Generalized P Colonies with passive environment

Lucie Ciencialová, Luděk Cienciala, and Petr Sosík

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

Summary. We study two variants of P colonies with initial content of P colony and so called passive environment: P colonies with two objects inside each agent that can only consume or generate objects, and P colonies with one object inside each agent using rewriting and communication rules. We show that the first kind of P colonies with one consumer agent and one sender agent can generate all sets of natural numbers computed by register machines, and hence they are computationally universal in the Turing sense. Similarly, also the second kind of systems with three agents with rewriting/consuming rules is computationally complete. The paper improves previously published universality results concerning generalized P colonies, and it also extends our knowledge about very simple multi-agent systems capable of universal computation.

Key words: P colony, computational completeness, register machine

1 Introduction

P colony was introduced in [9] as a very simple variant of membrane systems inspired by so called *colonies* of formal grammars. See [11] for more information about membrane systems and [7] for details on grammar systems theory. There are three basic entities in the P colony model: objects, agents and the environment. A P colony is composed of agents, each containing a collection of objects embedded in a membrane. The objects can be placed in the environment, too. Agents are equipped with programs composed of rules that allow interactions of objects. The number of objects inside each agent is set by definition and it is usually very low – 1, 2 or 3. The environment of P colony serves as a communication channel for agents: an agent is able to affect the behaviour of another agent by sending objects via the environment. There is also a special type of *environmental objects* denoted by e which are present in the environment in an unlimited number of copies.

A specific variant of P colony called *eco-P colony* with two object inside each agent, where the environment can change independently on the agents, was introduced in [1]. The evolution of the environment is controlled by a 0L scheme

applying context free rules in parallel to all possible objects in the environment which are unused by the agents in the current step of computation.

The activity of agents is based on rules that can be rewriting, communication or checking; these three types was introduced in [9]. Furthermore, generating, consuming and transporting rules were introduced in [5].

Rewriting rule $a \to b$ allows an agent to rewrite (evolve) one object a placed inside the agent to object b.

Communication rule $a \leftrightarrow b$ exchanges one object c placed inside the agent for object d from the environment.

Checking rule r_1/r_2, where each of r_1, r_2 is a rewriting or a communication rule, sets a priority between these two rules. The agent try to apply the first rule and if it cannot be performed, the agent executes the second rule.

Generating rule $a \to bc$ creates two objects b, c from one object a.

Consuming rule $ab \to c$ rewrites two objects a, b to one object c.

Transporting rule of the form $(a\ in)$ or $(a\ out)$ is used to transport one object from the environment into the agent, or from the agent to the environment, respectively. The rule is always associated with a consuming/generating rule to keep a constant number of object inside the agent.

The rules are combined into programs in such a way that all object inside the agent are affected by execution of the rules in every step. Consequently, the number of rules in the program is the same as the number of object inside the agent. The programs that contain consuming rules are called consuming programs and the programs with generating rules are called generating programs. The agent that only contains consuming resp. generating programs is called consumer resp. sender.

P colonies with senders and consumers without evolving environment were studied in [5] and the authors proved their computational completeness (in the Turing sense), as well as computational completeness of P colonies with senders and consumers with 0L scheme for the environment. Many papers were devoted to P colonies with rewriting and communication rules without evolving environment, e.g., [4, 6, 8], and there are two book chapters in [2] and [11] describing this topic.

In this paper we focus on P colonies with initial content of P colony with "passive" environment. The paper is structured as follows: The second section is devoted to definitions and notations used in the paper. The third section contains results obtained during studies of P colonies with senders and consumers. In the fourth section we study P colonies with one object inside the agent and rewriting/communication rules. The paper concludes with a summary of presented results.

2 Definitions

Throughout the paper we assume the reader is familiar with basic of formal automata and language theory. We introduce notation used in the paper.

We use $\mathbb{N}\cdot\mathsf{RE}$ to denote the family of recursively enumerable sets of natural numbers and $\mathbb{N}$ to denote the set of natural numbers.

Σ is a notation for the alphabet. Let Σ^* be set of all words over alphabet Σ (including the empty word ε). For the length of the word $w \in \Sigma^*$ we use the notation $|w|$ and the number of occurrences of symbol $a \in \Sigma$ in w is denoted by $|w|_a$.

A *multiset* of objects M is a pair $M = (V, f)$, where V is an arbitrary (not necessarily finite) set of objects and f is a mapping $f : V \to N$; f assigns to each object in V its multiplicity in M. The set of all multisets over the set of objects V is denoted by V^*. The cardinality of M, denoted by $card(M)$, is defined by $card(M) = \sum_{a \in V} f(a)$. Any multiset of objects M with the set of objects $V = \{a_i, \dots a_n\}$ 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 M, and ε represents the empty multiset.

The mechanism of evolution of the environment is based on a *0L scheme.* It is a pair (Σ, P), where Σ is the alphabet of 0L scheme and P is the set of context free rules fulfilling the condition $\forall a \in \Sigma\ \exists \alpha \in \Sigma^*$ such that $(a \to \alpha) \in P$. For $w_1, w_2 \in \Sigma^*$ we write $w_1 \Rightarrow w_2$ if $w_1 = a_1 a_1 \dots a_n, w_2 = \alpha_2 \alpha_2 \dots \alpha_n$, for $a_i \to \alpha_i \in P, 1 \leq i \leq n$.

A *register machine* [10] is the construct $M = (m, H, l_0, l_h, P)$ where:
- m is a number of registers, H is a set of instruction labels,
- l_0 is an initial/start label, l_h is the final label,
- P is a finite set of instructions injectively labelled with the elements from the given 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 contents 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 is not empty, then subtract 1 from its contents and go to instruction l_2, otherwise proceed to instruction l_3.

l_h : *HALT* Stop the machine. The final label l_h is only assigned to this instruction.

Without loss of generality, one can assume that in each *ADD*-instruction $l_1 : (ADD(r), l_2, l_3)$ and in each conditional *SUB*-instruction $l_1 : (SUB(r), l_2, l_3)$ the labels l_1, l_2, l_3 are mutually distinct. The register machine M computes a set $N(M)$ of numbers in the following way: we start with all registers empty (hence storing the number zero) with the instruction with label l_0 and we proceed to apply the instructions as indicated by the labels (and made possible by the contents of registers). If we reach 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 nondeterminism in choosing the continuation of the computation in the case of *ADD*-instructions, $N(M)$ can be an infinite set.) The family of sets of numbers computed by register machines is denoted by $\mathbb{N}\cdot RM$.

Theorem 1. [10] $\mathbb{N}\cdot RM = \mathbb{N}\cdot\mathsf{RE}$.

2.1 Generalized P colonies

Definition 1. *A P colony with capacity $c \geq 1$ is the structure*

$$\Pi = (\Sigma, e, f, v_E, D_E, B_1, \ldots, B_n), \text{ where}$$

- *Σ is the alphabet of the colony, its elements are called objects,*
- *e is the basic (environmental) object of the colony, $e \in \Sigma$,*
- *f is final object of the colony, $f \in \Sigma$,*
- *v_E is the initial content of the environment, $v_E \in (\Sigma - \{e\})^*$,*
- *D_E is 0L scheme (Σ, P_E), where P_E is the set of context free rules,*
- *B_i, $1 \leq i \leq n$, are the agents, every agent is the structure $B_i = (o_i, P_i)$, where o_i is the multiset over Σ, it defines the initial state (content) of the agent B_i and $|o_i| = c$ and $P_i = \{p_{i,1}, \ldots, p_{i,k_i}\}$ is the finite set of programs of three types:*
  - *(1) generating program with generating rules $a \to bc$ and transporting rules d out - the number of generating rules is the same as the number of transporting rules.*
  - *(2) consuming program with consuming rules $ab \to c$ and transporting rules d in - the number of consuming rules is the same as the number of transporting rules.*
  - *(3) rewriting/communication program can contain three types of rules:*
    - *$\diamond$ $a \to b$, called a rewriting rule,*
    - *$\diamond$ $c \leftrightarrow d$, called a communication rule,*
    - *$\diamond$ r_1/r_2, called a checking rule; each of r_1, r_2 is a rewriting or a communication rules.*

Every agent has only one type of programs. The agent with generating programs is called *sender* and the agent with consuming programs is called *consumer.* The capacity of P colony with senders and consumers must be even number.

The *initial configuration* of a P colony is the $(n+1)$-tuple $(o_1, \ldots, o_n, v_E)$, with symbols $o_1, \ldots, o_n, v_E$ as in Definition 1. In general, the *configuration* of the P colony Π is defined as $(n+1)$-tuple $(w_1, \ldots, w_n, w_E)$, where w_i represents the multiset of objects inside i-th agent, $|w_i| = c$, $1 \leq i \leq n$, and $w_E \in (\Sigma - \{e\})^*$ is the multiset of objects different from e placed in the environment.

At each step of the (parallel) computation every agent tries to find one of its programs to apply. If the number of applicable programs is higher than one, the agent non-deterministically chooses one. At each step of computation, the set of active agents executing a program must be maximal, i.e., no further agent can be added to it.

By applying programs, the P colony passes from one configuration to another configuration. Objects in the environment unaffected by any program in the given step are rewritten by the 0L scheme D_E. A sequence of configurations starting from the initial configuration is called a *computation.* A configuration is *halting* if the P colony has no applicable program. Each halting computation has associated

a *result* – the number of copies of the final object placed in the environment in halting configuration.

$$N(\Pi) = \{|w_E|_f \mid (o_1, \ldots, o_n, v_E) \Rightarrow^* (w_1, \ldots, w_n, w_E)\},$$

where $(o_1, \ldots, o_n, v_E)$ is the initial configuration, $(w_1, \ldots, w_n, w_E)$ is the final configuration, and $\Rightarrow^*$ denotes reflexive and transitive closure of $\Rightarrow$.

Let us denote $NEPCOL(i, j, k, u, v, w)$ the family of the sets computing by P colonies with at most $j \geq 1$ agents with $i \geq 1$ objects inside the agent and with at most $k \geq 1$ programs associated with each agent such that:

- $u = check$ if the P colony uses rewriting/communication rules with checking rules
- $u = no\text{-}check$ if the P colony uses rewriting/communication rules without checking rules
- $u = s/c/sc$ if the P colony contains only sender / only consumer / both sender and consumer agents
- $v = pas$ if the rules of 0L scheme are of type $a \to a$ only,
- $v = act$ if the set of rules of 0L scheme contains at least one rule of another type than $a \to a$,
- $w = ini$ if the environment or agents contain initially objects different from e, otherwise w is omitted,

If a numerical parameter is unbounded, we denote it by a $*$.

In [5] the authors deal with P colonies with senders and consumers with "passive" environment, they show that

$$NEPCOL(2, 3, *, sc, pas) = \mathbb{N}\cdot\mathsf{RE}.$$

In [1] there are results of P colonies with "active" environment:

$$NEPCOL(2, 2, *, c, act, ini) = \mathbb{N}\cdot\mathsf{RE}$$

$$NEPCOL(2, 2, *, sc, pas, ini) \supseteq \mathbb{N}\cdot RM_{pb}.$$

Other results are shown for P colonies with "passive" environment and rewriting/communication rules and with only one object inside the agent in [3]

$$NEPCOL(1, 4, *, check, pas) = \mathbb{N}\cdot\mathsf{RE}$$

and in [5]

$$NEPCOL(1, 6, *, no\text{-}check, pas) = \mathbb{N}\cdot\mathsf{RE}.$$

3 P colonies with senders and consumers

In this section we study computational power of P colonies with two objects inside the agent - consumer or sender. We extend the previous results reported in [1].

Theorem 2. $NEPCOL(2,2,*,sc,pas,ini) = \mathbb{N}\cdot\mathsf{RE}$.

Proof. Consider register machine $M = (m, H, l_0, l_h, P)$. All labels from the set H are objects in P colony. The content of register r is represented by the number of copies of objects a_r placed in the environment.

Let u be a mapping $u : H \to \{a_r \mid 1 \le r \le m\} \cup \{L_i \mid l_i \in H\}$ defined as

$$u(l_i) = \begin{cases} a_r \text{ for } l_i : (ADD(r), l_j, l_k) \\ L_i \text{ for } l_i : (SUB(r), l_j, l_k) \end{cases}$$

We construct the P colony $\Pi = (\Sigma, e, a_1, a_2^w, D_E, B_1, B_2)$ with:

- $\Sigma = \{l_i, L_i, L_i^1, L_i^2, W_i^0, W_i^1, W_i^2, l_i^0, l_i^1 \mid l_i \in H\} \cup \{a_i \mid 1 \le i \le m\} \cup$ $\cup \{e, C, Q\}$,
- $B_1 = (l_0 u(l_0), P_1)$,
- $B_2 = (ee, P_2)$.

At the beginning of computation the agent B_1 contains object l_0 representing the label of the initial instruction of M.

An instruction $l_i = (ADD(r), l_j, l_k)$ is simulated by agent B_1 by using following programs:

B_1 :

$1 : \langle l_i \to l_j u(l_j); a_r\ out\rangle;$
$2 : \langle l_i \to l_k u(l_k); a_r\ out\rangle;$

The computation is done in the following way: agent B_1 (sender) simulates the addition of one to the content of register r (sending one copy of object a_r to the environment), and it generates the object l_j or l_k – the label of instruction which will the simulated register machine M execute next. Simultaneously it "precomputes" objects for the execution of the next instruction.

An instruction $l_i : (SUB(r), l_j, l_k)$ is simulated by the following rules and programs:

B_1 :

$3 : \langle l_i \to W_i^0 e, L_i\ out\rangle;$
$4 : \langle W_i^0 \to W_i^1 L_i^1, e\ out\rangle;$
$5 : \langle W_i^1 \to W_i^2 L_i^2, L_i^1\ out\rangle;$
$6 : \langle W_i^2 \to l_j^0 l_j^0, L_i^2\ out\rangle;$
$7 : \langle W_i^2 \to l_k^0 l_k^0, L_i^2\ out\rangle;$
$8 : \langle l_j^0 \to l_j^1 e, l_j^0\ out\rangle;$
$9 : \langle l_k^0 \to l_k^1 e, l_k^0\ out\rangle;$
$10 : \langle l_j^1 \to l_j^2 e, e\ out\rangle;$
$11 : \langle l_k^1 \to l_k^2 e, e\ out\rangle;$
$12 : \langle l_j^2 \to l_j^3 e, e\ out\rangle;$
$13 : \langle l_k^2 \to l_k^3 e, e\ out\rangle;$
$14 : \langle l_j^3 \to l_j u(l_j), e\ out\rangle;$
$15 : \langle l_k^3 \to l_k u(l_k), e\ out\rangle;$

B_2 :

$A : \langle ee \to e; L_i\ in\rangle;$
$B : \langle L_i e \to L_i; a_r\ in\rangle;$
$C : \langle L_i a_r \to L_i^1; L_i^1\ in\rangle;$
$D : \langle L_i e \to L_i^2; L_i^2\ in\rangle;$
$E : \langle L_i^1 L_i^1 \to c; l_j^0\ in\rangle;$
$F : \langle L_i^1 L_i^1 \to q; l_k^0\ in\rangle;$
$G : \langle L_i^2 L_i^2 \to q; l_j^0\ in\rangle;$
$H : \langle L_i^2 L_i^2 \to c; l_k^0\ in\rangle;$
$I : \langle l_j^0 c \to c; L_i^2\ in\rangle;$
$J : \langle l_k^0 c \to c; L_i^1\ in\rangle;$
$K : \langle L_i^1 c \to e; e\ in\rangle;$
$L : \langle L_i^2 c \to e; e\ in\rangle;$
$M : \langle l_j^0 q \to q; e\ in\rangle;$
$N : \langle l_k^0 q \to q; e\ in\rangle;$
$O : \langle qe \to q; e\ in\rangle;$

If there are objects l_i (the label of *SUB*-instruction) and L_i inside the agent B_1, the agent sends object L_i (using the rule labelled 3) to the environment. This is the message for the agent B_2 to try to consume one copy of object a_r from the environment (try to subtract one from the content of register r.)

If the agent B_2 is successful (using the program labelled B), then the second agent consumes L_i^1. If there is no a_r in the environment, the agent has to wait one step and then it consumes object L_i^2.

The agent B_1 generates object l_j^0 or l_k^0, non-deterministically choosing the instruction to be simulated next (program 6 or 7). If the non-deterministic choice was wrong (the agent generates l_j^0 and register r was empty or the agent generates l_k^0 and the register was nonempty), the agent B_2 would use program labelled O and the computation never halts.

If the register r stores nonzero value:

	B_1	B_2	Env	P_1	P_2
1.	l_iL_i	ee	$a_r^x w$	3	–
2.	$W_i^0 e$	ee	$L_i a_r^x w$	4	A
3.	$W_i^1 L_i^1$	$L_i e$	$a_r^x w$	5	B
4.	$W_i^2 L_i^2$	$L_i a_r$	$L_i^1 a_r^{x-1} w$	6 or 7	C
5.	$l_j^0 l_j^0$	$L_i^1 L_i^1$	$L_i^2 a_r^{x-1} w$	8	–
6.	$l_j^1 e$	$L_i^1 L_i^1$	$l_j^0 L_i^2 a_r^{x-1} w$	10	E
7.	$l_j^2 e$	cl_j^0	$L_i^2 a_r^{x-1} w$	12	I
8.	$l_j^3 e$	cL_i^2	$a_r^{x-1} w$	14	L
9.	$l_j u(l_j)$	ee	$a_r^{x-1} w$	?	–

If the register r stores zero:

	B_1	B_2	Env	P_1	P_2
1.	l_iL_i	ee	w	3	–
2.	$W_i^0 e$	ee	$L_i w$	4	A
3.	$W_i^1 L_i^1$	$L_i e$	w	5	–
4.	$W_i^2 L_i^2$	$L_i e$	$L_i^1 w$	6 or 7	–
5.	$l_j^0 l_j^0$	$L_i e$	$L_i^1 L_i^2 w$	8	D
6.	$l_j^1 e$	$L_i^2 L_i^2$	$l_j^0 L_i^1 w$	10	G
7.	$l_j^2 e$	ql_j^0	$L_i^1 w$	12	M
8.	$l_j^3 e$	qe	$L_i^1 w$	14	O
9.	$l_j u(l_j)$	qe	$L_i^1 w$	?	O

	B_1	B_2	Env	P_1	P_2
1.	l_iL_i	ee	$a_r^x w$	3	–
2.	$W_i^0 e$	ee	$L_i a_r^x w$	4	A
3.	$W_i^1 L_i^1$	$L_i e$	$a_r^x w$	5	B
4.	$W_i^2 L_i^2$	$L_i a_r$	$L_i^1 a_r^{x-1} w$	7 or 6	C
5.	$l_k^0 l_k^0$	$L_i^1 L_i^1$	$L_i^2 a_r^{x-1} w$	9	–
6.	$l_k^1 e$	$L_i^1 L_i^1$	$l_k^0 L_i^2 a_r^{x-1} w$	11	E
7.	$l_k^2 e$	ql_k^0	$L_i^2 a_r^{x-1} w$	13	I
8.	$l_k^3 e$	qe	$a_r^{x-1} w$	15	O
9.	$l_k u(l_k)$	qe	$a_r^{x-1} w$	?	O

	B_1	B_2	Env	P_1	P_2
1.	l_iL_i	ee	w	3	–
2.	$W_i^0 e$	ee	$L_i w$	4	A
3.	$W_i^1 L_i^1$	$L_i e$	w	5	–
4.	$W_i^2 L_i^2$	$L_i e$	$L_i^1 w$	7 or 6	–
5.	$l_k^0 l_k^0$	$L_i e$	$L_i^2 L_i^1 w$	9	D
6.	$l_k^1 e$	$L_i^2 L_i^2$	$l_k^0 L_i^1 w$	11	H
7.	$l_k^2 e$	cl_k^0	$L_i^1 w$	13	J
8.	$l_k^3 e$	cL_i^1	w	15	K
9.	$l_k u(l_k)$	ee	w	?	–

No program is needed in $P_1 \cup P_2$ to simulate the instruction l_h :*HALT*. The P colony Π starts its computation with object l_0 in the environment and it simulates the instruction labelled l_0. By the programs it places and deletes from the environment the objects a_r and it halts its computation only after object l_h appears in the environment. The result of computation is the number of copies of

object a_1 placed in the environment at the end of computation. No other halting computation can be executed in the P colony.

4 P colonies with rewriting/communication rules

In this section we deal with P colonies with passive environment and with one object inside each agent. We prove that such a P colony with three agents can generate every recursively enumerable set of natural numbers.

Theorem 3. $NEPCOL(1, 3, *, no\text{-}check, pas, ini) = \mathbb{N}\cdot\mathsf{RE}$.

Proof. Let us consider register machine $M = (m, H, l_0, l_h, P)$. For all labels from the set H we construct corresponding objects in P colony Π. The content of register r will be represented by the number of copies of objects a_r placed in the environment.

We construct the P colony $\Pi = (\Sigma, e, a_1, d, D_E, B_1, B_2, B_3)$ with:

- $\Sigma = \{l_i, l_i', l_i'', \overline{l_i}, \overline{\overline{l_i}}, \underline{l_i}, l_i^1, l_i^2, l_i^3, l_i^4, M_i, M_i^1, M_i^2, M_i^3, M_i^4, N_i, N_i^1, N_i^2, N_i^3, N_i^4 \mid l_i \in H\} \cup \{a_i \mid 1 \le i \le m\} \cup \{e, d, f, g\}$,
- $B_1 = (l_0, P_1)$,
- $B_2 = (d, P_2)$,
- $B_3 = (e, P_3)$.

The object l_0 corresponds to the label of the first instruction executed by the register machine.

The instruction $l_i : (ADD(r), l_j, l_k)$ will be simulated by the agents B_1 and B_2 by using following programs:

B_1 :

$1 : \langle l_i \to l_i' \rangle$;
$2 : \langle l_i' \leftrightarrow e \rangle$;
$3 : \left\langle e \to \underline{l_i''} \right\rangle$;
$4 : \left\langle \underline{l_i''} \to \underline{\underline{l_i''}} \right\rangle$;
$5 : \left\langle \underline{\underline{l_i''}} \leftrightarrow l_i''' \right\rangle$;
$6 : \left\langle \underline{\underline{l_i''}} \to e \right\rangle$;
$7 : \left\langle \overline{\overline{l_i'''}} \to l_i^{iv} \right\rangle$;
$8 : \langle l_i^{iv} \to l_i^v \rangle$;
$9 : \langle l_i^v \to l_j \rangle$;
$10 : \langle l_i^v \to l_k \rangle$;

B_2 :

$A : \langle d \leftrightarrow l_i' \rangle$;
$B : \langle l_i' \to l_i''' \rangle$;
$C : \langle l_i''' \leftrightarrow e \rangle$;
$D : \langle e \leftrightarrow l_i'' \rangle$;
$E : \langle l_i'' \to a_r \rangle$;
$F : \langle a_r \leftrightarrow d \rangle$;

The simulation of ADD-instruction starts by rewriting the object l_i to l_i' by the first agent. The agent B_2 consumes the object l_i', changes it to l_i''' and sends it to the environment. The agent B_1 rewrites the object e to some l_j'', for $l_j \in H$. If this l_j'' has the same index as l_i''' placed in the environment (i.e., $i = j$), the computation passes to the next phase. If $i \neq j$, the agent B_1 tries to generate another l_j''. When the computation gets over this checking step, agent B_2 generates one copy of object a_r and places it to the environment (adding 1 to the content of register i). Then agent B_1 non-deterministically chooses to generate object l_j or l_k.

The instruction $l_i : (SUB(r), l_j, l_k)$ is simulated by using the following rules and programs:

B_1 :

$11 : \langle l_i \to l'_i \rangle;$	$15 : \langle \underline{l''_i} \to l''_i \rangle;$	$19 : \langle l^{iv}_i \to M_i \rangle;$	$23 : \langle \underline{L_i} \leftrightarrow d \rangle;$
$12 : \langle l'_i \leftrightarrow e \rangle;$	$16 : \langle \underline{\underline{l''_i}} \leftrightarrow l'''_i \rangle;$	$20 : \langle M_i \leftrightarrow d \rangle;$	$24 : \langle d \leftrightarrow L^2_i \rangle;$
$13 : \langle e \to \underline{l''_i} \rangle;$	$17 : \langle l''_i \to e \rangle;$	$21 : \langle d \leftrightarrow L_i \rangle;$	$25 : \langle L^2_i \to l_j \rangle;$
$14 : \langle \underline{l''_i} \to \underline{\underline{l''_i}} \rangle;$	$18 : \langle l'''_i \to l^{iv}_i \rangle;$	$22 : \langle L_i \to \underline{L_i} \rangle;$	$26 : \langle d \leftrightarrow L^3_i \rangle;$
			$27 : \langle L^3_i \to l_k \rangle;$

B_2 :

$G : \langle d \leftrightarrow l'_i \rangle;$	$K : \langle l''_i \to L_i \rangle;$	$N : \langle a_r \to h \rangle;$	$Q : \langle L^2_i \leftrightarrow d \rangle;$
$H : \langle l'_i \to l'''_i \rangle;$	$L : \langle L_i \leftrightarrow a_r \rangle;$	$O : \langle h \leftrightarrow \underline{L_i} \rangle;$	$R : \langle M_i \to N_i \rangle;$
$I : \langle l'''_i \leftrightarrow e \rangle;$	$M : \langle L_i \leftrightarrow M_i \rangle;$	$P : \langle \underline{L_i} \to L^2_i \rangle;$	$S : \langle N_i \leftrightarrow d \rangle;$
$J : \langle e \leftrightarrow l''_i \rangle;$			

B_3 :

$A' : \langle e \leftrightarrow N_i \rangle;$	$C' : \langle L^3_i \leftrightarrow \underline{L_i} \rangle;$	$E' : \langle e \leftrightarrow h \rangle;$	$G' : \langle y \leftrightarrow M_i \rangle;$
$B' : \langle N_i \to L^3_i \rangle;$	$D' : \langle \underline{L_i} \to e \rangle;$	$F' : \langle h \to y \rangle;$	$H' : \langle M_i \to e \rangle;$

The simulation starts by generating the objects l''_i, l'''_i in the same way as in the addition part described above. Then the agent B_2 simulates subtraction (if the subtracted register is nonzero). If there was some a_r in the environment, the agent generates object L^2_i. This object agent B_1 can rewrite to l_j). If the register r was empty, the agent B_2 generates object N_i and this object can be rewritten by agent B_3 to object L^3_i. Finally, agent B_1 can rewrite object L^3_i to l_k.

If the register r stores nonzero value:

	B_1	B_2	B_3	Env	P_1	P_2	P_3
1.	l_i	d	e	$da_r^x w$	11	–	–
2.	l_i'	d	e	$da_r^x w$	12	–	–
3.	e	d	e	$l_i' da_r^x w$	13	G	–
4.	$\underline{l_i''}$	l_i'	e	$dda_r^x w$	14	H	–
5.	$\underline{l_i''}$	l_i'''	e	$dda_r^x w$	15	I	–
6.	$\underline{\underline{l_i''}}$	e	e	$l_i''' dda_r^x w$	16	–	–
7.	l_i'''	e	e	$l_i'' dda_r^x w$	18	J	–
8.	l_i^{iv}	l_i'''	e	$dda_r^x w$	19	K	–
9.	M_i	L_i	e	$dda_r^x w$	20	L	–
10.	d	a_r	e	$M_i L_i da_r^{x-1} w$	21	N	–
11.	L_i	h	e	$M_i dda_r^{x-1} w$	22	–	–
12.	$\underline{L_i}$	h	e	$M_i dda_r^{x-1} w$	23	–	–
13.	d	h	e	$\underline{L_i} M_i da_r^{x-1} w$	–	O	–
14.	d	$\underline{L_i}$	e	$h M_i da_r^{x-1} w$	–	P	E'
15.	d	L_i^2	h	$M_i da_r^{x-1} w$	–	Q	F'
16.	d	d	x	$L_i^2 M_i a_r^{x-1} w$	24	–	G'
17.	L_i^2	d	M_i	$dya_r^{x-1} w$	25	–	H'
18.	l_j	d	e	$dya_r^{x-1} w$	?	–	–

If the register r stores value zero:

	B_1	B_2	B_3	Env	P_1	P_2	P_3
1.	l_i	d	e	dw	11	–	–
2.	l_i'	d	e	dw	12	–	–
3.	e	d	e	$l_i'dw$	13	G	–
4.	l_i''	l_i'	e	ddw	14	H	–
5.	$\overline{l_i''}$	l_i'''	e	ddw	15	I	–
6.	$\overline{\overline{l_i''}}$	e	e	$l_i'''ddw$	16	–	–
7.	l_i'''	e	e	$l_i''ddw$	18	J	–
8.	l_i^{iv}	l_i'''	e	ddw	19	K	–
9.	M_i	L_i	e	ddw	20	–	–
10.	d	L_i	e	M_idw	–	M	–
11.	d	M_i	e	L_iddw	21	R	–
12.	L_i	N_i	e	ddw	22	S	–
13.	$\underline{L_i}$	d	e	N_idw	23	–	A'
14.	d	d	N_i	$\underline{L_i}w$	–	–	B'
15.	d	d	L_i^3	$\underline{L_i}w$	–	–	C'
16.	d	d	$\underline{L_i}$	L_i^3w	26	–	D'
17.	L_i^3	d	e	dw	27	–	–
18.	l_k	d	e	dw	?	–	–

No program is needed in $P_1 \cup P_2 \cup P_3$ to simulate the instruction l_h :$HALT$. When l_h appears, the computation halts since no agent can execute a program. The result is the number of objects a_1 placed in the environment and it corresponds to the result of a successful computation of the register machine.

5 Conclusions

In this paper we presented the results obtained during the research of P colonies with passive environment. We have shown that P colonies with with one consumer and one sender agent can generate all sets of natural numbers computable by register machines.

Analogously, if we place three agents with one object inside each of them and with no-checking rewriting/communication programs into the passive environment, the obtained P colony is again computationally complete in the Turing sense.

Acknowledgments.

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4Innovations excellence

in science - LQ1602, and by the Silesian University in Opava under the Student Funding Scheme, project SGS/13/2016.

References

1. L. Cienciala and L. Ciencialová. Eco-P colonies. In G. Păun, M. Pérez-Jiménez, and A. Riscos-Núñez, editors, *Pre-Proceedings of the 10th Workshop on Membrane Computing, Curtea de Arges, Romania*, pages 201–209, 2009.
2. L. Cienciala and L. Ciencialová. P colonies and their extensions. In J. Kelemen and A. Kelemenová, editors, *Computation, Cooperation, and Life – Essays Dedicated to Gheorghe Paun on the Occasion of His 60th Birthday*, volume 6610 of *Lecture Notes in Computer Science*, pages 158–169, Berlin Heidelberg, 2011. Springer-Verlag.
3. L. Cienciala, L. Ciencialová, and A. Kelemenová. On the number of agents in P colonies. In *Membrane Computing*, volume 4860 of *LNCS*, pages 193–208. Springer, 2007.
4. L. Ciencialová, L. Cienciala, E. Csuhaj-Varjú, A. Kelemenová, and V. György. On very simple P colonies. In *Proceeding of The Seventh Brainstorming Week on Membrane Computing*, volume 1, pages 97–108, 2009.
5. L. Ciencialová, E. Csuhaj-Varjú, A. Kelemenová, and G. Vaszil. Variants of P colonies with very simple cell structure. *Int. J. of Computers, Communications & Control*, 3(IV):224–233, 2009.
6. R. Freund and M. Oswald. P colonies working in the maximally parallel and in the sequential mode. In *Proceedings - Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2005*, pages 419–426, Sept 2005.
7. J. Kelemen and A. Kelemenová. A grammar-theoretic treatment of multiagent systems. *Cybern. Syst.*, 23(6):621–633, Nov. 1992.
8. J. Kelemen and A. Kelemenová. On P colonies, a biochemically inspired model of computation. *Proc. of the 6th International Symposium of Hungarian Researchers on Computational Intelligence*, pages 40–56, 2005.
9. J. Kelemen, A. Kelemenová, and G. 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)*, pages 82–86. Boston, Mass, 2004.
10. M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.
11. G. Păun, G. Rozenberg, and A. Salomaa. *The Oxford Handbook of Membrane Computing*. Oxford University Press, Inc., New York, NY, USA, 2010.