P Colonies with a Bounded Number of Cells and Programs

Erzsébet Csuhaj-Varjú[1]

Maurice Margenstern [2]

György Vaszil[1]

[1]Computer and Automation Research Institute
Hungarian Academy of Sciences
Kende utca 13–17, H-1111 Budapest, Hungary
E-mail:{csuhaj,vaszil}@sztaki.hu

[2]Université Paul Verlaine - Metz, LITA, EA 3097
Île du Saulcy, 57045 Metz Cedex 1, France
E-mail: margens@univ-metz.fr

Motivations

P colonies intend to model

- a **community of very simple cells** dynamically changing and interacting with a dynamically changing **common, shared environment**,
- a **class of collection of simple membrane systems** *similar to the so-called colonies* of simple formal grammars that are models of **very simple reactive agents** with **emergent behaviour**, and
- **distributed and communicating systems** of as **simple computing agents** as possible.

(Kelemen, Kelemenová, Păun, 2004)

The Model

P colonies consist of **very simple computational devices** (elementary cells) **restricted** in both their **forms and functioning capabilities**.

- At any moment of time, each cell is allowed to contain only a **certain, fixed number** (two or three) **of objects.** The cell itself is not structured.
- Any **rule** of the cell is able to **modify only one object**, either by **rewriting** (point mutation) or by **communication with the environment** (exchange of objects with the environment).
- **Alternative rule application** is allowed by combination of two rules: if the first rule is not applicable then the second one should be applied. (**checking rules**)
- A set of **programs composed of rules** is associated to each cell. A **program** is a set of **rules that have to be performed in parallel to the objects** found inside the cell.

  (In the case of systems consisting of cells with only two objects inside, each program consists of two rules; when considering cells with three objects inside, then the programs consist of three rules.)

Computation

The cells of a P colony execute a **computation by synchronously applying their programs** to the objects inside the cells and objects outside in the environment.

Result of the computation

The **result of the computation** is read as the number of **certain types of objects present in the environment** when no more rules can be applied.

Power and Size of P colonies

- **P colonies** are able to compute **any recursively enumerable set of numbers**.
  - The statement holds even if at the starting configuration the P colonies contain an **infinite number of copies of a certain object in the environment** and **two or three copies of the same object inside the cells**.

    In this case, to reach this power, either the **number of necessary cells** or the number of necessary programs is **unbounded**.

    (Kelemen, Kelemenová, Păun, 2004)

    (Csuhaj-Varjú, Kelemen, Kelemenová, Păun, Vaszil, 2005)
  - **P colonies** are also able to compute any recursively enumerable set of numbers when being restricted in certain size parameters if they use a **functioning mode different from the generic one**.

    (Freund, Oswald, 2005)

Question

Are **P colonies** able to compute **any recursively enumerable set of natural numbers** with **a bounded number of cells and a bounded number of programs in each cell**?

Answer

We prove that the **answer is positive**, moreover, we **give bounds** for the number of cells and the number of programs in each cell.

The values of the bounds presented in our results depend on the type of rules the P colonies are allowed to use and seem to suggest **a trade-off** between the **number of necessary cells** and **the number of necessary programs in each cell**.

P colony - the Formal Model

$\Pi = (V, e, o_f, I_E, C_1, \ldots, C_n),\ n \geq 1$, where

- V is an alphabet (its elements are called **objects**),
- e (the **environmental object**) and o_f (the **final object**) are two distinguished objects of V,
- $I_E \in (V - \{e\})^\circ$ is a finite multiset of **objects initially present in the environment** besides the **infinitely many copies of** e,
- and $C_1, \ldots, C_n$ are the **cells** of the colony.

Cell

$C_i = (O_i, P_i)$, $1 \leq i \leq n$, where

- O_i is a multiset over $\{e\}$ having the same cardinality for all i, $1 \leq i \leq n$, called the **initial state of the cell**,
- P_i is a finite set of **programs**; each program is a set of rules of the forms
  - $a \to b$ - **internal point mutation**,
  - $c \leftrightarrow d$ - **one object exchange with the environment**,
  - $c \leftrightarrow d / c' \leftrightarrow d'$ - **checking rule for one object exchange with the environment**, or
  - $c \leftrightarrow d / a \to b$ - **checking rule for one object exchange with the environment or internal point mutation**,

  where $a, b, c, d, c', d' \in V$.

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, $1 \leq i \leq n$.

Restricted program

It contains

- **one point mutation rule** of the form $a \rightarrow b$, and
- – **either one exchange rule** of the form $c \leftrightarrow d$, or
  – **one checking rule** of the form $c \leftrightarrow d/c' \leftrightarrow d'$.

A **P colony** is called **restricted** if it contains **two objects in each cell** and **has only restricted programs**.

Two object colonies with non-restricted programs, or **three object colonies** are called **non-restricted.**

Functioning of the P-colony

- The **programs of the cells** are used in the **non-deterministic maximally parallel way** usual in membrane computing: in each time unit, each cell which can use one of its programs should use one.
- When **using a program**, **each of its rules must be applied to distinct objects** of the cell.
- In this way, we get **transitions** among the **configurations** of the colony.
- A sequence of transitions is a **computation**.
- A **computation is halting** if it reaches a configuration where no cell can use any program.
- The **result of a halting computation** is the **number of copies of the object o_f present in the environment** in the halting configuration.

Configuration

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

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

where

- w_i represents the **multiset of objects from cell** C_i, $1 \leq i \leq n$ ($w_i \in V^2$ in two-objects colonies and $w_i \in V^3$ in three-objects colonies), and
- $w_E \in (V - \{e\})^*$ represents the **multiset of objects from the environment different object** e.

The **initial configuration** is $(ee, \ldots, ee; I_E)$ in the case of two-objects colonies and $(eee, \ldots, eee; I_E)$ in the case of three-objects colonies where $I_E \in (V - \{e\})^*$.

Auxiliary notations

Let the programs of each P_i be labeled in a one-to-one manner by labels in the set $lab(P_i)$ in such a way that $lab(P_i) \cap lab(P_j) = \emptyset$ for $i \neq j,\ 1 \leq i, j \leq n$.

For a rule r, and a multiset $w \in V^\circ$,

- $left(r, w) = a$ and $right(r, w) = b$
  if r is a point mutation rule $r = (a \to b)$, or a
  checking rule $r = (c \leftrightarrow d / a \to b)$ with $d \notin w$,
- and let $left(r, w) = right(r, w) = \varepsilon$ otherwise.

For a rule r and a multiset $w \in V^\circ$

- $export(r, w) = c$ and $import(r, w) = d$
  if r is an exchange rule $r = (c \leftrightarrow d)$, or a
  checking rule $r = (c \leftrightarrow d / c' \leftrightarrow d')$ with $d \in w$.

If r is a checking rule as above with $d \notin w$ but $d' \in w$, then let $export(r, w) = c'$, $import(r, w) = d'$.
Let $export(r, w) = import(r, w) = \varepsilon$ in all other cases.

For a program p and any $\alpha \in \{left, right, export, import\}$, let $\alpha(p, w) = \bigcup_{r \in p} \alpha(r)$.

Transition

A **transition** from a **configuration to another one** is denoted as

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

where

- There is a set of program labels P with $|P| \leq n$, such that for $p, p' \in P$, $p \neq p'$, $p \in lab(P_j)$ implies $p' \notin lab(P_j)$, and for each $p \in P$, $p \in lab(P_j)$, $left(p, w_E) \cup export(p, w_E) = w_j$, and $\bigcup_{p \in P} import(p, w_E) \subseteq w_E$.

- The chosen set P is **maximal**, that is, if any other program $r \in \bigcup_{1 \leq i \leq n} lab(P_i)$, $r \notin P$, is added to P, then the conditions above are not satisfied.

For each j, $1 \leq j \leq n$, for which there exists a $p \in P$ with $p \in lab(P_j)$, let

$$w'_j = right(p, w_E) \cup import(p, w_E).$$

If there is no $p \in P$ with $p \in lab(P_j)$ for some j, $1 \leq j \leq n$, then let

$$w'_j = w_j,$$

and moreover, let

$$w'_E = w_E - \bigcup_{p \in P} import(p, w_E) \cup \bigcup_{p \in P} export(p, w_E).$$

A configuration is **halting** if the set of program labels P satisfying the conditions above cannot be chosen to be other than the empty set, $\emptyset$.

Result of the computation

The **set of numbers computed by a P colony** Π is defined as

$$N(\Pi) = \{|v_E|_{o_f} \mid (w_1, \ldots, w_n, I_E) \Rightarrow^* (v_1, \ldots, v_n, v_E)\}$$

where $(w_1, \ldots, w_n, I_E)$ is the initial configuration, $(v_1, \ldots, v_n, v_E)$ is a halting configuration, and $\Rightarrow^*$ denotes the reflexive and transitive closure of $\Rightarrow$.

Example

Cell

$$C = (ee, \{(e \to a, e \leftrightarrow e), (e \to a, a \leftrightarrow e), (e \to b, a \leftrightarrow e)\}$$

The environment initially: $I_E = c$

Derivation

$$(c, (ee)) \Longrightarrow (c, (ae)) \Longrightarrow (ca, (ea)) \Longrightarrow (caa, (ae)) \Longrightarrow (caaa, (ba))$$

Notations

The **family of all sets of numbers** computed as above by P colonies with k-objects ($k = 2, 3$) of degree at most $n \geq 1$ having at most $h \geq 1$ programs in the cells without using checking rules, is denoted by $IPCol\_k(n, h)$.

In the case of **two-objects colonies**, if we use **only restricted programs**, then we write $IPCol\_2R$ instead of $IPCol\_2$.

If **checking rules** are allowed, then we write $IPCol\_Ch\_k$ instead of $IPCol\_k$.

(Thus, for instance, $IPCol\_Ch\_2R(n, h)$ will be the family of numbers computed by restricted two-objects P colonies with at most n cells, each having at most h programs where the use of checking rules is allowed.)

Aim

Comparison of families

$IPCol\_\alpha(n,h)$ and $\mathbb{N}RE$, where $\alpha \in \{2, 3, 2R, Ch\_2, Ch\_3, Ch\_2R\}$

Method of Comparison

Simulation of register machines by P colonies

Register Machines

A **register machine** consists of

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

Instructions of a register machine

There are several types of instructions which can be used.

- l_i : ($\texttt{ADD}(r), l_j, l_k$) - **add** 1 to register r and then go to one of the instructions with labels l_j or l_k, non-deterministically chosen,
- l_i : ($\texttt{SUB}(r), l_j$) - if register r is **non-empty, then subtract**1 from it, otherwise **leave it unchanged**, and go to the instruction with label l_j in both cases,
- l_i : ($\texttt{CHECK}(\texttt{r}), l_j, l_k$) - if the **value** of register r is **zero**, go to instruction l_j, otherwise go to l_k,
- l_i : ($\texttt{CHECKSUB}(r), l_j, l_k$) - if register r is **non-empty, then subtract 1** from it and go to the instruction with label l_j, otherwise go to the instruction with label l_k,
- l_h : `HALT` - halt the machine.

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.

Computation by a register 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.

The power of register machines

- We can compute all sets of numbers which are Turing computable by using instructions of type `ADD`, `CHECKSUB`, and `HALT`.

  (Minsky, 1967)

- There exist universal register machines with a small number of registers and a small number of instructions, the exact numbers depend on the chosen set of instructions and the chosen notion of universality.

Small universal register machines by Ivan Korec (1996)

Let $\mathbb{M}$ be the set of register machines. Then, there are **register machines** $U_1,\ U_2,\ U_3$ with **8 registers** and a **recursive function** $g : \mathbb{M} \to \mathbb{N}$ such that for each $M \in \mathbb{M}$,

$$N(M) = N(U_i(g(M)))$$

where $N(U_i(g(M)))$ denotes the **set of numbers computed by** $U_i,\ 1 \leq i \leq 3$**,** with **initially containing** $g(M)$ in the second register.

All these machines have one `HALT` instruction labeled by l_h, one instruction of the type `ADD` labeled by l_0, and

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

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

Figure 1: Flowchart of strongly universal machine U_{32}

Ideas of the constructions

- The idea is to simulate any partial recursive function by register machines belonging to a **very restricted class of machines**, called $R3a$-machines (with 3 instructions) which simulate any partial recursive function.

- It can be proved that functions on the integers $x \mapsto 2^x$ and $x \mapsto \log_2 x$ can also be computed by $R3a$-machines.

- The idea of the simulation is to encode the list of instructions into a finite sequence of numbers which can be extracted from a unique positive integer x, according to number theoretic considerations.

- An additional property is used: it is required that the expected numbers are realised as the remainders of $x{+}1$ modulo non-divisors of $x{+}1$ whose ranks represent the instructions.

  If $F(x, y)$ denotes the function given by $(x, y) \mapsto (x{+}1) \bmod \mathbf{nd}(x{+}1, y)$, where $\mathbf{nd}(z, j)$ is the j^{th} non-divisor of z, then it is not difficult to compute $F(x, i)$: we stop when the i^{th} non-zero test of the division of $x{+}1$ by k is positive, k being incremented at each test.

- Next, both the next state j of an instruction and the instruction I itself is coded as $3j + \nu(I)$ where $\nu(I)$ ranges in $\{0..2\}$ as only three types of instructions are used.

- The universal machine consists of three blocks of instructions.

  The first bloc extracts $F(x, i)$ and gives it as k.

  The second bloc extracts j and $\nu(I)$ from $k = 3j + \nu(I)$.

  Then, instruction I is executed in the third block, possibly transforming j into $j{-}1$, depending on I. And as, the new label is known, we find the next code of an instruction by computing $F(x, j)$.

- There are infinitely many representations of a given $R3a$-machines by a number x. The set of all the representations is not recursively enumerable but it contains recursive subsets in which there is at least one representative for each machine.

A necessary modification for P colonies

These machines compute functions of non-negative integers in such a way that the **argument of the function is initially** present in the **third register**, thus we **need modifications in our constructions** to conform to the number generating definition we have given above.

We **add a new start label** l_0' and a **separate non-deterministic** `ADD` **instruction**, $l_0' : (\texttt{ADD}(r_3), l_0', l_0)$, to **produce an argument** $x \in \mathbb{N}$ in the **third register** before the actual computation begins, that is, to make the resulting universal machine generate any value from the range of the function computed by the simulated register machine.

The universality of P colonies with a bounded number of cells

P colonies with checking rules

Restricted Programs

1. $IPCol\_Ch\_2R(23,5) = IPCol\_Ch\_2R(22,6) = \mathbb{N}RE$.
2. $IPCol\_Ch\_2R(1,142) = \mathbb{N}RE$.

Non-Restricted Programs

1. $IPCol\_Ch\_2(22,5) = \mathbb{N}RE$.

Proof idea of $IPCol\_Ch\_2R(23,5) = \mathbb{N}RE$.

- Let $L \in \mathbb{N}RE$ and let M be a register machine with $L = N(M)$.
- We **simulate the universal register machines** from the main Theorem in **(Korec, 1996)**.
- By **placing the code of** M, the value $g(M)$, in the **second register**, they compute $N(M)$, producing the **result** in their **first** register.
- We **construct P colonies** which **simulate** the computation of U_2 and U_3.
- Consider a **P colony** $(V, e, a_1, I_E, C_1, \ldots, C_n)$ where V contains the **special object** e, two symbols l_i and l_i' for **each instruction label** l_i of the universal machine, and **one symbol** $a_j,\ 1 \leq j \leq 8$, **for each register**.
- The **number of** a_j **symbols in the environment** corresponds to the **value of register** j**.**
- The **initial contents of the environment**, I_E is $g(M)$ **copies of the object** a_2 **and the label of the initial instruction** l_0**.**
- The **result** of the computation can be read as the **number of** a_1 **objects** corresponding to the value of the first register in a halting configuration.

Simulation of $l_i : (\texttt{ADD}(r), l_{i,1}, l_{i,2})$

This means **increasing the number of objects corresponding to the value of register r by one** and by **changing the instruction label** l_i present in the environment to $l_{i,1}$ or $l_{i,2}$.

The simulating cell

$$\begin{aligned} P_i \;=\; & \{\langle e \to a_r; e \leftrightarrow l_i\rangle,\ \langle l_i \to l_{i,1}; a_r \leftrightarrow e\rangle,\ \langle e \to e; l_{i,1} \leftrightarrow e\rangle, \\ & \langle l_i \to l_{i,2}; a_r \leftrightarrow e\rangle,\ \langle e \to e; l_{i,2} \leftrightarrow e\rangle\}. \end{aligned}$$

Simulation (Example)

$$\begin{aligned} & (\ldots,(e,e),\ldots,u_E l_i) \Longrightarrow (\ldots,(a_r,l_i),\ldots,u_E) \\ \Longrightarrow\ & (\ldots,(e,l_{i,1}),\ldots,u_E a_r) \Longrightarrow (\ldots,(e,e),\ldots,u_E a_r l_{i,1}) \end{aligned}$$

Simulation of $l_i : (\texttt{CHEKSUB}(r), l_{i,1}, l_{i,2})$

This means **exchanging the label l_i to $l_{i,1}$**and **decreasing the number of a_r objects by one**, or **if the number of a_r objects is zero**, then **exchange l_i to $l_{i,2}$.**

The simulating cell

$$\begin{aligned} P_i \;=\; & \{\langle e \to e; e \leftrightarrow l_i\rangle,\ \langle l_i \to l_{i,1}; e \leftrightarrow a_r / e \leftrightarrow e\rangle,\ \langle a_r \to e; l_{i,1} \leftrightarrow e\rangle, \\ & \langle l_{i,1} \to l_{i,2}; e \leftrightarrow e\rangle,\ \langle e \to e; l_{i,2} \leftrightarrow e\rangle\}. \end{aligned}$$

Simulation (Examples)

$$\begin{aligned} & (\ldots,(e,e),\ldots,u_E a_r l_i) \Longrightarrow (\ldots,(e,l_i),\ldots,u_E a_r) \\ \Longrightarrow\ & (\ldots,(a_r,l_{i,1}),\ldots,u_E) \Longrightarrow (\ldots,(e,e),\ldots,u_E l_{i,1}) \end{aligned}$$

(The value of register r was decreased by one and the next instruction label is $l_{i,1}$.)

$$\begin{aligned} & (\ldots,(e,e),\ldots,u_E l_i) \Longrightarrow (\ldots,(e,l_i),\ldots,u_E) \\ \Longrightarrow\ & (\ldots,(e,l_{i,2}),\ldots,u_E) \Longrightarrow (\ldots,(e,e),\ldots,u_E l_{i,2}) \end{aligned}$$

(The value of register r was zero and the next instruction label is $l_{i,2}$.)

Simulation of $l_i : (\texttt{CHECK}(r), l_{i,1}, l_{i,2})$

This means **checking whether the value of register r is empty or not**. This simulated by six programs as follows.

The simulating cell

$$\begin{aligned} P_i \;=\; & \{\langle e \to e; e \leftrightarrow l_i\rangle,\ \langle l_i \to l'_{i,1}; e \leftrightarrow a_r / e \leftrightarrow e\rangle,\ \langle l'_{i,1} \to l_{i,1}; a_r \leftrightarrow e\rangle, \\ & \langle e \to e; l_{i,1} \leftrightarrow e\rangle,\ \langle l'_{i,1} \to l_{i,2}; e \leftrightarrow e\rangle,\ \langle e \to e; l_{i,2} \leftrightarrow e\rangle\}. \end{aligned}$$

Simulation (Examples)

$$\begin{aligned} (\ldots,(e,e),\ldots,u_E a_r l_i) &\Longrightarrow (\ldots,(e,l_i),\ldots,u_E a_r) \Longrightarrow (\ldots,(a_r,l'_{i,1}),\ldots,u_E) \\ &\Longrightarrow (\ldots,(e,l_{i,1}),\ldots,u_E a_r) \Longrightarrow (\ldots,(e,e),\ldots,u_E a_r l_{i,1}) \end{aligned}$$

(The value of register r was not zero and the next instruction label is $l_{i,1}$.)

$$\begin{aligned} (\ldots,(e,e),\ldots,u_E l_i) &\Longrightarrow (\ldots,(e,l_i),\ldots,u_E) \Longrightarrow (\ldots,(e,l'_{i,1}),\ldots,u_E) \\ &\Longrightarrow (\ldots,(e,l_{i,2}),\ldots,u_E) \Longrightarrow (\ldots,(e,e),\ldots,u_E l_{i,2}) \end{aligned}$$

(The value of register r was zero and the next instruction label is $l_{i,2}$.)

There are **no programs for the halting label l_h,** thus, its appearance **ends the computation** which otherwise never stops.

Simulation of U_2

$$\Pi_2 = (V, e, a_1, I_E, C_0, \ldots, C_{22}),$$

where

- V and I_E as above,
- and one cell with the programs P_0 for the initial `ADD` instruction labeled with l_0 which fills the input register,
- 9 cells with P_i, $1 \leq i \leq 9$, for the simulation of the rest of the `ADD` instructions, and
- 13 cells with $10 \leq i \leq 22$, for simulating the `CHECKSUB` instructions.

This gives us $1 + 9 + 13 = 23$ cells with at most 5 programs.

Simulation of U_3

$$\Pi_3 = (V, e, a_1, I_E, C_0, \ldots, C_{21})$$

where

- V, I_E and P_0 are as above,
- 8 cells with $P_i,\ 1 \leq i \leq 8$, for the simulation of the `ADD` instructions,
- 1 cell with P_9 for the `CHECK` instruction, and
- 12 cells with $P_i,\ 10 \leq i \leq 21$, for simulating the `CHECKSUB` instructions.
- This gives us $1 + 8 + 1 + 12 = 22$ cells, this time with at most 6 programs.

P colonies without checking rules

The use of checking rules can be avoided:

- $IPCol\_2(35, 8) = IPCol\_2R(57, 8) = \mathbb{N}RE$.
- $IPCol\_3(35, 7) = \mathbb{N}RE$.

Proof idea of $IPCol\_2(35, 8) = \mathbb{N}RE$

- **Universal register machines** U_1 and U_3 from (Korec, 1996) are simulated.
- The `ADD` **instructions** are simulated **without checking rules**, thus we can take cells from the previous constructions.
- The **simulation of** `CHECK` **and** `CHECKSUB` **instructions** need **two cooperating cells** per each instruction.

Simulation of $l_9 : (\texttt{CHECK}(r), l_{9,1}, l_{9,2})$

We need **two cells** having **at most 8 restricted programs** as follows.

The simulating cells

$$\begin{aligned} P'_9 \;=\; & \{\langle e \to l'_{9,2}; e \leftrightarrow l_9\rangle,\ \langle l_9 \to l'_9; l'_{9,2} \leftrightarrow e\rangle,\ \langle l'_9 \to l''_9; e \leftrightarrow a_r\rangle, \\ & \langle l''_9 \to l_{9,1}; a_r \leftrightarrow e\rangle,\ \langle l_{9,1} \to l_{9,1}; e \leftrightarrow l''_{9,2}\rangle,\ \langle l'_9 \to l_{9,2}; e \leftrightarrow l''_{9,2}\rangle, \\ & \langle l''_{9,2} \to e; l_{9,1} \leftrightarrow e\rangle,\ \langle l''_{9,2} \to e; l_{9,2} \leftrightarrow e\rangle\}, \end{aligned}$$

$$P'_{22} \;=\; \{\langle e \to l''_{9,2}; e \leftrightarrow l'_{9,2}\rangle,\ \langle l'_{9,2} \to e; l''_{9,2} \leftrightarrow e\rangle\}.$$

The **interplay of these two cells** produces the desired "checking" effect, they **exchange the symbol** l_9 **to** $l_{9,1}$ if there is **at least one** a_r **symbol in the environment**, otherwise $l_{9,2}$ is released.

Simulation of $l_i : (\texttt{CHECKSUB}(r), l_{i,1}, l_{i,2})$

In this case we also need **two cells** with **at most eight programs** as follows.

The simulating cells

$$\begin{aligned} P'_i \;=\; & \{\langle e \to l'_{i,2}; e \leftrightarrow l_i\rangle,\ \langle l_i \to l'_i; l'_{i,2} \leftrightarrow e\rangle,\ \langle l'_i \to l''_i; e \leftrightarrow a_r\rangle, \\ & \langle a_r \to e; l''_i \to l_{i,1}\rangle,\ \langle l_{i,1} \to l_{i,1}; e \leftrightarrow l''_{i,2}\rangle,\ \langle l'_i \to l_{i,2}; e \leftrightarrow l''_{i,2}\rangle, \\ & \langle l''_{i,2} \to e; l_{i,1} \leftrightarrow e\rangle,\ \langle l''_{i,2} \to e; l_{i,2} \leftrightarrow e\rangle\}, \end{aligned}$$

$$P'_{13+i} \;=\; \{\langle e \to l''_{i,2}; e \leftrightarrow l'_{i,2}\rangle,\ \langle l'_{i,2} \to e; l''_{i,2} \leftrightarrow e\rangle\}.$$

These **pairs work** together very **similarly to the previous ones**. However, they **not only check**, but if possible, also **decrease the number of a_r symbols** in the environment.

Conclusions

- **P colonies** are able to **simulate universal register machines**, provided they are initialized as follows: besides the environmental object, a finite number of objects are placed in the environment.
- Thus, **P colonies** are able to generate any recursively enumerable set of nonnegative integers with a **bounded number of cells, each containing a bounded number of programs of a bounded length**.
- Our results are only a **first approximation of the number of necessary cells and programs**. To decrease the present bounds, or to prove that they are sharp ones, remains the topic of further research.

References

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

[2] Csuhaj-Varjú, E., Kelemen, J., Kelemenová, A., Păun, Gh., Vaszil, Gy.: Computing with cells in environment: P colonies. Journal of Multiple Valued Logic and Soft Computing (to appear)

[3] Kelemen, J., Kelemenová, A.: A grammar-theoretic treatment of multiagent systems. Cybernetics and Systems **23** (1992) 621–633

[4] Kelemen, J., Kelemenová, A., Păun, Gh.: 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) (M. Bedau et al., eds.) Boston Mass. (2004) 82–86

[5] Korec, I.: Small universal register machines. Theoretical Computer Science **168** (1996) 267-301

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

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