A Note on a New Class of APCol Systems

Lucie Ciencialová[1], Erzsébet Csuhaj-Varjú[2], and György Vaszil[3]

[1] Institute of Computer Science, Silesian University in Opava, Czech Republic
lucie.ciencialova@fpf.slu.cz
[2] Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary
csuhaj@inf.elte.hu
[3] Faculty of Informatics, University of Debrecen, Hungary
vaszil.gyorgy@inf.unideb.hu

Summary. We introduce a new acceptance mode for APCol systems (Automaton-like P colonies), variants of P colonies where the environment of the agents is given by a string and during functioning the agents change their own states and process the string similarly to automata. In case of the standard variant, the string is accepted if it can be reduced to the empty word. In this paper, we define APCol systems where the agents verify their environment, a model resembling multihead finite automata. In this case, a string of length n is accepted if during every halting computation the length of the environmental string in the configurations does not change and in the course of the computation every agent applies a rule to a symbol on position i of some of the environmental strings for every i, $1 \leq i \leq n$ at least once. We show that these verifying APCol systems simulate one-way multihead finite automata.

1 Introduction

Automaton-like P colonies (APCol systems, for short), introduced in [1], are variants of of P colonies (introduced in [9]) - very simple membrane systems inspired by colonies of formal grammars. The interested reader is referred to [12] for detailed information on P systems (membrane systems) and to [10] and [5] for more information to grammar systems theory; for more details on P colonies consult [8] and [4].

An APCol system consists of a finite number of agents - finite collections of objects in a cell - and a shared environment. The agents have programs consisting of rules. These rules are of two types: they may change the objects of the agents and they can be used for interacting with the joint shared environment of the agents. While in the case of standard P colonies the environment is a multiset of objects, in case of APCol systems it is represented by a string. 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 an indirect communication channel for the agents as well, since through the string, the agents

are able to affect the behaviour of another agent. The reader may easily observe that APCol systems resembling automata as well, the current configuration of the system (the objects inside the agents) and the current environmental string correspond to the current state of an automaton and the currently processed input string.

The agents may perform 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 performs 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. Thus, the number of rules in the program is the same as the number of objects inside the agent.

The computation in APCol systems starts with the an input string, representing the initial state of the environment, and with each agents having only symbols e inside.

A computational step means a maximally parallel action of the active agents, i.e., agents that can apply their rules. Every symbol can be object of the action of only one agent. The computation ends if the input string is reduced to the empty word, there are no more applicable programs in the system, and meantime at least one of the agents is in so-called final state. This mode of computation is called accepting. APCol systems can also be used not only for accepting but generating strings. For more detailed information on APCol systems we refer to [2, 3].

In this paper, we define a new variant of APCol systems, a model resembling multihead finite automata, where the agents verify their environment. In this case, a string of length n is accepted if during every halting computation the length of the environmental string in the configurations does not change and in the course of the computation every agent applies a rule to a symbol on position i of some of the environmental strings for every i, $1 \leq i \leq n$ at least once. We show that these verifying APCol systems simulate one-way multihead finite automata.

2 Preliminaries and Basic Notions

Throughout the paper we assume the reader to be familiar with the basics of the formal language theory and membrane computing [13, 12].

For an alphabet Σ, the set of all words over Σ (including the empty word, ε), is denoted by Σ^*. We denote the length of a word $w \in \Sigma^*$ by $|w|$ and the number of occurrences of the symbol $a \in \Sigma$ in w by $|w|_a$.

For a string $x \in \Sigma^*$, $x[i]$ denotes the symbol at ith position of x, i.e., if $x = x_1 \dots x_n$, $x_i \in \Sigma$, then $x[i] = x_i$. For every string $x \in \Sigma^*$, $x[0] = \varepsilon$.

For every string $x \in \Sigma^*$, $perm(x)$ denotes the set of all permutations of x and $pref(x)$ denotes the set of prefixes of x.

A multiset of objects M is a pair $M = (O, f)$, where O is an arbitrary (not necessarily finite) set of objects and f is a mapping $f : O \to N$; f assigns to each object in O its multiplicity in M. Any multiset of objects M with the set of objects $O = \{x_1, \dots x_n\}$ can be represented as a string w over alphabet O with $|w|_{x_i} = f(x_i)$; $1 \leq i \leq n$. Obviously, all words obtained from w by permuting the letters can also represent the same multiset M, and ε represents the empty multiset.

2.1 One-way Multihead Finite Automata

We recall some basic notions concerning multi-head finite automata based on [7]. A non-deterministic one-way k-head finite automaton (a 1NFA(k), for short) is a construct $M = (Q, \Sigma, k, \delta, \triangleright, \triangleleft, q_0, F)$, where Q is the finite set of states, Σ is the set of input symbols, $k \geq 1$ is the number of heads, $\triangleright \notin \Sigma$ and $\triangleleft \notin \Sigma$ are the left and the right endmarkers, respectively, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states, and δ is the partial transition function which maps $Q \times (\Sigma \cup \{\triangleright, \triangleleft\})^k$ into subsets of $Q \times \{0, 1\}^k$, where 1 means that the head moves one tape cell to the right and 0 means that it remains at the same position. We note that the heads can never move to the left of the left endmarker and to the right of the right endmarker.

A configuration of a 1NFA(k) $M = (Q, \Sigma, k, \delta, \triangleright, \triangleleft, q_0, F)$ is a triplet $c = (w, q, p)$, where $w \in \Sigma^*$ is the input, $q \in Q$ is the current state, and $p = (p_1, \dots, p_k) \in \{0, 1, \dots, |w| + 1\}^k$ gives the head positions. If a position p_i is 0, then head i is scanning the symbol $\triangleright$, if $1 \leq p_i \leq |w|$, then head i scans the p_ith letter of w, and if $p_i = |w| + 1$, then the ith head is scanning the symbol $\triangleleft$.

The initial configuration for an input $w \in \Sigma^*$ is $(w, q_0, (1, \dots, 1))$, that is, a 1NFA(k) starts processing a nonempty input word with all of its heads positioned on the first symbol of w.

In the course of the computation, M performs direct changes of its configurations. Let $w = a_1 \dots a_n$, be the input, $a_0 = \triangleright$, $a_{n+1} = \triangleleft$. For two configurations, $c_1 = (w, q, (p_1, \dots, p_k))$ and $c_2 = (w, q', (p'_1, \dots, p'_k))$, we say that c_2 directly follows c_1, denoted by $c_1 \vdash c_2$, if $(q', (d_1, \dots, d_k)) \in \delta(q, (a_{p_1}, \dots, a_{p_k}))$ and $p'_i = p_i + d_i$, $1 \leq i \leq k$. The reflexive transitive closure of $\vdash$ is denoted by $\vdash^*$. Note that due to the restriction of the transition function, the heads cannot move beyond the endmarkers.

The language $L(M)$ accepted by a 1NFA(k) $M = (Q, \Sigma, k, \delta, \triangleright, \triangleleft, q_0, F)$ is the set of words w such that there is a computation which starts with $\triangleright w \triangleleft$ on the input tape and ends when M reaches an accepting state, i.e.,

$$L(M) = \{w \in \Sigma^* \mid (w, q_0, (1, \dots, 1)) \vdash^* (w, q_f, (p_1, \dots, p_k)),\ q_f \in F\}.$$

The class of languages accepted by 1NFA(k), for $k \geq 1$, is denoted by $\mathcal{L}$(1NFA(k))

According to the definition of a 1NFA(k) $M = (Q, \Sigma, k, \delta, \triangleright, \triangleleft, q_0, F)$, the heads do not need to move away from the scanned tape cell after reading the input

symbol. For technical reasons, we use a modified but equally powerful version of this definition in such a way that the automaton reads the input symbols only in the case when the head moves away from the tape cell containing the symbol. Otherwise, it "scans" the empty word ε, that is, the symbol does not have any role in the determination of the next configuration of the machine. For details and the proof of the equivalence the reader is referred to [6].

Thus, we can simplify the notation for the elements of the transition relation of one-way k-head finite automata, since we can assume that if $(q', (d_1, \ldots, d_k)) \in \delta(q, a_1, \ldots, a_k)$, then $d_j = 0$ if and only if $a_j = \varepsilon$, $1 \leq j \leq k$. As $(d_1, \ldots, d_k) \in \{0, 1\}^k$, we can simply denote the above transition as $q' \in \delta(q, a_1, \ldots, a_k)$: If $a_j \neq \varepsilon$ for some j, $1 \leq j \leq k$, then the jth reading head is moved to the right, otherwise, if $a_j = \varepsilon$ it remains in its current position.

2.2 APCol Systems

In the following we recall the notion of an APCol system (an automaton-like P colony) where the environment of the agents is given in the form of a string [1].

As standard P colonies, agents of the APCol systems contain objects, each of them is an element of a finite alphabet. Every agent is associated with a set of programs, every program consists of two rules that can be one of the following two types. The first one, called an evolution rule, is of the form $a \rightarrow b$. This means that object a inside of the agent is rewritten to object b. The second type, called a communication rule, is of the form $c \leftrightarrow d$. When this rule is applied, object c inside the agent and a symbol d in the string representing the environment (the input string) are exchanged. If $c = e$, then the agent erases d from the input string and if $d = e$, symbol c is inserted into the string.

The computation in APCol systems starts with an input string, representing the environment, and with each agent having only symbols e inside.

A computational step means a maximally parallel action of the active agents, i.e., agents that can apply their rules. Every symbol can be object of the action of only one agent. The computation ends if the input string is reduced to the empty word, there are no more applicable programs in the system, and meantime at least one of the agents is in so-called final state.

An APCol system, for is a construct

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

- O is an alphabet; its elements are called the objects,
- $e \in O$, called the basic object,
- A_i, $1 \leq i \leq n$, are agents. Each agent is a triplet $A_i = (\omega_i, P_i, F_i)$, where
  - 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.

In the following we explain the work of an APCol system.

During the work of the APCol system, the agents perform programs. Since both rules in a program can be communication rules, an agent can work with two objects in the string in one step of the computation. In the case of program $\langle a \leftrightarrow b; c \leftrightarrow d\rangle$, a substring bd of the input string is replaced by string ac. Notice that although the order of rules in the programs is usually irrelevant, here it is significant, since it expresses context-dependence. 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. Thus, the agent is allowed to act only at one position of the string in the one step of the computation and the result of its action to the string depends both on the order of the rules in the program and on the interacting objects. In particular, we have the following types of programs with two communication rules:

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

At the beginning of the work of the APCol system (at the beginning of the computation), the environment is given by a string ω of objects which are different from e. This string represents the initial state of the environment. Consequently, an initial configuration of the APCol system is an $(n+1)$-tuple $c = (\omega; \omega_1, \ldots, \omega_n)$ where w 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 the of agents.

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 inside the ith agent and $w \in (O - \{e\})^*$ is the string to be processed.

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

By applying programs, the APCol system passes from one configuration to another configuration. A sequence of configurations started 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 case of accepting mode a computation is called accepting if and only

if at least one agent is in final state and the string to be processed is ε. Hence, the string ω is accepted by the 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 the computation ends by halting in the configuration $(\varepsilon; w_1, \ldots, w_n)$, where at least one of $w_i \in F_i$ for $1 \leq i \leq n$.

In [1] it was shown that the family of languages accepted by jumping finite automata (introduced in [11]) is properly included in the family of languages accepted by APCol systems with one agent, and it was proved that any recursively enumerable language can be obtained as a projection of a language accepted by an APCol system with two agents.

3 Verifying APCol Systems

In this section we introduce a new variant of acceptance for APCol systems, motivated by the behaviour of multihead finite automata. In case of standard APCol systems acceptance means identifying and erasing symbols of the current environmental string. In case of verifying APCol systems, the agents only indicate that they "visit" a certain position in the current environmental string, i.e., rewrite the symbol at that position to some symbol but not to the empty word. A string is verified if the computation process is halting and for every i, $1 \leq i \leq n$ - supposed that the length of the input string is n -, each agent rewrites a symbol at position i in some of the environmental strings occurring in the computation process. This means that the agents "visit" each position (of the input string or that of its descendant), i.e., they verify that the environment. To perform a transition, the APCol systems work with the maximally parallel mode.

Definition 1. *Let $\Pi = (O, e, A_1, \ldots, A_n)$, $n \geq 1$, be and APCol system working with the maximally parallel mode. We say that Π verifies input string ω if the following conditions hold.*

- *There exists a halting computation c in Π where $c = (\omega; \omega_1, \ldots, \omega_n) \Longrightarrow \left(\omega^{(1)}; \omega_1^{(1)}, \ldots, \omega_n^{(1)}\right) \Longrightarrow \ldots \left(\omega^{(s)}; w_1^{(s)}, \ldots, w_n^{(s)}\right)$, $s \geq 1$, such that $|\omega| = |\omega^{(j)}|$ for $1 \leq j \leq s$.*
- *Computation c satisfies the following property. Let $\omega^{(0)} = \omega$. For every agent A_k, $1 \leq k \leq n$ and for every i, $1 \leq i \leq m$ where $|\omega| = m$, there exists j, $1 \leq j \leq s$ such that the symbol at the ith position of $\omega^{(j-1)}$ is letter b and A_k applies a rule $a \leftrightarrow b$ to this position of $\omega^{(j-1)}$.*
  Computation c is called a verification or a verifying computation.

The set of all words that can be verified by Π is called the language verified by Π. We call a word, resp. a language strongly verified if every computation of the word, resp. of every word in the language is verifying.

In the following we show that verifying APCol systems simulate one-way multihead finite automata.

Theorem 1. *Let $M = (Q, \Sigma, n, \delta, \triangleright, \triangleleft, q_0, F)$, $n \geq 1$, be an n-head finite automaton. Then we can construct an APCol system Π with $n+2$ agents such that any word w that can be accepted by M can strongly be verified by Π.*

Proof. To prove the statement, we construct an APCol system $\Pi = (O, e, A_{ini}, A_1, \ldots, A_n, A_{fin})$ such that each agent A_j, $1 \leq j \leq n$ simulates the work of the jth reading head of M and only that. Agent A_{ini} serves for initializing the simulation and agent A_{fin} checks whether the agents visited every position in the environmental string. The verifying process in Π corresponds to an accepting process in M: if a symbol a was scanned by reading heads $\{i_1, \ldots, i_r\} \subseteq \{1, \ldots, n\}$, then this fact will be indicated by a symbol $a^{(x)}$ in the environmental string of Π, where $x \in perm(i_1 \ldots 1_r)$; $i_1, \ldots, i_r$ are the numbers of reading heads that scanned symbol a. Every computation step in M is simulated by a sequence of computation steps performed by agents $A_1, \ldots A_n$.

The input word for Π is of the form $\triangleright w$.

To help the easier reading, we will present only the agents together with their programs.

Π has agent A_{ini} for initializing the simulation and also for assisting the checking whether every position has been visited or not.

It has the following programs:

(1) $\langle e \to q_{0,1}, e \to e \rangle$,

(2) $\langle q_{0,1} \leftrightarrow \triangleright, e \to e \rangle$,

(3) $\left\langle e \to e, \bar{a}^{(x)} \to a'^{(x)} \right\rangle$,

where q_0 is the initial state and $x \in perm(1 \ldots n)$, $a \in \Sigma$.

Programs (1) and (2) are for initializing the simulation, programs of type (3) are for checking whether or not all positions have been visited.

Every agent A_j simulates the work of the reading head j, $1 \leq j \leq n$.

For every transition relation

$$s \in \delta(q, a_1, \ldots, a_n)$$

of M, where $a_1, \ldots, a_n \in \Sigma \cup \{\varepsilon\}$, agent A_j, $1 \leq j \leq n-1$ has the following programs.

(We recall that if $a_j \neq \varepsilon$ for some j, $1 \leq j \leq k$, then the jth reading head is moved to the right, otherwise, if $a_j = \varepsilon$ it remains in its current position.)

(0) $\langle e \to \triangleright_j, e \to \triangleright'_j \rangle$,

(1) $\langle \triangleright'_j \leftrightarrow s_j, e \to e \rangle$,

(1a) $\langle \triangleright'_j \leftrightarrow s_j, \triangleright_j \leftrightarrow e \rangle$,

(1b) $\langle \triangleright'_1 \leftrightarrow q_1, e \to e \rangle$,

(1c) $\langle \triangleright'_1 \leftrightarrow q_1, \triangleright_1 \leftrightarrow e \rangle$,

(2) $\left\langle s_j \to s'_j, e \to a_j^{(xj)} \right\rangle$,
(2a) $\left\langle s_j \to s'_j, e \to a_j^{(x)} \right\rangle$,
(2b) $\left\langle q_1 \to s'_j, e \to a_j^{(xj)} \right\rangle$,
(2c) $\left\langle q_1 \to s'_j, e \to a_j^{(x)} \right\rangle$,
(3) $\left\langle a_j^{xj} \leftrightarrow \triangleright_j, s'_j \leftrightarrow a_j^{(x)} \right\rangle$,
(3a) $\left\langle s'_j \leftrightarrow \triangleright_j, a_j^{x} \leftrightarrow a_j^{(x)} \right\rangle$,

(4) $\left\langle \triangleright_j \leftrightarrow s'_j, a_j^{(x)} \to s_{j+1} \right\rangle$,
(4a) $\left\langle \triangleright_j \leftrightarrow s'_j, a_j^{(xj)} \to s_{j+1} \right\rangle$,

(5) $\left\langle s_{j+1} \leftrightarrow \triangleright'_j, s'_j \to e \right\rangle$,

where $x \in pref(perm(1 \dots n))$, $|x|_j = 0$, if $a_j \neq \varepsilon$ and $x \in pref(perm(1 \dots n))$ and $|x|_j = 1$ if $a_j = \varepsilon$. Furthermore, $y \in pref(perm(1 \dots n))$ and $|y|_j = 1$.

For $j = n$ and $s = (q', p_1, \dots, p_n)$, agent A_j has the same programs (0)-(3a), and programs (4),(4a) and (5) are changed as follows:

(5) $\langle \triangleright_n \to p'_1, e \to e \rangle$,
(6) $\langle p'_1 \leftrightarrow \triangleright_n, e \to e \rangle$,

We present a brief explanation of the programs. The first program, (0) is used for initialization. it generates two symbols of $\triangleright_j$ - the first to mark the position of reading head and the second to exchange for symbol of the simulated transition. The simulation of the move of the jth reading head starts with program (1). If it is simulation of the first step and the first head, then program (1c) is used. For the first use of other heads the program (1a) is used. The program (1b) is executed when the system simulates not the first step of computation. The input string has the form $q_j\alpha$, where q is the state in transition relation $s \in \delta(q, a_1, \dots, a_n)$, its subscript j refers to the jth reading head, and $\alpha \in \Sigma^*$ such that $|\alpha| = |w|$ (w is the input word). Then q_j is changed for $\triangleright_j$ in the environmental string, implying that no action of some other agent can be performed. Meantime, A_j changes s_j for s'_j and makes a guess whether symbol a_j is to be scanned or the reading head will remain at the same position. This is done by introducing $a_j^{(xj)}$ or $a_j^{(x)}$, programs (2) or (2a). Superscript xj refers to that letter a_j has not been scanned by reading head j (x is the sequence of the number of reading heads that already scanned this symbol). When agent A_1 does the same thing (uses one of programs (2b), (2c)), it also nondeterministically chooses transition s from the possible transitions $\delta(q, a_1, \dots, a_n)$ for given $q \in Q$ and arbitrary sequence of symbols $a_1, \dots, a_n$. After then, programs (3) or (3a) perform the corresponding action, namely change $\triangleright_j a_j^{(x)}$ to $a_j^{(xj)} s'_j$ or leave the two letters unchanged. (Notice that the order of rules in this type of programs is important). In the first case we simulate that the jth reading head scanned a_j, in the second case it remained at the same position. Note that if the reading head is in the position of cell with

symbol a, the symbol a is not marked as read in this moment. The symbol a is read when head is leaving the cell with this symbol. After then, by programs (4), (5), agent A_j will return to state $(\triangleright'_j, e)$ and the simulation of the move of the next reading head in transition s starts, i.e., the environmental string will have s_{j+1} as first letter. If $j = n$, then the first letter in the environmental string is changed to q'_1, meaning that M entered state q' and the simulation of the first reading head starts.

The computation is successful if it is halting and all positions have been visited by each agents. This is checked by agent A_{fin} and then A_{init}. The programs of A_{fin} are as follows.

(0) $\langle e \to e, e \to h \rangle$,
(1) $\langle e \leftrightarrow q_{f,1}, e \to h \rangle$,
(2) $\langle h \leftrightarrow c^{(z)}, e \to e \rangle$,
(3) $\langle c^{(x)} \to \bar{c}^{(x)}, e \to e \rangle$,
(4) $\langle \bar{c}^{(x)} \leftrightarrow h, e \to e \rangle$,
(5) $\langle c^{(y)} \leftrightarrow \#, e \to e \rangle$,
(6) $\langle h \leftrightarrow \triangleright_j, e \to e \rangle$,
(7) $\langle \triangleright_j \to e, e \leftrightarrow h \rangle$,
(8) $\langle \# \to \#, e \to e, \rangle$,

where $1 \leq j \leq n$, $c \in \Sigma$, $z = pref(perm(1 \dots n))$, $x = perm(1 \dots n)$, and $y \neq perm(1 \dots n)$.

Agent A_{fin} nondeterministically consumes all symbols in the environmental string. It checks if each of them has been visited by every agent. This is done by introducing symbol h in the environmental string. Then, programs of the form (3) and (4), indicates that the symbol was visited by all agents. If there is a symbol which does not satisfy this property (program (5)), then symbol # is introduced which implies that the system will never stop (program (8)). Suppose that this is not the case, then after a while A_{fin} will not be able to perform any of its programs. It is easy to see that A_{fin} visited all letters in the environmental word. To complete the proof, A_{init} has to visit all symbols as well. This is done by its programs of type (3). Since A_{fin} visited all symbols, A_{init} will do that too, thus, the computation halts.

The reader my observe that the agents simulate the transitions of M and that no agent can work simultaneously. Furthermore, programs of different agents cannot interfere. Thus, the language accepted by M can be verified by Π. Furthermore, every accepting computation of a word in Π is a verifying computation, thus Π strongly verifies the language accepted by Π.

4 Conclusions

In this paper we demonstrated a further connection between APCol systems and automata, as we proved that verifying APCol systems simulate one-way multihead finite automata. The new concept, the verifying computation opens further research directions: describing two-way multihead finite automata, jumping multihead finite automata in terms of APCol systems. We plan investigations in these topics in the future.

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. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E.: Towards P Colonies Processing Strings. In: Proc. BWMC 2014, Sevilla, 2014. pp. 103–118. Fénix Editora, Sevilla, Spain (2014)
2. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E.: P colonies processing strings. Fundamenta Informaticae 134(1-2), 51–65 (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. L. Ciencialová, E. Csuhaj-Varjú, L. Cienciala, and P. Sosík. P colonies. Bulletin of the International Membrane Computing Society 1(2):119–156 (2016).
5. 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)
6. Csuhaj-Varjú, E., Vaszil, G.: Finite dP Automata versus Multi-head Finite Automata In: Gheorghe, M. et. al. (eds.) CMC 2011, LNCS, vol. 7184, pp. 120-138. Springer-Verlag, Berlin Heidelberg (2012)
7. Holzer, M., Kutrib, M., Malcher, A.: Complexity of multi-head finite automata: Origins and directions, Theoretical Computer Science 412, 83–96 (2011)
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 (1992),
11. Meduna, A., Zemek, P.: Jumping Finite Automata. Int. J. Found. Comput. Sci. 23(7), 1555–1578 (2012)
12. Păun, Gh., Rozenberg, G., Salomaa, A.(eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)
13. Rozenberg, G., Salomaa, A.(eds.): Handbook of Formal Languages I-III. Springer Verlag., Berin-Heidelberg-New York (1997)