Further Results on the Power of Generating APCol Systems

Lucie Ciencialová[1], Luděk Cienciala[1], and Erzsébet Csuhaj-Varjú[2]

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

Summary. In this paper we continue our investigations in APCol systems (Automaton-like P colonies), variants of P colonies where the environment of the agents is given by a string and the functioning of the system resembles to the functioning of standard finite automaton. We first deal with the concept of determinism in these systems and compare deterministic APCol systems with deterministic register machines. Then we focus on generating non-deterministic APCol systems with only one agent. We show that these systems are as powerful as 0-type grammars, i.e., generate any recursively enumerable language. If the APCol system is non-erasing, then any context-sensitive language can be generated by a non-deterministic APCol systems with only one agent.

1 Introduction

Automaton-like P colonies (APCol systems, for short), introduced in [1], are variants of of P colonies (introduced in [10]) - very simple membrane systems inspired by colonies of formal grammars. The interested reader is referred to [14] for detailed information on membrane systems (P systems) and to [11] and [5] for more information to grammar systems theory. For more details on P colonies consult the surveys [9] and [4].

An APCol system consists of a finite number of agents - finite collections of objects in a cell - and their joint 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 string representing the environment 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. It can easily

be observed that APCol systems resemble automata. The current configuration of the system (the objects inside the agents) and the current string representing the environment correspond to the current state of the automaton and the currently processed input string.

The agents may perform rewriting, communication or checking rules [10]. A rewriting rule $a \to b$ allows the agent to rewrite one object a to object b. Rewriting rules are also called evolution rules. 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 the first part of this paper, we deal with both variants of modes of computation. In general, a computation of APCol system is non-deterministic. It means that in every configuration one set of maximal sets of applicable programs is non-deterministically chosen to be executed. We focus on such APCol systems that there exists only one maximal set of applicable programs in each configuration - deterministic APCol systems. The second part of this paper is devoted to non-deterministic generating APCol systems with one agent only.

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 [15, 14].

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 $|w|_a$.

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 Register machine

Definition 1. [13] *A register machine is a construct* $M = (m, H, l_0, l_h, P)$ *where*

- m *is the number of registers,*
- H *is the set of instruction labels,*
- l_0 *is the start label,*
- l_h *is the final label,*
- P *is a finite set of instructions injectively labelled with the elements from the set* H*.*

The instructions of the register machine are one of the following forms:

$l_1 : (ADD(r), l_2, l_3)$ Add 1 to the content of the register r and proceed to the instruction (labelled with) l_2 or l_3.

$l_1 : (SUB(r), l_2, l_3)$ If the register r stores the value different from zero, then subtract 1 from its content and go to instruction l_2, otherwise proceed to instruction l_3.

$l_h : HALT$ Halt the machine. The final label l_h is only assigned to this instruction.

If every ADD-instructions of M is of the form ADD-instruction $l_1 : (ADD(r), l_2)$, then M is called a deterministic register machine.

Register machine M accepts a set $N(M)$ of numbers in the following way: it starts with number $x \in N$ in the first register and all the other registers are empty (hence storing the number zero) and with the instruction labelled l_0. Then it proceeds to apply the instructions as indicated by the labels (and made possible by the contents of registers). If it reaches the halt instruction and all registers are empty, the input is said to be accepted by M and hence it is introduced in $N(M)$.

It is known that any recursively enumerable set of natural numbers can be accepted by a deterministic register machine with at most three registers.

Register machines can also generate sets of natural numbers. In this case the first register is dedicated as the output register. The register machine starts with empty registers (registers storing zero) and with instruction l_0. Then it proceeds with executing instructions, according to their labels. After halting, the generated number is the value of stored in the first register.

2.2 APCol systems

In the following we recall the notion of an APCol system (an automaton-like P colony) [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 or a rewriting 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 of rules, 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 computation step means a maximally parallel action of the active agents, i.e., a maximal number of agents that can perform at least one of their programs, has to execute such an action parallel. 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 and there are no more applicable programs in the whole system, and meantime at least one of the agents is in so-called final state.

An APCol system 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 the same 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 [12]) is properly included in the family of languages accepted by APCol systems with one agent. It was also proved that any recursively enumerable language can be obtained as a projection of a language accepted by an APCol system with two agents.

In the case of generating mode, the string w_F is generated by Π iff there exists 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 $1 \leq i \leq n$.

In both cases, instead of a string, we can work with the number of symbols in the string as the result of the computation. The set of natural numbers accepted or generated with an APCol system Π is denoted $N_{acc}(\Pi)$ or $N_{gen}(\Pi)$, respectively.

The family of sets of numbers generated by APCol systems with n agents is denoted by $NAPCol_{gen}(n)$, if we consider only restricted APCol systems, then we use notation $NAPCol_{gen}R(n)$. The family of recursively enumerable sets of natural numbers is denoted by NRE, and the family of sets of natural numbers acceptable by partially blind register machines is denoted by NRM_{pb}.

Results have been obtained about the generative power of APCol systems[3]:

- Restricted APCol systems with two agents working in generating mode generate any recursively set of natural numbers and conversely. Thus,

$$NAPCol_{gen}R(2) = NRE.$$

- The family of sets of natural numbers acceptable by partially blind register machines can be generated by restricted APCol systems with one agent and conversely. Thus,

$$NRM_{pb} \subseteq NAPCol_{gen}R(1).$$

3 Generative Power of APCol Systems

3.1 Deterministic APCol systems

The concept of determinism can be interpreted for APCol systems in several ways. Here we consider the following concept:

Let $c = (w_1, \ldots, w_n; w_E)$ be an arbitrary configuration of an APCol system $\Pi = (O, e, A_1, \ldots, A_n)$, $n \geq 1$. We say that Π is deterministic if there is only one maximal applicable multiset of programs M_P in Π that can be applied to c.

We can construct an n-tuple x_c of strings of length two, $x_i y_i$ corresponding to the string that agent i consumes from the environmental string by applying a program from M_P. If there is rewriting rule in the program, then e appears in the string $x_i y_i$. If some agent has no applicable program then it is represented by ee in the x_c. Let O be a set of objects and Σ is input alphabet, then let $f : O \to \Sigma^*$ be a function defined as follows: $\forall x \in O - \{e\} f(x) = a; f(e) = \varepsilon$ and

$$u_0\ a_{i_1} b_{i_1}\ u_1\ a_{i_2} b_{i_2}\ u_2 \ldots u_{n-1}\ a_{i_n} b_{i_n}\ u_n = w_E \tag{1}$$

We focus on deterministic APCol system working in generating mode.

The deterministic APCol system working in generating mode starts its computation in initial configuration given by initial contents of agents and with empty string as environmental string. By execution of programs it passes from one configuration to another one. Notice that to every configuration there is only one maximal set of programs such that environmental string is formed as (1). The result of the computation - environmental string - is obtained only if APCol system halts and at least one agent is in final state.

Theorem 1. *Let M be a deterministic register machine. Then there exists a deterministic APCol system Π with two agents such that M and Π generate the same set of natural numbers.*

Idea of the proof:

The environmental string stores information about the contents of the registers. It is in a form $\#111\ldots1222\ldots2333\ldots n\#'$. When ADD-instruction is performed on register r, then an agent puts $\downarrow$ just after $\#$ and moves $\downarrow$ through the string from the left to the right until the agent consumes the number $s \geq r$. Then the agent insert new symbol r just before s, deletes $\downarrow$ and generates the label of the next instruction performed by register machine.

The idea how to do zero-check, and thus subtraction $(l_1 : (SUB(r), l_2, l_3))$ is the following: Content of register r is represented by the number of objects r in the environmental string. If the agent needs to erase some r, then it places mark $\uparrow$ just after $\#$ and moves it through the string. If there is any object r, then the agent erases it and generates label l_2. If there is no r, then the agent consumes $\uparrow$ together with s $(s > r)$ or $\#'$ it generates label l_3.

If the next instruction is the halt instruction, then the agent exchanges $\#$ with $\updownarrow$ and pushes it through the string. it leaves symbol 1 unchanged, the symbols representing the contents of other registers are deleted. Finally, if agent consume $\#'$, then it erases $\updownarrow$ and stops working.

It can be seen that the instruction of the register machine can be simulated by the deterministic APCol system.

3.2 APCol systems with one agent

In this part we deal with APCol systems with only one agent and working in non-deterministic manner.

By the analogy of a non-decreasing Chomsky grammar, we introduce the notion of a non-decreasing APCol systems working in the generating mode. We say that an APCol system Π is non-decreasing, if no agent of Π has a rule of the form $e \leftrightarrow y$, i.e., there is no rule for erasing a symbol from the string representing the environment.

We first show that any ε-free context-sensitive language can be generated by an APCol system with only one agent. Furthermore, the APCol system is non-decreasing.

Theorem 2. *Any ε-free context-sensitive language can be generated by a non-decreasing APCol system with only one agent.*

Sketch of the proof:

We show that to every context-sensitive grammar $G = (N, T, P, S)$ in Kuroda normal form there exists an APCol system Π with one agent working in generating

mode such that $L(G) = L(\Pi)$. To do this, we construct an APCol system Π such that every generation in G can be simulated by a computation in Π and any successful computation in Π corresponds to a terminating derivation in G. To help the easier reading, we provide only the description of the simulation of the rules of G, the other components of Π can be extracted from the descriptions below.

At the beginning of the simulating computation in Π we need to initialize environmental string - there is only starting non-terminal at the beginning of derivation in G. Let symbols $X, X' \notin N \cup T$.

Initialization

Agent	Program	String
(ee)	$\langle e \to S; e \to X' \rangle$	ε
(SX')	$\langle S \leftrightarrow e; X' \to X \rangle$	ε
(eX)		S

There are four types of rules in grammar in Kuroda normal form: $AB \to CD$; $A \to BC; A \to B$ and $A \to a$, $a \in T$, $A, B, C, D \in N$.

To every rule in form $AB \to CD$, to the set of programs of the agent A, we add the following set of programs:

$p_i : AB \to CD$

Agent	Program	String
(eX)	$\langle e \to p_i; X \to X' \rangle$	$u\ AB\ v$
(p_iX')	$\langle p_i \leftrightarrow A; X' \leftrightarrow B \rangle$	$u\ p_i'X'\ v$
(AB)	$\langle A \to p_i''; B \to C \rangle$	$u\ p_i'X'\ v$
$(p_i''C)$	$\langle p_i'' \to p_i'''; C \leftrightarrow p_i' \rangle$	$u\ p_i'X'\ v$
$(p_i'''p_i')$	$\langle p_i''' \to q_i; p_i' \to D \rangle$	$u\ CX'\ v$
(q_iD)	$\langle q_i \to q_i'; D \leftrightarrow X' \rangle$	$u\ CX'\ v$
$(q_i'X')$	$\langle q_i' \to X; X \to e \rangle$	$u\ CD\ v$
(Xe)		$u\ CD\ v$
$(p_i''C)$	$\langle p_i'' \to A; C \to B \rangle$	$u\ p_i'X'\ v$

The program $\langle p_i'' \to A; C \to B \rangle$ can be used just after program $\langle p_i'' \to p_i'''; C \leftrightarrow p_i' \rangle$ and these two programs can cause loop in computation. This is because there can exist more than one rule with AB on the left side and the agent can generate label of a rule different from p_i.

For every rule in a form $A \to BC$ we add the following programs to the set of programs of A:

$p_i : A \to BC$

Agent	Program	String
(eX)	$\langle e \to p_i; X \to X' \rangle$	$u\ A\ v$
(p_iX')	$\langle p_i \to p_i'; X' \leftrightarrow A \rangle$	$u\ A\ v$
$(p_i'A)$	$\langle p_i' \to p_i''; A \to B \rangle$	$u\ X'\ v$
$(p_i''B)$	$\langle p_i'' \leftrightarrow X'; B \leftrightarrow e \rangle$	$u\ X'\ v$
$(X'e)$	$\langle X' \leftrightarrow p_i''; e \to e \rangle$	$u\ Bp_i''\ v$
$(p_i''e)$	$\langle p_i'' \to p_i'''; e \to C \rangle$	$u\ BX'\ v$
$(p_i'''C)$	$\langle p_i''' \to p_i'''; C \leftrightarrow X' \rangle$	$u\ BX'\ v$
$(p_i'''X')$	$\langle p_i''' \to e; X' \to X \rangle$	$u\ BC\ v$

For every rule in a form $A \to B$ or $A \to a$ we add the following programs to the set of programs of A (α is non-terminal or terminal symbol):

$p_i : A \to \alpha$	Agent Program	String
(eX)	$\langle e \to p_i; X \to \alpha \rangle$	$u\ A\ v$
$(p_i\alpha)$	$\langle p_i \to p_i'; \alpha \leftrightarrow A \rangle$	$u\ A\ v$
$(p_i'A)$	$\langle p_i' \to e; A \to X \rangle$	$u\ \alpha\ v$

At the end we add one more set of programs to the set of programs of A that is of the form

$$\langle e \to Y; X \leftrightarrow A \rangle ; \langle Y \to Y; A \to A \rangle$$

where $Y \notin N \cup T$ and $A \in N$. By execution of the first program, the computation enters a loop and the computation never halts. We have to add these programs to ensure that the computation will not halt if the derivation of a string in the grammar stops with a non-terminal in the string.

The agent simulates execution of rules and computation ends only when there is no non-terminal symbol in environmental string. By the definition of the rules and the programs above, it can easily be seen that the program set of agent A can only simulate the rules of G, furthermore any computation in Π successfully halts only if the corresponding derivation successfully terminates in G. We also observe that Π is non-decreasing.

It is known that any recursively enumerable language can be generated by a Kuroda-like normal form grammar $G = (N, T, P, S)$, where the rules are one of the forms $AB \to CD$, $A \to BC, A \to B$, $A \to a$, and $A \to \varepsilon$, where $a \in T$, $A, B, C, D \in N$. Modifying the proof of the above theorem (simulation of rule $A \to \alpha$), we can extend the proof to obtain any recursively enumerable language. The modification is the following: in the case of $A \to \varepsilon$ we use rule $e \leftrightarrow A$ in the corresponding rule set.

We also note that to generate a context-sensitive language which contains ε, we have an extension of the Kuroda normal form grammar where the rules are one of the forms $AB \to CD$, $A \to BC, A \to B$, $A \to a$, and $S \to \varepsilon$, where $a \in T$, $A, B, C, D \in N$, and S does not appear at the right-hand side of any rule. It is easy to see that the proof of the above theorem can be modified to be a proof of this statement as well.

Let CS, CS^{ε}, RE denote the family of ε-free context-sensitive, context-sensitive, and recursively enumerable languages, respectively. Since APCol systems (both in the generating and in the accepting mode) can be simulated by Turing machines, we obtain the following statement

Theorem 3.

$$CS \subset CS^{\varepsilon} \subset RE = APCol_{gen}(1).$$

4 Conclusion

In this paper we examined APCol systems working in generating mode. We defined a deterministic version of APCol systems and showed that they are able to simulate

functioning of deterministic register machines. In second part of paper we focused on generating APCol systems with only one agent. We have showed that these systems generate the family of recursively enumerable languages, and if they are non-decreasing (have no rule for decreasing the length of the environment).

Acknowledgments.

The work of L. Ciencialová and L. Cienciala 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/11/2019. The work of E. Csuhaj-Varjú was supported by Grant No. K 120558 of the National Research, Development, and Innovation Office, Hungary.

References

1. 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)
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. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading, Mass. (1979)
9. 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)
10. 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)
11. Kelemen, J., Kelemenová, A.: A Grammar-Theoretic Treatment of Multiagent Systems. Cybern. Syst. 23(6), 621–633 (1992),
12. Meduna, A., Zemek, P.: Jumping Finite Automata. Int. J. Found. Comput. Sci. 23(7), 1555–1578 (2012)

13. Minsky, Marvin L.: Computation: Finite and Infinite Machines. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1967)
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)