Journal of Membrane Computing (2019) 1:178–197
https://doi.org/10.1007/s41965-019-00019-w

SURVEY PAPER

P colonies

Survey

Lucie Ciencialová[1] · Erzsébet Csuhaj-Varjú[2] · Luděk Cienciala[1] · Petr Sosík[1]

Received: 23 December 2018 / Accepted: 20 July 2019 / Published online: 6 August 2019
© Springer Nature Singapore Pte Ltd. 2019

Abstract
P colonies are abstract computing devices modelling communities of very simple reactive agents living and acting in a joint shared environment. The concept was motivated by so-called colonies, grammar systems based on interplay of very simple agents, on one hand, and by membrane systems, massively parallel computational models inspired by cell biology, on the other hand. Some variants of P colonies also allow the environment to participate actively in the system's evolution. In this paper we summarize the most important results on P colonies, present open problems concerning these constructs, and suggest new research directions in their study.

Keywords P colony · Membrane systems · PCol automata · 2D P colonies

1 Introduction

In contemporary computer science, there has been a growing demand for reliable and efficient computing devices to describe the behaviour of communities of dynamically changing agents which are in interaction with their shared environment. Multi-agent systems with very simple reactive agents are of special interest, in particular with respect to their emerging behaviour and the limits of their power.

P colonies, introduced in [44], were motivated by these problems. They are variants of very simple tissue-like P systems, where the agents (the cells) have only one region and they interact with their shared environment using programs (collections of rules of special form). P systems (or membrane systems), introduced in [51], are a family of computing devices inspired by biology and biochemistry of cells. Colonies of simple formal grammars, also motivating P colonies, were introduced in [42].

During the years, P colonies have been studied in detail; a summary of results can be found in [45].

Although several variants of P colonies have been developed, all of them have some common basic features. Inside each agent (each cell) there is a finite multiset of objects. These objects are processed by a finite set of programs associated to the agent. The number of objects inside each agent is constant (does not change) during the functioning of the agent community and it is called the capacity of the P colony. The agents share an environment which is represented by a multiset of objects. One type of these objects, called the environmental object, is distinguished, and it is supposed to be in a countably infinite number of copies in the environment. (In the literature, the reader may also find that the environmental symbol appears in an arbitrarily large number of copies in the environment).

Using their programs, the agents can change the objects present at their disposal and can exchange some of their objects with objects present in the environment. These synchronized actions correspond to a configuration change (a transition) of the P colony; a finite sequence of consecutive configuration changes starting from the initial configuration

✉ Lucie Ciencialová
lucie.ciencialova@fpf.slu.cz

Erzsébet Csuhaj-Varjú
csuhaj@inf.elte.hu

Luděk Cienciala
ludek.cienciala@fpf.slu.cz

Petr Sosík
petr.sosik@fpf.slu.cz

[1] Institute of Computer Science and Research Institute of the IT4Innovations Centre of Excellence, Silesian University in Opava, Opava, Czech Republic

[2] Department of Algorithms and Their Applications, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary

is a computation. The result of the computation is the number of copies of a distinguished object, called the final object, present in the environment in a final configuration of the P colony.

It can easily be seen that the environment is both a communication channel for the agents and a storage for the objects. It plays strategic role in synchronizing the work of the agents during the computation.

One major research topic in the theory of P colonies is the study of their computational power related to their descriptional complexity. These investigations focus on the question of identifying how many components are necessary and to what extent the programs can be simplified to obtain a certain computational power. In addition to these problems, the working modes of P colonies have obtained attention as well, whether or not parallelism in the joint work of the agents plays significant role in increasing the expressive power of P colonies.

The rules of P colonies demonstrate strong similarities with instructions or rules of some well-known computing devices (register machines, rewriting systems based on point mutations, other variants of membrane systems), thus comparisons of these constructs with other classical and non-classical computing devices are also of interest.

P colonies, due to their original motivation, model multi-agent systems (complex systems) acting in an environment. According to the basic definitions, the objects present in the environment have significant role in the change of the states of the agents. Therefore, one of the research directions in the theory is devoted to studying the role of the dynamics of the environment in the behaviour of P colonies, i.e., the case when the objects in the environment are provided step-by-step not only by the actions of the agents but by some special object provider device.

Due to their simplicity and distributed nature, P colonies are convenient tools for modelling complex systems as robot collections, sender and consumer systems, eco-systems. We expect several new areas of applications in the future.

This paper is an extended and revised version of survey [11].

2 Notations

We assume that the reader is familiar with formal language and automata theory, computability, and the basics of membrane computing [50, 54].

Throughout the paper we use the following notions and notations. Let Σ be the alphabet and let Σ^* be the set of all words over Σ (including the empty word ε). The length of a word $w \in \Sigma^*$ is denoted 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 $M = (V, f)$, 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. Each multiset of objects M with the set of objects $V' = \{a_1, \dots, a_n\}$ can be represented as a string w over alphabet V', where $|w|_{a_i} = f(a_i)$; $1 \le i \le n$. Obviously, all words obtained from w by permuting the letters represent the same multiset M. Symbol ε represents the empty multiset. The set of all multisets with the set of objects V is denoted by V^*. The cardinality of M, denoted by $|M|$, is defined by $|M| = \sum_{a \in V} f(a)$.

The set of all non-negative integers is denoted by N. We use REG, CF and RE as notations for the families of regular, context-free and recursively enumerable languages. The family of languages accepted by matrix grammars without appearance checking and with erasing rules is denoted by $\mathrm{MAT}^{\varepsilon}$ and the family of languages generated by interactionless L systems is denoted by $0L$. NRE denotes the family of recursively enumerable set of non-negative integers.

Definition 1 [48] 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 of the following forms:

$l_1 : (\mathrm{ADD}(r), l_2, l_3)$	Add 1 to the content of the register r and proceed to the instruction (labelled with) l_2 or l_3.
$l_1 : (\mathrm{SUB}(r), l_2, l_3)$	If the register r stores a value different from zero, then subtract 1 from its content and go to instruction l_2, otherwise proceed to instruction l_3.
$l_h : \mathrm{HALT}$	Halt the machine. The final label l_h is only assigned to this instruction.

Without loss of generality, one can assume that in each ADD-instruction $l_1 : (\mathrm{ADD}(r), l_2, l_3)$ and in each SUB-instruction $l_1 : (\mathrm{SUB}(r), l_2, l_3)$ the labels l_1, l_2, l_3 are mutually distinct.

The register machine M computes a set $N(M)$ of numbers in the following way: it starts with all registers empty (hence storing the number zero) with the instruction labelled l_0 and 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, then the number stored at that

time in the register 1 is said to be computed by M and hence it is introduced in $N(M)$. It is known (see e.g. [48]) that in this way we compute all Turing computable sets of numbers.

The register machine is called partially blind if the SUB -instruction is executed as follows: if register r stores a non-zero value then this value is decreased by one and the next instruction will be l_2 or l_3, otherwise the computation aborts. When the partially blind register machine enters the final state, the result obtained in the first register is only taken into account if the remaining registers store value zero. The family of sets of non-negative integers generated by partially blind register machines is denoted by $\mathrm{NRM}_{\mathrm{pb}}$. The partially blind register machines accept a proper subfamily of *NRE*.

3 The basic models of P colonies

In this section we provide the generic version of a P colony and some of its important variants.

3.1 P colonies with rewriting and communication rules

The original concept of a P colony was introduced in [44] and presented in a developed form in [31, 43].

Definition 2 A P colony of capacity k, $k \geq 1$, is a construct $\Pi = (A, e, f, v_E, B_1, \ldots, B_n)$, where

- A is an alphabet, its elements are called objects;
- $e \in A$ is the basic (or environmental) object of the colony;
- $f \in A$ is the final object of the colony;
- v_E is a finite multiset over $A - \{e\}$, called the initial state (or initial content) of the environment;
- B_i, $1 \leq i \leq n$, are agents, where each agent $B_i = \left(o_i, P_i\right)$ is defined as follows:
  - o_i is a multiset over A consisting of k objects, the initial state (or the initial content) of the agent;
  - $P_i = \left\{p_{i,1}, \ldots, p_{i,k_i}\right\}$ is a finite set of programs, where each program consists of k rules, which are in one of the following forms each:
    - $a \to b$, $a, b \in A$, called an evolution rule;
    - $c \leftrightarrow d$, $c, d \in A$, called a communication rule;
    - r_1/r_2, called a checking rule; r_1, r_2 are both evolution rules or both communication rules.

We add some brief explanations to the components of the P colony.

We first note that throughout the paper, we use term "object a inside agent B" and term "$a \in w$, where w is the state of agent B" as equivalent.

The first type of rules associated to the programs of the agents, the evolution rules, are of the form $a \to b$. This means that object a inside the agent is rewritten to (evolved to be) object b.

The second type of rules, the communication rules, are of the form $c \leftrightarrow d$. If a communication rule is performed, then object c inside the agent and object d in the environment swap their location. Thus, after executing the rule, object d appears inside the agent and object c is located in the environment.

The third type of rules are the checking rules. A checking rule is formed from two rules of one of the two previous types. If a checking rule r_1/r_2 is performed, then the rule r_1 has higher priority to be executed over the rule r_2. This means that the agent checks whether or not rule r_1 is applicable. If the rule can be executed, then the agent must use this rule. If rule r_1 cannot be applied, then the agent uses rule r_2.

We note that these types of rules are the basic ones; in some variants of P colonies other types of rules have been also considered. We will discuss them in later sections.

The program determines the activity of the agent: the agent can change its state and/or the state of the environment.

The environment is represented by a finite number (zero included) of copies of non-environmental objects and a countably infinite copies of the environmental object e.

When an agent executes a program, then each object inside the agent is affected. Depending on the rules in the program, the program execution may affect the environment as well. This interaction between the agents and the environment is the key factor of the functioning of the P colony.

The functioning of the P colony starts from its initial configuration (initial state).

The initial configuration of a P colony is an $(n+1)$-tuple of multisets of objects present in the P colony at the beginning of the computation. It is given by the multisets o_i for $1 \leq i \leq n$ and by multiset v_E. Formally, the configuration of the P colony Π is given by $(w_1, \ldots, w_n, w_E)$, where $|w_i| = k$, $1 \leq i \leq n$, w_i represents all the objects present inside the ith agent, and $w_E \in (A - \{e\})^*$ represents all the objects in the environment different from the object e.

At each step of the computation (at each transition), the state of the environment and that of the agents changes in the following manner: in the maximally parallel derivation mode, each agent which can use any of its programs should use one (non-deterministically chosen), whereas in the sequential derivation mode, only one agent at a time is allowed to use one of its programs (non-deterministically chosen). If the number of applicable programs for an agent is higher than one, then the agent non-deterministically chooses one of the programs.

A sequence of transitions is called a computation. A computation is said to be halting, if a configuration is obtained where no program can be applied anymore. With

a halting computation, we associate a result which is given as the number of copies of the objects f present in the environment in the halting configuration.

Because of the non-determinism in choosing the programs, starting from the initial configuration we obtain several computations, hence, with a P colony we can associate a set of numbers, denoted by $N(\Pi)$, computed by all possible halting computations of given P colony.

In the original model (see [44]) the number of objects inside each agent is set to two, and the programs were formed from only two rules. Moreover, the initial configuration was defined as $(n+1)$-tuple $(ee, \dots, ee, \varepsilon)$ so at the beginning of the computation the environment of the P colony is "empty", it is without an input information.

The number of agents in a given P colony is called the degree of Π; the maximal number of programs of an agent of Π is called the height of Π and the number of the objects inside an agent is the capacity of Π. The family of all sets of numbers $N(\Pi)$ computed as above by P colonies of capacity at most $c \geq 0$, degree at most $n \geq 0$ and height at most $h \geq 0$, using checking programs, and working in the sequential mode is denoted by $\mathrm{NPCOL}_{\mathrm{seq}}K(c, n, h)$; whereas the corresponding families of P colonies working in the maximally parallel way are denoted by $\mathrm{NPCOL}_{\mathrm{par}}K(c, n, h)$. If one of the parameters n or h is not bounded, then we replace it with $*$. If only P colonies using programs without checking rules are considered, then we omit the K.

Although P colonies are very simple computing devices, due to their (mainly parallel) working mode and distributed nature they demonstrate large expressive (computational) power. In most cases, computational completeness can be obtained with these constructs even with very few components and very few restrictions on the programs. In this section, we briefly summarize some important results concerning their expressive power. Most of the statements are based on simulations of register machines, thus providing further knowledge on the nature of these classical computing devices as well.

To demonstrate a connection between P colonies and register machines, we add an example how the *ADD*-instruction of a register machine can be simulated by a P colony.

Example 1 Let $\Pi = (A, e, f, v_E, B)$ be the P colony with capacity two and let the current content of the agent be $l_1 e$. Let $M = (m, H, l_0, l_h, P)$ be a non-deterministic register machine with m registers. The *ADD*-instruction $l_1 = (\mathrm{ADD}(r), l_2, l_3)$ of M can be simulated by the following programs associated with the agent:

P :

$$\begin{aligned} 1 &: \left\langle l_1 \to l_1'; e \to a_r \right\rangle; \\ 2 &: \left\langle l_1' \to l_2; a_r \leftrightarrow e \right\rangle; \\ 3 &: \left\langle l_1' \to l_3; a_r \leftrightarrow e \right\rangle. \end{aligned}$$

At the beginning, objects l_1 and e are placed inside the agent. The content of register r is encoded to the number of objects a_r placed in the environment. The computation is done in such a way that the agent rewrites its content to $l_1' a_r$ using the first program (there are two rewriting rules in it). In the second step the agent rewrites l_1' to object corresponding to the label of the next instruction l_2 (or l_3) to be executed and it puts object a_r into the environment.

3.1.1 Restricted P colonies

By [44], P colonies of capacity two are computationally complete. Furthermore, their programs have special forms: one of the rules is an evolution rule and the other one is either a communication rule or a checking rule with two communication rules.

These variants of P colonies are called restricted P colonies.

The family of all sets of numbers computed by restricted P colonies without checking rules and with parameters c, n, h and working modes par and seq, see above, is denoted by $\mathrm{NPCOL}_{\mathrm{par}}R(c, n, h)$ or $\mathrm{NPCOL}_{\mathrm{seq}}R(c, n, h)$, respectively. If the restricted P colonies are with checking rules, then we add K in front of R.

Let us have one more example.

Example 2 Let $\Pi = (A, e, f, v_E, B)$ be the P colony with capacity two and let the current contents of the agent be $l_1 e$. Let $M = (m, H, l_0, l_h, P)$ be a non-deterministic register machine with m registers. The *ADD*-instruction $l_1 = (\mathrm{ADD}(r), l_2, l_3)$ can be simulated by the following programs associated with the agent:

P :

$$\begin{aligned} 1 &: \langle e \to a_r; l_1 \leftrightarrow e \rangle; & 3 &: \langle l_1 \to l_2; d \leftrightarrow e \rangle; \\ 2 &: \langle e \to d; a_r \leftrightarrow l_1 \rangle; & 4 &: \langle l_1 \to l_3; d \leftrightarrow e \rangle. \end{aligned}$$

At the beginning, objects l_1 and e are inside the agent. The content of register r is encoded to the number of objects a_r placed in the environment. The computation is done in such a way that at one computational step the agent must rewrite one of the objects inside it and should exchange the other one with an object from the environment. At the first step

the agent rewrites e to object a_r and sends object l_1 into the environment. In the second step it rewrites e to auxiliary object d and exchanges object a_r and l_1 from the environment. At the last step the agent rewrites object l_1 to the object corresponding to the label of the next instruction l_2 (or l_3) to be executed and it puts object d into the environment.

For restricted P colonies, using the maximally parallel working mode, the following results hold:

- $NPCOL_{par}KR(2,*,5) = NRE$ in [31, 44],
- $NPCOL_{par}R(2,*,5) = NRE$ in [35],
- $NPCOL_{par}K(2,*,4) = NRE$ in [31],
- $NPCOL_{par}KR(2,1,*) = NRE$ in [35],
- $NPCOL_{par}R(2,2,*) = NRE$ in [16].

The reader can easily see that the family of sets of natural numbers computed by restricted P colonies with or without the use of checking rules having at most five programs associated with agent equals to NRE. If we remove the restriction on the type of rules in the programs, P colonies need only at most four programs associated with every agent to obtain computational completeness. The difference in the last two results demonstrates the power of checking rules and the power of synchronized cooperation. To generate NRE, the restricted P colonies need only one agent if the agent can use checking rules and two agents if they are not equipped with checking rules.

The maximally parallel application of rules does not necessarily add power, as the following results demonstrate:

- $NPCOL_{seq}KR(2,*,5) = NRE$ in [35],
- $NPCOL_{seq}KR(2,1,*) = NRE$ in [35],
- $NPCOL_{seq}K(3,*,6) = NRE$ in [31, 43].

However, if only restricted P colonies with the sequential working modes are considered, the maximal computation power to be obtained is equal to the recognition power of blind counter machines, thus significantly reduced, irrespectively from the number of programs and agents in the P colony.

Notice that the property "restricted" demonstrates strong similarity to some normal forms of variants of regulated grammars, where some of the production is used for programming the action and some other production is responsible for its execution. Using some well-organized synchronizing mechanisms, simulation of standard P colonies with restricted ones can be demonstrated, thus, we may consider restricted P colonies as "normal forms" for the family of P colonies.

We note that the idea of restriction can be extended, with prescribing the ratio of evolution and communication rules in the programs of capacity k, $k \geq 2$.

3.1.2 Homogeneous P colonies

If each program in the P colony consists of rules of the same type, then we can call the P colony homogeneous. For a P colony with capacity two, this means that the program is formed from two evolution rules, or two communication rules, or two checking rules of the same type.

Indicating by symbol H that homogeneous P colonies are considered, the following results were obtained:

- $NPCOL_{par}KH(2,*,4) = NRE$ in [17],
- $NPCOL_{par}KH(2,1,*) = NRE$ in [17].

As for the previous variants, we provide an example.

Example 3 Let $\Pi = (A, e, f, v_E, B)$ be the P colony with capacity two and let the current content of the agent be $l_1 e$. Let $M = (m, H, l_0, l_h, P)$ be a non-deterministic register machine with m registers. The ADD-instruction $l_1 = (ADD(r), l_2, l_3)$ can be simulated by the following programs associated with the agent:

P :

$$\begin{array}{llll} 1: \langle l_1 \to l_1'; e \to a_r \rangle; & & 4: \langle l_1' \to l_2; e \to e \rangle; \\ 2: \langle l_1' \leftrightarrow e; a_r \leftrightarrow e \rangle; & & 5: \langle l_1' \to l_3; e \to e \rangle. \\ 3: \langle e \leftrightarrow l_1'; e \leftrightarrow e \rangle; & & \end{array}$$

At the beginning, objects l_1 and e are placed inside the agent. The content of register r is encoded to the number of objects a_r placed in the environment. The computation is done in such a way that the agent at one computational step must rewrite all object inside it or must exchange all of its objects with objects from the environment. At the first step, the agent rewrites multiset $l_1 e$ to multiset $l_1' a_r$. In the second step, it sends both objects of multiset $l_1' a_r$ into the environment. At the third step, it consumes objects of multiset $l_1' e$ and at the last step the agent rewrites object l_1 to the object corresponding to the label of the next instruction to be executed, namely, l_2 (or l_3).

The results that have been recalled so far concern mainly P colonies with agents of capacity at least two. It is a challenging question, whether the work of agents with capacity one, i.e., with agents having only one object inside can be organized in such way that they obtain the same power as P colonies in the general sense. Notice that in this case the objects play more important role in the synchronization of the work of the agents. The following results give positive answer to this question.

- $NPCOL_{par}K(1,*,5) = NRE$ in [18],
- $NPCOL_{par}KH(1,*,6) = NRE$ in [17],

Table 1 Computational complete classes of P colonies

No.	Mode of comp.	Capacity	Degree	Height	Checking rules/restricted programs/homogeneous programs			
*Results with one * parameter*								
1.	par	1	*	5	K			in [18]
2.	par	1	*	6	K		H	in [17]
3.	par	1	4	*	K			in [16]
4.	par	1	6	*				in [23]
5.	par	2	*	8				in [31]
6.	par	2	*	5	K	R		in [31, 44]
7.	seq	2	*	5	K	R		in [35]
8.	par	2	*	5		R		in [35]
9.	par	2	*	4	K			in [31]
10.	par	2	*	4	K		H	in [17]
11.	seq	2	*	4	K			in [43]
12.	seq/par	2	1	*	K	R		in [31, 35]
13.	par	2	2	*		R		in [16]
14.	seq/par	2	1	*	K		H	in [17]
15.	seq/par	3	*	3	K			in [31, 43]
16.	par	3	2	*			H	in [8]
Results with all parameters bounded								
17.	par	1	3	325	K		H	in [9]
18.	par	2	23	5	K	R		in [32]
19.	par	2	22	6	K	R		in [32]
20.	par	2	22	5	K			in [32]
21.	par	2	92	3			H	in [9]
22.	par	2	70	5			H	in [9]
23.	seq/par	2	1	74	K	R		in [9]
24.	seq/par	2	1	66	K			in [9]
25.	par	2	2	163			H	in [9]
26.	par	2	35	8				in [32]
27.	par	2	57	8		R		in [32]
28.	par	3	35	7				in [32]

- $NPCOL_{par}K(1, 4, *) = NRE$ in [16],
- $NPCOL_{par}(1, 6, *) = NRE$ in [23].

Finally, we provide two more interesting results dealing with P colonies with capacity three [8, 31].

- $NPCOL_{par}K(3, *, 3) = NRE$ in [31],
- $NPCOL_{par}H(3, 2, *) = NRE$ in [8].

In Table 1 the reader can find a summarized list of results concerning the computational complete variants of P colonies.

3.2 P colonies with senders and consumers

In [23] new types of programs for P colonies with two objects inside each agent were introduced. The first of them is a deletion program—$\langle a_{in}; bc \rightarrow d\rangle$; using this program an agent consumes one object (a) from the environment and transforms the two objects (b, c) inside the agent into a new one (d). The second type is an insertion program in the form $\langle a_{out}; b \rightarrow cd\rangle$. By executing this program, the agent sends to the environment one object (a) and generates two new objects (c, d) from the other object (b). The concept resembles to the provider/customer architecture.

Example 4 [23] (a) A P colony with one sender cell can generate the Parikh set of a regular language $L \subseteq T^*$. Let $G = (N, T, P, S)$ be a regular grammar such that $L(G) = L$.

For generating the Parikh vectors of the words in L, we use, for each $S \rightarrow aB$ of P, the programs $\langle e, out; e \rightarrow eS\rangle$, $\langle e, out; S \rightarrow aB\rangle$ and then $\langle x, out; A \rightarrow aB\rangle, x \in T$ for every $A \rightarrow aB$ in P. Finally, for every rule of the form $A \rightarrow a$ we need $\langle x, out; A \rightarrow aF\rangle, x \in T$, $\langle a, out; F \rightarrow FF\rangle$, where $F \notin T \cup N$.

(b) A P colony with one consumer cell can "consume" the Parikh set of a regular language L. To see this, let $M = (Q, T, \delta, q_0, F)$ be a deterministic finite automaton such that $L(M) = L$.

We need the program $\langle e, in; ee \to q_0\rangle$, and to every transition $\delta(q_i, a) = q_j$ in M, we introduce $\langle a, in; xq_i \to q_j\rangle, x \in T \cup \{e\}$. If $q_j \in F$ in $\delta(q_i, a) = q_j$ we have to add the programs $\langle a, in; xq_i \to E\rangle, x \in T$, where $E \notin Q \cup T$.

In [8] the authors showed that P colonies with one sender and one consumer and some initial content in the environment are computationally complete. In [23] the authors proved that P colonies with senders and consumers with three agents and with only environmental objects in the initial configuration can generate every recursively enumerable set of natural numbers.

- $NPCOL_{sc}(3, *) = NRE$ in [23].
- $NPCOL_{sc}(2, *, ini) = NRE$ in [8, 24].

3.3 P colonies with evolving environment and generalized P colonies

The environment is static in the basic model, it can be changed only by the activity of the agents. Eco-P colonies were constructed as a natural extension of P colonies with dynamically evolving environment, the evolution does not depend only on the activity of agents. The mechanism of evolution in the environment is based on an 0L scheme. An 0L scheme is a pair (Σ, P), where Σ is the alphabet of 0L scheme and P is the set of context-free rules. It fulfils the following condition: for all $a \in \Sigma$ there exists $\alpha \in \Sigma^*$ such that $(a \to \alpha) \in P$. For $w_1, w_2 \in \Sigma^*$ we write $w_1 \Rightarrow w_2$ if $w_1 = a_1a_2 \dots a_n, w_2 = \alpha_1\alpha_2 \dots \alpha_n$, for $a_i \to \alpha_i \in P, 1 \le i \le n$.

Definition 3 A generalized P colony with capacity $k \ge 1$ is a construct

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

- A is the alphabet of the generalized P colony, its elements are called objects,
- e is the basic (environmental) object of the generalized P colony, $e \in A$,
- f is the final object of the generalized P colony, $f \in A$,
- v_E is the initial content of the environment, $v_E \in (A - \{e\})^*$,
- D_E is an 0L scheme (A, P_E), where P_E is the set of context-free rules,
- B_i, $1 \le i \le n$, are the agents, every agent is a construct $B_i = (o_i, P_i)$, where o_i is a multiset over A, it defines the initial state (content) of agent B_i and $|o_i| = k$ and $P_i = \{p_{i,1}, \dots, p_{i,k_i}\}$ is the finite set of programs of three types $(a, b, c, d \in A)$:

  1. generating program with generating rules $a \to bc$ and transporting rules $d\ out$ - the number of generating rules is the same as the number of transporting rules.
  2. consuming program with consuming rules $ab \to c$ and transporting rules $d\ in$ - the number of consuming rules is the same as the number of transporting rules.
  3. rewriting/communication program which can contain three types of rules:

     ⋄ $a \to b$, called a rewriting rule,
     ⋄ $c \leftrightarrow d$, called a communication rule,
     ⋄ r_1/r_2, called a checking rule; each of r_1, r_2 is a rewriting or a communication rule.

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

The initial configuration of a P colony is the $(n+1)$-tuple $(o_1, \dots, o_n, v_E)$, with the same interpretation of the symbols $o_1, \dots, o_n, v_E$ as in Definition 3. In general, the configuration of the P colony Π is defined as $(n+1)$-tuple $(w_1, \dots, w_n, w_E)$, where w_i represents the multiset of objects inside the i-th agent, $|w_i| = k$, $1 \le i \le n$, and $w_E \in (A - \{e\})^*$ is the multiset of objects different from e placed in the environment.

By applying programs, the generalized P colony passes from one configuration to some other configuration. Objects in the environment unaffected by any program in the given step are rewritten by the 0L scheme D_E. (Notice that in this case the 0L scheme is considered as a multiset rewriting mechanism). At each step, every agent tries to find one of its programs to apply. If the number of applicable programs is higher than one, then the agent non-deterministically chooses one program. At each step of, the set of active agents executing a program must be maximal, i.e., no further agent can be added to it.

A sequence of consecutive configurations starting from the initial configuration is called a computation. A configuration is halting if the P colony has no applicable program. Each halting computation has an associated a result – the number of copies of the final object placed in the environment in a halting configuration.

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

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

Let NEPCOL(i, j, h, u, v, w) be the family of the sets of numbers computed by generalized P colonies with at most $j \geq 1$ agents with $i \geq 1$ objects inside the agent and with at most $h \geq 1$ programs associated with each agent such that:

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

If a numerical parameter is not bounded, we use notation $*$.

Example 5 Let $M = (m, H, l_0, l_h, P)$ be a non-deterministic register machine with m registers. The *ADD*-instruction $l_1 = (\mathrm{ADD}(r), l_2, l_3)$ will be simulated by the following rules:

ENV :

$$\begin{aligned} 1 &: l_1 i \to a_r l_1' D; \\ 2 &: l_1' \to \overline{l_2 l_3} D; \\ 3 &: \overline{l_2} \to l_2 D; \\ 4 &: \overline{l_3} \to l_3 D. \end{aligned}$$

B :

$$\begin{aligned} 5 &: \left\langle Pe \to P; \overline{l_2}\ in \right\rangle; \\ 6 &: \left\langle Pe \to P; \overline{l_3}\ in \right\rangle; \\ 7 &: \left\langle P\overline{l_2} \to P; e\ in \right\rangle; \\ 8 &: \left\langle P\overline{l_3} \to P; e\ in \right\rangle. \end{aligned}$$

The computation is done in such a way that the 0L scheme works in the environment, it adds one to the contents of register r (generates one copy of object a_r - the rule number 1) and generates objects l_2 and l_3, labels of all instructions which will be possibly executed in the next steps of computation of the register machine M (the rule 2). In the next step, consumer agent B takes one of these objects inside the agent - the rule 5 or 6. Then, instruction l_2 or l_3 will be simulated.

Generalized P colonies with two agents (senders and consumers) with passive environment (0L scheme contains the rules of type $a \to a$ only) are computationally complete. If the environment is active, then the family of generalized P colonies is computationally complete if the systems have two consumers and the initial contents of their environment is different from e.

- NEPCOL$(2, 2, *, \text{sc,pas,ini}) = \text{NRE}$ in [24],
- NEPCOL$(2, 2, *, \text{c,act,ini}) = \text{NRE}$ in [7],
- $\text{NRM}_{\text{pb}} \subseteq$ NEPCOL$(2, 1, *, \text{c,act,ini})$ in [25],
- NEPCOL$(1, 2, *, \text{ no-check }, \text{act,ini}) = \text{NRE}$ in [25],
- $NRM_{\text{pb}} \subseteq$ NEPCOL$(1, 1, *, \text{check}, \text{act,ini})$ in [25].

3.4 Relation of P colonies and other P systems

Generalized P colonies have been related to other variants of P systems. We briefly summarize their main features, with only the necessary details.

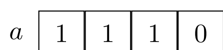
Catalytic P systems are an important type of symbol-object P systems (already considered in the original definition of a membrane system, due to the relevance of chemical catalysts, see [50], chapter 4). In these systems a set of objects is distinguished, called catalysts, that do not change during the functioning of the P system, but their presence is necessary to perform some of the rules. If each rule has occurrence of at least one catalyst, then we speak of purely catalytic P systems, and in case of multi-stable catalytic P systems catalysts are allowed to change only to some other, distinguished catalysts. A catalytic P system is extended if the catalytic objects are not counted to the result of a computation.

In [25] it was shown that for an arbitrary extended catalytic P system with one catalyst there exists a generalized P colony with checking rules and one agent containing one object such that the two constructs determine the same set of numbers. In [34], P colonies and P systems with multi-stable catalysts are compared to each other. It is shown that, using a general framework of P systems [33], both models have identical representation, and therefore both models can be related using a bi-simulation.

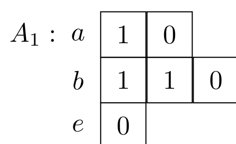
P colonies have also been interpreted in terms of kernel P systems [28]. Kernel P systems are a framework integrating the most commonly used features of membrane systems (compartments, dynamically changing structure, rules with application conditions, execution strategies, etc.) The concept has obtained recently much interest, due to its broad scope of applicability. In [28] connections among several classes of P colonies and kernel have been demonstrated, and P colonies have been represented as kernel P systems. In particular, the famous producer/consumer problem has also been approached, namely its representation using P colony with components having sender programs and/or consumer programs and with kP systems has been presented.

3.5 Logical representation of P colonies

In [12] the authors introduce an approach how to express computation in P colonies in terms of propositional logic. To represent if object is present in P colony we use value 1 (or 0 if it is not present). Let a be an object in P colony, $a \in O$ and there are three copies of such object placed in the environment. We construct a stack called "a" and put value 0 into the bottom of stack. For every copy of object a in the environment we push one copy of 1 to the stack. The presence of object a can be expressed as literal a interpreted as TRUE, otherwise it is FALSE.



An agent of capacity k is represented by an array of $|O|$ stacks. The sum of 1s in all stacks is k. For example agent A_1 with capacity 3 working with alphabet $O = \{e, a, b\}$ and with objects bba inside the agent has following representation:



The presence of object a inside the agent A_i can be expressed as as an interpretation of literal $A_i[a]$.

A rewriting rule $a \to b$ is applicable if there is an object a inside the agent A_i. It means that the rule is applicable if literal $A_i[a]$ is true. The communication rule $a \leftrightarrow b$ is applicable if there is an object a inside the agent and object b in the environment. In the terms of logic we can write the condition as $A_1[a] \wedge b$. If $b = e$ we omit b in condition (there is always some copy of e in the environment). A condition of application of rewriting or communication rule is called elementary condition of application. Let r_1/r_2 be a checking rule, where r_1, r_2 are rewriting or communication rules with conditions of application c_1, c_2 then we can express condition of application for the checking rule as $c_1 \vee c_2$.

The condition of applicability of the program $p_{i,l}$ of i-th agent is $c_{i,l} : c_1 \wedge c_2 \wedge \cdots \wedge c_k$, where c_x is the condition of application of the x-th rule in the program. A condition is in the form: $A_i[a]$ (rewriting rule), $A_i[a] \wedge b$ (communication rule), $c_{x_1} \vee c_{x_2}$ (checking rule).

If the program contains a checking rule, we can write the condition $c_{i,l}$ in the disjunctive normal form (DNF) $c_{i,l} : (c_1 \wedge c_2 \wedge \cdots \wedge c_{j_1} \wedge c_{j+1} \wedge \cdots \wedge c_k) \vee (c_1 \wedge c_2 \wedge \cdots \wedge c_{j_2} \wedge c_{j+1} \wedge \cdots \wedge c_k)$.

Furthermore, consider a k-tuple of elementary rules corresponding to a conjunction. Generally, j rules $(1 \le j \le k)$ may depend on the presence of the same object a inside the agent, hence the program is applicable only if the agent contains at least j objects a. Therefore, we introduce a literal $A_i[a][j], 1 \le j \le k$, which is TRUE when the j-th position in the stack "a" of the agent A_i exists and contains 1. Similarly, $b[j]$ is the literal which is TRUE when the j-th position in stack "b" is 1, i.e., when the environment contains at least j objects b. Therefore, in each conjunction in the final DNF of the condition $c_{i,l}$, literals $A_i[a]$ must be substituted/indexed for $A_i[a][j]$, where j is the order of occurrence of $A_i[a]$ in the conjunction. Similarly, each literal b is substituted for $b[j]$.

Since DNF is represented as a string, we can order the conjunctions in DNF due to decreasing priority among rules. The first will be the conjunction with elementary conditions for the first rule in checking programs, then we can take conjunctions with one second rule from checking rules in random order followed by conjunctions with three second rules from checking order etc.

Clearly, the logical condition whether an agent A_i can be active (i.e., apply some of its programs) can be expressed as a disjunction of conditions for all programs of that agent: $c_i = c_{i,1} \vee c_{i,2} \vee \cdots \vee c_{i,k_i}$, where k_i is the number of programs of the agent A_i.

Given a P colony Π, the condition whether Π can perform a computational step can be expressed in a disjunctive normal form with $\sum_{i=1}^{n} \sum_{j=1}^{k_i} 2^{d_{i,j}}$ conjunctions, where $d_{i,j}$ is the number of checking rules in the program $p_{i,j}$.

The process of logical representation of one computational step of the P colony under maximally parallel mode is demonstrated on the following example ([12]).

Let $\Pi = (O, e, f, V_E, A_1)$ be a P colony with capacity two and one agent and with $O = \{a, b, c, d, e, f\}$, ε, $A_1 = (ee, \{\langle a \leftrightarrow c/c \leftrightarrow d; c \leftrightarrow f/a \leftrightarrow e\rangle; \langle a \to b; e \leftrightarrow b\rangle\})$.

Let us construct a condition of application of the program $\langle a \to b; e \leftrightarrow b\rangle$: It is formed from one rewriting and one communication rule.

Rule	Elementary condition of application
$a \to b$	$A_1[a]$
$e \leftrightarrow b$	$A_1[e] \wedge b$

The condition of application of the program after the substitution (indexing) of literals is $A_1[a][1] \wedge A_1[e][1] \wedge b[1]$.

The condition of application of the program $\langle a \leftrightarrow c/c \leftrightarrow d; c \leftrightarrow f/a \leftrightarrow e\rangle$ is formed from two checking rules, each formed from two communication rules.

Rule	Elementary condition of application
$a \leftrightarrow c$	$c_{11} : A_1[a] \wedge c$
$c \leftrightarrow d$	$c_{12} : A_1[c] \wedge d$
$c \leftrightarrow f$	$c_{21} : A_1[c] \wedge f$
$a \leftrightarrow e$	$c_{22} : A_1[a]$
$a \leftrightarrow c/c \leftrightarrow d$	$(A_1[a] \wedge c) \vee (A_1[c] \wedge d)$
$c \leftrightarrow f/a \leftrightarrow e$	$(A_1[c] \wedge f) \vee A_1[a]$

The condition of application of the program is formed from four conjunctions: $c_{11} \wedge c_{21}$ with highest priority, $c_{12} \wedge c_{21}$ and $c_{11} \wedge c_{22}$, and $c_{12} \wedge c_{22}$ with lowest priority. After indexing of literals we obtain

$$(A_1[a][1] \wedge c[1] \wedge A_1[c][1] \wedge f[1]) \vee$$
$$\vee\ (A_1[c][1] \wedge d[1] \wedge A_1[c][2] \wedge f[1]) \vee$$
$$\vee\ (A_1[a][1] \wedge c[1] \wedge A_1[a][2]) \vee$$
$$\vee\ (A_1[c][1] \wedge d[1] \wedge A_1[a][1])$$

The execution of a multiset of rules can be understood as an action of a rule-based production system: as sensory precondition we use condition of application and an action can be constructed from functions `push` and `pop` as it is usual for stacks. Function `push`(x) means put 1 to the top of stack x. Function `pop`(x) means remove 1 from the top of stack x. Rules for execution of programs in our example are:

- `if` $A_1[a][1] \wedge c[1] \wedge A_1[c][1] \wedge f[1]$ `then` (pop$(A_1[a]) \wedge$ push$(A_1[c]) \wedge$ pop$(c) \wedge$ push$(a) \wedge$ pop$(A_1[c]) \wedge$ push$(A_1[f]) \wedge$ pop$(f) \wedge$ push(c))
- `if` $A_1[c][1] \wedge d[1] \wedge A_1[c][2] \wedge f[1]$ `then` (pop$(A_1[c]) \wedge$ push$(A_1[d]) \wedge$ pop$(d) \wedge$ push$(c) \wedge$ pop$(A_1[c]) \wedge$ push$(A_1[f]) \wedge$ pop$(f) \wedge$ push(c))
- `if` $A_1[a][1] \wedge c[1] \wedge A_1[a][2]$ `then` (pop$(A_1[a]) \wedge$ push$(A_1[c]) \wedge$ pop$(c) \wedge$ push$(a) \wedge$ pop$(A_1[a]) \wedge$ push$(A_1[e]) \wedge$ push(a))
- `if` $A_1[c][1] \wedge d[1] \wedge A_1[a][1]$ `then` (pop$(A_1[c]) \wedge$ push$(A_1[d]) \wedge$ pop$(d) \wedge$ push$(c) \wedge$ pop$(A_1[a]) \wedge$ push$(A_1[e]) \wedge$ push(a))

In [12] the authors showed that if a P colony does not use checking rules the problem whether configuration is a halting configuration is in **P** while if P colony uses checking rules the problem whether configuration is a halting configuration is in **NP**.

4 P colony models related to automata

The concept of P colonies has been extended to automaton-like computing devices.

4.1 PCol automata

The basic motivation of P colonies was to model multi-agent systems with very simple agents interacting with their shared environment. The interaction was realized in communicating objects, and the description of the result of the activity of the P colony was defined as the multiset of distinguished objects in the environment when no more action could be performed.

Interaction of the environment and the collection of agents can also be described as the sequence of multisets of non-environmental agents that are found in the environment during the computation, i.e., the sequence of computational steps. From this point of view, the concept of a P colony can be extended to the notion of a PCol automaton (a P colony automaton), motivated by P automata from membrane computing [50] and classical finite automata [55].

In reference to the finite automaton, the concept of the P colony was extended by an input tape and the generating device was changed to an accepting one [13]. The agents of the P colony work according to the actual symbol read from the input tape. To do this, they have rules which can "read" the input tape, we call them tape rules or T-rules. The other rules, which are rules of standard P colonies, are called non-tape rules or N-rules. An input symbol is said to be read if at least one agent processed it (using its corresponding T-rule).

Now we recall the notion of a PCol automaton.

Definition 4 A PCol automaton of capacity k and with n agents, $k, n \geq 1$, is a construct $\Pi = (A, e, v_E, (o_1, P_1), \ldots, (o_n, P_n), F)$ where

- A is an alphabet, the alphabet of the PCol automaton, its elements are called objects;
- $e \in A$ is the environmental object of the PCol automaton; $v_E \in (A - \{e\})^*$ is a string representing the multiset of objects different from e, called the initial state of the environment ;
- (o_i, P_i), $1 \leq i \leq n$, is the i-th agent; where
  - o_i is a multiset over V, the initial state (contents) of the agent,
  - P_i is a set of programs, where every program consists of k rules, each of them is one of the following types:
    - tape rules of the form $a \xrightarrow{T} b$ or $a \overset{T}{\leftrightarrow} b$, called rewriting tape rules and communication tape rules, respectively; or
    - non-tape rules of the form $a \to b$, or $c \leftrightarrow d$, called rewriting (non-tape) rules and communication (non-tape) rules, respectively.

  and
- F is a set of accepting configurations of the PCol automaton.

For each i, $1 \leq i \leq n$, we distinguish tape programs and non-tape programs. The set of tape programs (T-programs), denoted by P_i^T, are formed from one tape rule and $k-1$ non-tape rules, the set of non-tape programs

(N-programs) which contain only non-tape rules, is denoted by P_i^N, thus, $P_i = P_i^T \cup P_i^N$ and $P_i^T \cap P_i^N = \emptyset$.

The computation starts in the initial configuration, i.e., when the input word is on the input tape and all agents are in initial state.

For a configuration $(w_E, w_1, \dots, w_n)$ and an input symbol a, the sets of applicable programs, $\mathcal{P}$, can be constructed. To pass from one configuration to some other one in one step we define the following types of transitions:

- *t-transition* $\Rightarrow_t^a$: There exists at least one set of applicable programs $P \in \mathcal{P}$ such that every $p \in P$ is T-program with T-rule in the form $x \overset{T}{\rightarrow} a$ or $x \overset{T}{\leftrightarrow} a, x \in A$ and the set P is maximal.
- *n-transition* $\Rightarrow_n$: There exists at least one set of applicable programs $P \in \mathcal{P}$ such that every $p_i \in P$ is N-program and the set P is maximal.
- *tmin-transition* $\Rightarrow_{tmin}^a$: There exists at least one set of applicable programs $P \in \mathcal{P}$ such that there is at least one T-program in P in the form $x \overset{T}{\rightarrow} a$ or $x \overset{T}{\leftrightarrow} a, x \in A$ and possibly N-programs. The set P is maximal.
- *tmax-transition* $\Rightarrow_{tmax}^a$: There exists at least one set of applicable programs $P \in \mathcal{P}$ such that P contain as many T-programs (they are in a form $x \overset{T}{\rightarrow} a$ or $x \overset{T}{\leftrightarrow} a, x \in A$) as possible, P can contain N-programs too, and the set P is maximal.

We say that a PCol automaton works in *t (tmax, tmin) mode* of computation if it uses only *t- (tmax-, tmin-) transitions*. It works in *nt (ntmax or ntmin)* working mode if it uses *t-(tmax- or tmin-)* transitions and if there is no set of applicable T-programs it can use n-transition. PCol automaton works in *init* mode if it performs only t-transitions and after reading all the input symbols it makes n-transitions.

If the PCol automaton works in t, *tmax* or *tmin* mode, then it reads one input symbol in every step of computation. Consequently, the length of the computation equals to the length of the input string. Notice that this property strongly resembles to some property of ε-free finite automata.

The computation by a PCol automaton may end in a final state. It is successful if the whole input tape is read and the PCol automaton is in some configuration in F.

Let $M = \{t, \text{nt}, \text{tmax}, \text{ntmax}, \text{tmin}, \text{ntmin}, \text{init}\}$.

The *language accepted by a PCol automaton* Π, given as above, is defined as the set of strings which can be read during a successful computation:

$$L(\Pi, \text{mode}) = \{w \in A^* | (w; v_E, o_1, \dots, o_n) \text{ can be transformed by } \Pi \text{ into } (\varepsilon; w_E, w_1, \dots, w_n) \in F \text{ with a computation in mode mode} \in M\}.$$

Let $\mathcal{L}$(PColA,mode) denote the class of languages accepted by PCol automata in the computational mode mode $\in M$.

Language classes of the Chomsky hierarchy can be described by PCol automata as follows [13].

- For every regular language L there exists a PCol automaton working in the t-mode having only one agent accepting all words from L.
- There exists a context-free language that can be accepted by a PCol automaton with only one agent and working in the t-mode.
- The family of languages accepted by PCol automata with one agent working in the t-mode is a subfamily of the family of context-sensitive languages.

It is open question whether the family of context-sensitive languages is equal to the family of languages accepted by PCol automata with one agent working in the t-mode. Notice that unlike other variants of P colonies PCol automata working in the t-mode are not computationally complete.

In [13], it was shown that class of languages accepted by PCol automata working in the nt, ntmin or ntmax mode equals to the class of recursively enumerable languages, respectively. The workspace needed to obtain this computational power is provided by the interaction between the agents and the environment.

- $\mathcal{L}$(PColA,nt) = RE in [13],
- $\mathcal{L}$(PColA,ntmin) = RE in [13],
- $\mathcal{L}$(PColA,ntmax) = RE in [13],
- $\mathcal{L}$(PColA,init) = RE in [8].

4.2 Generalized PCol automaton

In [39] a model, called generalized P colony automaton (genPCol automaton, for short) was introduced that combines features of P colonies and P automata. (For detailed information on P automata consult [50]). In the following we describe the main features of this construct, for the technical details the reader is referred to [39, 41].

In case of P colony automata there is an input string given, while in case of P automata the accepted string is defined as a map of the sequence of multisets entering the P system during the successful (usually halting) computation. An idea similar to P automaton is employed in the concept of generalized P colony automaton, namely, determining the accepted strings through the sequences of multisets processed during computations. The computations of the P colony define accepted multiset sequences, which are turned into accepted strings by mapping the multiset sequences to strings over some previously given alphabet. The rules of the underlying P colony that describes the communication with the environment are of two types: standard rules and

so-called tape rules. The application of a tape rule also represents the reading of the processed symbol from the input, but unlike the original model, the P colony automaton is allowed to read more than one such symbol in a single computational step. This means that during a computation consisting of a sequence of computational steps, a sequence of multisets is read from the input. This sequence of multisets then can be mapped into a string (a sequence of symbols) in a similar way as in P automata.

In [39] the so-called permutation mapping (also known from the field of P automata) is used to create the accepted strings from the accepted multiset sequences. Some basic variants of the model were introduced and studied.

In [41] the authors considered three possible ways of dealing with tape rules in the programs: the unrestricted case, the case when each program contains at least one tape rule (all-tape programs), and the case when all communication rules are tape rules (com-tape programs). It was shown that in the unrestricted case, even systems with capacity one are able to characterize the class of recursively enumerable languages. For capacities greater than two, all-tape and com-tape genPCol automata behave differently: all-tape systems describe the class of recursively enumerable languages, while the power of com-tape systems is bounded by the power of so-called restricted logarithmic space Turing machines. (For these variants of a Turing machines see [29]). It is also shown that com-tape systems of capacity two are already able to accept languages that are not possible to be accepted by P automata.

4.3 APCol systems

In [10] the authors make one step further in combining properties of P colonies and automata. While the behaviour of the agents of PCol automata is determined both by the string to be processed and the environment consisting of multisets of symbols, in the case of APCol systems (Automaton-like P colonies), the agents act only on the input string. This interaction between the agents of the P colony and the input string is realized by exchanging symbols between the objects of the agents and that of the string (communication rules), and the states of the agents can change both by communication and evolution; the latter one is an application of a rewriting rule to an object. The distinguished symbol, e (in the previous models the environmental symbol) has a special role: whenever it is exchanged by a symbol in the environmental string, this symbol is erased. An evolution rule is of the form $a \rightarrow b$. It means that object a inside the agent is rewritten (evolved) to the object b. The second type of rules are called communication rules. A communication rule is in the form $c \leftrightarrow d$. When this rule is performed, the object c inside the agent and a symbol d in the string are exchanged, so, we can say that the agent rewrites symbol d to symbol c in the input string. 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 of the agents having only symbols e in their state. (Note that the initial states of the agents can be chosen not to consist of only e.)

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.

Definition 5 An Automaton-like P colony (an APCol system, for short) is a construct

$\Pi = (A, e, B_1, \dots, B_n), n \geq 1,$ where

- A is an alphabet; its elements are called the objects,
- $e \in A$, called the basic object,
- B_i, $1 \leq i \leq n$, are agents. Each agent is a triplet $B_i = \left(o_i, P_i, F_i\right)$, where
  - o_i is a multiset over A, describing the initial state (content) of the agent, $|o_i| = 2$,
  - $P_i = \left\{p_{i,1}, \dots, p_{i,k_i}\right\}$ 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 $a, b \in A$, called an evolution rule,
    - $c \leftrightarrow d$, where $c, d \in A$, called a communication rule,
  - $F_i \subseteq A^*$ is a finite set of final states (contents) of agent B_i.

As in the case of other variants of P colonies, the number of objects inside the agents are called the capacity of the APCol system, which is 2.

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. 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 one step of the computation and the change of 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 computation of the APCol system the environment is given by a string ω of objects which are different from e. Consequently, an initial configuration of the APCol system is an $(n+1)$-tuple $c = (\omega; o_1, \dots, o_n)$ where ω is the input string and the other n components are multisets of strings of objects, given in the form of strings, the initial states of the agents.

A configuration of an APCol system Π is given by $(w;w_1, \dots, w_n)$, where $|w_i| = 2$, $1 \le i \le n$, w_i represents the state of the i-th agent and $w \in (A - \{e\})^*$ is the string to be processed.

The language $L_{acc}(\Pi)$ accepted by Π is the set of words over $(A - \{e\})$ which are accepted by Π. A string ω is accepted by the APCol Π if there exists a computation by Π such that it starts in the initial configuration $(\omega;o_1, \dots, o_n)$ and the computation ends by halting in the configuration $(\varepsilon;w_1, \dots, w_n)$, where at least one of $w_i \in F_i$ for $1 \le i \le n$.

APCol systems are powerful computational devices as it is shown in [10]:

Let A be an alphabet and let $L \subseteq A^*$ be a recursively enumerable language. Let $L' = S \cdot L \cdot E$, where $S, E \notin A$. Then there exists an APCol system Π with two agents such that $L' = L(\Pi)$ holds.

APCol systems can also be used not only for accepting but generating strings. A string w_F is generated by an APCol system Π if there exists a computation starting in an initial configuration $(\varepsilon;ee, \dots, ee)$ and the computation ends by halting in configuration $(w_F;w_1, \dots, w_n)$, where $w_i \in F_i$ for at least one w_i, $1 \le i \le n$. The language $L_{gen}(\Pi)$ generated by Π is the set of words over $(A - \{e\})$ which are generated by Π.

Particularly important are those variants, where the programs are restricted (as defined for standard P colonies).

We denote by $\mathrm{APCol}_{\mathrm{acc}}R(n)$ (or $\mathrm{APCol}_{\mathrm{acc}}(n)$) the family of languages accepted by APCol systems having at most n agents, $n \ge 1$, with restricted programs only (or without this restriction). Analogously, we denote by $\mathrm{APCol}_{\mathrm{gen}}R(n)$ the family of languages generated by APCol systems having at most n agents, $n \ge 1$, with restricted programs only, and $\mathrm{APCol}_{\mathrm{gen}}(n)$ denotes the case when the programs are without any restriction.

We may associate sets of numbers to APCol systems working in the generating or the accepting mode in the usual manner.

For an APCol system Π, $\mathrm{NL}_{\mathrm{acc}}(\Pi)$ and $\mathrm{NL}_{\mathrm{gen}}(\Pi)$ denote the length sets of $L_{\mathrm{acc}}(\Pi)$ and $L_{\mathrm{gen}}(\Pi)$, respectively. The family of length sets of languages accepted or generated by restricted APCol systems with at most n agents, $n \ge 1$, is denoted by $\mathrm{NAPCol}_x R(n)$, $x \in \{\mathrm{acc},\mathrm{gen}\}$, respectively, and $\mathrm{NAPCol}_x(n)$ denotes the case when the programs are without any restriction.

The following results were obtained in [10]:

- $\mathrm{NAPCol}_{\mathrm{gen}}R(2) = \mathrm{NRE}$.
- $\mathrm{NRM}_{\mathrm{pb}} \subseteq \mathrm{NAPCol}_{\mathrm{gen}}R(1)$.
- $\mathrm{APCol}_{\mathrm{gen}}R(1) \subseteq \mathrm{MAT}^{\varepsilon}$.

In case of the original concept of APCol systems, the input string is accepted if it can be reduced to the empty word. Recently, a new variant of acceptance by APCol systems has been introduced where the agents explore and verify their common environment, i.e. the input the string. The notion was introduced as verifying APCol system (or APCol systems with verifier agents) [14, 15]. In this case, an input string of length n is accepted if there is a halting computation c such that the length of the environmental string remains unchanged during the computation and for every agent and for each position each i, $1 \le i \le n$, there is an environmental string obtained by c such that the agent applies a rule to position i. It is shown that APCol systems with verifier agents simulate nondeterministic two-way multihead automata. The result implies that any language in NSPACE(log n)can be accepted by an APCol system with verifier agents.

4.3.1 APCol systems with teams

In [21] the authors introduced the concept of APCol systems with coloured teams. The concept of teams in P colonies was first proposed in [27]. APCol systems with 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 (agents) should perform a program in parallel. 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. By this the APCol systems join 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.

Red-green Turing machines were introduced in [47] and they 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. In [47], 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. 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.

In [21] the authors showed that any red-green counter machine can be simulated with an APCol system with coloured teams with 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.

4.3.2 APCol systems with agent creation

In [20], the author introduced the programs for agent creation. For this purpose, a new special object @ was defined. If an agent contains object @, the agent makes a copy of itself. This action is done by executing a program formed from two rewriting rules. The order of rules in a program determines whether the rewriting rule without @ is used before or after the creation of the child-agent. Let $x@$ be a contents of agent A with program $p_1 = \langle @ \rightarrow b; x \rightarrow y \rangle$. After execution of the program p_1 there is one new child-agent in the APCol system with the same label and the same set of programs as the parent-agent A has. The contents of the parent-agent after the execution of the program is by while the contents of the child-agent is bx. If the parent-agent has a program $p_2 = \langle x \rightarrow y; @ \rightarrow b \rangle$, then after the execution of the program p_2 the contents of parent-agent is by and the contents of the child-agent is by, too.

When an agent contains the object @ the agent must create a new agent in the next step of the computation if there is some applicable program in its set of programs.

Here we provided the main ideas, for the technical details the reader is referred to [20].

In [20] the author showed that APCol systems with agent creation can solve 3SAT in linear time (3SAT is a famous NP complete problem).

5 Other models raised from P colonies

In this section we focus to the models related to P colonies and using different type of rules.

5.1 P colonies with prescribed teams

P colonies with prescribed teams were introduced in [36]. Unlike the original variants of P colonies, the agents use finite sets of rules called teams instead of programs; with each agent a finite set of teams is given, with priorities (pri) among them. The used rules can be communicating (com), rewriting (rew), and so-called membrane rules (mem). The membrane rules are in a form $a \rightarrowtail b$ (a goes out and becomes b) or $b \leftarrowtail a$ (a goes in and becomes b).

The P colony can work in sequential (seq) or parallel (par) manner. The rules are applied by the team in parallel manner with various stop conditions: $*$ (stop after arbitrary number of derivation steps), $\leq l$, $\geq l$, resp. $= l$ (stop after at most l, at least l resp. after exactly l derivation steps) and t_0 (the team becomes inactive when it is no longer able to work as a team.)

At each step of the computation, the contents of the environment and the contents of every agent changes in the following way: in the maximally parallel derivation mode, each agent which can use any of its teams should use one (non-deterministically chosen) in the mode d, while in the sequential derivation mode, one agent uses one of its teams in the mode d at a time (non-deterministically chosen). As in the usual case, any copy of an object can be involved in only one rule. Using the teams as described above, with all agents acting simultaneously or sequentially, non-deterministically choosing the team(s) to be applied, the P colony changes its configuration.

In [36], the authors showed that the families of P colonies with prescribed teams are computationally complete if some conditions hold. These conditions concern, for example, the working mode, the number of objects in the agents using rewriting and communication rules, the priority among the teams, the number of teams.

The following table summarizes the list of results on computational completeness of P colonies with prescribed teams which use rewriting and communication rules [36].

In the tables, below, d indicates that the results hold for any of the modes.

Computational mode	Capacity	Max. number of sets in the team	Max. number of rules in the set	Number of agents	Number of teams	Priorities	Mode
seq	2	2	1	*	6	pri	d
par	2	2	1	*	5		d
seq	2	2	1	1	*	pri	d
seq	2	2	2	1	*		t_0

The following table contains a list of results from [36] concerning P colonies using membrane rules only.

Computational mode	Capacity	Max. number of sets in the team	Max. number of rules in the set	Number of agents	Number of teams	Priorities	Mode
seq	2	2	1	*	12	pri	d
seq	2	2	1	1	*	pri	d
par	2	3	1	*	10		d
par	2	3	2	*	5		d
seq	1	2	2	1	*		t_0
seq	1	2	1	1	*	pri	t_0

5.2 2D P colonies

In [19] a new model, called 2D P colony was introduced. As in the original model, the P colony is of capacity two and the agents are equipped with sets of the programs formed from rules—communication and evolution. The main change is in the environment. Namely, the authors put the agents into the 2D grid of square cells and they provide the agent with the possibility to move—the motion rule. The direction of the movement of the agent is determined by the contents of cells surrounding the cell in which the agent is placed.

The program can contain at most one motion rule. To achieve the greatest simplicity in agent behaviour, one other condition was set. If the agent moves, it cannot communicate with the environment. So if the program contains a motion rule, then the other rule is an evolution rule.

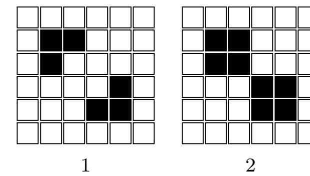


Fig. 1 Pattern beacon changes in two consecutive steps

Definition 6 A 2D P colony is a construct

$\Pi = (A, e, Env, B_1, \dots, B_k, f), k \geq 1,$ where

- A is an alphabet of the colony, its elements are called objects,
- $e \in A$ is the basic environmental object of the 2D P colony,
- Env is a pair $(m \times n, w_E)$, where $m \times n, m, n \in N$ is the size of the environment and w_E is the initial contents of environment, it is a matrix of size $m \times n$ of multisets of objects over $A - \{e\}$.
- B_i, $1 \leq i \leq k$, are agents, each agent is a construct $B_i = \left(o_i, P_i, [o, p]\right)$, $0 \leq o \leq m$, $0 \leq p \leq n$, where
  - o_i is a multiset over A, it determines the initial state (contents) of the agent, $|o_i| = 2$,
  - $P_i = \left\{p_{i,1}, \dots, p_{i,l_i}\right\}, l \geq 1, 1 \leq i \leq k$ is a finite set of programs, where each program contains exactly 2 rules, which are in one of the following forms each:
    - $a \to b$, called the evolution rule, $a, b \in A$,
    - $c \leftrightarrow d$, called the communication rule, $c, d \in A$,
    - $\left[a_{q,r}\right] \to s, 0 \leq q, r \leq 2, s \in \{\Leftarrow, \Rightarrow, \Uparrow, \Downarrow\}$, called the motion rule,
- $f \in A$ is the final object of the colony.

A configuration of the 2D P colony is given by the state of the environment—matrix of type $m \times n$ with multisets of objects over $A - \{e\}$ as its elements—and by the state of all agents—pairs of objects from alphabet A and the coordinates of the agents. An initial configuration is given by the definition of the 2D P colony.

A computational step consists of three parts. The first part lies in determining the set of applicable programs according to the current configuration of the 2D P colony. In the second part, we have to select from this set one program for each agent, in such a way that there is no collision between the communication rules belonging to different programs. The third part is the execution of the chosen programs.

A change of the configuration is triggered by the execution of programs and it involves changing the state of the environment, contents and placement of the agents.

A computation is non-deterministic and maximally parallel. The computation ends by halting when there is no agent with applicable program.

The result of the computation is the number of copies of the final object placed in the environment at the end of the computation.

The aim of introducing 2D P colonies is not studying their computational power but monitoring their behaviour during the computation.

In [19] an example for a 2D P colony simulating a kind of cellular automata—Conway's Game of Life ([37]) is presented. The following example is the pattern called beacon (see Fig. 1).

Let Π_2 be 2D P colony defined as follows: $\Pi_2 = (A, e, Env, B_1, \dots, B_{16}, f)$, where

- $A = \{e, f, D, S, Z, M, O, L, N\}$,
- $e \in A$ is the basic environmental object of the 2D P colony,
- $Env = (6 \times 6, w_E)$,
- $w_E = \begin{bmatrix} D & D & D & D & D & D \\ D & S & S & D & D & D \\ D & S & S & D & D & D \\ D & D & D & S & S & D \\ D & D & D & S & S & D \\ D & D & D & D & D & D \end{bmatrix}$,
- $B_1 = \big(ee, P_1, [1,1]\big)$, $B_2 = \big(ee, P_2, [1,2]\big)$, ..., $B_{16} = \big(ee, P_{16}, [4,4]\big)$,
- $f \in A$ is the final object of the 2D P colony.

The states of the automata are stored inside the cells (D—dead automaton, S—live automaton). There is only one kind of agent in this 2D P colony, so there are 16 identical agents located in the matrix 4×4 of inner cells with following programs:

The first program is to initialize the agent $< e \leftrightarrow e; e \rightarrow Z >$;

we sort the programs using the number of copies of object S in the condition of the motion rule.

1. When neighbouring automata are dead—a single program for both dead as well as alive automaton $\left\langle \begin{bmatrix} D & D & D \\ D & e & D \\ D & D & D \end{bmatrix} \rightarrow \Uparrow; Z \rightarrow M \right\rangle$.
2. When there is one alive neighbouring automaton—there are eight possible programs for dead and alive automata $\left\langle \begin{bmatrix} S & D & D \\ D & e & D \\ D & D & D \end{bmatrix} \rightarrow \Uparrow; Z \rightarrow M \right\rangle$ and seven other combinations.
3. When there are 2 alive neighbouring automata—28 programs for alive automata $\left\langle \begin{bmatrix} S & S & D \\ D & S & D \\ D & D & D \end{bmatrix} \rightarrow \Uparrow; Z \rightarrow O \right\rangle$ and other 27 combinations.
4. When there are 2 alive neighbouring automata—28 programs for dead automata $\left\langle \begin{bmatrix} S & S & D \\ D & D & D \\ D & D & D \end{bmatrix} \rightarrow \Uparrow; Z \rightarrow M \right\rangle$ and other 27 combinations.
5. When there are 3 alive neighbouring automata—56 possible programs for dead and alive automata $\left\langle \begin{bmatrix} S & S & S \\ D & e & D \\ D & D & D \end{bmatrix} \rightarrow \Uparrow; Z \rightarrow O \right\rangle$ and other 55 combinations.
6. When there are 4 alive neighbouring automata—seventy possible programs for dead and alive automata $\left\langle \begin{bmatrix} S & S & S \\ S & e & D \\ D & D & D \end{bmatrix} \rightarrow \Uparrow; Z \rightarrow M \right\rangle$ and other 69 combinations.
7. When there are at least 5 alive neighbouring automata—56 possible programs for dead and alive automata $\left\langle \begin{bmatrix} S & S & S \\ S & e & S \\ e & e & e \end{bmatrix} \rightarrow \Uparrow; Z \rightarrow M \right\rangle$ and other 55 combinations.

After executing one of the above programs, all agents move one step upwards and rewrite one of their objects e to object M (automaton will be dead) or to object O (automaton will be live). The following programs are for downward movement and for updating the state of an automaton, i.e., the replacement of the object in the cell for an object in the agent to change the state of the automaton.

$$\left\langle \begin{bmatrix} e & e & e \\ e & e & e \\ e & e & e \end{bmatrix} \rightarrow \Downarrow; O \rightarrow S \right\rangle; \left\langle \begin{bmatrix} e & e & e \\ e & e & e \\ e & e & e \end{bmatrix} \rightarrow \Downarrow; M \rightarrow D \right\rangle;$$

$\langle e \rightarrow L; S \leftrightarrow S \rangle$; $\langle e \rightarrow L; D \leftrightarrow S \rangle$; $\langle S \rightarrow e; L \rightarrow e \rangle$; $\langle D \rightarrow e; L \rightarrow e \rangle$;
$\langle e \rightarrow L; S \leftrightarrow D \rangle$; $\langle e \rightarrow L; D \leftrightarrow D \rangle$.

(see Fig. 2).

6 Applications of P colonies

Robot controllers P colonies and PCol automata were introduced as robot controllers in [6]. The authors followed two ideas of controlling a robot with use of P colonies.

Fig. 2 The sequence of configurations of the 2D P colony simulating beacon

The first controller model used the PCol automaton with instructions for the robot on the input tape. The agents have to read the current information from the tape and together with objects in the environment coming from the receptors, they generate objects—commands for actuators.

The agents are assembled into modules. All the modules are controlled by the main control unit. Each input symbol on the input tape represents a single instruction which has to be done by the robot, so the input string is the sequence of the actions which guides the robot in reaching its goal; performing all the actions. In this meaning the computation ends by halting, and it is successful if the whole input tape is read.

The second idea was to use original model of the P colony and put all information to the environment. They also used module-oriented structure of agents, each module performs the individual functions in the control of the robot. The authors constructed a P colony with four modules: The control unit, the left actuator controller, the right actuator controller and the infra-red receptors. The controller (the P colony) is completed by the input and output filter. The input filter codes signals from the robot receptors and spread the coded signal into the environment. In the environment there is the coded signal used by the agents. The output filter decodes the signal from the environment which the actuator controllers sent into it. Decoded signal is forwarded to the robot actuators.

Surface runoff 2D P colonies appear to be suitable to simulate multi-agent systems. In [5] the authors presented hydrological modelling flow of liquid over the earth's surface using 2D P colonies.

The issue of the flow of liquid over the Earth's surface is studied by experts from two areas - hydrology and geoinformatics. Both of these disciplines work closely together on the issue of the so-called "surface runoff". Surface runoff is the water flow that occurs when the soil is saturated to full capacity and excess water from rain, meltwater, or other sources flows over the land.

Agents in the model have capacity 2, the agent contains two objects. Each of the objects carries the information about the state of the agent. One of the objects stores information about the activity of the agent. At this stage of the simulation it is the information that the agent "flows" down the terrain or the agent is still inactive (belonging to the future rainfall). The other object stores information about the previous direction of flow. This information can further modify the way of the agent as inertia.

Based on the entered data—the slope surface, a source of fluid and quantity—they simulated the fluid distribution in the environment.

The research continues in [22] where the model can fill sinks (places without output).

7 Open problems and conclusions

We recalled the idea and functioning of the basic model of P colonies. This model was introduced in [44] in 2004. Since that time many papers and studies about the model and its variations have been published.

Almost all these works are focused on describing the computational power of more or less restricted variants of P colonies. Although extensive investigations have been made in this direction, some important questions remain open: what about deterministic P colonies? Furthermore, how to define determinism in P colonies? Another interesting question can be the problem of reversibility: how to define reversible P colonies and, thus, reversible computation in P colonies?

One reasonable idea is to introduce these concepts on the analogy of deterministic and/or reversible P systems. By [3], a P system is strongly reversible if every configuration has in-degree at most one and it is called reversible if every reachable configuration has in-degree at most one. We call a P system strongly deterministic if every configuration has out-degree at most one; and it is called deterministic if every reachable configuration has out-degree at most one. (By a reachable configuration we mean a configuration that is reachable from the initial configuration; the in-degree means all pre-images of the configuration not only the reachable.) We note that these notions were considered for a particular variant of (symport/antiport) P systems, but obviously, they can be considered for other variants as well (see, for example [2]). An extensive study of deterministic and/or reversible P systems has been performed during the years, focusing on the computational power of these systems, decidability questions, existence of syntactic criteria for deciding whether or not the P system is (strongly) deterministic or reversible. For more information, the reader is referred to [2, 3]. These problems and questions are interesting for P colonies and its variants as well. Furthermore, the so-called k-determinism, where every configuration has at most k successor configurations ($k > 1$) would also be interesting. For this concept, in the theory of P systems, we refer to [4, 49].

Automaton-like P colonies (PCol automata and their variants) are topics of further study as well. Although they have been compared to some classical and non-classical automata variants, precise descriptions of their relation to P automata and to some further variants of classical and non-classical automata would be very useful. Comparison of automaton-like P colonies and dP automata was initiated in [38], pp. 564-565. P automata are variants of antiport P systems accepting strings in an automaton-like fashion (for a summary on P automata, see Chapter 6 [50]). The notion of a dP automaton (distributed system of P automata) was introduced in [52]. A dP automaton consists of a finite number of component P automata which have their separate inputs and which also may communicate with each other by means of special antiport-like rules (roughly speaking, an antiport rule exchanges multisets of objects u and v where u is in the parent region and v is in the child region). A string accepted by a dP automaton is the concatenation of the strings accepted by the individual components during a computation performed by the system. For more information on dP automata, the interested reader is referred to the survey [26] and to [53]. A dP automaton is called finite if it has only a finite number of different configurations. A comparative study of automaton-like P colonies (APCol systems, PCol automata) and finite dP automata with several acceptance modes would be especially interesting.

P colonies can also be compared to several types of P systems, e.g. tissue-like constructs. Generalized communicating P systems, where the rules of this tissue-like membrane systems describe the move of pairs of objects from pairs of compartments to new locations (each object of the pair is moved to a new compartment) demonstrate functional similarity to P colonies. Initial steps in this direction have been made in [46].

2D P colonies are also important extensions of P colonies. This model was found suitable for simulations of multi-agent systems. One of the simulation introduced in [5] is the simulation of surface runoff.

In the future, many ways appear for improving the model of the 2D P colonies. One way is to assign the number to objects in addition to the type. This number will indicate the value of the parameter that the object represents. Another possibility is to extend the environment with a mechanism which is able to change the object in the environment independently from the activity of the agents.

The concepts and results reported in this survey demonstrate that P colonies (and their variants) are simple but very powerful devices. In addition, they can serve as modelling tools as well. The reader is welcome to contribute in exploring this fruitful research area.

Acknowledgements This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science—LQ1602, and by the Silesian University in Opava under the Student Funding Scheme, project SGS/11/2019. The work of E. CS-V. was supported by NKFIH (National Research, Development, and Innovation Office), Hungary, Grant no. K 120558.

References

1. Alhazov, A., Aman, B., Freund, R., & Păun, Gh. (2014). Matter and Anti-matter in Membrane Systems. In H. Jürgensen, J. Karhumäki, & A. Okhotin A (Eds.) *Descriptional complexity of formal systems: 16th international workshop*, DCFS 2014, Turku, Finland, August 5-8, 2014. Proceedings (pp. 65–76). Cham: Springer
2. Alhazov, A., Feund, R., & Morita, K. (2012). Sequential and maximally parallel multiset rewriting: Reversibility and determinism. *Natural Computing*, *11*, 95–106.
3. Alhazov, A., & Morita, K. (2010). On reversibility and determinism in P systems. In G. Păun, M. J. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, & A. Salomaa (Eds.), *Membrane computing, 10th international workshop, WMC 2009. Lecture notes in computer science* (Vol. 5957, pp. 158–168). Berlin: Springer.
4. Cienciala, L., Ciencialová, L., Frisco, P., & Sosík, P. (2007). On the power of deterministic and sequential communicating P systems. *International Journal of Foundations of Computer Science*, *18*(2), 415–431.
5. Cienciala, L., Ciencialová, L., & Langer, M. (2014). Modellingof surface run off using 2D P colonies. Modelling of surface runoff using 2D P colonies. In A. Alhazov, S. Cojocaru, M. Gheorghe, Y. Rogozhin, G. Rozenberg, & A. Salomaa (Eds.), *Membrane computing. CMC2013. Lecture Notes in computer science* (Vol. 8340, pp. 101–116). Berlin: Springer.

6. Cienciala, L., Ciencialová, L., Langer, M., & Perdek, M. (2014). The abilities of P colony based models in robot control. The abilities of P colony based models in robot control. In M. Gheorghe, G. Rozenberg, A. Salomaa, P. Sosík, & C. Zandron (Eds.), *Membrane computing. CMC 2014. Lecture notes in computer science* (Vol. 8961, pp. 179–193). Cham: Springer.
7. Cienciala, L., & Ciencialová, L. (2009). Eco-P colonies. In G. Păun, M. Pérez-Jiménez, A. Riscos-Núñez (Eds.) *Pre-proceedings of the 10th workshop on membrane computing*, Curtea de Arges, Romania (pp. 201–209)
8. Cienciala, L., & Ciencialová, L. (2011). P colonies and their extensions. In J. Kelemen & A. Kelemenová (Eds.), *Computation, cooperation, and life. Lecture notes in computer science* (Vol. 6610, pp. 158–169). Berlin: Springer.
9. Cienciala, L., & Ciencialová, L. (2016). Some new results of P colonies with bounded parameters. *Natural Computing*, *17*, 1–12.
10. Cienciala, L., Ciencialová, L., & Csuhaj-Varjú, E. (2014). P colonies processing strings. *Fundamenta Informaticae*, *134*(1–2), 51–65.
11. Ciencialová, L., Csuhaj-Varjú, E., Cienciala, L., & Sosík, P. (2016). P Colonies. *Bulletin of the International Membrane Computing Society*, *2*, 129–156.
12. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E., & Sosík, P. (2018). A logical representation of P colonies: An introduction. In C. Graciani, A. Riscos-Núñez, G. Păun, G. Rozenberg, & A. Salomaa (Eds.), *Enjoying natural computing. Lecture notes in computer science* (Vol. 11270, pp. 66–76). Cham: Springer.
13. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E., & Vaszil, Gy. (2010). PCol automata: recognizing strings with P colonies. In M. A. del Amor, G. Păun, I. P. H. de Mendoza, & A. R. Núñez (Eds.), *Eighth brainstorming week on membrane computing. Sevilla, 2010 RGNC Report 01/2010* (pp. 65–76). Sevilla: Fénix Editora.
14. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E., & Vaszil, Gy. (2018). Verifying APCol Systems. In Hinze, T., Behre, J. (eds.) *Proceedings of the nineteenth international conference on membrane computing (CMC19)*, Pro BUSINESS Verlag, pp. 247–258
15. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E., & Vaszil, Gy. (2019). APCol systems with verifier agents. In T. Hinze, G. Rozenberg, A. Salomaa, & C. Zandron (Eds.), *Membrane computing. CMC 2018. Lecture notes in computer science* (Vol. 11399, pp. 95–107). Cham: Springer.
16. Cienciala, L., Ciencialová, L., & Kelemenová, A. (2007). Onthe number of agents in P colonies. In G. Eleftherakis, P. Kefalas, Gh Păun, G. Rozenberg, & A. Salomaa (Eds.), *Membrane computing. WMC 2007. Lecture notes in computer science* (Vol. 4860, pp. 193–208). Berlin: Springer.
17. Cienciala, L., Ciencialová, L., & Kelemenová, A. (2008). Homogeneous P colonies. *Computing and Informatics*, *27*(3+), 481–496.
18. Cienciala, L., Ciencialová, L., & Langer, M. (2012). Modularityin P colonies with checking rules. In M. Gheorghe, G. Păun, G. Rozenberg, A. Salomaa, & S. Verlan (Eds.), *Membrane computing. CMC2011. Lecture notes in computer science* (Vol. 7184, pp. 104–119). Heidelberg: Springer.
19. Cienciala, L., Ciencialová, L., & Perdek, M. (2012). 2D Pcolonies. In E. Csuhaj-Varjú, M. Gheorghe, G. Rozenberg, A. Salomaa, & G. Vaszil (Eds.), *Membrane computing. CMC 2012. Lecture notes in computer science* (Vol. 7762, pp. 161–172). Berlin: Springer.
20. Ciencialová, L. (2019). APCol systems with agent creation. In T. Hinze, G. Rozenberg, A. Salomaa, & C. Zandron (Eds.), *Membrane computing. CMC 2018. Lecture notes in computer science* (Vol. 11399, pp. 84–94). Cham: Springer.
21. Ciencialová, L., Cienciala, L., & Csuhaj-Varjú, E. (2018). APCol systems with teams. In M. Gheorghe, G. Rozenberg, A. Salomaa, & C. Zandron (Eds.), *Membrane computing. CMC 2017. Lecture notes in computer science* (Vol. 10725, pp. 88–104). Cham: Springer.
22. Ciencialová, L., Cienciala, L., & Perdek, M. (2014). 2D P colonies used in hydrology simulations. *International Multidisciplinary Scientific GeoConference Surveying Geology and Mining Ecology Management, SGEM*, *1*(2), 3–10.
23. Ciencialová, L., Csuhaj-Varjú, E., Kelemenová, A., & Vaszil, Gy. (2009). Variants of P colonies with very simple cell structure. *International Journal of Computers, Communications and Control*, *4*(3), 224–233.
24. Ciencialová, L., Cienciala, L., & Sosík, P. (2016). Generalized P colonies with passive environment. In C. Graciani, D. Orellana-Martín, A. Riscos-Núñez, Á. Romero-Jiménez, L. Valencia-Cabrera (Eds.) *Fourteen brainstorming week on membrane computing*, pp. 151–162
25. Ciencialová, L., Cienciala, L., & Sosík, P. (2017). Pcolonies with evolving environment. In A. Leporati, G. Rozenberg, A. Salomaa, & C. Zandron (Eds.), *Membrane computing. CMC 2016. Lecture notes in computer science* (Vol. 10105, pp. 151–164). Cham: Springer.
26. Csuhaj-Varjú, E. (2003). P and dP Automata: Unconventional versus Classical Automata. *International Journal of Foundations of Computer Science*, *24*(7), 995–1008.
27. Csuhaj-Varjú, E. (2016). Extensions of P colonies (extended abstract). In: A. Leporati, & C. Zandron (Eds.) Proc. CMC17, Milan, 2016. University Milano-Bicocca & IMCS, Italy, pp. 281–286
28. Csuhaj-Varjú, E., Gheorghe, M., & Lefticaru, R. (2018). P colonies and kernel p systems. *International Journal of Advances in Engineering Sciences and Applied Mathematics*, *10*(3), 181–192.
29. Csuhaj-Varjú, E., Ibarra, O. H., & Vaszil, Gy. (2006). On the computational complexity of P automata. *Natural Computing*, *5*(2), 109–126.
30. Csuhaj-Varjú, E., Kántor, K., & Vaszil, Gy. (2018). Deterministic parsing with P colony automata. In C. Graciani, A. Riscos-Núñez, Gh Păun, G. Rozenberg, & A. Salomaa (Eds.), *Enjoying natural computing. Lecture notes in computer science* (Vol. 11270, pp. 88–98). Cham: Springer.
31. Csuhaj-Varjú, E., Kelemen, J., Kelemenová, A., Păun, G., & Vaszil, G. (2006). Computing with cells in environment: P colonies. *Journal of Multiple-Valued Logic and Soft Computing*, *12*(3), 201–215.
32. Csuhaj-Varjú, E., Margenstern, M., & Vaszil, Gy. (2006). Pcolonies with a bounded number of cells and programs. In H. J. Hoogeboom, G. Păun, G. Rozenberg, & A. Salomaa (Eds.), *Membrane computing. WMC 2006. Lecture notes in computer science* (Vol. 4361, pp. 352–366). Heidelberg: Springer.
33. Csuhaj-Varjú, E., & Verlan, S. (2018). Computationally complete generalized communicating P systems with three cells. In M. Gheorghe, G. Rozenberg, A. Salomaa, & C. Zandron (Eds.), *Membrane computing. CMC 2017. Lecture notes in computer science* (Vol. 10725, pp. 118–128). Cham: Springer.
34. Csuhaj-Varjú, E., & Verlan, S. (2018). Bi-simulation between PColonies and P systems with multi-stable catalysts. In M. Gheorghe, G. Rozenberg, A. Salomaa, & C. Zandron (Eds.), *Membrane computing. CMC 2017. Lecture notes in computer science* (Vol. 10725, pp. 88–104). Cham: Springer.
35. Freund, R., & Oswald, M. (2005). P colonies working in the maximally parallel and in the sequential mode. In D. Zaharie, D. Petcu, V. Negru, T. Jebelean, G. Ciobanu, A. Cicortas, A. Abraham, & M. Paprzycki (Eds.), *Seventh international symposium on symbolic and numeric algorithms for scientific computing (SYNASC 2005), 25–29 September 2005* (pp. 419–426). Timisoara, Romania: IEEE Computer Society.

36. Freund, R., & Oswald, M. (2006). P colonies and prescribed teams. *International Journal of Computer Mathematics*, *83*(7), 569–592.
37. Gardner, M. (1970). Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, *223*, 120–123.
38. Gheorghe, M., Păun, Gh, Pérez-Jiménez, M. J., & Rozenberg, G. (2013). Research frontiers of membrane computing: Open problems and research topics. *International Journal of Foundations of Computer Science*, *24*(5), 547–624. (Section 5. P Colonies and dP Automata by Erzsébet Csuhaj-Varjú).
39. Kántor, K., & Vaszil, Gy. (2014). Generalized P colony automata. *Jornal of Automata, Language and Combinatorics*, *19*(1–4), 145–156.
40. Kántor, K., & Vaszil, Gy. (2018). Generalized P colony automata and their relation to P automata. In M. Gheorghe, G. Rozenberg, A. Salomaa, & C. Zandron (Eds.), *Membrane computing. CMC 2017. Lecture notes in computer science* (Vol. 10725, pp. 167–182). Cham: Springer.
41. Kántor, K., & Vaszil, G. (2018). On the classes of languages characterized by generalized P colony automata. *Theoretical Computer Science*, *724*, 35–44.
42. Kelemen, J., & Kelemenová, A. (1992). A grammar-theoretic treatment of multiagent systems. *Cybernetics and Systems*, *23*(6), 621–633.
43. Kelemen, J., & Kelemenová, A. (2005). On P colonies, a biochemically inspired model of computation. In *Proc. of the6th International Symposium of Hungarian Researchers on Computational Intelligence, Budapest TECH* (pp. 40–56). Hungary
44. Kelemen, J., Kelemenová, A., & Păun, G. (2004). 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, Massachusetts, USA
45. Kelemenová, A. (2010). P Colonies, chap. 23.1, pp. 584–593. In: [50]
46. Krishna, S. N., Gheorghe, M., Ipate, F., Csuhaj-Varjú, E., & Ceterchi, R. (2017). Further results on generalised communicating P systems. *Theoretical Computer Science*, *701*, 146–160.
47. van Leeuwen, J., & Wiedermann, J. (2012). Computation as an unbounded process. *Theoretical Computer Science*, *429*, 202–212.
48. Minsky, M. L. (1967). *Computation: Finite and infinite machines*. Upper Saddle River, NJ, USA: Prentice-Hall Inc.
49. Oswald, M. (2003). P automata. PhD dissertation, Technological University of Vienna, Vienna
50. Păun, Gh, Rozenberg, G., & Salomaa, A. (2010). *The Oxford handbook of membrane computing*. New York, NY, USA: Oxford University Press Inc.
51. Păun, Gh. (2000). Computing with membranes. *Journal of Computer and System Sciences*, *61*(1), 108–143.
52. Păun, Gh, & Pérez-Jiménez, M. J. (2010). Solving problems in a distributed way in membrane computing: dP systems. *International Journal of Computers, Communication and Control*, *5*(2), 238–252.
53. Păun, Gh, & Pérez-Jiménez, M. J. (2012). P automata revisited. *Theoretical Computer Science*, *454*, 222–230.
54. Rozenberg, G., & Salomaa, A. (1997). *Handbook of formal languages: Beyonds words. Handbook of formal languages*. Berlin: Springer.
55. Rozenberg, G., & Salomaa, A. (Eds.). (1997). *Handbook of Formal languages: Word, language, grammar*. New York: Springer.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Lucie Ciencialová joined Institute of Computer Science of Silesian University in Opava in 2006 where she works as an assistant professor. She graduated in computer science at Silesian University in 2005. She finished her Ph.D. studies at Silesian University in 2008. She teaches theoretical computer science, mathematics, and logic. Her main research activity is in the fields of natural and unconventional computing. In particular, she studies the computational power and efficiency of computing models inspired from the structure and functioning of living cells.

Erzsébet Csuhaj-Varjú is a full professor at the Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary and Head of the Doctoral School of Informatics of the university. Her main research interests are formal languages and applications, natural computing, in particular, bio-inspired computing, distributed systems, and natural language processing. In these areas she authored and co-authored more than 200 publications (articles in international journals and edited volumes and a monograph) and she was editor or co-editor of 18 volumes. She has been co-founder of the area of grammar systems, a formal language-theoretic counterpart of the theory of multi-agent systems. She also has important contributions to bio-inspired computing; among others together with her co-authors, she has launched research vistas theory of networks of language processors and P automata (membrane automata) theory. She has been the supervisor (principal investigator) and participant of several Hungarian and bilateral granted research projects, team leader or leader of EU projects. She is the chair of the Advisory Board of the International Membrane Computing Society. She is member of the editorial board of International Journal of Foundations of Computer Science and Journal of Membrane Computing. In the last 12 years, she has been programme committee member and organizer of more than 50 international workshops and conferences, among them she was programme committee co-chair and chair of the organizing committee of FCT 2007, AFL 2008, CMC13, CiE 2014, MFCS 2014, AFL 2017. She is serving as member in several scientific or educational committees of the Eötvös Loránd University and she is the chair of the Committee of Information Science, Section of Mathematics of the Hungarian Academy of Sciences.

Luděk Cienciala works as an associate professor of computer science at the Institute of Computer Science at the Faculty of Philosophy and Science, Silesian University in Opava, Czech Republic. He graduated at Faculty of Science in Univerzity of Ostrava in 1996. In 2006, he received his Ph.D. in Applied Mathematics at Faculty of Science, University of Ostrava. He teaches logic, graph theory, computer graphics, and mathematics. His main areas of research are theoretical computer science, natural computing, and computer graphics.

Petr Sosík is an associate professor of computer science at the Faculty of Philosophy and Science, Silesian University in Opava, Czech Republic. He received his Ph.D. (1997) in computer science at Charles University in Prague, Czech Republic. His professional experience includes research stays at Canadian and European universities, as well as commercial software projects at the beginning of his career. His scientific work has been recognized with awards at various international conferences in research areas such as formal languages and applications, discrete mathematics, molecular and generally nature-inspired computing.