruction $l_2$ or to instruction $l_3$; if in register $r$ there is stored zero when attempting to decrement register $r$, then the program ends without yielding a result.

When the partially blind register machine reaches the final state, the result obtained in the first register is only taken into account if the remaining registers store value zero. The family of sets of non-negative integers generated by partially blind register machines is denoted by $NRM_{pb}$. Partially blind register machines accept a proper subset of *NRE*, the family of recursively enumerable sets of numbers.

**Definition 2.** *A context-free matrix grammar is a construct*

$$G = (N, T, S, M, F), \text{ where}$$

- $N$ *and* $T$ *are sets of non-terminal and terminal symbols, respectively, with* $N \cap T = \emptyset$,
- $S \in N$ *is the start symbol,*
- $M$ *is a finite set of matrices,* $M = \{m_i \mid 1 \leq i \leq n\}$, *where matrices* $m_i$ *are sequences of the form* $m_i = (m_{i,1}, \ldots, m_{i,n_i}), n_i \geq 1,\ 1 \leq i \leq n$, *and* $m_{i,j},\ 1 \leq j \leq n_i,\ 1 \leq i \leq n$, *are context-free rules over* $(N \cup T)$,
- $F$ *is a subset of all productions occurring in the elements of* $M$, *i.e.* $F \in \{m_{i_j} \mid 1 \leq i \leq n, 1 \leq j \leq n_i\}$.

We say that $x \in (N \cup T)^+$ *directly derives* $y \in (N \cup T)^*$ *in the appearance checking mode* by application of $m_{i,j} = A \to w \in m_i$ – denoted by $x \Rightarrow^{ac} y$, – if one of the following conditions hold:

$$x = x_1 A x_2 \text{ and } y = x_1 w x_2 \quad \text{or} \quad A \text{ does not appear in } x, m_{i,j} \in F \text{ and } x = y.$$

For $m_i = (m_{i,1}, \ldots, m_{i,n_i})$ and $v, w \in (N \cup T)^*$ we define $v \Rightarrow_{m_i} w$ if and only if there are $w_0, w_1, \ldots, w_{n_i} \in (N \cup T)^*$ such that

$$v = w_0 \Rightarrow^{ac}_{m_{i,1}} w_1 \Rightarrow^{ac}_{m_{i,2}} w_2 \Rightarrow^{ac}_{m_{i,3}} \ \ldots \ \Rightarrow^{ac}_{m_{i,n_i}} w_{n_i} = w$$

The language generated by $G$ is

$$L(G) = \{w \in T^* \mid S \Rightarrow_{m_{i_1}} w_1 \cdots \Rightarrow_{m_{i_k}} w_k,\ w_k = w,\\ w_j \in (N \cup T)^*,\ m_{i_j} \in M \text{ for } 1 \leq j \leq k,\ k \geq 1,\ 1 \leq i \leq n\}$$

The family of languages generated by matrix grammars with appearance checking is denoted by $MAT^{\lambda}_{ac}$. The superscript $\lambda$ indicates that erasing rules ($\lambda$-rules) are allowed.

We say that $M$ is a matrix grammar without appearance checking if and only if $F = \emptyset$. The family of languages generated by matrix grammars without appearance checking is denoted by $MAT^{\lambda}$.

The following results are known about matrix languages:

- $CF \subset MAT \subseteq\ MAT^{\lambda}\ \subset RE$
- $MAT \subset MAT_{ac} \subset CS$
- $MAT^{\lambda}\ \subset\ MAT^{\lambda}_{ac}\ =\ CS$, where $CF$, $CS$, $RE$ are the context-free, context-sensitive, and recursively enumerable language classes, respectively
- $NRM_{pb} = NMAT^{\lambda}$, where $NMAT^{\lambda}$ is class of the length sets associated with matrix languages without appearance checking (in [11]).

Further details about matrix grammars can be found in [9].

### 2.2 Automaton-like P colony

In the following we recall the concept of an automaton-like P colony (an APCol system, for short) where the environment of the agents is given in the form of a string [2].

As in the case of standard P colonies, agents of APCol systems contain objects, each being an element of a finite alphabet. With every agent, a set of programs is associated. There are two types of rules in the programs. The first one, called an evolution rule, is of the form $a\ \ \rightarrow b$. It means that object $a$ inside of the agent is rewritten (evolved) to the object $b$. The second type of rules, called a communication rule, is in the form $c \leftrightarrow d$. When this rule is performed, the object $c$ inside the agent and a symbol $d$ in the string are exchanged, so, we can say that the agent rewrites symbol $d$ to symbol $c$ in the input string. If $c = e$, then the agent erases $d$ from the input string and if $d = e$, symbol $c$ is inserted into the string.

An automaton-like P colony works successfully, if it is able to reduce the given string to $\varepsilon$, i.e., to enter a configuration where at least one agent is in accepting state and the processed string is the empty word.

**Definition 3.** *[2] An automaton-like P colony (an APCol system, for short) is a construct*

$$\Pi = (O, e, A_1, \ldots, A_n), \text{ where}$$

- $O$ *is an alphabet; its elements are called the objects,*
- $e \in O$, *called the basic object,*
- $A_i$, $1 \leq i \leq n$, *are agents. Each agent is a triplet* $A_i = (\omega_i, P_i, F_i)$, *where*
  - $\omega_i$ *is a multiset over* $O$, *describing the initial state (content) of the agent,* $|\omega_i| = 2$,
  - $P_i = \{p_{i,1}, \ldots, p_{i,k_i}\}$ *is a finite set of programs associated with the agent, where each program is a pair of rules. Each rule is in one of the following forms:*
    - $a \to b$, *where* $a, b \in O$, *called an evolution rule,*
    - $c \leftrightarrow d$, *where* $c, d \in O$, *called a communication rule,*
  - $F_i \subseteq O^*$ *is a finite set of final states (contents) of agent* $A_i$.

During the work of the APCol system, the agents perform programs. Since both rules in a program can be communication rules, an agent can work with two objects in the string in one step of the computation. In the case of program $\langle a \leftrightarrow b; c \leftrightarrow d\rangle$, a sub-string $bd$ of the input string is replaced by string $ac$. If the program is of the form $\langle c \leftrightarrow d; a \leftrightarrow b\rangle$, then a sub-string $db$ of the input string is replaced by string $ca$. That is, the agent can act only in one place in a computation step and the change of the string depends both on the order of the rules in the program and on the interacting objects. In particular, we have the following types of programs with two communication rules:

- $\langle a \leftrightarrow b; c \leftrightarrow e\rangle$ - $b$ in the string is replaced by $ac$,
- $\langle c \leftrightarrow e; a \leftrightarrow b\rangle$ - $b$ in the string is replaced by $ca$,
- $\langle a \leftrightarrow e; c \leftrightarrow e\rangle$ - $ac$ is inserted in a non-deterministically chosen place in the string,
- $\langle e \leftrightarrow b; e \leftrightarrow d\rangle$ - $bd$ is erased from the string,
- $\langle e \leftrightarrow d; e \leftrightarrow b\rangle$ - $db$ is erased from the string,
- $\langle e \leftrightarrow e; e \leftrightarrow d\rangle$; $\langle e \leftrightarrow e; c \leftrightarrow d\rangle$, ...- these programs can be replaced by programs of type $\langle e \to e; c \leftrightarrow d\rangle$.

The program is said to be *restricted* if it is formed from one rewriting and one communication rule. The APCol system is restricted if all the programs the agents have are restricted.

At the beginning of the work of the APCol system (at the beginning of the computation), there is an input string placed in the environment, more precisely, the environment is given by a string $\omega$ of objects which are different from $e$. This string represents the initial state of the environment. Consequently, an initial configuration of the automaton-like P colony is an $(n+1)$-tuple $c = (\omega; \omega_1, \ldots, \omega_n)$ where $\omega$ is the initial state of the environment and the other $n$ components are multisets of strings of objects, given in the form of strings, the initial states of the agents.

A configuration of an APCoL system $\Pi$ is given by $(w; w_1, \ldots, w_n)$, where $|w_i| = 2,\ 1 \leq i \leq n$, $w_i$ represents all the objects placed inside the $i$-th agent and $w \in (O - \{e\})^*$ is the string to be processed.

At each step of the computation every agent attempts to find one of its programs to use. If the number of applicable programs is higher than one, the agent non-deterministically chooses one of them. At every step of computation, the maximal possible number of agents have to perform a program.

By applying programs, the automaton-like P colony passes from one configuration to another configuration. A sequence of configurations starting from the initial configuration is called a computation. A configuration is halting if the APCol system has no applicable program.

## 3 Accepting and generating mode of computation

The result of a computation depends on the mode in which the APCol system works. In the case of accepting mode, a computation is called accepting if and only if at least one agent is in final state and the string obtained is $\varepsilon$. Hence, the string $\omega$ is accepted by the automaton-like P colony $\Pi$ if there exists a computation by $\Pi$ such that it starts in the initial configuration $(\omega; \omega_1, \ldots, \omega_n)$ and the computation ends by halting in the configuration $(\varepsilon; w_1, \ldots, w_n)$, where at least one of $w_i \in F_i$ for $1 \leq i \leq n$.

The situation is different when the APCol system works in the generating mode. A computation is called successful if only if it is halting and at least one agent is in final state. The string $w_F$ is generated by $\Pi$ iff there exists computation starting in an initial configuration $(\varepsilon; \omega_1, \ldots, \omega_n)$ and the computation ends by halting in the configuration $(w_F; w_1, \ldots, w_n)$, where at least one of $w_i \in F_i$ for $1 \leq i \leq n$.

We denote by $APCol_{acc}R(n)$ (or $APCol_{acc}(n)$) the family of languages accepted by APCol system having at most $n$ agents with restricted programs only (or without this restriction). Similarly we denote by $APCol_{gen}R(n)$ the family of languages generated by APCol systems having at most $n$ agents with restricted programs only.

APCol system $\Pi$ can generate or accept a set of numbers $|L(\Pi)|$.

By $NAPCol_xR(n), x \in \{acc, gen\}$, is denoted the family of sets of natural numbers accepted or generated by APCol systems with at most $n$ agents.

In [2] the authors proved that the family of languages accepted by jumping finite automata (introduced in [18]) is properly included in the family of languages accepted by APCol systems with one agent, and it is proved that any recursively enumerable language can be obtained as a projection of a language accepted by an automaton-like P colony with two agents.

**Theorem 1.** *[2] The family of languages accepted by automaton-like P colonies with one agent properly includes the family of languages accepted by jumping finite automata.*

**Theorem 2.** *[2] Any recursively enumerable language can be obtained as a projection of a language accepted by an automaton-like P colony with two agents.*

## 4 The power of restricted generating APCol systems

In this section we compare the computational power of automaton-like P colonies working in generating mode with one and two agents and that of register machine and matrix grammars with erasing rules. We start with the comparison of APCol systems and register machines.

**Theorem 3.** $NAPCol_{gen}R(2) = NRE$

*Proof.* Let us consider a register machine $M$ with $m$ registers. We construct an APCol system $\Pi = (O, e, A_1, A_2)$ simulating the computations of register machine $M$. To help the easier understanding of the simulation, we provide the components together with some explanations of their role. Let

- $O = \{G\} \cup \{l_i, l_i^I, l_i^{II}, l_i^{III}, l_i^{IV}, l_i^V, l_i^{VI}, \underline{l_i}, \underline{\underline{l_i}}, L_i, L_i', L_i'', F_i \mid l_i \in H\} \cup$
  $\cup \{r \mid 1 \leq r \leq m\}$,
- $A_1 = (ee, P_1, \{eG\})$
- $A_2 = (ee, P_2, \{ee\})$

At the beginning of the computation the first agent generates object $l_0$ (the label of starting instruction of $M$). Then it starts to simulate instruction labelled by $l_0$ and it generates the label of the next instruction. The number stored at the register $r$ corresponds to the number of symbols $r$ placed on the input string. The set of programs is as follows:

(1) For initializing the simulation there is one program in $P_1$:

$P_1$

$1 : \langle e \to l_0; e \leftrightarrow e \rangle$

The initial configuration of $\Pi$ is $(\varepsilon; ee, ee)$. After the first step of computation (only the program 1 is applicable) the system enters configuration $(\varepsilon; l_0 e, ee)$.

(2) For every $ADD$-instruction $l_1 : (ADD(r), l_2, l_3)$ we add to $P_1$ the next programs:

$P_1$

$2 : \langle e \to r; l_1 \leftrightarrow e \rangle$, $3 : \langle e \to a; r \leftrightarrow l_1 \rangle$,
$4 : \langle l_1 \to l_2; a \leftrightarrow e \rangle$, $5 : \langle l_1 \to l_3; a \leftrightarrow e \rangle$

When there is object $l_1$ inside the agent, it generates one copy of $r$, puts it to the environment and generates the label of the next instruction (it non-deterministically chooses one of the last two programs 4 and 5)

| | configuration of $\Pi$ | | | labels of applicable programs | |
|---|---|---|---|---|---|
| | $A_1$ | $A_2$ | string | $P_1$ | $P_2$ |
| 1. | $l_1 e$ | $ee$ | $r^x$ | 2 | – |
| 2. | $re$ | $ee$ | $l_1 r^x$ | 3 | – |
| 3. | $al_1$ | $ee$ | $r^{x+1}$ | **4** *or* 5 | – |
| 4. | $l_2 e$ | $ee$ | $r^{x+1}a$ | | |

(3) For every $SUB$-instruction $l_1 : (SUB(r), l_2, l_3)$, the next programs are added to sets $P_1$ and $P_2$:

| $P_1$ : | | $P_2$ : |
|---|---|---|
| $6 : \langle l_1 \to l_1^I; e \leftrightarrow e \rangle$ | $12 : \langle l_1^{VI} \to \underline{l_2}; e \leftrightarrow L_1'' \rangle$ | $19 : \langle e \to L_1; e \leftrightarrow l_1^I \rangle$ |
| $7 : \langle e \to l_1^{II}; l_1^I \leftrightarrow e \rangle$ | $13 : \langle L_1'' \to l_2; \underline{l_2} \leftrightarrow e \rangle$ | $20 : \langle l_1^I \to L_1'; L_1 \leftrightarrow l_1^{II} \rangle$ |
| $8 : \langle e \to l_1^{III}; l_1^{II} \leftrightarrow e \rangle$ | $14 : \langle l_1^{VI} \to \underline{l_3}; e \leftrightarrow L_1 \rangle$ | $21 : \langle l_1^{II} \to L_1''; L_1' \leftrightarrow r \rangle$ |
| $9 : \langle l_1^{III} \to l_1^{IV}; e \leftrightarrow e \rangle$ | $15 : \langle L_1 \to F_3; \underline{l_3} \leftrightarrow e \rangle$ | $22 : \langle r \to e; L_1'' \leftrightarrow L_1 \rangle$ |
| $10 : \langle l_1^{IV} \to l_1^V; e \leftrightarrow e \rangle$ | $16 : \left\langle e \to \underline{\underline{l_3}}; F_3 \leftrightarrow \underline{l_3} \right\rangle$ | $23 : \langle L_1 \to e; e \leftrightarrow e \rangle$ |
| $11 : \langle l_1^V \to l_1^{VI}; e \leftrightarrow e \rangle$ | $17 : \left\langle \underline{l_3} \to F_3'; \underline{\underline{l_3}} \leftrightarrow e \right\rangle$ | $24 : \langle l_1^{II} \to e; L_1' \leftrightarrow F_3 \rangle$ |
| | $18 : \langle F_3' \to l_3; e \leftrightarrow e \rangle$ | $25 : \langle F_3 \to e; e \leftrightarrow e \rangle$ |

At the first phase of the simulation of the $SUB$ instruction the first agent generates object $l_1'$, which is consumed by the second agent. The agent $A_2$ generates symbol $L_1$ and tries to consume one copy of symbol $r$. If there is any $r$, the agent sends to the environment object $L_1''$ and consumes $L_1$. After this step the first agent consumes $L_1''$ or $L_1$ and rewrites it to $l_2$ or $l_3$.

Instruction $l_1 : (SUB(r), l_2, l_3)$ is simulated by the following sequence of steps.

If the register $r$ stores non-zero value:

| | configuration of $\Pi$ | | | labels of applicable programs | |
|---|---|---|---|---|---|
| | $A_1$ | $A_2$ | string | $P_1$ | $P_2$ |
| 1. | $l_1 e$ | $ee$ | $r^x$ | 6 | – |
| 2. | $l_1^I e$ | $ee$ | $r^x$ | 7 | – |
| 3. | $l_1^{II} e$ | $ee$ | $l_1^I r^x$ | 8 | 19 |
| 4. | $l_1^{III} e$ | $L_1 l_1^I$ | $l_1^{II} r^x$ | 9 | 20 |
| 5. | $l_1^{IV} e$ | $L_1' l_1^{II}$ | $L_1 r^x$ | 10 | 21 |
| 6. | $l_1^V e$ | $L_1'' r$ | $L_1 L_1' r^{x-1}$ | 11 | 22 |
| 7. | $l_1^{VI} e$ | $e L_1$ | $L_1'' r^{x-1}$ | 12 | 23 |
| 8. | $\underline{l_2} L_1''$ | $ee$ | $r^{x-1}$ | 14 | – |
| 9. | $l_2 e$ | $ee$ | $r^{x-1} \underline{l_2}$ | | |

If the register $r$ stores value zero :

| | configuration of $\Pi$ | | | labels of applicable programs | |
|---|---|---|---|---|---|
| | $A_1$ | $A_2$ | string | $P_1$ | $P_2$ |
| 1. | $l_1 e$ | $ee$ | | 6 | – |
| 2. | $l_1^I e$ | $ee$ | | 7 | – |
| 3. | $l_1^{II} e$ | $ee$ | $l_1^I$ | 8 | 19 |
| 4. | $l_1^{III} e$ | $L_1 l_1^I$ | $l_1^{II}$ | 9 | 20 |
| 5. | $l_1^{IV} e$ | $L_1' l_1^{II}$ | $L_1$ | 10 | – |
| 6. | $l_1^V e$ | $L_1' l_1^{II}$ | $L_1$ | 11 | – |
| 7. | $l_1^{VI} e$ | $L_1' l_1^{II}$ | $L_1$ | 13 | – |
| 8. | $\underline{l_3} L_1$ | $L_1' l_1^{II}$ | | 15 | – |
| 9. | $F_3 e$ | $L_1' l_1^{II}$ | $\underline{l_3}$ | 16 | – |
| 10. | $\underline{\underline{l_3}}\ \underline{l_3}$ | $L_1' l_1^{II}$ | $F_3$ | 17 | 24 |
| 11. | $F_3' e$ | $F_3 e$ | $\underline{l_3} L_1'$ | 18 | 25 |
| 12. | $l_3 e$ | $ee$ | $\underline{\underline{l_3}} L_1'$ | | |

(4) For halting instruction $l_h$ there are programs belonging to the sets $P_1$ and $P_2$. After that agent $A_1$ generates the object $l_h$, it writes the symbol $G$ to the tape. After consuming it, the second agent erases all the symbols from the tape except these ones which correspond to the first register of the register machine.

| $P_1$ : | $P_2$ : |
|---|---|
| $26 : \langle e \to G; l_h \leftrightarrow e \rangle$ $27 : \langle e \to e; e \leftrightarrow l_h \rangle$ | $28 : \langle e \to e; l_h \leftrightarrow X \rangle$ |
| | $29 : \langle X \to e; e \leftrightarrow l_h \rangle$ |

$X \in \{a\} \cup \{L'_i, \underline{l_i}, \underline{\underline{l_i}} \mid 0 \leq i \leq p, \text{ where } |H| = p\} \cup \{r \mid 1 < r \leq m\}$

The APCol system $\Pi$ starts computation in the initial configuration with empty tape. It starts the simulation of register machine $M$ with instruction labelled by $l_0$ and it proceeds the simulation according to the instructions of the register machine. After $M$ reaches the halting instruction, then agent $A_2$ in the APCol system $\Pi$ erases from the tape all the symbols except symbols 1 and then APCol systems halts. So the length of the word placed on the tape in the last configuration corresponds to the number stored in the first register of $M$ at the end of its computation.

We proved that the family of length sets of languages generated by restricted APCol systems with two agents equals to *NRE*. If the APCol system is formed from only one agent, there are some limitation for generated languages. In the following we show the limits of the computational power of restricted APCol systems with only one agent.

**Theorem 4.** $NRM_{PB} \subseteq NAPCol_{gen}R(1)$

*Proof.* Let us consider a partially blind register machine $M$ with $m$ registers. We construct an APCol system $\Pi = (O, e, A)$ simulating the computations of register machine $M$ with:

- $O = \{G\} \cup \{l_i, l_i^I, l_i^{II}, l_i^{III}, l_i^{IV}, l_i^V, l_i^{VI}, \underline{l_i}, \underline{\underline{l_i}}, L_i, L'_i, L''_i, F_i \mid l_i \in H\} \cup$
  $\cup \{r \mid 1 \leq r \leq m\}$,
- $A = (ee, P, \{eG\})$

The functioning of the constructed APCol system is very similar to the one from the proof of previous theorem. At the beginning of the computation, the agent generates the object $l_0$ (the label of starting instruction of $M$). Then it starts to simulate instruction labelled by $l_0$ and generates the label of the next instruction. The number stored at register $r$ corresponds to the number of symbols $r$ placed on the input string. The set of programs is given as follows:

(1) For initializing the simulation there is one program in $P_1$:

| $P$ : |
|---|
| $1 : \langle e \to l_0; e \leftrightarrow e \rangle$ |

The initial configuration of $\Pi$ is $(\varepsilon; ee)$. After the first step of computation (only the program 1 is applicable) the system enters configuration $(\varepsilon; l_0 e)$.

(2) For every *ADD*-instruction $l_1 : (ADD(r), l_2, l_3)$ we add to $P$ the next programs:

| $P$ | |
|---|---|
| $2 : \langle e \to r; l_1 \leftrightarrow e \rangle$, | $3 : \langle e \to a; r \leftrightarrow l_1 \rangle$, |
| $4 : \langle l_1 \to l_2; a \leftrightarrow e \rangle$, | $5 : \langle l_1 \to l_3; a \leftrightarrow e \rangle$ |

When there is object $l_1$ inside the agent, it generates one copy of $r$, puts it to the environment and generates the label of the next instruction (it non-deterministically chooses one of two programs 4 and 5)

| | configuration of $\Pi$ | | labels of applicable programs |
|---|---|---|---|
| | $A$ | string | $P$ |
| 1. | $l_1 e$ | $r^x$ | 2 |
| 2. | $re$ | $l_1 r^x$ | 3 |
| 3. | $al_1$ | $r^{x+1}$ | **4** *or* 5 |
| 4. | $l_2 e$ | $r^{x+1}a$ | |

(3) For every *SUB*-instruction $l_1 : (SUB(r), l_2, l_3)$, the next programs are added to set $P$:

$P$ :

$6 : \langle l_1 \to l_1^I; e \leftrightarrow r \rangle$ $7 : \langle r \to l_2; l_1^I \leftrightarrow e \rangle$ $8 : \langle r \to l_3; l_1^I \leftrightarrow e \rangle$

The agent generates object $l_1'$ even if there is at least one object $r$ in the environment. Then it rewrites $r$ to $l_2$ or $l_3$. If there is no $r$ in the environment, the computation halts and the agent is in non-final state.

(4) For halting instruction $l_h$ there are programs belonging to the set $P$:

$P$ :

$9 : \langle l_h \to G; e \leftrightarrow e \rangle$ $10 : \langle e \to e; G \leftrightarrow X \rangle$ $11 : \langle X \to e; e \leftrightarrow G \rangle$
$X \in \{a\} \cup \{l_i^I \mid 0 \leq i \leq p, \text{ where } |H| = p\} \cup \{r \mid 1 < r \leq m\}$

After the object $l_h$ appears inside agent $A$, it generates the symbol $G$. Using $G$, the agent erases all the symbols from the tape except those ones which correspond to the first register of the register machine.

The APCol system $\Pi$ starts computation in the initial configuration with empty tape. It starts the simulation of the partially blind register machine $M$ with instruction labelled by $l_0$ and it proceeds the simulation according to the instructions of register machine. After $M$ reaches the halting instruction, the agent $A$ in the APCol system $\Pi$ erases from the tape all the symbols except symbols 1 and then the APCol system halts. So the length of the word placed on the tape in the last configuration corresponds to the number stored in the first register of $M$ at the end of its computation.

In the following we will examine the language generating power of restricted APCoL systems.

**Theorem 5.** $APCol_{gen}R(1) \subseteq MAT^{\lambda}$

*Proof.* Let $\Pi = (O, e, A)$ be a restricted APCol system with one agent. We construct a matrix grammar $G = (N, T, S, M)$ simulating $\Pi$ as follows:

The symbol on the tape $a \neq e$ is represented by $\boxed{a} \in N$ and at the end of simulation it can be rewritten to $a \in T$. The content of the agent is represented

by a non-terminal symbol $\boxed{AB} \in N$, where $a, b \in O$ are objects placed inside the agent. As in the previous cases, we provide only the necessary details.

The first applied matrix is $\left(S \to C\ \boxed{EE}\right)$, representing the initial content of the agent.

For every program of type $\langle a \to b; c \leftrightarrow d\rangle\,,\ c, d \neq e$ there is a matrix in $M$:

$$\left(C \to C, \boxed{AC} \to \boxed{BD}, \boxed{d} \to \boxed{c}\right)$$

For every program of type $\langle a \to b; e \leftrightarrow d\rangle\,,\ d \neq e$ there is a matrix in $M$:

$$\left(C \to C, \boxed{AC} \to \boxed{BD}, \boxed{d} \to \varepsilon\right)$$

For every program of type $\langle a \to b; e \leftrightarrow e\rangle$ there is a matrix in $M$:

$$\left(C \to C, \boxed{AE} \to \boxed{BE}\right)$$

For every program of the type $\langle a \to b; c \leftrightarrow e\rangle\,,\ c \neq e$ there is a matrix in $M$:

$$\left(C \to C, \boxed{AC} \to \boxed{BE}\,\textcircled{c}\right)$$

and a set of matrices generating $\boxed{c}$ somewhere in the string and deleting $\textcircled{c}$:

$$\left(C \to C, \boxed{x} \to \boxed{x}\,\boxed{c}, \textcircled{c} \to \varepsilon\right),$$
$$\left(C \to C, \boxed{x} \to \boxed{c}\,\boxed{x}, \textcircled{c} \to \varepsilon\right),$$

for all $\boxed{x}$ such that $x \in T$.

When the APCol system reaches the halting configuration, the matrix grammar generates the corresponding string. The string is formed from only non-terminals. The matrix grammar has to rewrite the rammed terminal symbols to terminals and to delete non-terminal representing the content of the agent and the non-terminal $C$. The halting configuration can be presented by a string $AB \cdot w$, where $|w|_a = 1$ for all $a \in T$ such that $a$ is present in this halting configuration and $AB$ is content of the agent such that $ab \in F$. The set of such a representations is finite.

For each representation $AB \cdot a_1 a_2 \ldots a_p,\ p \leq |T|$, we add the following matrices to the matrix grammar:

$$\left(C \to [\ \boxed{AB} a_1 a_2 \ldots a_p]\right)$$
$$\left([\ \boxed{AB} a_1 a_2 \ldots a_q] \to [\ \boxed{AB} a_1 a_2 \ldots a_q], \boxed{a_q} \to a_q\right),$$
$$\left([\ \boxed{AB} a_1 a_2 \ldots a_q] \to [\ \boxed{AB} a_1 a_2 \ldots a_{q-1}]\right), 1 < q \leq p,$$
$$\left([\ \boxed{AB} a_1] \to [\ \boxed{AB} a_1], \boxed{a_1} \to a_1\right),\ \left([\ \boxed{AB} a_1] \to [\ \boxed{AB}\ ]\right)$$
$$\left([\ \boxed{AB}\ ] \to \varepsilon, \boxed{AB} \to \varepsilon\right)$$

In this way all non-terminal symbols are rewritten to the corresponding terminals and the non-terminal symbol corresponding to the contents of the agent is deleted.

If the restricted APCol system $\Pi$ generates a string $\omega$, then the matrix grammar is able to generate it, too. If the APCol system halts and the agent is not in final state, then the matrix grammar cannot generate a string consisting only of terminals.

The last theorem is devoted to the relationship of $MAT^{\lambda}_{ac}$ to restricted APCol systems with two agents.

**Theorem 6.** $APCol_{gen}R(2) \subseteq MAT^{\lambda}_{ac}$

*Proof.* Let $\Pi = (O, e, A_1, A_2)$ be the restricted APCol system with two agents. We construct a matrix grammar $G = (N, T, S, M)$ simulating $\Pi$ as follows:

The symbol on the tape $a \neq e$ is represented by $\boxed{a} \in N$ and at the end of the simulation it is rewritten to $a \in T$. The contents of the agent $A_1$ is represented by a non-terminal symbol $\boxed{AB} \in N$, where $a, b \in O$ are the objects placed inside the first agent. The contents of the agent $A_2$ is represented by a non-terminal symbol $(AB) \in N$, where $a, b \in O$ are the objects placed inside the second agent.

We add label $p_i$ from the set of labels $P$ to every program associated with both agents. Let $P \cap (N \cup T) = \emptyset$. We add a set of new non-terminals $\{P_i \mid p_i \in P\}$ to the set $N$ . To control the derivation, we add non-terminals $S$, $C$, $C_P$ and $\#$.

The first applied matrix is $\big(S \to C\boxed{EE}(EE)\big)$, representing the initial states of the agents.

For every possible pair of states of agents, we add the following types of matrices to $M$. Let $ab$ be a state of the agent $A_1$. In this state there are three applicable types of programs: $\langle a \to c; b \leftrightarrow d\rangle$ – (A), $\langle a \to c; b \leftrightarrow e\rangle$ – (B). The first type includes the case where $b = e$ and $\langle a \to c; e \leftrightarrow d\rangle$ – (A').

We divide the execution of such a program into three phases. The first phase is to choose two programs to apply. The corresponding matrices (forming the set $M_1 \subset M$) are of the form

$$\big(C \to C, \boxed{AB} \to \boxed{ABv}, (XY) \to (XYz)\big)\,, \tag{1}$$

where $v$ and $z$ depend on the type of the used programs. If the program is of the type $\langle a \to c; b \leftrightarrow d\rangle$ or $\langle a \to c; e \leftrightarrow d\rangle$, $v$ (or $z$) is $d$ and it means that $d$ will be erased in following derivation steps. If the program is of the type $\langle a \to c; b \leftrightarrow e\rangle$, $v$ (or $z$) is $\sqcup$ and it means that there is nothing to erase from string.

Because of maximal parallelism in the computation, an agent can be in a state when there is no program to use. For this situation, we construct another set of matrices. We choose from the set $M_1$ all matrices corresponding to the application of a program of the type $\langle a \to c; b \leftrightarrow d\rangle$ by the "sleeping agent". We need not to include matrices for the programs of type $\langle a \to c; b \leftrightarrow e\rangle$ because these programs in state $ab$ are always applicable. To the selected matrices of the type (1), we add the following matrices:

$$\big(C \to C_P, \boxed{AB} \to \boxed{ABN}P_i, (XY) \to (XYz)\big)\,, \tag{2}$$

for the first agent with no applicable program and

$$\left(C \to C_P, \boxed{AB} \to \boxed{ABv}, \textcircled{XY} \to \textcircled{XYN} P_j\right), \tag{3}$$

for the second agent with no applicable program and $p_i \in P$, resp. $p_j \in P$ is the label of program corresponding to triplet $\boxed{ABv}$ resp. to $\textcircled{XYz}$.

To check whether in the given state the agent has no applicable program, we perform the following construction. Let $P_i'$ be the set of all applicable programs of the type (A) in the state $ab$ of the agent. Then matrices

$$(C_P \to C, P_i \to \varepsilon, d_1 \to \#, d_2 \to \#, \ldots, d_k \to \#), \tag{4}$$

are for checking the inactivity of the agent. $d_1, \ldots, d_k \in N, k = |P_i'|$ are non-terminals corresponding to the objects that the agent needs to consume from the string applying any program from $P_i'$. All the rules of the type $d_l \to \#, 1 \leq l \leq k$ are in the set $F$. If $\#$ appears in the string, then the derivation cannot end with a terminal word.

The last phase of the simulation of execution of programs in the APCol system $\Pi$ corresponds to the change of the state of the agents and to the changes changes in the string. We add to $M$ the following matrices: For the pair of programs $(\langle a \to c; b \leftrightarrow d\rangle, \langle x \to u; y \leftrightarrow z\rangle)$

$$\left(C \to C, \boxed{ABd} \to \boxed{CD}, \boxed{d} \to \boxed{b}, \textcircled{XYz} \to \textcircled{UZ}, \boxed{z} \to \boxed{y}\right) . \tag{5}$$

For the pair of programs $(\langle a \to c; e \leftrightarrow d\rangle, \langle x \to u; y \leftrightarrow z\rangle)$

$$\left(C \to C, \boxed{ABd} \to \boxed{CD}, \boxed{d} \to \varepsilon, \textcircled{XYz} \to \textcircled{UZ}, \boxed{z} \to \boxed{y}\right) \tag{6}$$

For the pair of programs $(\langle a \to c; b \leftrightarrow e\rangle, \langle x \to u; y \leftrightarrow z\rangle)$

$$\left(C \to C, \boxed{AB\sqcup} \to \boxed{CD}\textcircled{b}, \textcircled{XYz} \to \textcircled{UZ}, \boxed{z} \to \boxed{y}\right) \tag{7}$$

For the remaining program combinations we add another six types of matrices.

We also add the set of matrices for generating $\boxed{c}$ somewhere in the string and for deleting $\textcircled{c}$:

$$\left(C \to C, \boxed{x} \to \boxed{x}\,\boxed{c}, \textcircled{c} \to \varepsilon\right),$$
$$\left(C \to C, \boxed{x} \to \boxed{c}\,\boxed{x}, \textcircled{c} \to \varepsilon\right), \text{ for all } q, \boxed{x} \text{ such that } x \in T.$$

When the APCol system reaches the halting configuration, the matrix grammar generates the corresponding string. The string is formed from non-terminals only. The matrix grammar has to rewrite the rammed terminal symbols to terminals and to delete the non-terminals representing the contents of the agents and non-terminal $C$. The halting configuration can be represented by a string $AB \cdot XY \cdot w$, where $|w|_a = 1$ for all $a \in T$ such that $a$ is present in this halting configuration and $AB, XY$ are the contents of the agents such that $ab \in F_1$ and $xy \in F_2$. The set of such a representations is finite.

For each representation $AB \cdot XY \cdot a_1a_2 \dots a_p,\ p \leq |T|$, we add the following matrices to the matrix grammar:

$$\begin{gathered}
\left(C \to [\ \boxed{AB}\,(XY)\,a_1a_2 \dots a_p]\right) \\
\left([\ \boxed{AB}\,(XY)\,a_1a_2 \dots a_q] \to [\ \boxed{AB}\,(XY)\,a_1a_2 \dots a_q], \boxed{a_q} \to a_q\right), \\
\left([\ \boxed{AB}\,(XY)\,a_1a_2 \dots a_q] \to [\ \boxed{AB}\,(XY)\,a_1a_2 \dots a_{q-1}]\right), 1 < q \leq p \\
\left([\ \boxed{AB}\,(XY)\,a_1] \to [\ \boxed{AB}\,(XY)\,a_1], \boxed{a_1} \to a_1\right),\ \left([\ \boxed{AB}\,(XY)\,a_1] \to [\ \boxed{AB}\,(XY)\ ]\right) \\
\left([\ \boxed{AB}\,(XY)\ ] \to \varepsilon, \boxed{AB} \to \varepsilon, (XY) \to \varepsilon\right)
\end{gathered}$$

In this way all non-terminal symbols are rewritten to the corresponding terminals and the non-terminals corresponding to the contents of the agents are deleted.

If the restricted APCol system $\Pi$ generates the string $\omega$, then the matrix grammar can generate it, too. If the APCol system halts and the agent is not in final state, then the matrix grammar cannot generate a string consisting of only terminals.

## 5 Conclusions

We developed the concept of automaton-like P colonies (APCol systems) - variants of P colonies that work on a string. We introduced the generating mode of computation of these systems and compared the generative and computational power of automaton-like P colonies and the generative power of context-free matrix grammars with and without appearance checking and the computational power of variants of register machines. The results of this paper can be summarized as follows:

- $NRM_{PB} \subseteq NAPCol_{gen}R(1)$
- $APCol_{gen}R(1) \subseteq MAT^{\lambda}$
- $NAPCol_{gen}R(2) = NRE$
- $APCol_{gen}R(2) \subseteq MAT^{\lambda}_{ac}$

*Remark 1.* This work was partially supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/-02.0070), by SGS/24/2013 and by project OPVK no. CZ.1.07/2.2.00/28.0014.

## References

1. Ciencialová, L., Cienciala, L.: Variation on the theme: P colonies. In: Kolář, D., Meduna, A. (eds.) Proc. 1st Intern. Workshop on Formal Models, pp. 27–34. Ostrava (2006)

2. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E.: P Colonies Processing Strings. Fundam. Inform. 134(1-2), 51–65 (2014)
3. Ciencialová, L., Csuhaj-Varjú, E., Kelemenová, A., Vaszil, Gy.: Variants of P colonies with very simple cell structure. International Journal of Computers, Communication and Control 4(3), 224–233 (2009)
4. Ciencala, L., Ciencialová, L., Csuhaj-Varjú, E., Vaszil, Gy.: PCol Automata: Recognizing strings with P colonies. In: Martinez-del-Amor, M. A. et al. (eds.) Proc. BWMC 2010, pp. 65–76. Fénix Editora, Sevilla (2010)
5. Cienciala, L., Ciencialová, L., Kelemenová, A.: Homogeneous P colonies. Computing and Informatics 27, 481–496 (2008)
6. Cienciala, L., Ciencialová, L., Kelemenová, A.: On the number of agents in P colonies. In: Eleftherakis,G. et. al (eds.) Membrane Computing, 8th International Workshop, WMC 2007, Thessaloniki, Greece, June 25-28, 2007 Revised Selected and Invited Papers. LNCS, vol. 4860, pp. 193–208. Springer (2007)
7. Csuhaj-Varjú, E., Kelemen, J., Kelemenová, A., Păun, Gh., Vaszil, Gy.: Computing with cells in environment: P colonies. Journal of Multi-Valued Logic and Soft Computing 12, 201–215 (2006)
8. Csuhaj-Varjú, E.,Margenstern, M., Vaszil, Gy.: P colonies with a bounded number of cells and programs. In: Hoogeboom, H-J. et. al (eds.) Membrane Computing, 7th International Workshop, WMC 2006, Leiden, The Netherlands, July 17-21, 2006, Revised, Selected, and Invited Papers. LNCS, vol. 4361, pp. 352–366. Springer(2006)
9. Dassow, J., Păun, Gh.: Regulated Rewriting in Formal Language Theory. EATCS Monographs in Theoretical Computer Science 18. Springer-Verlag Berlin (1989)
10. Fischer, P. C.: Turing machines with restricted memory access. Information and Control 9, 364–379 (1966)
11. Freund, R., Oswald, M.: P colonies working in the maximally parallel and in the sequential mode. In: Ciobanu, G., Păun, Gh. (eds.) Pre-Proc. 1st Intern. Workshop on Theory and Application of P Systems, pp. 49–56. Timisoara, Romania (2005)
12. Freund, R., Oswald, M.: P colonies and prescribed teams. International Journal of Computer Mathematics 83, 569–592 (2006)
13. Greibach, S. A.: Remarks on blind and partially blind one-way multicounter machines. Theoretical Computer Science 7(1), 311–324 (1978)
14. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading, Mass. (1979)
15. Kelemen, J., Kelemenová, A.: A grammar-theoretic treatment of multi-agent systems. Cybernetics and Systems 23, 621–633 (1992)
16. Kelemen, J., Kelemenová, A., Păun, Gh.: Preview of P colonies: A biochemically inspired computing model. In: Bedau, M. et al. (eds.) Workshop and Tutorial Proceedings. Ninth International Conference on the Simulation and Synthesis of Living Systems (Alife IX), pp. 82–86. Boston Mass. (2004)
17. Kelemenová, A.: P Colonies. Chapter 23.1, In: Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) The Oxford Handbook of Membrane Computing, pp. 584–593. Oxford University Press (2010)
18. Meduna, A., Zemek, P.: Jumping Finite Automata. Int. J. Found. Comput. Sci. 23, 1555–1578 (2012)
19. Minsky, M.: Computation – Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, NJ (1967)
20. Păun, Gh., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press (2010)