Journal of Membrane Computing (2019) 1:233–240
https://doi.org/10.1007/s41965-019-00022-1

REGULAR PAPER

Synchronization of rules in membrane computing

Bogdan Aman[1] · Gabriel Ciobanu[2]

Received: 17 July 2019 / Accepted: 1 October 2019 / Published online: 17 October 2019

Abstract
We modify the most used evolution strategy in membrane systems (namely that of maximal parallelism) by imposing a *synchronization* between rules. A synchronization over a set of rules can be applied only if each rule of the set can be applied at least once. For membrane systems working in the accepting mode, this synchronization is powerful enough to provide the computational completeness without any other ingredient (no catalysts, promoters, inhibitors, etc). The modeling power of synchronization is described by simulating the basic arithmetic operations (addition, subtraction, multiplication and division).

Keywords Membrane computing · Synchronization of the rules · Computational completeness · Arithmetic operations

1 Introduction

Membrane systems (known also as P systems) are able to model massively parallel systems inspired by the structure and behaviour of biological cells [23]. A membrane system can be represented as a hierarchical structure of regions (membranes) contained inside a unique outermost membrane called *skin*. Various classes of membrane systems (motivated by different features of the biological cells: catalytic entities, electric charges, antiport/symport communications, etc.) are presented in [24]. Several books including theoretical results and various applications in the field of membrane computing were published over the last years [5, 21, 27]. The main research directions considered in the field of membrane computing are: modeling power [3, 15], computational power with respect to the computational power of Turing machines using a limited number of resources [4], and efficiency by providing algorithms to solve NP-complete problems (weak [6] or strong [7]) by trading space for time, namely using an exponential space to obtain a polynomial time solution. Over the years, several operational and denotational semantics were defined for membrane systems [13, 16].

In this paper, we consider the class of P systems defined in [24] in which the various regions of the membrane structure contain multisets of objects and sets of evolution rules. Every region has its own task, such that all regions working in parallel achieve the general task of the entire system. The specific rules of each region modify its objects. The evolution of this class of P systems is given by applying the rules in a maximally parallel way [23]. The maximal parallelism ensures that the multiset of applicable rules chosen in a computation step cannot be further extended by adding further rules. This feature was preserved in many of the variants defined in the last twenty years, being a useful feature in obtaining computational completeness. Choosing the rules to be applied in a maximally parallel way is done non-deterministically, by respecting also some restrictions (e.g., priority relation among rules) or value-based criteria (e.g., the guards used in adaptive P systems [8] or kernel P systems [22]).

Synchronization is ubiquitous in nature: e.g., pacemaker cells in the heart, β-cells in the pancreas, long-range synchronization across brain during perception, contractions in the pregnant uterus, cellular clocks, quorum sensing. A biological motivation for such a synchronization can be found in the field of membrane computing in a statement published in [9]: coordinated gene-expression (and hence phenotypic change) in bacteria is best understood by noticing that a

This work was presented at 20th Conference on Membrane Computing (CMC20)

✉ Bogdan Aman
bogdan.aman@iit.academiaromana-is.ro

Gabriel Ciobanu
gabriel@info.uaic.ro

[1] Alexandru Ioan Cuza University of Iaşi, Iasi, Romania

[2] Romanian Academy, Institute of Computer Science, Iasi, Romania

colony is successful only by pooling together the activity of a quorum of cells. The synchronization between the evolution of regions was defined in [25], namely all regions use their rules in parallel in the maximal mode. In this paper, we introduce a different synchronization which was not yet considered in membrane computing: a synchronization among the rules of the same membrane. More exactly, a rule synchronizing with a non-empty set of rules is applicable at least once only if each rule from the set of rules is applicable at least once. This means that synchronization over a set of rules can be applied only if each rule of the set is applicable at least once. Just like for priorities (in membrane systems), this synchronization is given as a partial relation over the set of rules, specifying which rules are synchronized. The approach is conservative; the systems without a synchronization relation are in fact systems evolving according to the usual maximal evolution strategy.

An interesting aspect is that synchronization over rules is powerful enough to provide the computational completeness. Such a result is nice and surprising. For synchronized P systems working in the accepting mode, the synchronization is powerful enough to achieve the computational completeness in the absence of any other ingredients (no catalysts, promoters or inhibitors, for instance). This represents an improvement for the computational power of membrane systems. For instance, at least two catalysts are needed to achieve the computational completeness when the maximal parallelism strategy is used [19]. The number of catalysts can be reduced from two to one to obtain the computational completeness, but this is possible using a rather complicated control mechanisms [20].

To illustrate the modeling power of the new synchronized P systems, we show how the synchronization over rules can be used to implement arithmetic operations on numbers given in unary base.

2 Synchronization in membrane computing

A multiset over a finite alphabet O of objects is defined as a mapping $u : O \to \mathbb{N}$, where $\mathbb{N}$ denotes the set of non-negative integers. The empty multiset ε is defined such that $\varepsilon(a) = 0$ for all $a \in O$. As it is usual in the membrane computing community, we represent the multisets as strings; for example, the string $abaaca$ is the representation of the multiset u in which $u(a) = 4$, $u(b) = 1$ and $u(c) = 1$. Given a string u as a representation for a multiset, then the same multiset can be represented also by any permutation of the string u. Given a finite alphabet $O = \{a_1, \dots, a_n\}$, the set of all multisets over O is denoted by O^*, while the set of all non-empty multisets is denoted by $O^+ = O^* \backslash \varepsilon$. Given two multisets u and v over O, the multiset union is defined as $(u + v)(a) = u(a) + v(a)$ for all $a \in O$, and the multiset difference is defined as $(u - v)(a) = max\{0, u(a) - v(a)\}$ for all $a \in O$. Also, $u \le v$ if $u(a) \le v(a)$ for all $a \in O$.

For the sake of simplicity, in what follows, we consider the flat P systems (namely, P systems with only one membrane). Actually, according to [1], any P system can be flattened to a system with only one membrane.

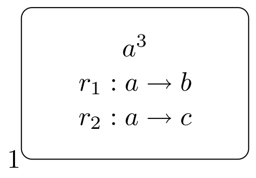
Definition 1 A *synchronized P system* of degree 1 is a tuple

$\Pi = (O, H, \mu, w_1, (R_1, \rho_1))$, where

- O and $H = \{1\}$ are finite non-empty sets of objects and labels for membranes, respectively;
- $\mu = [\,]_1$ is the membrane structure describing the fact that the system has only one membrane labeled 1;
- $w_1 \in O^*$ is the multiset of objects initially placed in the membrane 1;
- R_1 is a finite set of rules over the objects from O placed in membrane 1, and ρ_1 is a partial relation defined over the set R_1 of rules specifying the synchronization relation over the rules; given the multisets $u \in O^+$ and $v \in O^*$, a rule has the form $u \to v$, meaning that the multiset of objects u can be rewritten into the (possibly empty) multiset of objects v.

Synchronization means that a rule which needs to be synchronized with a non-empty set of rules is applicable (at least once) only if each rule from the set is applicable at least once. In what follows we give some examples that illustrate how the maximal parallelism behaviour is modified when synchronization of rules is used.

Example 1 Consider the membrane system depicted below:



Using the maximal parallel strategy, there are four possibilities to evolve in one step from the initial multiset a^3, as follows:

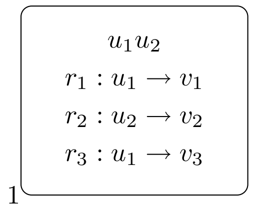
$$\begin{tikzcd}
b^3 & & b^2c \\
& a^3 \arrow[ul, "r_1^3"] \arrow[ur, "r_1^2r_2"] \arrow[dl, "r_2^3"'] \arrow[dr, "r_1r_2^2"'] & \\
c^3 & & bc^2
\end{tikzcd}$$

By adding to the initial system the synchronization $r_1 \otimes r_2$, the possibilities to evolve in one step from the initial multiset a^3 are:

$$\begin{tikzcd}
& b^2c \\
a^3 \arrow[ur, "r_1^2r_2"] \arrow[dr, "r_1r_2^2"'] & \\
& bc^2
\end{tikzcd}$$

Notice that synchronization reduces the number of possible evolutions.

Example 2 Consider the membrane system depicted below:



Using the maximal parallel strategy, there are two possibilities to evolve in one step from the initial multiset u_1u_2, as follows:

$$\begin{tikzcd}
& v_3v_2 \\
u_1u_2 \arrow[ur, "r_3r_2"] \arrow[dr, "r_1r_2"'] & \\
& v_1v_2
\end{tikzcd}$$

By adding the synchronization $r_1 \otimes r_2$ to the initial system, the possibilities to evolve in one step from the initial multiset u_1u_2 are:

$$\begin{tikzcd}
& v_3u_2 \\
u_1u_2 \arrow[ur, "r_3"] \arrow[dr, "r_1r_2"'] & \\
& v_1v_2
\end{tikzcd}$$

Notice that using synchronization, the system evolves differently (by producing new multisets) than using the maximal parallel strategy.

More details about the synchronization are provided in the next section in which the arithmetic operations are modeled using only one membrane.

3 Arithmetic operations using synchronized P systems

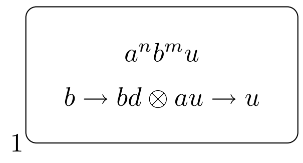
In this section, we define some (flat) synchronized P systems able to model the basic arithmetic operations for numbers given in unary base. For this purpose, we use the multiset natural encoding that assigns to each unit an object in the membrane system; in this way, a number n is encoded as a multiset of n similar objects. This encoding represents the encoding of natural numbers in base one. Just like in [11], the addition and subtraction are trivial, and the new defined synchronization relation is not needed for these operations (due to the fact that at most one rule is needed). The simplest implementation of addition requires no rule (and thus, no synchronization); the result is obtained by just counting all the objects contained in membrane 1. In a similar manner, subtraction of n (given by objects a) and m (given by objects b) is performed using only the rule $ab \to \varepsilon$ that deletes a pair of objects ab (synchronization is not necessary). Due to the maximal parallel manner of applying the rules, the rule $ab \to \varepsilon$ erases in the same computational step all the pairs ab, and the result is obtained by just counting all the objects contained in membrane 1. The time complexity of these simple arithmetic operations is $O(1)$.

3.1 Multiplication

The multiplication operation is a little bit more complex than the addition and subtraction operations presented previously. In Fig. 1, we describe a synchronized P system that is able to model the multiplication of n (given by objects a) by m (given by objects b). The result is obtained by counting all the objects d contained in membrane 1.

The use of the synchronization $b \to bd \otimes au \to u$ ensures that by applying any of the two rules at least once, the other rule is applied also at least once. As the two rules do not compete for the same objects and the maximal evolution strategy is used, this ensures that in each computational step, all the available objects b are rewritten by the rule $b \to bd$, while

Fig. 1 Multiplication using synchronization



only an object a is removed using the rule $au \to u$ (due to the existence of only one u object). The application of the two rules is repeated until all the objects a are consumed. Note that at each computational step, consuming one object a is done in parallel with the creation of m objects d. After n steps, even if there are resources for the rule $b \to bd$ to be applicable, the synchronization with the rule $au \to u$ for which there are no resources available means that the rule $b \to bd$ cannot be applied anymore, and the computation stops.

In [11], the multiplication operation is modeled in two ways: one way is without promoters but using priority between rules, while another way uses promoters and priority between rules. The time complexity is $O(m \cdot n)$ and $O(n)$, respectively. It is worth noting that using the synchronization relation, neither promoters nor priorities among rules are used. The synchronization has a similar effect as using promoters in [11], and the time complexity remains $O(n)$.

Example 3 Let us consider the multiplication of $n = 2$ by $m = 3$. The initial multiset is a^2b^3u. Applying once $b \to bd \otimes au \to u$, the initial multiset is rewritten to ab^3d^3u. Applying once more $b \to bd \otimes au \to u$, it is obtained the multiset b^3d^6u. Since there are no more a objects, the evolution stops and the result 6 is given by the number of d objects.

3.2 Division

This arithmetic operation is implemented as repeated subtractions. In Fig. 2, we depict a synchronized P system computing the quotient (objects c) and the remainder (objects r) of n (objects a) divided by m (objects b).

Regardless of the values of n and m, the evolution starts by applying the synchronized rules $b \to bd$ and $u_1 \to u_2$. The rule $b \to bd$ ensures that the number n of objects b remains unchanged, while the same number n of objects d is created to be used in the next steps. The synchronization with rule $u_1 \to u_2$ is used just to mark the fact that the objects b can be rewritten to the set of objects bd just in this step of the computation. The evolution continues with a subtraction step modeled by the synchronized rules $ab \to \varepsilon$, $u_2 \to u_3u_0$ and $d \to r$. The object u_0 is used for choosing the correct path depending on the objects present in the system at a given moment. Three cases are distinguished, depending on the number of objects a, b and/or r present in membrane 1 after the above two computational steps:

- If there are only a and r objects, namely all the objects b were consumed in the previous step by the rule $ab \to \varepsilon$, then the rule $au_3 \to au_{3a}$ is applied in parallel with the rule $u_0 \to u_0'$. Notice that the rule $bu_3 \to bu_{3b}$ is not applicable because there are no b objects. Also, the rule $u_3 \to c$ is not applicable as it requires the synchronization with the rule $u_0' \to \varepsilon$ that is not applicable yet because the object u_0' is not present in the system. This is followed by the application of the synchronized rules $u_{3a} \to u_1c$, $r \to b$ and $u_0' \to \varepsilon$. After applying all these rules, a subtraction was performed (marked by the creation of an object c) and the system can start another one (the system contains both a and b objects) by applying the first two sets of rules described above.
- If there are only b and r objects, namely all the objects a were consumed in the previous step by the rule $ab \to \varepsilon$, then the rule $bu_3 \to bu_{3b}$ is applied in parallel with the rule $u_0 \to u_0'$. This is followed by the application of the synchronized rules $u_{3b} \to \varepsilon$, $br \to \varepsilon$ and $u_0' \to \varepsilon$. This means that none of the objects u_i $(1 \le i \le 4)$ is present in the system, and so the computation halts. As in the previous case, none of the other rules rewriting u_3 can be applied.
- If there are only r objects, namely all the a and b objects were consumed in the previous step by the rule $ab \to \varepsilon$, then the synchronized rules $u_3 \to c$, $r \to \varepsilon$ and $u_0' \to \varepsilon$ are applied. This leads to the removal of all the objects r (there is no remainder after the division operation), and creation of another object c. This means that none of the objects u_i $(1 \le i \le 4)$ is available in the system, and so the computation halts. As argued in the first case, none of the other rules rewriting u_3 can be applied.

$$\boxed{\begin{array}{c} a^nb^mu_1 \\ b \to bd \otimes u_1 \to u_2 \\ ab \to \varepsilon \otimes u_2 \to u_3u_0 \otimes d \to r \\ u_0 \to u_0' \quad au_3 \to au_{3a} \quad bu_3 \to bu_{3b} \\ u_{3a} \to u_1c \otimes r \to b \otimes u_0' \to \varepsilon \\ u_{3b} \to \varepsilon \otimes br \to \varepsilon \otimes u_0' \to \varepsilon \\ u_3 \to c \otimes r \to \varepsilon \otimes u_0' \to \varepsilon \end{array}}\ _1$$

Fig. 2 Division using synchronization

The time complexity for the division operation is $O(3(c + 1))$. An improvement with respect to the division presented in [11] is given by the reduction of the number of membranes (one instead of two), and the fact that there is no need for promoters, inhibitors and/or priority.

Example 4 Let us consider the division of $n = 3$ by $m = 2$. The initial multiset is $a^3b^2u_1$. By applying the synchronization $b \to bd \otimes u_1 \to u_2$, it is obtained the multiset $a^3b^2d^2u_2$. The evolution continues using the synchronization $ab \to \varepsilon \otimes u_2 \to u_3u_0 \otimes d \to r$, leading to the multiset

$ar^2u_3u_0$. As there are only a and r objects available, then the rules $u_0 \to u'_0$ and $au_3 \to au_{3a}$ are applicable, and so the multiset $ar^2u_{3a}u'_0$ is obtained. By applying the synchronization $u_{3a} \to u_1c \otimes r \to b \otimes u'_0 \to \varepsilon$, it is obtained the multiset ab^2u_1c.

The above computation denotes a performed subtraction, and so the system can start another one. By applying the synchronization $b \to bd \otimes u_1 \to u_2$, the multiset $ab^2d^2u_2c$ is obtained. The evolution continues using the synchronization $ab \to \varepsilon \otimes u_2 \to u_3u_0 \otimes d \to r$, leading to the multiset $br^2u_3u_0c$. As there are only b and r objects available, then the rules $u_0 \to u'_0$ and $bu_3 \to bu_{3b}$ are applicable, and the multiset $br^2u_{3b}u'_0c$ is obtained. Applying the synchronization $u_{3b} \to \varepsilon \otimes br \to \varepsilon \otimes u'_0 \to \varepsilon$, it is obtained the multiset cr. In this case, the computation stops, and the result can be read as follows: the quotient is 1 (number of objects c), while the remainder is 1 (number of objects r).

4 Computational power of synchronized P systems

Given a string x over an alphabet $O = \{a_1, \dots, a_n\}$, its length is defined as $|x| = \Sigma_{a_i}|x|_{a_i}$, where $|x|_{a_i}$ is the number of a_i's appearing in the string x. The Parikh vector associated with x with respect to the set O is denoted by $(|x|_{a_1}, \dots, |x|_{a_n})$. Given a language L over an alphabet O, its Parikh image is $Ps(L) = \{(|x|_{a_1}, \dots, |x|_{a_n}) \mid x \in L\}$. Given a family of languages FL, its family of Parikh images is $PsFL = \{Ps(L) \mid L \in FL\}$. The family of languages RE contains the recursively enumerable string languages. More details on formal languages theory can be found in [26].

A *register machine* with m registers is a tuple $M = (m, B, l_0, l_h, P)$, where $l_0, l_h \in B$ represents the initial and final labels, and P is the set of instructions bijectively labeled by the labels from the set B. Given three labels $l_1 \in B \backslash \{l_h\}, l_2, l_3 \in B$, and a register j (with $1 \le j \le m$), the labeled instructions from the set P of the register machine M can be of the forms:

- $l_1 : (ADD(j), l_2, l_3)$.
  The effect of performing this instruction (usually called *increment*) is to increase by one the value of register j, and then continue the execution by either instruction l_2 or l_3. The choice between the instructions l_2 and l_3 is performed in a non-deterministic manner.
- $l_1 : (SUB(j), l_2, l_3)$.
  The effect of performing this instruction depends on the value of register j. The first case (usually called *zero-test*) is when the register j is empty: the value of the register j remains the same, and the execution continues with instruction l_3. The other case (usually called *decrement*) is when the register j is non-empty: its value is decreased by one, and the execution continues with instruction l_2.
- $l_h : HALT$. When this instruction is encountered, the execution stops.

A *configuration* of a register machine M with m registers is given as a tuple containing the values of the m registers and the label of the instruction to be executed in the next step. The execution of the instruction l_0 marks the start of a computation, while the halt instruction l_h marks the termination of a computation.

Given a synchronized P system Π, the set $Ps_{acc}(\Pi)$ contains the Parikh vectors over $\mathbb{N}_+$ (the set of non-negative integers) accepted as result of a halting computation in Π. When a P system accepts as a result of a halting computation all the vectors over $\mathbb{N}_+$ given as input, it is said to be working in the accepting case. In what follows, *textitPs*$_{acc}OP_m(synch)$ denotes the families of sets $Ps_{acc}(\Pi)$ obtained as result of halting computations in synchronized P systems with no more than m membranes.

The next theorem illustrates how a register machine can be simulated by a synchronized P systems working in an accepting mode. The synchronization is powerful enough to get the computational completeness without using any additional catalyst, promoter or inhibitor in the synchronized P system.

Theorem 1 *For any $m \ge 1$, $Ps_{acc}OP_m(synch) = PsRE$.*

Proof Let us consider a register machine $M = \left(m, B, l_0, l_h, P\right)$. We present the steps to construct a synchronized P system Π with only one membrane that is able to simulate the computation of the register machine M. The membrane system Π is the tuple $(O, H, \mu, w_1, (R_1, \rho_1))$ with

- $O = \{a_r, u_r, v_r \mid 1 \le r \le m\} \cup \{l, l', \bar{l} \in B\}$;
- $H = \{1\}$; $\mu = [\,]_1$;
- $w_1 = l_0a_1^{k_1} \dots a_m^{k_m}$.

The initial configuration of the membrane system Π is given by the membrane content, namely the multiset w_1 that is composed of:

- an object l_0 that marks the beginning of the evolution in the simulated register machine;
- the multiset $a_1^{k_1} \dots a_m^{k_m}$ representing the vector $(k_1, \dots, k_m)$ that needs to be accepted by the (constructed) synchronized P system; the content of register r is represented by the number k_r of copies of the objects a_r, where $1 \le r \le m$.

The rules of R_1 given below are used to simulate the instructions of the simulated register machine M. Note that for the *SUB* operation, the synchronization relation ρ_1 contains synchronizations between some of the rules.

The instructions of the simulated register machine are:

- $l_1 : (ADD(r), l_2, l_3)$, with $l_1 \in B \setminus \{l_h\}, l_2, l_3 \in B, 1 \leq j \leq m$ is simulated by the rules

$r_{11} : l_1 \rightarrow a_r l_2,$
$r_{12} : l_1 \rightarrow a_r l_3.$

If the current instruction to be executed in the register machine M is $l_1 : (ADD(r), l_2, l_3)$, it is simulated by the rules r_{11} and r_{12} of our membrane system. Choosing which of the two rules is applied is done non-deterministically. As a result of applying the rule r_{11} or r_{12}, we get the creation of a new object a_r and one of the objects l_2 or l_3, respectively. The possible execution is depicted in Fig. 3, where for simplicity we omit the object u and the objects modeling the other registers not involved in the evolution.

- $l_1 : (SUB(r), l_2, l_3)$, with $l_1 \in B \setminus \{l_h\}, l_2, l_3 \in B, 1 \leq r \leq m$ is simulated by the rules

$r_{20} : l_1 \rightarrow \overline{l_1} u_r,$
$r_{21} : \overline{l_1} \rightarrow l_1',$
$r_{22} : a_r u_r \rightarrow v_r,$
$r_{23} : l_1' \rightarrow l_2,$
$r_{24} : v_r \rightarrow \varepsilon,$
$r_{25} : l_1' \rightarrow l_3 u,$
$r_{26} : u_r \rightarrow \varepsilon,$

where $\{r_{23} \otimes r_{24}\} \subseteq \rho_1$ and $\{r_{25} \otimes r_{26}\} \subseteq \rho_1$.

If the current instruction to be executed in the register machine M is $l_1 : (SUB(r), l_2, l_3)$, the simulation begins by executing the rule r_{20} that creates an additional object u_r used for the zero test. If there is no object a_r present in membrane 1, then no rule is executed in parallel with rule r_{21}. If there exists at least one a_r object, then the rule r_{22} is executed in parallel with rule r_{21}. Application of the rule r_{22} removes an object a_r together with the rewriting of the unique object u_r into the unique object v_r. The unique object u_r ensures that only one a_r object can be deleted when simulating the instruction $l_1 : (SUB(r), l_2, l_3)$. Due to the fact that rule r_{23} is synchronized with rule r_{24}, the object l_1' is replaced by object l_2, and also the object v_r is removed in order to be used in subsequent simulation of *SUB* instruction. If there was no object a_r in the system, namely the rule r_{22} was not applied, then the object u_r remains unchanged while object l_1 is rewritten to object l_1'. This means that the synchronized rules r_{25} and r_{26} are now applicable, leading to the creation of the object l_3, while removing the unique object u_r from the system. Due to the synchronization, after every computational step there exists no object u_r nor v_r in the system. The evolution is depicted in Fig. 4.

- $l_h : HALT$ is simulated by $r_h : l_h \rightarrow \varepsilon$.

If the current instruction to be executed is $l_h : HALT$ marking the end of the computation for the register machine M, this step is simulated by removing the object l_h such that no other rule is applicable anymore. The result of the computation is given by the number of objects a_r $(1 \leq r \leq m)$ found in membrane 1 after consuming the object l_h.

□

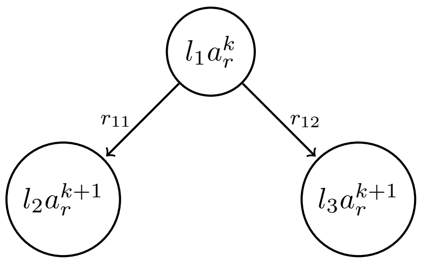


Fig. 3 Simulating *ADD* instruction

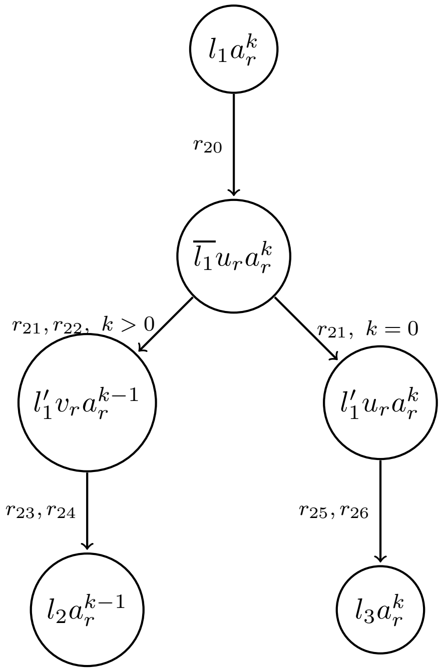


Fig. 4 Simulating *SUB* instruction

5 Conclusion and related work

In [25], the authors introduced the synchronization between membranes by considering that all the regions use their rules in parallel in a maximal way. A similar idea of synchronization of computation among various membranes is treated in [2, 17]. In this paper, we defined *synchronized P systems* by introducing a different synchronization relation to control the application of the evolution rules in the same membrane. We just adjust the use of the rules in a maximally parallel way by considering an additional synchronization over the rules. A synchronization over a set of rules can be applied only if each rule of the set can be applied at least once. According to our knowledge, this natural synchronization of the rules is for the first time defined and studied in membrane computing. For the sake of simplicity, we considered flat P systems with symbol objects and rewriting rules, but the synchronization relation can be defined in any class of membrane systems. Studying the effect of the synchronized relation in other classes of membrane systems represents future work.

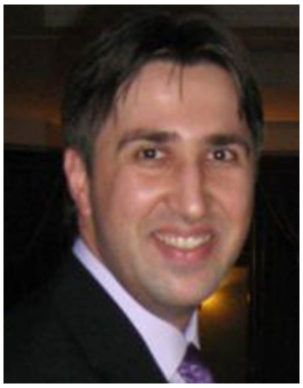
To illustrate the modeling power of the synchronized P systems, we described the arithmetic operations on numbers given in unary base. Comparing the multiplication operation with that presented in [11], it is worth noting that the synchronization is able to simulate properly the use of promoters. The computational completeness of the synchronized P systems can be obtained in the accepting case without using any additional ingredient (catalysts, for instance). This marks an improvement because we know from [19] that two catalysts are needed to achieve computational completeness when the maximal parallelism strategy is used. It is possible to reduce the number of catalysts to one, but this implies the use of complicated control mechanisms [20].

There exist in the membrane computing literature certain approaches in which the maximal parallelism was not taken as the principal strategy for evolution. Sequential systems presented in [18] defined a class of membrane systems that impose the application of only one rule in each computational step. A version stronger than sequential P systems but weaker than maximal P systems is the class of non-synchronized P systems in which any number of rules can be used in each computational step. The sequential and non-synchronized systems require additional control mechanisms (e.g., priorities) in order to obtain the computational completeness. Similar control mechanisms (e.g., promoters, bi-stable catalysts) are used in time-free P systems [12] to synchronize rules having unknown evolution times. The non-synchronized P systems are taken a step further such that for a given number k, in each computational step, one can apply either exactly, at least or at most k rules (see [15]). A class of P systems using boundary rules imposes a parallelism of type (k, q) in the application of the rules [10]. A parallelism of type (k, q) imposes for each computational step a bound of at most k membranes that can evolve using at most q rules inside each of the m membranes. The minimal parallelism defined in [14] proposes a different condition: if a set of rules contains at least an applicable rule, then at least one rule is actually applied; more rules can be used as there exists no upper bound on the number of used rules.

References

1. Agrigoroaiei, O., & Ciobanu, G. (2010). Flattening the transition P systems with dissolution. *Lecture Notes in Computer Science*, *6501*, 53–64.
2. Alhazov, A., Margenstern, M., & Verlan, S. (2009). Fast synchronization in P systems. *Lecture Notes in Computer Science*, *5391*, 118–128.
3. Aman, B., & Ciobanu, G. (2008). Describing the immune system using enhanced mobile membranes. *Electronic Notes in Theoretical Computer Science*, *194*(3), 5–18.
4. Aman, B., & Ciobanu, G. (2009). Turing completeness using three mobile membranes. *Lecture Notes in Computer Science*, *5715*, 42–55.
5. Aman, B., & Ciobanu, G. (2011). *Mobility in Process Calculi and Natural Computing*. Berlin: Springer.
6. Aman, B., & Ciobanu, G. (2011). Solving a weak NP-complete problem in polynomial time by using mutual mobile membrane systems. *Acta Informatica*, *48*(7–8), 409–415.
7. Aman, B., & Ciobanu, G. (2017). Efficiently solving the bin packing problem through bio-inspired mobility. *Acta Informatica*, *54*(4), 435–445.
8. Aman, B., & Ciobanu, G. (2019). Adaptive P systems. *Lecture Notes in Computer Science*, *11399*, 57–72.
9. Bernardini, F., Gheorghe, M., & Krasnogor, N. (2007). Quorum sensing P systems. *Theoretical Computer Science*, *371*, 20–33.
10. Bernardini, F., Romero-Campero, F.J., Gheorghe, M., Pérez-Jiménez, M.J., Margenstern, M., Verlan, S., & Krasnogor, N. On P systems with bounded parallelism. In *IEEE Computer Society Proceedings 7th SYNASC* (pp. 399–406).
11. Bonchiş, C., Ciobanu, G., & Izbaşa, C. (2006). Encodings and arithmetic operations in membrane computing. *Lecture Notes in Computer Science*, *3959*, 621–630.
12. Cavaliere, M., & Sburlan, D. (2005). Time and synchronization in membrane systems. *Fundamenta Informaticae*, *64*(1–4), 65–77.
13. Ciobanu, G. (2010). Semantics of P systems. In *The Oxford Handbook of Membrane Computing*, Oxford: Oxford University Press (pp. 413–436).
14. Ciobanu, G., Pan, L., Păun, G., & Pérez-Jiménez, M. J. (2007). P systems with minimal parallelism. *Theoretical Computer Science*, *378*, 117–130.
15. Ciobanu, G., Păun, G., & Pérez-Jiménez, M. J. (Eds.). (2006). *Applications of Membrane Computing*. Berlin: Springer.
16. Ciobanu, G., & Todoran, E. N. (2017). Denotational semantics of membrane systems by using complete metric spaces. *Theoretical Computer Science*, *701*, 85–108.
17. Dinneen, M. J., Kim, Y.-B., & Nicolescu, R. (2012). Faster synchronization in P systems. *Natural Computing*, *11*(1), 107–115.

18. Freund, R. (2005). Asynchronous P systems and P systems working in the sequential mode. *Lecture Notes in Computer Science*, *3365*, 36–62.
19. Freund, R., Kari, L., Oswald, M., & Sosík, P. (2005). Computationally universal P systems without priorities: two catalysts are sufficient. *Theoretical Computer Science*, *330*, 251–266.
20. Freund, R., & Păun, G. (2013). How to obtain computational completeness in P systems with one catalyst. *Electronic Proceedings in Theoretical Computer Science*, *128*, 47–61.
21. Frisco, P., Gheorghe, M., & Pérez-Jiménez, M. J. (Eds.). (2014). *Applications of Membrane Computing in Systems and Synthetic Biology*. Berlin: Springer.
22. Gheorghe, M., & Ipate, F. (2014). A kernel P systems survey. *Lecture Notes in Computer Science*, *8340*, 1–9.
23. Păun, G. (2002). *Membrane Computing: An Introduction*. Berlin: Springer.
24. Păun, G., Rozenberg, G., & Salomaa, A. (Eds.). (2010). *The Oxford Handbook of Membrane Computing*. Oxford: Oxford University Press.
25. Păun, G, & Sheng, Y. (1999). On synchronization in P systems. *Fundamenta Informaticae*, *38*(4), 397–410.
26. Rozenberg, G., & Salomaa, A. (Eds.). (1997). *Handbook of Formal Languages* (Vol. 3). Berlin: Springer.
27. Zhang, G., Pérez-Jiménez, M. J., & Gheorghe, M. (2017). *Real-Life Applications with Membrane Computing*. Berlin: Springer.



Bogdan Aman graduated in 'A.I. Cuza' University of Iaşi, Faculty of Mathematics in 2007 and completed his PhD thesis in 2009 under the supervision of Professor Gabriel Ciobanu at the Romanian Academy (Iaşi branch). He has received a public recognition for his research with the 2013 Grigore Moisil Award of the Romanian Academy of Sciences. His main research fields are membrane computing, natural computing, process algebra, type systems, and other theoretical aspects of computer science.

Gabriel Ciobanu is a researcher at the Romanian Academy of Sciences (Iasi branch) and at 'A.I.Cuza' University of Iasi. His research articles on distributed systems (process calculi), formal methods (semantics, logics), and natural computing (membrane systems). For his scientific contributions, he received awards from the Romanian Academy (2000, 2004, and 2013) and Ad-Astra Association (2018). He is the Editor-in-Chief of the Scientific Annals of Computer Science, and a member of Academia Europaea (the Academy of Europe).