Int. J. of Computers, Communications & Control, ISSN 1841-9836, E-ISSN 1841-9844
Vol. IV (2009), No. 3, pp. 224-233

Variants of P Colonies with Very Simple Cell Structure

Lucie Ciencialová, Erzsébet Csuhaj-Varjú, Alica Kelemenová, György Vaszil

Lucie Ciencialová, Alica Kelemenová
Institute of Computer Science, Faculty of Philosophy and Science, Silesian University in Opava
Bezručovo nám. 13, 74601 Opava, Czech Republic
E-mail: {lucie.ciencialova, alica.kelemenova}@fpf.slu.cz

Erzsébet Csuhaj-Varjú, György Vaszil
Computer and Automation Research Institute of the Hungarian Academy of Sciences
Kende utca 13-17, 1111 Budapest, Hungary
E-mail: {csuhaj, vaszil}@sztaki.hu

Erzsébet Csuhaj-Varjú
Department of Algorithms and Their Applications, Faculty of Informatics, Eötvös Loránd University
Pázmány Péter sétány 1/c, 1117 Budapest, Hungary

Alica Kelemenová
Department of Computer Science, Catholic University Ružomberok
Nám. A. Hlinku 56, 03401 Ružomberok, Slovakia

Received: April 5, 2009
Accepted: May 30, 2009

Abstract: We study two very simple variants of P colonies: systems with only one object inside the cells, and systems with insertion-deletion programs, so called P colonies with senders and consumers. We show that both of these extremely simple types of systems are able to compute any recursively enumerable set of vectors of non-negative integers.
Keywords: P systems, colonies, P colonies, register machines

1 Introduction

P colonies form a class of abstract computing devices modeling a community of simple agents acting and evolving in a shared environment. They were introduced in [5] as very simple membrane systems, similar in simplicity and architecture to so called colonies of formal grammars. (See [7] for more information on membrane systems and [2, 4] for details on grammar systems theory.)

A P colony consists of a collection of cells, each having a number of objects inside and an associated set of rules through which it can process these objects. Communication between the cells is only possible indirectly through the environment which is common to all of them.

The capabilities of the computing agents are very restricted, and the number of objects present inside a cell during the functioning of the system is previously fixed: it is usually one, two or three. The rules are also of a very simple form. As we will see, they allow the transformation of objects inside the cells and the transportation of objects between the cells and the environment. The rules are grouped into programs. A program contains exactly as many rules, as the number of objects allowed to be present inside the cell. The rules of the programs are applied to the objects inside the associated cells in parallel, and this also affects the objects which are in the environment.

The P colony executes a computation by synchronously applying the programs to the objects inside the cells and outside in the environment until a halting configuration is reached. The result of the computation is obtained as the vector of copies of certain "final" objects present in the environment after the system halts.

Research supported in part by the Czech Science Foundation, "GAČR", project no. 201/09/P075 (L. Ciencialová), by the Hungarian Scientific Research Fund, "OTKA", project no. K75952 (E. Csuhaj-Varjú, Gy. Vaszil), and by the Scientific Grant Agency of Slovakia, "VEGA", project

In the following, after providing the formal definitions, we first give a short overview of results on the computational completeness of the different P colony variants. Then we present new results about two types of systems: first about the simplest possible P colonies, those which only have one object inside every cell, and then about a new type called P colonies with senders and consumers, which have special rules for insertion-deletion. We show that both kinds of these very simple devices are able to compute any recursively enumerable set of vectors of non-negative integers.

2 Preliminaries

Let V be an alphabet, let V^* be the set of all words over V, and let ε denote the empty word. We denote the number of occurrences of a symbol $a \in V$ in w by $|w|_a$. The set of non-negative integers is denoted by $\mathbb{N}$.

A multiset over an arbitrary (not necessarily finite) 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.

We will also need the notion of a register machine which consists of a finite number of registers each of which can hold an arbitrarily large non-negative integer (we say that the register is empty if it holds zero), and a set of labeled instructions which specify how the numbers stored in the registers can be changed.

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

- $l_i : (ADD(r), l_j, l_k)$ - *nondeterministic add*: Add one to register r and then go to one of the instructions with labels l_j or l_k, non-deterministically chosen.
- $l_i : (SUB(r), l_j, l_k)$ - *subtract*: If register r is non-empty, then subtract one from it and go to the instruction with label l_j, if the value of register r is zero, go to instruction l_k.
- $l_h : HALT$ - *halt*: Stop the machine.

A register machine M computes a set $N(M)$ of numbers in the following way: It starts with empty registers by executing the instruction with label l_0 and proceeds by applying instructions as indicated by the labels (and made possible by the contents of the registers). If the halt instruction is reached, then the number stored at that time in register 1 is said to be computed by 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., [6]) that in this way we can compute all sets of numbers which are Turing computable.

If a set of output registers $i_1, \ldots, i_r$, $1 \leq r \leq m$, $i_j \in \{1, \ldots, m\}$ is specified, then M computes a set of vectors of non-negative integers as follows. If the halt instruction is reached, then $(v_1, \ldots, v_r)$, where v_k is the number stored in register i_k, $1 \leq k \leq r$, is the vector of numbers computed by M, i.e., the result of that computation.

Now we recall the definition of a P colony from [5]. A *P colony* is a construct $\Pi = (V, e, F, C_1, \ldots, C_n)$, $n \geq 1$, where V is an alphabet (its elements are called *objects*). There are two kinds of distinguished objects: $e \in V$ (the environmental object), and the objects in $F \subseteq V$ (the set of final objects). The *cells* of the colony are denoted by $C_1, \ldots, C_n$. Each cell is a pair $C_i = (O_i, P_i)$, where O_i is a multiset over $\{e\}$

having the same cardinality *called capacity* (here we only consider $|O_i| \in \{1,2\}$) for all i, $1 \leq i \leq n$ (the initial state of the cell), and P_i is a finite set of *programs*. Each program consists of rules of the following forms:

- $a \to b$ (internal point mutation), specifying that an object $a \in V$ inside the cell is changed to $b \in V$.
- $c \longrightarrow d$ (one object exchange with the environment), specifying that if $c \in V$ is contained inside the cell and $d \in V$ is present in the environment, then c is sent out of the cell while d is brought inside.
- $c \longrightarrow d/c \longrightarrow d'$ (checking rule for one object exchange with the environment), specifying that if $c \in V$ is inside the cell then it is exchanged with $d \in V$ from the environment, or if there is no d outside but $d' \in V$ is present, then c is exchanged with d'.
- $c \longrightarrow d/c \to d'$ (checking rule for one object exchange with the environment or internal point mutation), specifying that if the exchange of $c \in V$ inside and $d \in V$ outside is not possible, then c is changed to $d' \in V$.

The programs contain one rule for each element of O_i, thus, the number of rules of a program coincides with the cardinality of O_i, $1 \leq i \leq n$.

In addition, P colonies with capacity of two may have programs of the form

- $\langle a, in; bc \to d \rangle$ with $a,b,c,d \in V$ (deletion programs), specifying that if bc is present inside the cell and a is present in the environment, then the objects inside are changed to d and a is brought in (a is "deleted" from the environment).
- $\langle a, out; b \to cd \rangle$ with $a,b,c,d \in V$ (insertion programs), specifying that if ab is inside the cell, then a is sent out (a is "inserted" into the environment) and b is changed to cd.

The programs of the cells are used in the non-deterministic maximally parallel manner: in each time unit, each cell which is able to use one of its programs should use one. The use of a program means the application of the rule(s) of the program to the object(s) in the cell.

This way, transitions among the configurations of the colony are obtained. A sequence of transitions is a *computation* which is halting if it reaches a configuration where no cell can use any program. The result of a halting computation is obtained from the number of copies of objects from F present in the environment in the halting configuration. Because of the non-determinism in choosing the programs, several computations can be obtained from a given initial configuration, hence with a P colony Π we can associate a set of vectors of non-negative integers computed by all possible halting computations of Π.

Initially, the environment contains arbitrarily many copies of the environmental object e, and the cells also contain one or two copies of e inside, depending on the capacity of the P colony.

For a P colony $\Pi = (V, e, F, C_1, \ldots, C_n)$ as above, a configuration can be formally written as an $(n+1)$-tuple

$$(w_1, \ldots, w_n; w_E),$$

where $w_i \in V^*$ represents the multiset of objects from cell C_i, $1 \leq i \leq n$, and $w_E \in (V - \{e\})^*$ represents the multiset of objects from the environment different from the environmental object e. The initial configuration is $(e^i, \ldots, e^i; \varepsilon)$ where $i \in \{1,2\}$ is the capacity of the cells.

A transition from a configuration to another is denoted as

$$(w_1, \ldots, w_n; w_E) \Rightarrow (w'_1, \ldots, w'_n; w'_E)$$

where w'_E and each w'_i is obtained from w_i, $1 \leq i \leq n$, and w_E by executing one of the programs of P_i.

The set of vectors in $\mathbb{N}^m$, $m = |F|$, $F = \{o_1, \ldots, o_m\}$, computed by a P colony Π is defined as

$$N(\Pi) = \{(|v_E|_{o_1}, \ldots, |v_E|_{o_m}) \mid (e^i, \ldots, e^i; \varepsilon) \Rightarrow^* (v_1, \ldots, v_n, v_E)\}$$

where $(e^i, \ldots, e^i, \varepsilon)$, $i \in \{1,2\}$, is the initial configuration, $(v_1, \ldots, v_n, v_E)$ is a halting configuration, and $\Rightarrow^*$ denotes the reflexive and transitive closure of $\Rightarrow$.

Let us denote by $PCOL(i, j, k, check)$ and $PCOL(i, j, k, no\text{-}check)$ the classes of sets of vectors generated by P colonies with at most $j \geq 1$ cells of capacity $i \in \{1,2\}$, having at most $k \geq 1$ programs associated to a cell which contain or do not contain checking rules, respectively. If a numerical parameter is unbounded, we denote it by a $*$.

P colonies can simulate register machines with a rather limited number of programs per cell. In [3], it was shown that

$$PCOL(2, *, 4, check) = PCOL(3, *, 3, check) = \mathbb{N}RE$$

where $\mathbb{N}RE$ denotes the class of recursively enumerable sets of integer vectors. Even one cell is enough, if it may have an arbitrarily large number of programs, that is,

$$PCOL(2, 1, *, check) = \mathbb{N}RE.$$

Similar results were also obtained without the use of checking rules. In this case we have

$$PCOL(2, *, 8, no\text{-}check) = PCOL(3, *, 7, no\text{-}check) = \mathbb{N}RE.$$

3 P colonies with one object

In [1] it was shown that if checking rules are allowed to be used, then all recursively enumerable sets of vectors can even be generated by P colonies with capacity one, that is,

$$PCOL(1, 4, *, check) = \mathbb{N}RE.$$

In the following we show that P colonies with six components generate all vectors even if checking rules are not used.

Theorem 1. $PCOL(1, 6, *, no\text{-}check) = \mathbb{N}RE$.

Proof. We construct a P colony simulating the computations of a register machine. Let us consider an m-register machine $M = (m, H, l_0, l_h, P)$ and represent the content of the register i by the number of copies of a specific object a_i in the environment. We construct the P colony $\Pi = (V, e, F, C_1, \ldots, C_6)$ with:

$$\begin{aligned} V &= \{e, l_i, l_i', l_i'', \bar{l}_i, K_i, L_i, L_i', L_i'', L_i''', E_i, F_i, \$_i \mid \text{for each } l_i \in H\} \cup \\ &\quad \{a_i, a_{i,j} \mid 1 \leq i \leq m,\ 1 \leq j \leq |H|\} \cup \{D, D', T\}, \\ F &= \{a_i \mid \text{register } i \text{ is an output register}\}, \text{ and} \\ C_i &= (e, P_i), \text{ for } 1 \leq i \leq 6. \end{aligned}$$

Because initially there are only copies of e in the environment and inside the cells, we have to initialize the simulation of the computation of M by generating the initial the label l_0, and an arbitrary number of l_i', l_i'' for all $l_i \in H$. These symbols are generated by C_1 and C_2 with the following programs:

$$\begin{aligned} P_1 \supset\ & \{\langle e \to l_r' \rangle, \langle l_r' \longrightarrow e \rangle, \langle e \to l_r'' \rangle, \langle l_r'' \longrightarrow e \rangle \mid l_r \in H\} \cup \\ & \{\langle e \longrightarrow D' \rangle, \langle D' \to l_0 \rangle, \langle l_0 \longrightarrow D \rangle\}, \end{aligned}$$

$$P_2 \supset \{\langle e \to D' \rangle, \langle D' \to D' \rangle, \langle D' \longrightarrow l_1' \rangle, \langle l_1' \to D \rangle, \langle D \longrightarrow l_1'' \rangle\}.$$

With these programs, from the configuration $(e,e,e,e,e,e;\varepsilon)$, we obtain $(D,l_1'',e,e,e,e;l_0w)$ where the environment contains the label of the initial instruction, l_0, and w, a multiset of primed and double primed instruction labels.

To simulate the instruction $l_i:(ADD(r),l_j,l_k)$, cells C_1 and C_3 cooperate to add one copy of object a_r and object l_j or l_k to the environment.

P_1				P_3			
$i_1:$	$\langle D \longrightarrow a_{r,i}\rangle$	$i_6:$	$\langle K_k \to l_k\rangle$	$i_1:$	$\langle e \longrightarrow l_i\rangle$	$i_6:$	$\langle l_i' \to K_k\rangle$
$i_2:$	$\langle a_{r,i} \to a_r\rangle$	$i_7:$	$\langle l_j \longrightarrow D\rangle$	$i_2:$	$\langle l_i \to a_{r,i}\rangle$	$i_7:$	$\langle K_j \longrightarrow e\rangle$
$i_3:$	$\langle a_r \longrightarrow K_j\rangle$	$i_8:$	$\langle l_k \longrightarrow D\rangle$	$i_3:$	$\langle a_{r,i} \longrightarrow l_i'\rangle$	$i_8:$	$\langle K_k \longrightarrow e\rangle$
$i_4:$	$\langle a_r \longrightarrow K_k\rangle$			$i_4:$	$\langle a_{r,i} \to t\rangle$	$i_9:$	$\langle t \to t\rangle$
$i_5:$	$\langle K_j \to l_j\rangle$			$i_5:$	$\langle l_i' \to K_j\rangle$		

It is not difficult to follow how the interplay of these two cells produce the configuration $(D,l_1'',e,e,e,e;l_ja_rw')$ or $(D,l_1'',e,e,e,e;l_ka_rw')$ from a configuration $(D,l_1'',e,e,e,e;l_iw)$ where w,w' are multisets of l_i',l_i'' for $l_i\in H$ and $a_r,\ 1\le r\le m$. If there is no l_i' present in the environment when the program i_3 of cell C_3 should be used, then the programs i_4 and i_9 do not allow the halting of the computation.

For each subtract instruction $l_f:(SUB(r),l_g,l_n)$ there are the following programs in P_1, P_4, P_5 and in P_6:

P_1		P_4				P_5		P_6	
$f_1:$	$\langle D \leftrightarrow L_f\rangle$	$f_1:$	$\langle e \longrightarrow l_f\rangle$	$f_9:$	$\langle L_f \to t\rangle$	$f_1:$	$\langle e \leftrightarrow L_f'\rangle$	$f_1:$	$\langle e \leftrightarrow L_f''\rangle$
$f_2:$	$\langle L_f \to E_f\rangle$	$f_2:$	$\langle l_f \to L_f\rangle$	$f_{10}:$	$\langle L_f' \to t\rangle$	$f_2:$	$\langle L_f' \to l_f'\rangle$	$f_2:$	$\langle L_f'' \to l_f'\rangle$
$f_3:$	$\langle E_f \to F_f\rangle$	$f_3:$	$\langle L_f \leftrightarrow l_f'\rangle$	$f_{11}:$	$\langle t \to t\rangle$	$f_3:$	$\langle l_f' \leftrightarrow a_r\rangle$	$f_3:$	$\langle l_f' \longrightarrow \$_f\rangle$
$f_4:$	$\langle F_f \to \$_f\rangle$	$f_4:$	$\langle l_f' \to L_f'\rangle$			$f_4:$	$\langle l_f' \leftrightarrow \$_f\rangle$	$f_4:$	$\langle \$_f \to l_g\rangle$
$f_5:$	$\langle \$_f \leftrightarrow D\rangle$	$f_5:$	$\langle L_f' \longrightarrow l_f''\rangle$			$f_5:$	$\langle \$_f \to \bar{l}_n\rangle$	$f_5:$	$\langle l_g \leftrightarrow e\rangle$
		$f_6:$	$\langle l_f'' \to L_f'''\rangle$			$f_6:$	$\langle a_r \to e\rangle$	$f_6:$	$\langle l_f' \longrightarrow \bar{l}_n\rangle$
		$f_7:$	$\langle L_f''' \to L_f''\rangle$			$f_7:$	$\langle \bar{l}_n \leftrightarrow e\rangle$	$f_7:$	$\langle \bar{l}_n \to l_n\rangle$
		$f_8:$	$\langle L_f'' \longrightarrow e\rangle$					$f_8:$	$\langle l_n \longrightarrow e\rangle$

In the following table we show how a subtract instruction can be simulated by the programs above. Since C_2 and C_3 cannot apply any of their rules in any step of the following simulation, we omit them from the table. The multiset of objects in the environment is denoted by $[\ldots]$, and for now we assume that it always contains a sufficient amount of l_i',l_i'' objects for any $l_i\in H$.

First we consider the case when there is at least one object a_r in the environment, that is, if the

simulation starts in a configuration $(D, l_1'', e, e, e, e; l_f a_r[\ldots])$.

	configuration of Π					programs to be applied			
	C_1	C_4	C_5	C_6	Env	P_1	P_4	P_5	P_6
1.	D	e	e	e	$l_f a_r[\ldots]$	$-$	f_1	$-$	$-$
2.	D	l_f	e	e	$a_r[\ldots]$	$-$	f_2	$-$	$-$
3.	D	L_f	e	e	$a_r[\ldots]$	$-$	f_3	$-$	$-$
4.	D	l_f'	e	e	$L_f a_r[\ldots]$	f_1	f_4	$-$	$-$
5.	L_f	L_f'	e	e	$Da_r[\ldots]$	f_2	f_5	$-$	$-$
6.	E_f	l_f''	e	e	$L_f' Da_r[\ldots]$	f_3	f_6	f_1	$-$
7.	F_f	L_f'''	L_f'	e	$Da_r[\ldots]$	f_4	f_7	f_2	$-$
8.	$\$_f$	L_f''	l_f'	e	$Da_r[\ldots]$	f_5	f_8	f_3	$-$
9.	D	e	a_r	e	$\$_f L_f''[\ldots]$	$-$	$-$	f_6	f_1
10.	D	e	e	L_f''	$\$_f[\ldots]$	$-$	$-$	$-$	f_2
11.	D	e	e	l_f'	$\$_f[\ldots]$	$-$	$-$	$-$	f_3
12.	D	e	e	$\$_f$	$[\ldots]$	$-$	$-$	$-$	f_4
13.	D	e	e	l_g	$[\ldots]$	$-$	$-$	$-$	f_5
14.	D	e	e	e	$l_g[\ldots]$	$-$	g_1	$-$	$-$

In 13 steps, from a configuration $(D, l_1'', e, e, e, e; l_f a_r[\ldots])$ we obtain $(D, l_1'', e, e, e, e; l_g[\ldots])$ where l_g is the label of the instruction which should follow the successful decrease of the value of the nonempty register r, and the environment contains a multiset of objects l_i', l_i'' for $l_i \in H$.

Now we consider the case when register r, which is the register to be decremented, stores zero, that is, if the simulation starts in a configuration $(D, l_1'', e, e, e, e; l_f[\ldots])$ where the environment does not contain any object a_r.

	configuration of Π					programs to be applied			
	C_1	C_4	C_5	C_6	Env	P_1	P_4	P_5	P_6
1.	D	e	e	e	$l_f[\ldots]$	$-$	f_1	$-$	$-$
2.	D	l_f	e	e	$[\ldots]$	$-$	f_2	$-$	$-$
3.	D	L_f	e	e	$[\ldots]$	$-$	f_3	$-$	$-$
4.	D	l_f'	e	e	$L_f[\ldots]$	f_1	f_4	$-$	$-$
5.	L_f	L_f'	e	e	$D[\ldots]$	f_2	f_5	$-$	$-$
6.	E_f	l_f''	e	e	$L_f' D[\ldots]$	f_3	f_6	f_1	$-$
7.	F_f	L_f'''	L_f'	e	$D[\ldots]$	f_4	f_7	f_2	$-$
8.	$\$_f$	L_f''	l_f'	e	$D[\ldots]$	f_5	f_8	$-$	$-$
9.	D	e	l_f'	e	$\$_f L_f''[\ldots]$	$-$	$-$	f_4	f_1
10.	D	e	$\$_f$	L_f''	$[\ldots]$	$-$	$-$	f_5	f_2
11.	D	e	$\bar{l}_n$	l_f'	$[\ldots]$	$-$	$-$	f_7	$-$
12.	D	e	e	l_f'	$\bar{l}_n[\ldots]$	$-$	$-$	$-$	f_6
13.	D	e	e	$\bar{l}_n$	$[\ldots]$	$-$	$-$	$-$	f_7
14.	D	e	e	l_n	$[\ldots]$	$-$	$-$	$-$	f_8
15.	D	e	e	e	$l_n[\ldots]$	$-$	n_1	$-$	$-$

Similarly to the previous case, in 14 steps we obtain a configuration $(D, l_1'', e, e, e, e; l_n[\ldots])$ where l_n is the label of the instruction which should follow l_f if register r is empty, that is, if the decrease of its value is not possible.

Consider now what happens if there is an insufficient amount of objects l_i', l_i'' for $l_i \in H$ is present in the environment. Notice that such symbols are needed in step 3 and 5 by cell C_4. If there is no more

available (not enough of them were produced in the initial phase by C_1 and C_2), then the programs f_9, f_{10}, and f_{11} do not allow the halting of the computation.

From these considerations we can see that after the initialization phase, all instructions of the register machine M can be simulated by the P colony. If the label of the halt instruction, l_h is produced, the computation halts since there is no program for processing the object l_h. The reader can immediately see that Π computes the same set of vectors as M. □

4 P colonies with senders and consumers

Now we continue with the investigation of two object P colonies with insertion-deletion programs. It is not too difficult to see that if we allow a cell to contain both types of programs, then we can simulate the other types of programs in two steps, thus, it is more interesting to consider P colonies having cells which contain either insertion or deletion programs, but not both types at the same time. We call these systems P colonies with senders and consumers. A sender is a cell with only insertion programs, a consumer is a cell with only deletion programs.

Let us denote by $PCOL(i, j, \text{s-c})$ the class of sets of numbers generated by P colonies with senders and consumers having at most $i \geq 1$ cells with at most $j \geq 1$ program each.

Example 2. *(a) A P colony with one sender cell can generate the Parikh set of a regular language $L \subseteq T^*$. Let $G = (N, T, P, S)$ be a regular grammar such that $L(G) = L$.*

For generating the Parikh vectors of the words in L, we use, for each $S \to aB$ of P, the programs $\langle e, out; e \to eS\rangle$, $\langle e, out; S \to aB\rangle$ and then $\langle x, out; A \to aB\rangle, x \in T$ for every $A \to aB$ in P. Finally, for every rule of the form $A \to a$ we need $\langle x, out; A \to aF\rangle, x \in T$, $\langle a, out; F \to FF\rangle$, where $F \notin T \cup N$.

(b) A P colony with one consumer cell can "consume" the Parikh set of a regular language L. To see this, let $M = (Q, T, \delta, q_0, F)$ be a deterministic finite automaton such that $L(M) = L$.

We need the program $\langle e, in; ee \to q_0\rangle$, and to every transition $\delta(q_i, a) = q_j$ in M, we introduce $\langle a, in; xq_i \to q_j\rangle, x \in T \cup \{e\}$. If $q_j \in F$ in $\delta(q_i, a) = q_j$ we have to add the programs $\langle a, in; xq_i \to F\rangle, x \in T$, where $F \notin Q \cup T$.

Now we show that three cells, one sender and two consumers are sufficient to generate all recursively enumerable sets of integer vectors.

Theorem 3. $PCOL(3, *, s\text{-}c) = \mathbb{N}RE$.

Proof. Consider an m-register machine $M = (m, H, l_0, l_h, P)$, $m \geq 1$. We simulate M by representing the content of the register i by the number of copies of a specific object a_i in the environment. We construct the P colony $\Pi = (V, e, F, C_1, C_2, C_3)$ with:

$$\begin{aligned} V &= \{e, l, l', l'', l''', l^{iv}, l^v, \bar{l}, \bar{\bar{l}} \mid l \in H\} \cup \{a_i \mid 1 \leq i \leq m\} \cup \{K, T_1, T_2, T_3, T_4, T_5\}, \\ F &= \{a_i \mid \text{register } i \text{ is an output register}\}, \text{ and} \\ C_i &= (ee, P_i) \text{ for } 1 \leq i \leq 3. \end{aligned}$$

The P colony Π starts its computation in the initial configuration $(ee, ee, ee; \varepsilon)$. We initialize the computation by generating the initial label l_0 with a program from P_1, $\langle e, out; e \to l_0 l_0\rangle \in P_1$ obtaining $(l_0 l_0, ee, ee; \varepsilon)$.

The simulation of an instruction with label l_i starts from a configuration $(l_i l_i, ee, ee; w)$ where $w \in V^*$, the multiset of objects in the environment, represents the counter contents of M.

To simulate an *ADD* instruction, we use the programs of P_1 and P_3. For each $l_i, l_j, l_k \in H$ with l_i being

the label of an instruction $l_i : (ADD(r), l_j, l_k)$, we have the following programs:

P_1		P_3	
$i_1:$	$\langle l_i, out; l_i \to a_r l_j \rangle$	$i_1:$	$\langle l_i, in; ee \to T_1 \rangle$
$i_2:$	$\langle l_i, out; l_i \to a_r l_k \rangle$	$i_2:$	$\langle e, in; l_i T_1 \to e \rangle$
$i_3:$	$\langle a_r, out; l_j \to l_j l_j \rangle$	$i_3:$	$\langle l_i, in; \bar{\bar{l}}_i T_5 \to T_1 \rangle$
$i_4:$	$\langle a_r, out; l_k \to l_k l_k \rangle$		

Using these programs, we obtain a sequence of configurations

$$(l_i l_i, ee, ee; w) \Rightarrow (a_r l, ee, ee; l_i w) \Rightarrow (ll, ee, l_i T_1; a_r w)$$

where l is the label of the next instruction, that is, we either have $(l_j l_j, ee, l_i T_1; a_r w)$, or the configuration $(l_k l_k, ee, l_i T_1; a_r w)$. The contents of cell C_3, $l_i T_1$, will change in the next step to ee independently of the several ways of the continuation of the computation, as we shall see later.

The program labeled with i_3 is used if the instruction simulated before l_i was a SUB instruction (see below). In this case, the configuration in which the simulation of l_i starts is $(l_i l_i, ee, \bar{l}_i T_4; \bar{\bar{l}}_i w)$ and we need the steps $(l_i l_i, ee, \bar{l}_i T_4; \bar{\bar{l}}_i w) \Rightarrow (a_r l, ee, \bar{\bar{l}}_i T_5; l_i w) \Rightarrow (ll, ee, l_i T_1; a_r w)$ and program i_3 to obtain the same configuration as before.

Now we show how to simulate a SUB instruction. For each $l_j, l_k, l_l \in H$ with l_j being the label of an instruction $l_j : (SUB(r), l_k, l_l)$, and for all labels $l_s \in H$, we have the following programs.

P_1	P_2		P_3	
$j_1 : \langle l_j, out; l_j \to l'_j l'_j \rangle$	$j_1:$	$\langle l_j, in; ee \to e \rangle$	$j_1:$	$\langle l'_j, in; ee \to T_1 \rangle$
$j_2 : \langle l'_j, out; l'_j \to l''_j l''_j \rangle$	$j_2:$	$\langle a_r, in; e l_j \to e \rangle$	$j_2:$	$\langle e, in; l'_j T_1 \to T_2 \rangle$
$j_3 : \langle l''_j, out; l''_j \to l'''_j l^{iv}_j \rangle$	$j_3:$	$\langle l''_j, in; e l_j \to e \rangle$	$j_3:$	$\langle l''_j, in; eT_2 \to T_3 \rangle$
$j_4 : \langle l'''_j, out; l^{iv}_j \to \bar{l}_k \bar{l}_k \rangle$	$j_4:$	$\langle l'''_j, in; a_r e \to e \rangle$	$j_{4,s}:$	$\langle \bar{l}_s, in; l''_j T_3 \to T_4 \rangle$
$j_5 : \langle l^{iv}_j, out; l'''_j \to \bar{l}_l \bar{l}_l \rangle$	$j_5:$	$\langle e, in; l'''_j e \to e \rangle$	$j_{5,s}:$	$\langle \bar{l}_s, in; eT_2 \to T_4 \rangle$
$j_6 : \langle \bar{l}_k, out; \bar{l}_k \to \bar{\bar{l}}_k \bar{\bar{l}}_k \rangle$	$j_6:$	$\langle l^{iv}_j, in; a_r e \to K \rangle$	$j_{6,s}:$	$\langle \bar{\bar{l}}_s, in; \bar{l}_s T_4 \to T_5 \rangle$
$j_7 : \langle \bar{\bar{l}}_k, out; \bar{\bar{l}}_k \to l_k l_k \rangle$	$j_7:$	$\langle e, in; l^{iv}_j K \to K \rangle$	$j_{7,s}:$	$\langle e, in; \bar{\bar{l}}_s T_5 \to e \rangle$
$j_8 : \langle \bar{l}_l, out; \bar{l}_l \to \bar{\bar{l}}_l \bar{\bar{l}}_l \rangle$	$j_8:$	$\langle e, in; eK \to K \rangle$		
$j_9 : \langle \bar{\bar{l}}_l, out; \bar{\bar{l}}_k \to l_l l_l \rangle$	$j_9:$	$\langle l'''_j, in; l''_j e \to K \rangle$		
	$j_{10}:$	$\langle e, in; l'''_j K \to K \rangle$		
	$j_{11}:$	$\langle l^{iv}_j, in; l''_j e \to e \rangle$		
	$j_{12}:$	$\langle e, in; l^{iv}_j e \to e \rangle$		

In the following tables we show how the programs above simulate the execution of the instruction $l_j : (SUB(r), l_k, l_l)$. To save space, we use the sign "/" to separate the different possible multisets which might appear in the same row of the table.

First we consider the case when register r is not empty, that is, when there is at least one object a_r present in the environment. We see that starting with a configuration where C_1 contains the objects $l_j l_j$ and the environment contains a_r, in six steps we obtain a configuration where the object a_r is removed from the environment, and C_1 either contains the label of the next instruction l_k, or because of the presence of program j_8, in P_2, the computation will never be able to halt.

	configuration of Π				programs to be applied		
	C_1	C_2	C_3	Env	P_1	P_2	P_3
1.	$l_j l_j$	ee	?	$a_r w'$	j_1	$-$	?
2.	$l'_j l'_j$	ee	?	$l_j a_r w''$	j_2	j_1	?
3.	$l''_j l''_j$	$l_j e$	ee	$l'_j a_r w$	j_3	j_2	j_1
4.	$l'''_j l^{iv}_j$	$a_r e$	$l'_j T_1$	$l''_j w$	j_4/j_5	$-$	j_2
5.	$\bar{l}_k \bar{l}_k / \bar{l}_l \bar{l}_l$	$a_r e$	eT_2	$(l'''_j/l^{iv}_j) l''_j w$	j_6/j_8	j_4/j_6	j_3
6.	$\bar{\bar{l}}_k \bar{\bar{l}}_k / \bar{\bar{l}}_l \bar{\bar{l}}_l$	$l'''_j e / l^{iv}_j K$	$l''_j T_3$	$(\bar{l}_k/\bar{l}_l) w$	j_7/j_9	j_5/j_7	$j_{4,k}/j_{4,l}$
7.	$l_k l_k / l_l l_l$	ee/eK	$(\bar{l}_k/\bar{l}_l) T_4$	$(\bar{\bar{l}}_k/\bar{\bar{l}}_l) w$	k_1/l_1	$-/j_8$	$j_{6,k}/j_{6,l}$
8.	$l'_k l'_k / l'_l l'_l$	ee/eK	$(\bar{\bar{l}}_k/\bar{\bar{l}}_l) T_5$	$(l_k/l_l) w$	k_2/l_2	k_1/j_8	$j_{7,k}/j_{7,l}$
9.	$l''_k l''_k / l''_l l''_l$	$(l_k/l_l)e/eK$	ee	$(l'_k/l'_l) w$	k_3/l_3	k_2/j_8	j_1

Now we show the simulation of the $l_j : (SUB(r), l_k, l_l)$ instruction when there is no object a_r is present in the environment, that is, when register r is empty. In this case, similarly to the previous one, we either get the objects $l_k l_k$ in the cell C_1, or the computation will not be able to halt.

	configuration of Π				rules to be applied		
	C_1	C_2	C_3	Env	P_1	P_2	P_3
1.	$l_j l_j$	ee	?	w	j_1	$-$	?
2.	$l'_j l'_j$	ee	?	$l_j w$	j_2	j_1	?
3.	$l''_j l''_j$	$l_j e$	ee	$l'_j w$	j_3	$-$	j_1
4.	$l'''_j l^{iv}_j$	$l_j e$	$l'_j T_1$	$l''_j w$	j_4/j_5	j_3	j_2
5.	$\bar{l}_k \bar{l}_k / \bar{l}_l \bar{l}_l$	$l''_j e$	eT_2	$(l'''_j/l^{iv}_j) w$	j_6/j_8	j_9/j_{11}	$-$
6.	$\bar{\bar{l}}_k \bar{\bar{l}}_k / \bar{\bar{l}}_l \bar{\bar{l}}_l$	$l'''_j K / l^{iv}_j e$	eT_2	$(\bar{l}_k/\bar{l}_l) w$	j_7/j_9	j_{10}/j_{12}	$j_{5,k}/j_{5,l}$
7.	$l_k l_k / l_l l_l$	eK/ee	$(\bar{l}_k/\bar{l}_l) T_4$	$(\bar{\bar{l}}_k/\bar{\bar{l}}_l) w$	k_1/l_1	$j_8/-$	$j_{6,k}/j_{6,l}$
8.	$l'_k l'_k / l'_l l'_l$	eK/ee	$(\bar{\bar{l}}_k/\bar{\bar{l}}_l) T_5$	$(l_k/l_l) w$	k_2/l_2	j_8/k_1	$j_{7,k}/j_{7,l}$
9.	$l''_k l''_k / l''_l l''_l$	$eK/(l_k/l_l)e$	ee	$(l'_k/l'_l) w$	k_3/l_3	j_8/k_2	j_1

The rules to be applied and the objects contained by the cell C_3 in row 1. and row 2. of the tables above depend on the instruction l_i which was simulated before l_j. If l_i is an ADD instruction, then we have $l_i T_1$ in the first row, and applying the program i_2 from P_3 we get ee in the second row, where no program is applied until the next step. Also, $w = w' = w''$ in this case.

If l_i is a SUB instruction, then (as we can also see from row 7. and row 8.) the contents of the cell C_3 is $\bar{l}_j T_4$ and $\bar{\bar{l}}_j T_5$ in the first two rows where the programs $i_{6,j}$ and $i_{7,j}$ are applied. In this case $w'' = \bar{\bar{l}}_j w$, and $w' = w$.

As we have seen above, the P colony successfully simulates each instruction of M and since there is no program to process l_h, the label of the halt instruction, it also halts when the computation of M is finished. It is also easy to see that M and Π compute the same set of vectors of non-negative integers. □

5 Conclusion

We have examined extremely simplified variants of P colonies: P colonies of capacity one with no checking rules, and P colonies with capacity two, but only with senders and consumers. We have shown that even these very simple variants are able to simulate arbitrary register machines, that is, to compute all Turing computable sets of vectors.

Bibliography

[1] L. Cienciala, L. Ciencialová, A. Kelemenová. On the number of agents in P colonies. In: *Membrane Computing. 8th International Workshop, WMC 2007. Thessaloniki, Greece, June 25-28, 2007. Revised Selected and Invited Papers.* Edited by G. Eleftherakis, P. Kefalas, Gh. Păun, G. Rozenberg, A. Salomaa. Volume 4860 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin-Heidelberg, 2007, 193-208.

[2] E. Csuhaj-Varjú, J. Dassow, J. Kelemen, Gh. Păun. *Grammar Systems – A Grammatical Approach to Distribution and Cooperation*. Gordon and Breach, London, 1994.

[3] E. Csuhaj-Varjú, J. Kelemen, A. Kelemenová, Gh. Păun, Gy. Vaszil. Computing with cells in environment: P colonies. *Journal of Multi-Valued Logic and Soft Computing* 12:201-215, 2006.

[4] J. Kelemen, A. Kelemenová. A grammar-theoretic treatment of multi-agent systems. *Cybernetics and Systems* 23:621-633, 1992.

[5] J. Kelemen, A. Kelemenová, Gh. Păun. Preview of P colonies: A biochemically inspired computing model. In: *Workshop and Tutorial Proceedings. Ninth International Conference on the Simulation and Synthesis of Living Systems (Alife IX)*. Edited by M. Bedau et al. Boston Mass., 2004, 82-86.

[6] M. Minsky. *Computation – Finite and Infinite Machines.* Prentice Hall, Englewood Cliffs, NJ, 1967.

[7] Gh. Păun. *Membrane Computing – An Introduction.* Springer-Verlag, Berlin, 2002.

Lucie Ciencialová has received her PhD. in 2008 and she works as an assistant professor at the Institute of Computer Science, Silesian University in Opava. Her interests include theoretical informatics and natural computing.

Erzsébet Csuhaj-Varjú D.Sc, dr. Habil., is head of the Theoretical Computer Science Research Group at the Computer and Automaton Research Institute of the Hungarian Academy of Sciences. She has also been affiliated with the Department of Algorithms and Applications of the Eötvös Loránd University, Budapest, Hungary, as science advisor. Her main research interests are formal languages, distributed systems, and nature-motivated computing. In these areas she has more than 150 papers, a monograph, and eleven edited volumes published in international publication forums.

Alica Kelemenová is an associated professor at the Institute of Computer Science, Silesian University in Opava, Czech republic and at the Department of Computer Science Catholic University in Ružomberok, Slovakia. Her main research interest is theoretical computer science, especially formal language theory and biologically motivated generative devices like L systems and P systems.

György Vaszil, PhD. is a senior research fellow at the Theoretical Computer Science Research Group of the Computer and Automation Research Institute of the Hungarian Academy of Sciences. His research interests are formal language and automata theory, nature motivated computational models.