APCol Systems with Teams

Lucie Ciencialová, Luděk Cienciala, and Erzsébet Csuhaj-Varjú

[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] Department of Algorithms and Their Applications, Faculty of Informatics
ELTE Eötvös Loránd University, Budapest, Hungary,
Pázmány Péter sétány 1/c, 1117
csuhaj@inf.elte.hu

Abstract. In this paper, we investigate the possibility of "going beyond" Turing in the terms of Automaton-like P Colonies (APCol systems, for short), variants of P colonies processing strings as their environments. We use the notion of teams of agents as a restriction for the maximal parallelism of the computation. In addition, we assign a colour to each team. In the course of the computation, the colour is changing according to the team that is currently active. We show that we can simulate red-green counter machines with APCol systems with two-coloured teams of minimal size. Red-green counter machines are computing devices with infinite run on finite input that exceed the power of Turing machines.

Keywords: Automaton-like P colonies, APCol systems, red-green counter machine, unbounded computation, teams

1 Introduction

Recently, both unconventional Turing equivalent computing devices and computational models which "go beyond" Turing, i.e., which are able to compute more than recursively enumerable sets of strings or numbers are in the focus of interest. In membrane computing, we can find examples for both types of such constructs.

APCol systems (Automaton-like P colonies) were introduced in [5] as an extension of P colonies (introduced in [9]) - a very simple variant of membrane systems inspired by colonies of formal grammars. (The reader is referred to [14] for more information in membrane systems and to [10] and [7] for details on grammar systems theory.) An APCol system consists of a finite number of agents - finite collections of objects embedded in a membrane - and a shared environment. The agents are equipped with programs which are composed from rules that allow them to interact with their environment that is represented by a string. For this reason, the agents use their own objects and the objects of the environment. The number of objects inside each agent is set by definition and it

is usually a very small number: 1, 2 or 3. The environmental string is processed by the agents and it is used as a communication channel for the agents as well. Through the string, the agents are able to affect the behaviour of another agent.

The activity of the agents is based on rules that can be rewriting, communication or checking rules [9]. A rewriting rule $a \to b$ allows the agent to rewrite (evolve) one object a to object b. Both objects are placed inside the agent. Communication rule $c \leftrightarrow d$ makes possible to exchange object c placed inside the agent with object d in the string. A checking rule is formed from two rules r_1, r_2 of type rewriting or communication. It sets a kind of priority between the two rules r_1 and r_2. The agent tries to apply the first rule and if it cannot be performed, then the agent executes the second rule. The rules are combined into programs in such a way that all objects inside the agent are affected by execution of the rules. Consequently, the number of rules in the program is the same as the number of objects inside the agent.

The interested reader can find more details on P colonies in [14], [8] and [4].

In this paper, we focus on APCol systems with agents forming teams; the concept was first proposed in [6]. The team is a finite number of agents of the APCol system. These collections can be so-called prescribed teams (given together with the components of the APCol system) or so-called free teams where only the size of the teams, i.e., the number of the agents in the team is given in advance. The notion is inspired by the concept of team grammar systems (see [15]). APCol systems with prescribed or with free teams function in the following manner: in every computation step only one team is allowed to work (only one team is active) and all of its components should perform a program in parallel.

Another interesting extension is to assign colours to programs, instructions or rules and observing how the currently used colour changes under the computation. This method is well-known for observing unbounded computations. Motivated by the notion of red-green Turing machines [12] (red-green register machines) and related notions in P systems theory [2], we introduce the concept of APCol systems with coloured teams. These constructs are APCol systems with teams where each team is associated with a colour. A string is accepted by an APCol system with coloured teams, if starting with the string as initial string the computation is unbounded and its teams with the final colour are active in an infinite number of steps and the teams of the other colours are active only in a finite number of steps.

Red-green Turing machines, introduced in [12] exceed the power of Turing machines since they recognize exactly the Σ_2-sets of the Arithmetical Hierarchy. These machines are deterministic and their state sets are divided into two disjoint sets, called the set of red states and the set of green states. Red-green Turing machines work on finite input words with the following recognition criterion on infinite runs: no red state is visited infinitely often and one or more green states are visited infinitely often. A change from a green state to a red state or reversely is called a mind change; we may speak of a change of the "colour". In [12], it is shown that every recursively enumerable language can be recognized by a red-green Turing machine with one mind change. It is also proved that if more than

one mind changes may take place, then red-green Turing machines are able to recognize the complement of any recursively enumerable language.

Our paper is structured as follows: the second section is devoted to definitions and notations used in the paper. The third section contains results obtained on APCol systems with coloured teams, namely, we show that any red-green counter machine can be simulated with an APCol system with coloured teams, where there are two colours. The teams either consist of only one agent and then the system works sequentially, or the APCol system has teams of at most two agents acting in parallel. Finally, some conclusions are derived.

2 Definitions

Throughout the paper we assume that the reader is familiar with the basics of formal language and automata theory; for further details consult [15]. We list the notations 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.

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

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 set V' is called the support of M and denoted by $supp(M)$ if for all $x \in V'$ $f(x) \neq 0$. 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 $V = \{a_i, \ldots, a_n\}$ 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 M, and ε represents the empty multiset.

2.1 Register and Counter Machines

We briefly recall the basic notions, following the notations used in [2].

A register machine [11] is a construct $M = (m, B, l_0, l_h, P)$, where m is the number of registers, B is a set of labels, $l_0 \in B$ is the initial label, $l_h \in B$ is the final label, and P is the set of instructions bijectively labelled by elements of B. The instructions of M can be of the following forms:

- $l_1 : (ADD(r), l_2, l_3)$, with $l_1 \in (B - \{l_h\})$, $l_2, l_3 \in B, 1 \leq j \leq m$. It increases the value of register r by one and the next instruction to be performed is non-deterministically chosen, it is labelled by l_2 or l_3. This instruction is called increment.
- $l_1 : (SUB(r), l_2, l_3)$, with $l_1 \in (B - \{l_h\})$, $l_2, l_3 \in B, 1 \leq j \leq m$. If the value of register r is zero, then the label of the next instruction to be performed is l_3; otherwise, the value of register r is decreased by one and the label of

the next instruction to be executed is l_2. The first case is called zero-test, the second case is called decrement.
- $l_h : HALT$. The register machine stops executing instructions.

A configuration of a register machine is described by the numbers stored in the registers and by the label of the next instruction to be performed. Computations start by executing the instruction l_0 of P, and terminate by execution of the $HALT$-instruction l_h.

This model of register machines can be extended by instructions for reading from an input tape and writing to an output tape containing strings over an input alphabet T_{in} and an output alphabet T_{out}, respectively, see [2]:

- $l_1 : (read(a), l_2)$, with $l_1 \in (B - \{l_h\}), l_2 \in B, a \in T_{in}$. This instruction reads symbol a from the input tape and the next instruction is l_2.
- $l_1 : (write(a), l_2)$, with $l_1 \in (B - \{l_h\}), l_2 \in B, a \in T_{out}$. This instruction writes symbol a to the output tape and the next instruction is l_2.

This extended register machine, working with strings is also called a counter automaton and is denoted by $M = (m, B, l_0, l_h, P, T_{in}, T_{out})$. If no output is written, T_{out} is not indicated.

It is known (see e.g. [11]) that register machines with (at most) three registers can compute all recursively enumerable sets of natural numbers. Counter automata with two registers can simulate the computations of Turing machines and thus characterize RE. All these results are obtained with deterministic register machines, where the ADD-instructions are of the form $l_1 : (ADD(r), l_2)$, with $l_1 \in (B - \{l_h\}), l_2 \in B, 1 \leq j \leq m$. More details can be found in [2].

2.2 Red-Green Turing Machines

We briefly recall the most important notions and statements concerning red-green Turing machines and their variants, following [12], [1], and [2].

Red-green Turing machines, introduced in [12], exceed the power of the standard Turing machines, since they recognize exactly the Σ_2-sets of the Arithmetical Hierarchy. As we told before, they are deterministic and their state sets are divided into two disjoint sets, namely, the set of red states and the set of green states. Red-green Turing machines work on finite inputs with the recognition criterion on infinite runs that no red state is visited infinitely often and one or more green states are visited infinitely often. A change from a green state to a red state or reversely is called a mind change; we may speak of a change of the "colour". In [12], it was shown that every recursively enumerable language can be recognized by a red-green Turing machine with one mind change. It was also proved that if more than one mind change may take place, then they are able to recognize the complement of any recursively enumerable language.

In the analogy of the concept of red-green Turing machines, red-green counter machines (red-green register machines) were defined and examined [1]. The authors proved that the computations of a red-green Turing machine TM can be simulated by a red-green register machine RM with two registers and with string

input in such a way that during the simulation of a transition of TM leading from a state p with colour c to a state p' with colour c' the simulating register machine uses instructions with labels (states) of colour c and only in the last step of the simulation changes the label (state) to colour c'. They showed that the reverse simulation works as well: the computations of a red-green register machine RM with an arbitrary number of registers and with string input can be simulated by a red-green Turing machine TM in such a way that during the simulation of a computation step of RM from an instruction with label (state) p with colour c to an instruction with label (state) p' with colour c', the simulating Turing machine TM are in states of colour c and only in the last step of the simulation changes to a state of colour c'.

In [2], the above notions were implemented for membrane systems: the notions of a red-green P automaton and its variants, as counterparts were introduced. It was shown that these devices are able to "go beyond" Turing, in the sense as red-green Turing machines are able to do.

2.3 APCol Systems

In the following we recall the concept of APCol systems, particular variants of P colonies, where the environment of the agents is given in the form of a string [5].

The agents of APCol systems contain objects, each object is 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 is of the form $a \rightarrow b$ and it is called an evolution rule. It means that object a inside of the agent is rewritten (evolved) to object b. The second type of rules is called a communication rule and it is in the form $c \leftrightarrow d$. When this rule is performed, then the object c inside the agent and a symbol d in the string are exchanged. If $c = e$, then the agent erases d from the input string and if $d = e$, then the symbol c is inserted into the string.

During the work of the APCol system, the agents perform programs. The number of objects inside the agents remain unchanged during the functioning of the system, it is usually 2.

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 substring 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 substring 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, the following types of programs with two communication rules are considered:

- $\langle a \leftrightarrow b; c \leftrightarrow e\rangle$ - 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 $\langle e \rightarrow e; c \leftrightarrow d\rangle$.

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 of the programs of the agents are restricted.

To help the reader in the easier understanding the technical details of the paper, we recall the formal definition of an APCol system.

Definition 1. *[5] An APCol system is a construct*
$\Pi = (O, e, A_1, \ldots, A_n)$, *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*
  - ω_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:*
    - $a \rightarrow b$, *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.

At the beginning of the computation of the APCol system is in initial configuration which is an $(n+1)$-tuple $c = (\omega; \omega_1, \ldots, \omega_n)$ where ω 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. The initial state of the environment does not contain object e.

A configuration of an APCol system Π 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.

In each computation step every agent attempts to find one of its programs to use. If it has applicable programs, then it non-deterministically chooses one of them and applies it. As usual in membrane computing, APCol systems work in the maximally parallel manner, i.e., as many agents perform one of its programs in parallel as possible. We note that other working modes can also be defined.

By applying programs, the APCol system 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.

The result of computation depends on the mode in which the APCol system works. In the accepting mode, a string ω is accepted by APCol system Π if there exists a computation by Π such that it starts in the initial configuration $(\omega; \omega_1, \ldots, \omega_n)$ and ends by halting in a configuration $(\varepsilon; w_1, \ldots, w_n)$,

where at least one of $w_i \in F_i$ for $1 \leq i \leq n$. In the generating mode, a string w_F is generated by Π if and only if there exists a 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 $1 \leq i \leq n$.

An APCol system Π can generate or accept a set of numbers as well, i.e., $|L(\Pi)|$.

In [5] the authors proved that the family of languages accepted by jumping finite automata (introduced in [13]) 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 APCol system with two agents.

In [3] the authors proved that restricted APCol systems with two agents working in the generating mode determine $\mathbb{N}\cdot\mathsf{RE}$, while if the APCol systems have only a single agent, then only a proper subset of $\mathbb{N}\cdot\mathsf{RE}$ can be obtained.

2.4 APCol Systems with Coloured Teams of Agents

As a restriction of the computation process, we can introduce teams into the concept of APCol system, as proposed in [6]. The team is a finite set of agents. These teams can be prescribed teams (given together with the components of the APC0L system) or free teams where only the size of the teams, i.e., the number of agents in the team is given in advance. The notion is inspired by the concept of team grammar systems (see [15]). APCol systems with prescribed or with free teams work in the following manner: at any computation step only one team is allowed to work (only one team is active) and all of its components should perform a program in parallel.

One other extension of the concept of APCol system is associating "colour" to the agents or, if the APCol system is with teams, to the teams. The concept is inspired by red-green Turing machines; the idea was first presented in [6], given in an informal manner, using only two colours, red and green.

Definition 2. *An APCol system with coloured teams is a construct*
$\Pi = (O, e, A_1, \ldots, A_n, C, f, B, B_{colours}, B_{teams})$, *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*
  - ω_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,
- C *is a set of labels of colours,*

- $f \in C$ *is the final colour,*
- B *is a set of labels of teams,*
- B_{colour} *is a set of pairs* (B_s, c_t) *assigning to every team its colour, where* $B_s \in B$, $c_t \in C$,
- B_{teams} *is a set of pairs* (A_i, B_s) *assigning the label of team* $B_s \in B$ *to each agent* A_i.

Accepting mode of infinite computations Due to the results of the computational power of APCol systems, it can easily be seen that for finite computations colours and teams do not add more, they can only be used for defining restricted classes of APCol systems. However, this is not the case if we consider unbounded computations.

We say that a string is accepted by an APCol system with coloured teams, if starting with the string as initial string the computation is unbounded and its teams with the final colour are active in an infinite number of steps and the teams of the other colours are active only in a finite number of steps.

Now we provide an illustrative example of APCol system with coloured teams.

Example 1. We construct an APCol system with three teams assigned to three different colours - red, green and orange, simulating work of streets lights connected with speed radar. The green light is on the traffic light at the beginning. If the vehicle is approaching faster than allowed, the traffic light changes to orange and red. In the case that the vehicle stops before the traffic lights (or it drives away while the red is on), the traffic light lights up orange and then green again.The input string for computation is a sequence of signals coming from speed radar. The signals are encoded into symbols in such a way that F means fast speed over limit, S means slow speed within the limits, Z means that car stopped and finally E which means that street is empty. The signals are encoded and inserted into the string with given frequency. Every input string starts with special symbol \$.

The constructed APCol system

$$\begin{aligned}\Pi = (&\{e, E, Z, S, F, o, r, g, \$, R, P\}, e,\\ &A_1, A_2, A_3, \{\mathit{green}, \mathit{orange}, \mathit{red}\}, \mathit{green},\\ &\{B_1, B_2, B_3\}, \{(B_1, \mathit{green}), (B_2, \mathit{orange}), (B_3, \mathit{red})\},\\ &\{(A_1, B_1), (A_2, B_2), (A_3, B_3)\})\end{aligned}$$

has three teams - one green, one orange and one red. Each team is formed from only one agent. Agent A_1 has initial configuration ge and the following programs:

$1 : \langle g \leftrightarrow \$; e \leftrightarrow X \rangle \quad X \in \{E, Z, S\}$

$2 : \langle \$ \leftrightarrow g; X \rightarrow e \rangle$

The green team is active only if the current symbol is in accordance with the speed limit or the street is empty. When the speed of the arriving vehicle is over the speed limit, only the orange team can work.

The initial configuration of the agent A_2 is oe and it executes following programs:

$3 : \langle o \leftrightarrow \$; e \leftrightarrow F \rangle$
$4 : \langle \$ \rightarrow \$; F \rightarrow R \rangle$
$5 : \langle \$ \rightarrow \$; R \leftrightarrow o \rangle$

The agent from the orange team consumes symbol F and replaces symbol \$ by R. When this symbol appears in the string, only the red team can work.

The initial configuration of the agent A_3 is re and it performs the following programs:

$6 : \langle r \leftrightarrow R; e \leftrightarrow X \rangle, \quad X \in \{E, F, S, Z\}$
$7 : \langle R \rightarrow R; Y \rightarrow e \rangle, \quad Y \in \{F, S\}$
$8 : \langle R \leftrightarrow r; e \rightarrow e \rangle$
$9 : \langle R \rightarrow P; K \rightarrow e \rangle, \quad K \in \{Z, E\}$
$10 : \langle P \leftrightarrow r; e \rightarrow e \rangle$

The agent from the red team consumes symbol R and the neighbouring symbol from the string. The following behaviour of the agent depends on consumed symbol. If the symbol is F or it is S, then the agent puts to the string symbol R and in this way it calls itself to work. In the case of symbol Z or E, (the vehicle stopped or the street is empty) then the agent puts the symbol P to the string and the agent from the orange team has an applicable program.

$11 : \langle \$ \rightarrow \$; o \leftrightarrow P \rangle$
$12 : \langle \$ \rightarrow \$; P \rightarrow e \rangle$
$13 : \langle \$ \leftrightarrow o; e \rightarrow e \rangle$

After executing program 13, symbol \$ appears in the string and it can be consumed by agent from the green team or the orange team. Although the computation over a finite string is not unbounded, but one can assume that if there is no output from the speed radar, encoder puts symbols E into the string with a given frequency (it is similar to the endless tape of Turing machine) and the computation can continue with executing programs of the agent from the green team.

3 APCol Systems with Coloured Teams and Red-Green Counter Machines

In this section we study the interconnection between red-green counter machines and APCol systems with coloured teams. First we present a result where the number of agents within every team is minimal, namely, one.

Theorem 1. *For every red-green counter machine*

$$CM = (m, B, B_{red}, B_{green}, l_0, P, T_{in})$$

we can construct an APCol system

$$\Pi = (O, e, A_1, \ldots, A_n, C, B, B_{colours}, B_{teams})$$

with one agent teams simulating the computations of CM.

Proof. Consider a red-green counter machine $CM = (2, B, B_{red}, B_{green}, l_0, P, T_{in})$ accepting language $L(CM)$. To every such counter machine there exists a red-green counter machine $CM' = (2, B \cup \{l'_0\}, B'_{red}, B'_{green}, l'_0, P, T_{in} \cup \{\#\})$ that accepts language $L(CM') = \# \cdot L(CM)$, where $\{\#\} \cap T_{in} = \emptyset$ and the first instruction of CM' to be executed is instruction $l'_0 : (\text{read}(\#), l_0)$. Then it continues the computation in the same way as machine CM. We construct an APCol system Π with coloured teams as follows: all labels from the set $B \cup T_{in}$ are objects of the APCol system. The content of register i is represented by the number of copies of objects i occurring in the string. All teams have one agent only. At the beginning of the computation only one team of agents can work - red team of one agent that generates symbols to the beginning of the string.

Team: B_1
Colour: *Red*
Agent: $A_1 = (ee, P_1, \emptyset)$
Programs:

$1 : \langle e \to \#_1; e \to O_1 \rangle$;
$2 : \langle \#_1 \leftrightarrow \#; O_1 \leftrightarrow e \rangle$;
$3 : \langle \# \to \#_2; e \to \$ \rangle$;
$4 : \langle \#_2 \leftrightarrow e; \$ \leftrightarrow O_1 \rangle$;
$5 : \langle O_1 \to R; e \to (R) \rangle$;
$6 : \langle (R) \to [R]; R \leftrightarrow e \rangle$;
$7 : \langle [R] \to (G); e \to G \rangle$;
$8 : \langle (G) \to [G]; G \leftrightarrow e \rangle$;
$9 : \langle [G] \to (L); e \to X_0 \rangle$;
$10 : \langle (L) \to [L]; X_0 \leftrightarrow e \rangle$;

Symbol X is an element from the set $\{l, r, g\}$ and it is selected as follows: let l_1 be the currently simulated instruction and let l_2 be the label of the next instruction. If l_2 is a read-instruction and the colour of instruction is red (or green) then $X = r$ (or $X = g$). Otherwise $X = l$.

The APCol system starts its computation on string $\#\omega$. Agent A_1 uses programs $1, 2, 3$ and 4 to replace symbol $\#$ by substring $\#_1\#_2\$$. Then it places three symbols (R, G, X_0) into random positions in the string.

Symbols R and G are consumed by two agents from two teams.

Team: B_2
Colour: *Red*
Agent: $A_2 = (ee, P_2, \emptyset)$
Programs: $11 : \langle e \leftrightarrow R; e \to e \rangle$;

Team: B_3
Colour: *Green*
Agent: $A_3 = (ee, P_3, \emptyset)$
Programs: $12 : \langle e \leftrightarrow G; e \to e \rangle$;

Let l_1 be a read-instruction $l_1 : (read(a), l_2)$. We construct two similar teams of different colours to execute the first phase of the simulation of the read-instruction. The agent from such a team checks whether the symbol currently read from the input string is a or not. The team of the working agent has the same colour as the previously simulated instruction.

Team: B_{X_1} for $X \in \{r, g\}$
Colour: B_{r_1} is *Red*, B_{g_1} is *Green*
Agent: $A_{x_1} = (ee, P_{x_1}, \emptyset)$
Programs: $13 : \langle e \leftrightarrow X_1; e \to e \rangle$;
$14 : \langle X_1 \to L_1'; e \to e \rangle$;
$15 : \langle L_1' \leftrightarrow \$; e \leftrightarrow y \rangle$;
for all $y \in T_{in}$

Team: B_2 or B_3
Colour: B_2 is *Red*, B_3 is *Green*
Agent: A_2 or A_3; $d \in \{R, G\}$
Programs: $16 : \langle d \leftrightarrow L_1'; e \to M_1 \rangle$;
$17 : \langle L_1' \to N_1; M_1 \to M_1 \rangle$;
$18 : \langle M_1 \leftrightarrow d; N_1 \leftrightarrow e \rangle$;

When agent A_{X_1} successfully finishes its work, then agent from a team with the same colour as l_1 inserts symbol l_2 into the string. In the other case, when the read-instruction cannot be performed, agent from red team starts to be active for an unbounded number of steps.

Team: B_{X_1} for $X \in \{r, g\}$
Colour: B_{r_1} is *Red*, B_{g_1} is *Green*
Agent: A_{X_1}
Programs: $19 : \langle \$ \leftrightarrow M_1; a \to e \rangle$; $22 : \langle \$ \leftrightarrow M_1; y \leftrightarrow N_1 \rangle$; $y \in T_{in} - \{a\}$;
$20 : \langle M_1 \to Q_1; e \leftrightarrow N_1 \rangle$; $23 : \langle M_1 \to W; N_1 \to e \rangle$;
$21 : \langle Q_1 \leftrightarrow e; N_1 \to E \rangle$; $24 : \langle W \leftrightarrow e; e \to e \rangle$;

Team: B_{l_1}
Colour: *Red* or *Green* (depends on l_1)
Agent: $A_{l_1} = (ee, P_{l_1}, \emptyset)$
Programs: $25 : \langle e \leftrightarrow Q_1; e \to X_2 \rangle$;
$26 : \langle X_1 \leftrightarrow e; Q_1 \to e \rangle$;
$X \in \{l, r, g\}$

Team: B_4
Colour: *Red*
Agent: A_4
Programs: $27 : \langle e \leftrightarrow W; e \to e \rangle$;
$28 : \langle W \to W; e \to e \rangle$;

For each ADD-instruction $l_1 : (ADD(r), l_2)$, there are two teams of agents of the same colour as the ADD-instruction has.

Team: B_{l_1}
Colour: *Red* or *Green* (depends on the instruction colour)
Agent: A_{l_1}
Programs: $29 : \langle e \leftrightarrow l_1; a \to e \rangle$; $32 : \langle \#_r \leftrightarrow M_1; r \leftrightarrow N_1 \rangle$;
$30 : \langle l_1 \to L_1; e \to e \rangle$; $33 : \langle M_1 \to X_2; e \to e \rangle$; $X \in \{l, r, g\}$
$31 : \langle L_1 \leftrightarrow \#_r; e \to r \rangle$; $34 : \langle X_2 \leftrightarrow e; e \to e \rangle$;

Team: B_2 or B_3
Colour: B_2 is *Red*, B_3 is *Green*
Agent: A_2 or A_3; $d \in \{R, G\}$
Programs: $35 : \langle d \leftrightarrow L_1; e \to M_1 \rangle$;
$36 : \langle M_1 \leftrightarrow d; L_1 \to e \rangle$;

The first agent consumes the corresponding symbol of the actually simulated instruction. At the following steps, the agent rewrites the symbol l_1 to L_1 and exchanges this symbol by $\#_r$. In the same time, the agent generates symbol r. Now it is time for the second team to work. The agent from the second team replaces symbol L_1 by R or G - it depends on the colour of the instruction,

rewrites it to symbol M_1 and puts the symbol M_1 to the string instead of symbol R or G. When symbol M_1 appears in the string, then the agent B_1 exchanges it by two symbols - $\#_r$ and r.

For SUB-instruction $l_1 : (SUB(r), l_2, l_3)$, there are two teams of the same colour, too. The first team with one agent is for execution of the instruction and the second team is preparing the symbols for further use (symbol L_1 is replaced with M_1).

Team:	B_{l_1}	Team:	B_2 or B_3
Colour:	*Red* or *Green*	Colour:	B_2 is *Red*, B_3 is *Green*
	(depends on the colour of l_1)	Agent:	A_2 or A_3; $d \in \{R, G\}$
Agent:	A_{l_1}	Programs:	$41 : \langle d \leftrightarrow L_1; e \rightarrow M_1 \rangle$;
Programs:	$37 : \langle e \leftrightarrow l_1; e \rightarrow e \rangle$;		$42 : \langle M_1 \leftrightarrow d; L_1 \rightarrow e \rangle$;
	$38 : \langle l_1 \rightarrow L_1; e \rightarrow e \rangle$;		
	$39 : \langle L_1 \leftrightarrow \#_r; e \leftrightarrow r \rangle$;		
	$40 : \langle L_1 \leftrightarrow \#_r; e \leftrightarrow Z \rangle$;		
	$Z \in \{\#_{r+1}, \$\}$		

The idea of simulation of SUB-instruction is that the agent consumes symbol $\#_r$ together with symbol r - if the counter r is not empty -, or with symbol $\#_{r+1}$ (or \$) - if the counter r is empty and it is not the last counter (or it is the last counter). According to its content, the agent generates the label of the next instruction.

Team:	B_{l_1}		
Colour:	*Red* or *Green* (depends on the colour of l_1)		
Agent:	A_{l_1}		
Programs:	$43 : \langle \#_r \rightarrow \#_r; r \rightarrow l'_2 \rangle$;	$48 : \langle \#_r \leftrightarrow M_1; Z \leftrightarrow N_1 \rangle$;	
	$44 : \langle \#_r \leftrightarrow M_1; l'_2 \rightarrow l''_2 \rangle$;	$33 : \langle M_1 \rightarrow Y_3; N_1 \rightarrow e \rangle$;	
	$45 : \langle M_1 \rightarrow e; l''_2 \rightarrow l'''_2 \rangle$;	$33 : \langle Y_3 \leftrightarrow e; e \rightarrow e \rangle$;	
	$46 : \langle e \leftrightarrow N_1; l'''_2 \rightarrow X_2 \rangle$;	$X, Y \in \{l, r, g\}$;	
	$47 : \langle N_1 \rightarrow e; X_2 \leftrightarrow e \rangle$;	$Z \in \{\#_{r+1}, \$\}$	

We construct the APCol system

$$\Pi = (O, e, A_1, \ldots, A_n, C, B, B_{colours}, B_{teams}) \text{ with:}$$

- $O = T_{in} \cup \{l_i, l'_i, l''_i, l'''_i, L_i, L'_i, M_i, N_i, g_i, r_i, Q_i | l_i \in H\} \cup \{i | 1 \leq i \leq m\} \cup$
  $\cup \{e, G, R, \textcircled{R}, \textcircled{G}, \boxed{R}, \boxed{G}, \textcircled{L}, \boxed{L}, W, \$, \#_1, \#2, O_i\}$,
- $n = |H| + 2 \times$ number of read-instructions $+ 4$
- $B = \{B_j\}, 1 \leq j \leq n$
- $C = \{Red, Green\}$
- The sets $B_{colours}, B_{teams}$ and the agents $A_1, \ldots, A_n$ are defined in the previous part of the text.

The computation of the APCol system starts with string $\$w$. The first steps are done by the red team B_1. Teams B_2 and B_3 must go through initialization

before they are used the first time during simulation of the first red or green instruction. It can imply only a finite number of mind changes. After initialization of these two agents, the APCol system goes through the same mind changes as the red-green counter machine CM goes through during the corresponding computation. Therefore, if red-green counter machine CM accepts string w, then APCol system Π accepts it too and vice versa.

□

Although the APCol system from proof of Theorem 1. uses the maximally parallel working mode, its work is limited to the use of one team at each step, therefore, to one agent. As a matter of fact, it works sequentially.

Next we provide another simulation of the red-green counter machines with APCol systems with teams and colours.

Theorem 2. *For every red-green counter machine*

$$CM = (m, B, B_{red}, B_{green}, l_0, P, T_{in})$$

we can construct an APCol system

$$\Pi = (O, e, A_1, \ldots, A_n, C, B, B_{colours}, B_{teams})$$

with at least one team formed from two agents simulating the computations of CM with the same result.

Proof. As in proof of Theorem 1, let us consider red-green counter machine

$$CM = (2, B, B_{red}, B_{green}, l_0, P, T_{in})$$

accepting language $L(CM)$. To every such a red-green counter machine there exists a red-green counter machine

$$CM' = (2, B \cup \{l'_0\}, B'_{red}, B'_{green}, l'_0, P, T_{in} \cup \{\#\})$$

that accepts language $L(CM') = \# \cdot L(CM)$, where $\{\#\} \cap T_{in} = \emptyset$ and the first instruction of the machine CM' to be executed is instruction $l'_0 : (\text{read}(\#), l_0)$. Then, it continues the computation in the same way as machine CM. We construct an APCol system Π with coloured teams as follows: All labels from the set $B \cup T_{in}$ are objects of the APCol system. The content of register i is represented by the number of copies of objects i in the string. At the beginning of the computation only one team of agents can work - red team of one agent that generates symbols to the beginning of the string.

Team: B_1
Colour: *Red*
Agent: $A_1 = (ee, P_1, \emptyset)$
Programs:

$1 : \langle e \to \#_1; e \to O_1 \rangle$;
$2 : \langle \#_1 \leftrightarrow \#; O_1 \leftrightarrow e \rangle$;
$3 : \langle \# \to \#_2; e \to \$ \rangle$;
$4 : \langle \#_2 \leftrightarrow e; \$ \leftrightarrow O_1 \rangle$;
$5 : \langle O_1 \to l_0; e \to T \rangle$;
$6 : \langle T \to T; l_0 \leftrightarrow e \rangle$;

The APCol system starts its computation on string $\#\omega$. Agent A_1 uses programs $1, 2, 3$ and 4 to replace symbol $\#$ by substring $\#_1\#_2\$$. Then it places symbol l_0 into some random position in the string.

Let l_1 be a read-instruction $l_1 : (read(a), l_2)$. We construct a team of the same colour as the read-instruction has. The team is formed from two agents. Because they work as a team, either they both execute their programs or none of them works.

Team:	B_{l_1}	
Colour:	*Red* or *Green* (it depends on the colour of l_1)	
Agent:	$A_{a_1} = (ee, P_{a_1}, \emptyset)$	Agent: $A_{b_1} = (ee, P_{b_1}, \emptyset)$
Programs:	$7 : \langle e \leftrightarrow l_1; e \to e\rangle$;	Programs: $11 : \langle e \to R_1; e \to e\rangle$;
	$8 : \langle l_1 \to \$; e \to e\rangle$;	$12 : \langle R_1 \leftrightarrow \$; e \to x\rangle$;
	$9 : \langle \$ \leftrightarrow R_1; e \to e\rangle$;	for all $x \in T_{in}$
	$10 : \langle R_1 \to e; e \to e\rangle$;	$13 : \langle \$ \to l_2; a \to e\rangle$;
		$14 : \langle \$ \to W; y \to e\rangle$;
		for all $y \in T_{in} - \{a\}$
		$15 : \langle l_2 \leftrightarrow e; e \to e\rangle$;
		$16 : \langle W \leftrightarrow e; e \to e\rangle$;

Although agent A_{b_1} has an applicable program it must stay inactive until the first agent has an applicable program, too, i.e., until symbol l_1 appears in the string.

Team:	B_1
Colour:	*Red*
Agent:	A_1
Programs:	$17 : \langle e \leftrightarrow W; T \to T\rangle$;
	$18 : \langle W \to W; T \to T\rangle$;

When the read-instruction cannot be performed, agent A_1 from red team starts working for an unbounded number of steps.

For each ADD-instruction $l_1 : (ADD(r), l_2)$, there is one team of agents of the same colour as the ADD-instruction has.

Team:	B_{l_1}	
Colour:	*Red* or *Green* (it depends on the colour of l_1)	
Agent:	$A_{a_1} = (ee, P_{a_1}, \emptyset)$	Agent: $A_{b_1} = (ee, P_{b_1}, \emptyset)$
Programs:	$19 : \langle e \leftrightarrow l_1; e \to L_1\rangle$;	Programs: $24 : \langle e \to r'; e \to e\rangle$;
	$20 : \langle L_1 \leftrightarrow \#_r; l_1 \to K_1\rangle$;	$25 : \langle r' \to r; e \to M_1\rangle$;
	$21 : \langle \#_r \to \#_r; K_1 \to K_2\rangle$;	$26 : \langle M_1 \leftrightarrow L_1; r \leftrightarrow e\rangle$;
	$22 : \langle \#_r \leftrightarrow M_1; K_2 \to K_3\rangle$;	$27 : \langle L_1 \to l_2; e \to e\rangle$;
	$23 : \langle M_1 \to e; K_3 \to e\rangle$;	$28 : \langle l_2 \leftrightarrow e; e \to e\rangle$;

The first agent consumes the symbol corresponding to the actually simulated instruction. In the same time, the second agent starts to generate symbol r. At the following steps, the first agent rewrites symbol l_1 to L_1 and exchanges this symbol by $\#_r$. The first agent can put symbol $\#_r$ back to the string only by

replacing it by symbol M_1 generated by the second agent. The second agent inserts the label of the next instruction at some random place in the string.

For SUB-instruction $l_1 : (SUB(r), l_2, l_3)$, there is one team of the same colour as the instruction has.

Team:	B_{l_1}	
Colour:	*Red* or *Green* (it depends on the colour of l_1)	
Agent:	$A_{a_1} = (ee, P_{a_1}, \emptyset)$	Agent: $A_{b_1} = (ee, P_{b_1}, \emptyset)$
Programs:	$29 : \langle e \leftrightarrow l_1; e \to L_1 \rangle$;	Programs: $37 : \langle e \to M_1; e \to K_1 \rangle$;
	$30 : \langle L_1 \leftrightarrow \#_r; l_1 \to K_1 \rangle$;	$38 : \langle M_1 \to M_1; K_1 \to e \rangle$;
	$31 : \langle \#_r \to \#_r; K_1 \to K_2 \rangle$;	$39 : \langle M_1 \leftrightarrow L_1; e \leftrightarrow d \rangle$;
	$32 : \langle \#_r \leftrightarrow M_1; K_2 \to K_2 \rangle$;	for all $d \in \{r, \#_{r+1}, \$\}$
	$33 : \langle M_1 \to M_1'; K_2 \to K_2 \rangle$;	$40 : \langle L_1 \to N_1; r \to K_1 \rangle$;
	$34 : \langle M_1' \leftrightarrow N_1; K_2 \to K_2 \rangle$;	$41 : \langle L_1 \to N_1; d' \to d' \rangle$;
	$35 : \langle N_1 \to e; K_2 \to K_2 \rangle$;	for all $d' \in \{\#_{r+1}, \$\}$
	$36 : \langle e \to e; K_2 \to e \rangle$;	$42 : \langle N_1 \leftrightarrow \#_r; K_1 \to K_2 \rangle$;
		$43 : \langle N_1 \leftrightarrow \#_r; d' \to d' \rangle$;
		$44 : \langle \#_r \to \#_r; K_1 \to K_2 \rangle$;
		$45 : \langle \#_r \to \#_r; K_2 \to K_3 \rangle$;
		$46 : \langle \#_r \leftrightarrow M_1'; K_3 \to l_2 \rangle$;
		$47 : \langle \#_r \to l_2; e \to e \rangle$;
		$48 : \langle N_1 \leftrightarrow \#_r; d' \leftrightarrow e \rangle$;
		$49 : \langle \#_r \to \#_r; e \to K \rangle$;
		$50 : \langle \#_r \leftrightarrow M_1'; K \to l_3 \rangle$;
		$51 : \langle l_2 \leftrightarrow e; M_1' \to e \rangle$;
		$52 : \langle l_3 \leftrightarrow e; M_1' \to e \rangle$;

The idea of simulation of SUB-instruction is that agent consumes symbol $\#_r$ together with the right neighbouring symbol. According to content of the agent, it generates the label of the next instruction.

We construct the APCol system

$$\Pi = (O, e, A_1, \ldots, A_n, C, \mathit{Green}, B, B_{colours}, B_{teams}) \text{ with:}$$

- $O = T_{in} \cup \{l_i, L_i, M_i, M_1', N_i, R_i | l_i \in H\} \cup \{i, i' | 1 \leq i \leq m\} \cup$
  $\cup \{e, K_1, K_2, K_3, W, \$, \#_1, \#2\}$,
- $n = 2 \times |H| + 1$
- $B = \{B_j\}, 1 \leq j \leq p;\ p = |H| + 1$
- $C = \{\mathit{Red}, \mathit{Green}\}$
- The sets $B_{colours}, B_{teams}$ and the agents $A_1, \ldots, A_n$ are defined in the previous part of the text.

The computation of the APCol system starts with string $\$w$. The first steps are done by the red team B_1. After initialization, the APCol system goes through

the same mind changes as the counter machine goes through during the corresponding computation. Therefore, if red-green counter machine CM accepts string w, then APCol system Π accepts string $\#w$ too, and vice versa.

□

4 Conclusions

In this paper, we investigated the possibility of "going beyond" Turing in the terms of APCol systems. We introduced the notion of teams of agents as a restriction for the maximal parallelism of computation. In addition, we assigned a colour to each team. The unbounded computation was described by the sequence of the colours associated to the acting teams. We have shown that we can simulate red-green counter machines with APCol systems with two-coloured teams. Red-green counter machines are computing devices with infinite run on finite input that exceed the power of Turing machines.

As we mentioned in the Introduction, there are concepts in P systems theory which are motivated and mimic the behaviour of red-green Turing machines, for example [2]. The proofs of the theorems in Section 3 demonstrate that finite communities of very simple and very small computing devices (i.e. agents and programs) in a suitable environment and using a simple cooperation protocol (based on colours) can produce a behaviour which may not be computable in the sense of Turing machines. These results add further information on the behaviour of communities of agents and ideas to constructs networks of computing agents.

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, by SGS/13/2016 and by Grant No. 120558 of the National Research, Development, and Innovation Office, Hungary.

References

1. Alhazov, A., Aman, B., Freund, R., Păun, G.: Matter and Anti-matter in Membrane Systems. In: Jürgensen, H., Karhumäki, J., Okhotin, A. (eds.) Descriptional Complexity of Formal Systems: 16th International Workshop, DCFS 2014, Turku, Finland, August 5-8, 2014. Proceedings. pp. 65–76. Springer International Publishing, Cham (2014),
2. Aman, B., Csuhaj-Varjú, E., Freund, R.: Red-Green P Automata. In: Gheorghe, M. et al. (eds.): CMC 2014, LNCS, vol. 8961, pp. 139–157, Springer (2014)
3. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E.: A Class of Restricted P Colonies with String Environment. Natural Computing 15(4), 541–549 (2016),
4. Cienciala, L., Ciencialová, L.: P Colonies and Their Extensions. In: Kelemen, J., Kelemenová, A. (eds.) Computation, Cooperation, and Life – Essays Dedicated to Gheorghe Paun on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 6610, pp. 158–169. Springer-Verlag, Berlin Heidelberg (2011),

5. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E.: Towards on P Colonies Processing Strings. In: Proc. BWMC 2014, Sevilla, 2014. pp. 102–118. Fénix Editora, Sevilla, Spain (2014)
6. Csuhaj-Varjú, E.: Extensions of P Colonies (Extended Abstract). In: Leporati, A. and Zandron, C. (Eds.) Proc. CMC17, Milan, 2014. pp. 281–286. University Milano-Bicocca & IMCS, Italy (2014)
7. Csuhaj-Varjú, E., Kelemen, J., Păun, Gh., Dassow, J.(eds.): Grammar Systems: A Grammatical Approach to Distribution and Cooperation. Gordon and Breach Science Publishers, Inc., Newark, NJ, USA (1994)
8. 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)
9. Kelemen, J., Kelemenová, A., Păun, G.: 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). pp. 82–86. Boston, Mass (2004)
10. Kelemen, J., Kelemenová, A.: A Grammar-Theoretic Treatment of Multiagent Systems. Cybern. Syst. 23(6), 621–633 (Nov 1992),
11. Minsky, M. L.: Computation: Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, NJ, 1967.
12. van Leeuwen, J., Wiedermann, J.: Computation as an Unbounded Process. Theoretical Computer Science 429, 202 – 212 (2012),
13. Meduna, A., Zemek, P.: Jumping Finite Automata. Int. J. Found. Comput. Sci. 23(7), 1555–1578 (2012)
14. Păun, Gh., Rozenberg, G., Salomaa, A.(eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)
15. Rozenberg, G., Salomaa, A.(eds.): Handbook of Formal Languages I-III. Springer Verlag., Berin-Heidelberg-New York (1997)