



When catalytic P systems with one catalyst can be computationally complete

Artiom Alhazov¹ · Rudolf Freund² · Sergiu Ivanov³

Received: 7 March 2021 / Accepted: 21 July 2021 / Published online: 3 August 2021
© The Author(s) 2021

Abstract

Catalytic P systems are among the first variants of membrane systems ever considered in this area. This variant of systems also features some prominent computational complexity questions, and in particular the problem of using only one catalyst in the whole system: is one catalyst enough to allow for generating all recursively enumerable sets of multisets? Several additional ingredients have been shown to be sufficient for obtaining computational completeness even with only one catalyst. In this paper, we show that one catalyst is sufficient for obtaining computational completeness if either catalytic rules have weak priority over non-catalytic rules or else instead of the standard maximally parallel derivation mode, we use the derivation mode *maxobjects*, i.e., we only take those multisets of rules which affect the maximal number of objects in the underlying configuration.

Keywords Catalytic rules · Derivation mode maxobjects · Priority of catalytic rules · P systems

1 Introduction

Membrane systems were introduced in [10] as a multiset-rewriting model of computing inspired by the structure and the functioning of the living cell. During two decades, now membrane computing has attracted the interest of many researchers, and its development is documented in two textbooks, see [11] and [12]. For actual information, see the P systems webpage [14] and the issues of the Bulletin of

the International Membrane Computing Society and of the Journal of Membrane Computing.

One basic feature of P systems already presented in [10] is the maximally parallel derivation mode, i.e., using non-extendable multisets of rules in every derivation step. The result of a computation can be extracted when the system halts, i.e., when no rule is applicable any more. Catalysts are special symbols which allow only one object to evolve in its context (in contrast to promoters, which allow all promoted objects to evolve) and in their basic variant never evolve themselves, i.e., a catalytic rule is of the form $ca \rightarrow cv$, where c is a catalyst, a is a single object and v is a multiset of objects. In contrast, non-catalytic rules in catalytic P systems are non-cooperative rules of the form $a \rightarrow v$.

From the beginning, the question how many catalysts are needed in the whole system for obtaining computational completeness has been one of the most intriguing challenges regarding (catalytic) P systems. In [5], it has already been shown that two catalysts are enough for generating any recursively enumerable set of multisets, without any additional ingredients like a priority relation on the rules as used in the original definition. As already known from the beginning, without catalysts, only regular (semi-linear) sets can be generated when using the standard halting mode, i.e., a result is extracted when the system halts with no rule being applicable any more. As shown, for example, in [7],

A preliminary version of this paper only containing the result on weak priority was presented at ICMC 2020, see [2].

✉ Rudolf Freund
rudi@emcc.at

Artiom Alhazov
artiom@math.md

Sergiu Ivanov
sergiu.ivanov@ibisc.univ-evry.fr

¹ Vladimir Andrunachievici Institute of Mathematics and Computer Science, Academiei 5, 2028 Chişinău, Moldova

² Faculty of Informatics, TU Wien, Favoritenstraße 9–11, 1040 Wien, Austria

³ IBISC, Université Évry, Paris-Saclay, 23, Boulevard de France, 91034 Évry, France

using various additional ingredients, i.e., additional control mechanisms, one catalyst can be sufficient: in P systems with label selection, only rules from one set of a finite number of sets of rules in each computation step are used; in time-varying P systems, the available sets of rules change periodically with time. On the other hand, for catalytic P systems with only one catalyst, a lower bound has been established in [8]: P systems with one catalyst can simulate partially blind register machines, i.e., they can generate more than just semi-linear sets.

In [2], we returned to the idea of using a priority relation on the rules, but took only a very weak form of such a priority relation: we only required that overall in the system, catalytic rules have weak priority over non-catalytic rules. This means that the catalyst c must not stay idle if the current configuration contains an object a with which it may cooperate in a rule $ca \rightarrow cv$; all remaining objects evolve in the maximally parallel way with non-cooperative rules. On the other hand, if the current configuration does not contain an object a with which the catalyst c may cooperate in a rule $ca \rightarrow cv$, c may stay idle and *all* objects evolve in the maximally parallel way with non-cooperative rules. Even without using more than this, weak priority of catalytic rules over the non-catalytic (non-cooperative) rules, we could establish computational completeness for catalytic P systems with only one catalyst. In this paper, we recall the result from [2], but show a somehow much stronger result using a similar construction as in [2]: we show computational completeness for catalytic P systems with only one catalyst using the derivation mode *maxobjects*, i.e., we only take those multisets of rules which affect the maximal number of objects in the underlying configuration.

2 Definitions

For an alphabet V , by V^* , we denote the free monoid generated by V under the operation of concatenation, i.e., containing all possible strings over V . The *empty string* is denoted by λ . A *multiset* M with underlying set A is a pair (A, f) where $f : A \rightarrow \mathbb{N}$ is a mapping. If $M = (A, f)$ is a multiset, then its *support* is defined as $supp(M) = \{x \in A \mid f(x) > 0\}$. A multiset is empty (respectively, finite) if its support is the empty set (respectively, a finite set). If $M = (A, f)$ is a finite multiset over A and $supp(M) = \{a_1, \dots, a_k\}$, then it can also be represented by the string $a_1^{f(a_1)} \dots a_k^{f(a_k)}$ over the alphabet $\{a_1, \dots, a_k\}$, and, moreover, all permutations of this string precisely identify the same multiset M . For further notions and results in formal language theory, we refer to textbooks like [4] and [13].

2.1 Register machines

Register machines are well-known universal devices for generating or accepting or even computing with sets of vectors of natural numbers.

Definition 1 A *register machine* is a construct

$$M = (m, B, l_0, l_h, P)$$

where

- m is the number of registers,
- B is a finite set of labels,
- $l_0 \in B$ is the initial label,
- $l_h \in B$ is the final label, and
- P is the set of instructions bijectively labeled by elements of B .

The instructions of M can be of the following forms:

- $p : (ADD(r), q, s)$, with $p \in B \setminus \{l_h\}, q, s \in B, 1 \leq r \leq m$.
Increase the value of register r by one, and non-deterministically jump to instruction q or s .
- $p : (SUB(r), q, s)$, with $p \in B \setminus \{l_h\}, q, s \in B, 1 \leq r \leq m$.
If the value of register r is not zero, then decrease the value of register r by one (*decrement case*) and jump to instruction q , otherwise jump to instruction s (*zero-test case*).
- $l_h : HALT$.
Stop the execution of the register machine.

A *configuration* of a register machine is described by the contents of each register and by the value of the current label, which indicates the next instruction to be executed. M is called *deterministic* if the ADD-instructions all are of the form $p : (ADD(r), q, q)$, simply also written as $p : (ADD(r), q)$.

In the *accepting* case, a computation starts with the input of an l -vector of natural numbers in its first l registers and by executing the first instruction of P (labeled with l_0); it terminates with reaching the *HALT*-instruction. Without loss of generality, we may assume all registers to be empty at the end of the computation.

In the *generating* case, a computation starts with all registers being empty and by executing the first instruction of P (labeled with l_0); it terminates with reaching the *HALT*-instruction and the output of a k -vector of natural numbers in its last k registers. Without loss of generality, we may

assume all registers except the last k output registers to be empty at the end of the computation.

In the *computing* case, a computation starts with the input of an l -vector of natural numbers in its first l registers and by executing the first instruction of P (labeled with l_0); it terminates with reaching the *HALT*-instruction and the output of a k -vector of natural numbers in its last k registers. Without loss of generality, we may assume all registers except the last k output registers to be empty at the end of the computation.

For useful results on the computational power of register machines, we refer to [9]; for example, to prove our main theorem, we need the following formulation of results for register machines generating or accepting recursively enumerable sets of vectors of natural numbers with k components or computing partial recursive relations on vectors of natural numbers:

Proposition 1 *Deterministic register machines can accept any recursively enumerable set of vectors of natural numbers with l components using precisely $l + 2$ registers. Without loss of generality, we may assume that at the end of an accepting computation, all registers are empty.*

Proposition 2 *Register machines can generate any recursively enumerable set of vectors of natural numbers with k components using precisely $k + 2$ registers. Without loss of generality, we may assume that at the end of an accepting computation the first two registers are empty, and, moreover, on the output registers, i.e., the last k registers, no *SUB*-instruction is ever used.*

Proposition 3 *Register machines can compute any partial recursive relation on vectors of natural numbers with l components as input and vectors of natural numbers with k components as output using precisely $l + 2 + k$ registers, where without loss of generality, we may assume that at the end of a successful computation the first $l + 2$ registers are empty, and, moreover, on the output registers, i.e., the last k registers, no *SUB*-instruction is ever used.*

In all cases, it is essential that the output registers never need to be decremented.

2.2 Partially blind register machines

We now consider one-way non-deterministic machines which have registers allowed to hold positive or negative integers and which accept by final state with all registers being zero. Such machines are called *blind* if their actions depend on the state (label) and the input only and not on the register configuration itself. They are called *partially*

blind if they block when any register is negative (i.e., only non-negative register contents is allowed) but do not know whether or not any of the registers contains zero.

Definition 2 A *partially blind register machine* is a construct

$$M = (m, B, l_0, l_h, P)$$

where

- m is the number of registers,
- B is a finite set of labels,
- $l_0 \in B$ is the initial label,
- $l_h \in B$ is the final label, and
- P is the set of instructions bijectively labeled by elements of B .

The instructions of M can be of the following forms:

- $p : (ADD(r), q, s)$, with $p \in B \setminus \{l_h\}$, $q, s \in B$, $1 \leq r \leq m$.
Increase the value of register r by one, and non-deterministically jump to instruction q or s .
- $p : (SUB(r), q)$, with $p \in B \setminus \{l_h\}$, $q \in B$, $1 \leq r \leq m$.
If the value of register r is not zero, then decrease the value of register r by one and jump to instruction q , otherwise abort the computation.
- $l_h : HALT$.
Stop the execution of the register machine.

Again, a *configuration* of a partially blind register machine is described by the contents of each register and by the value of the current label, which indicates the next instruction to be executed.

A computation works as for a register machine, yet with the restriction that a computation is aborted if one tries to decrement a register which is zero. Moreover, computing, accepting or generating now also requires all registers (except output registers) to be empty at the end of the computation.

Remark 1 For any register machine (even for a blind or a partially blind one), without loss of generality, we may assume that the first instruction is an *ADD*-instruction: in fact, given a register machine $M = (m, B, l_0, l_h, P)$ with having a *SUB*-instruction as its first instruction, we can immediately construct an equivalent register machine M' which starts with an increment immediately followed by a decrement of the first register:

$$\begin{aligned}
 M' &= (m, B', l'_0, l_h, P'), \\
 B' &= B \cup \{l'_0, l''_0\}, \\
 P' &= P \cup \{l'_0 : (ADD(1), l''_0, l''_0), l''_0 : (SUB(1), l_0, l_0)\}.
 \end{aligned}$$

2.3 Catalytic P systems

As in [8], the following definition cites Definition 4.1 in Chapter 4 of [12].

Definition 3 An *extended catalytic P system* of degree $m \geq 1$ is a construct

$$\Pi = (O, C, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0)$$

where

- O is the alphabet of objects;
- $C \subseteq O$ is the alphabet of catalysts;
- μ is a membrane structure of degree m with membranes labeled in a one-to-one manner with the natural numbers $1, \dots, m$;
- $w_1, \dots, w_m \in O^*$ are the multisets of objects initially present in the m regions of μ ;
- $R_i, 1 \leq i \leq m$, are finite sets of *evolution rules* over O associated with the regions $1, 2, \dots, m$ of μ ; these evolution rules are of the forms $ca \rightarrow cv$ or $a \rightarrow v$, where c is a catalyst, a is an object from $O \setminus C$, and v is a string from $((O \setminus C) \times \{here, out, in\})^*$;
- $i_0 \in \{0, 1, \dots, m\}$ indicates the output region of Π (0 indicates the environment).

The membrane structure and the multisets in Π at the current moment of a computation constitute a *configuration* of the P system; the *initial configuration* is given by the initial multisets w_1, \dots, w_m . A transition between two configurations is governed by the application of the evolution rules, which is done in a given derivation mode. In this paper, we consider the maximally parallel derivation mode (*max* for short), i.e., only applicable multisets of rules which cannot be extended by further rules are to be applied to the objects in all membrane regions, as well as the derivation mode *maxobjects* where in every membrane region only applicable multisets of rules which affect the maximal number of objects are to be applied. We remark that all the multisets which can be used in the derivation mode *maxobjects* can also be used in the derivation mode *max*.

The application of a rule $u \rightarrow v$ in a region containing a multiset M results in subtracting from M the multiset identified by u , and then in adding the multiset identified by

v . The objects can eventually be transported through membranes due to the targets *in* (i.e., to the region of an inner membrane) and *out* (i.e., to the region of the surrounding membrane) or else stay in the membrane where the rule has been applied (target *here*). We refer to [12] for further details and examples.

The P system continues with applying multisets of rules according to the derivation mode until there remain no applicable rules in any region of Π . Then, the system *halts*. We consider the number of objects from $O \setminus C$ contained in the output region i_0 at the moment when the system halts as the *result* of the underlying computation of Π . The system is called *extended* since the catalytic objects in C are not counted to the result of a computation. Yet, as often done in the literature, in the following, we will omit the term *extended* and just speak of *catalytic P systems*, especially as we will restrict ourselves to P systems with only one catalyst.

The set of results of all computations possible in Π using the derivation mode δ is called the set of natural numbers *generated by Π* using δ and it is denoted by $N(\Pi, \delta)$ if we only count the total number of objects in the output membrane; if we distinguish between the multiplicities of different objects, we obtain a set of vectors of natural numbers denoted by $Ps(\Pi, \delta)$.

Remark 2 As in this paper, we only consider catalytic P systems with only one catalyst, without loss of generality, we can restrict ourselves to one-membrane catalytic P systems with the single catalyst in the skin membrane, by taking into account the well-known flattening process, e.g., see [6].

Remark 3 Finally, we make the convention that a one-membrane catalytic P system with the single catalyst in the skin membrane and with internal output in the skin membrane, not taking into account the single catalyst c for the results, throughout the rest of the paper will be described without specifying the trivial membrane structure or the output region (assumed to be the skin membrane), i.e., we will just write

$$\Pi = (O, \{c\}, w, R)$$

where O is the set of objects, c is the single catalyst, w is the initial input specifying the initial configuration, and R is the set of rules.

As already mentioned earlier, the following result was shown in [8], establishing a lower bound for the computational power of catalytic P systems with only one catalyst:

Proposition 4 *Catalytic P systems with only one catalyst working in the derivation mode max have at least the computational power of partially blind register machines.*

Example 1 In [8], it was shown that the vector set

$$S = \{(n, m) \mid 0 \leq n, n \leq m \leq 2^n\}$$

(which is not semi-linear) can be generated by a P system Π working in the derivation mode max with only one catalyst and 19 rules as the multiset language

$$L = \{a^n b^m \mid 0 \leq n, n \leq m \leq 2^n\},$$

i.e., $Ps(\Pi, \max) = L$.

3 Weak priority of catalytic rules

In this section, we now study catalytic P systems with only one catalyst in which the catalytic rules have weak priority over the non-catalytic rules.

Example 2 To illustrate this weak priority of catalytic rules over the non-catalytic rules, consider the rules $ca \rightarrow cb$ and $a \rightarrow d$. If the current configuration contains $k > 0$ copies of a , then the catalytic rule $ca \rightarrow cb$ must be applied to one of the copies, while the rest of objects a may be taken up by the non-catalytic rule $a \rightarrow d$. In particular, if $k = 1$, only $ca \rightarrow cb$ may be applied.

We would like to highlight the fact that weak priority of catalytic rules is much weaker than the general weak priority, as the priority relation is only constrained by the types of rules.

Remark 4 The reverse weak priority, i.e., non-catalytic rules having priority over catalytic rules, is useless for our purposes, since it is equivalent to removing all catalytic rules for which there are non-catalytic rules with the same symbol on the left-hand side of the rule. In that way, we just end up with an even restricted variant of P systems with only one catalyst, although it might still be interesting to characterize the family of multiset languages obtained by this restricted variant of P systems: obviously, the lower bound still are the semi-linear languages, as for generating those we only need catalytic rules with one catalyst; on the other hand, the result stated in Proposition 4 cannot be obtained anymore with the methods used in [8].

3.1 Computational completeness with weak priority

We now are going to show that catalytic P systems with one catalyst only and with weak priority of catalytic rules are computationally complete.

Theorem 1 *Catalytic P systems with only one catalyst and with weak priority of catalytic rules over the non-cooperative rules when working in the derivation mode max are computationally complete.*

Proof Given an arbitrary register machine $M = (m, B, l_0, l_h, P)$, we will construct a corresponding catalytic P system with one membrane and one catalyst $\Pi = (O, \{c\}, w, R)$ simulating M . Without loss of generality, we may assume that, depending on its use as an accepting or generating or computing device, the register machine M , as stated in Proposition 1, Proposition 2, and Proposition 3, fulfills the condition that on the output registers, we never apply any *SUB*-instruction. The following proof is given for the most general case of a register machine computing any partial recursive relation on vectors of natural numbers with l components as input and vectors of natural numbers with k components as output using precisely $l + 2 + k$ registers, where without loss of generality, we may assume that at the end of a successful computation, the first $l + 2$ registers are empty, and, moreover, on the output registers, i.e., the last k registers, no *SUB*-instruction is ever used. In fact, the proof works for any number n of decrementable registers, no matter how many of them are the l input registers and the working registers, respectively.

The main idea behind our construction is that all the symbols except the catalyst c and the output symbols (representing the contents of the output registers) go through a cycle of length $n + 2$ where n is the number of decrementable registers of the simulated register machine. When the symbols are traversing the r -th section of the n sections, they “know” that they are to probably simulate a *SUB*-instruction on register r of the register machine M .

The alphabet O of symbols includes register symbols (a_r, i) for every decrementable register r of the register machine and only the register symbol a_r for each of the k output registers r , $m - k + 1 \leq r \leq m$.

Now let B_{ADD} denote the set of labels of *ADD*-instructions $p : (ADD(r), q, s)$ of arbitrary registers r and $B_{SUB(r)}$ denote the set of labels of all *SUB*-instructions $p : (SUB(r), q, s)$ of decrementable registers r . For every *ADD*-instruction $p : (ADD(r), q, s)$, i.e., $p \in B_{ADD(r)}$, of the register machine we take the state symbols (p, i) , $1 \leq i \leq n + 2$, into O . For every *SUB*-instruction $p : (SUB(r), q, s)$ of the register machine, i.e., $p \in B_{SUB(r)}$, we need the state symbols (p, i) for $1 \leq i \leq r + 1$ as well as $(p, j)^-$ and $(p, j)^0$ for $r + 2 \leq j \leq n + 2$.

Moreover, we use a decrement witness symbol e as well as the catalyst c and the trap symbol $\#$.

Observing that $n = m - k$, in total, we get the following set of objects:

$$O = \{a_r \mid n + 1 \leq r \leq m\} \cup \{(a_r, i) \mid 1 \leq r \leq n, 1 \leq i \leq n + 2\} \cup \{(p, i) \mid p \in B_{ADD}, 1 \leq i \leq n + 2\} \cup \{(p, i) \mid p \in B_{SUB(r)}, 1 \leq i \leq r + 1, 1 \leq r \leq n\} \cup \{(p, i)^-, (p, i)^0 \mid p \in B_{SUB(r)}, r + 2 \leq i \leq n + 2, 1 \leq r \leq n\} \cup \{c, e, \#\}.$$

The starting configuration of Π is

$$w = c(l_0, 1)\alpha_0,$$

where l_0 is the starting label of the machine and α_0 is the multiset encoding the initial values of the registers.

All register symbols $a_r, 1 \leq r \leq n$, representing the contents of decremtable registers, are equipped with the rules evolving them throughout the whole cycle:

$$(a_r, i) \rightarrow (a_r, i + 1), 1 \leq r \leq n + 1; \quad (a_r, n + 2) \rightarrow (a_r, 1). \tag{1}$$

The construction also includes the trap rule $\# \rightarrow \#$: once the trap symbol $\#$ is introduced, it will always keep the system busy and prevent it from halting and thus from producing a result.

For simulating *ADD*-instructions, we need the following rules:

Increment p : (ADD(r), q, s):

The (variants of the) symbol p cycles together with all the other symbols, always involving the catalyst:

$$c(p, i) \rightarrow c(p, i + 1), \quad 1 \leq i \leq n + 1. \tag{2}$$

At the end of the cycle, the register r is incremented and the non-deterministic jump to q or s occurs: for r being a decremtable register, we take

$$c(p, n + 2) \rightarrow c(q, 1)(a_r, 1), \quad c(p, n + 2) \rightarrow c(s, 1)(a_r, 1), \tag{3}$$

whereas for r being a register never to be decremented, we take

$$c(p, n + 2) \rightarrow c(q, 1)a_r, \quad c(p, n + 2) \rightarrow c(s, 1)a_r. \tag{4}$$

The output symbols need not undergo the cycle, in fact, they must not do that because otherwise the computation would never stop. When the computation of the register machine halts, only output symbols (of course, besides the catalyst c) will be present, as we have assumed that at the end of a computation all decremtable registers will be empty, i.e., no cycling symbols will be present any more in the P system. Finally, we have to mention that if q or s is the final label l_h ,

then we take λ instead, which means that also the P system will halt, because, as already explained above, the only symbols left in the configuration (besides the catalyst c) will be output symbols, for which no rules exist.

The state symbol is not allowed to evolve without the catalyst:

$$(p, i) \rightarrow \#, \quad 1 \leq i \leq n + 2. \tag{5}$$

Hence, in that way, it is guaranteed that the catalyst cannot be used in another way, i.e., affecting a symbol (a_r, i) as explained below during the simulation of a *SUB*-instruction on register r .

Decrement and zero-test p : (SUB(r), q, s):

The simulation of a *SUB*-instruction is carried out in three steps of the cycle, i.e., in steps $r, r + 1$, and $r + 2$.

Before reaching the simulation phase for register r , i.e., step r of the cycle, the state symbol goes through the cycle, necessarily involving the catalyst:

$$c(p, i) \rightarrow c(p, i + 1) \quad > \quad (p, i) \rightarrow \#, \quad 1 \leq i < r. \tag{6}$$

By definition, in the P system Π we construct, catalytic rules have priority over the non-cooperative rules, i.e., the catalytic rule $c(p, i) \rightarrow c(p, i + 1)$ has priority over the non-catalytic rule $(p, i) \rightarrow \#$; we indicate this general priority relation for the object (p, i) by the sign $>$ (and we use $<$ for the reverse relation) in order to make the situation even clearer.

In the first step of the simulation phase for register r , i.e., in step r , the state symbol releases the catalyst to try to perform the decrement and to produce a witness symbol e if register r is not empty:

$$(p, r) \rightarrow (p, r + 1), \quad c(a_r, r) \rightarrow ce. \tag{7}$$

Note that due to the counters identifying the position of the register symbols in the cycle, it is guaranteed that the catalytic rule transforming (a_r, r) picks the correct register symbol. Furthermore, due to the priority of the catalytic rules, one of the register symbols (a_r, r) *must* be transformed by the catalytic rule if present, instead of continuing along its cycle.

In the second step of simulation phase r , i.e., in step $r + 1$, the detection of the possible decrement happens. The outcome is stored in the state symbol:

$$ce \rightarrow c \quad > \quad e \rightarrow \#, \tag{8}$$

$$(p, r + 1) \rightarrow (p, r + 2)^- \quad < \quad c(p, r + 1) \rightarrow c(p, r + 2)^0.$$

If in the first step of the simulation phase, the catalyst did manage to decrement the register, it produced e . Thus, in the second step, i.e., in step $r + 1$, the catalyst has three choices:

1. the catalyst c correctly erases e , because otherwise this symbol will trap the computation, and to the program

- symbol $(p, r + 1)$ the rule $(p, r + 1) \rightarrow (p, r + 2)^-$ must be applied as the catalyst is not available for using it with the catalytic rule $c(p, r + 1) \rightarrow c(p, r + 2)^0$; all register symbols evolve in the usual way;
2. the catalyst c takes the program symbol $(p, r + 1)$ using the rule $c(p, r + 1) \rightarrow c(p, r + 2)^0$, thus forcing the object e to be trapped by the rule $e \rightarrow \#$, and all register symbols evolve in the usual way; this variant cannot lead to a halting computation due to the introduction of the trap symbol $\#$;
 3. the catalyst c takes a register object $(a_{r+1}, r + 1)$, thus leaving the object e to be trapped by the rule $e \rightarrow \#$, the program symbol $(p, r + 1)$ evolves with the rule $(p, r + 1) \rightarrow (p, r + 2)^-$, and all other register objects evolve in the usual way; this variant again cannot lead to a halting computation due to the introduction of the trap symbol $\#$.

On the other hand, if register r is empty, no object e is generated, and the catalyst c has only two choices:

1. the catalyst c takes the program symbol $(p, r + 1)$ using the rule $c(p, r + 1) \rightarrow c(p, r + 2)^0$, and all register symbols evolve in the usual way;
2. the catalyst c takes a register object $(a_{r+1}, r + 1)$ thereby generating e , the program symbol $(p, r + 1)$ evolves with the rule $(p, r + 1) \rightarrow (p, r + 2)^-$, and all other register objects evolve in the usual way; this variant leads to the situation that e will be trapped in step $r + 2$, as otherwise the program symbol will be trapped, hence, this variant in any case cannot lead to a halting computation due to the introduction of the trap symbol $\#$. We mention that in case no register object $(a_{r+1}, r + 1)$ is present we have to apply case 1 and thus have a correct computation step.

Observe that in this case of register r being empty, both rules $(p, r + 1) \rightarrow (p, r + 2)^-$ and $c(p, r + 1) \rightarrow c(p, r + 2)^0$ would be applicable, but due to the priority of the catalytic rules, the second rule must be preferred, thus producing $(p, r + 2)^0$. Therefore, the superscript of the state symbol correctly reflects the outcome of the *SUB*-instruction: it is “-” if the decrement succeeded, and “0” if it did not.

After the simulation of the *SUB*-instruction $p : (SUB(r), q, s)$ on register r , the state symbols evolve to the end of the cycle and produce the corresponding next state symbols:

$$\begin{aligned}
 c(p, i)^- &\rightarrow c(p, i + 1)^-, & r + 2 \leq i \leq n + 1, & & c(p, n + 2)^- &\rightarrow c(q, 1), \\
 c(p, i)^0 &\rightarrow c(p, i + 1)^0, & r + 2 \leq i \leq n + 1, & & c(p, n + 2)^0 &\rightarrow c(s, 1), \\
 (p, i)^- &\rightarrow \#, & (p, i)^0 &\rightarrow \#, & r + 2 \leq i \leq n + 2. &
 \end{aligned}
 \tag{9}$$

The additional two steps $n + 1$ and $n + 2$ are needed to correctly finish the decrement and zero-test cases even for $r = n$.

Finally, we again mention that if q or s is the final label l_h , then we take λ instead, which means that not only the register machine but also the *P* system halts, because, as already explained above, the only symbols — except the single catalyst c — left in the configuration will be output symbols, for which no rules exist. \square

The proof elaborated above is an improved version of the proof given in [3]. In this new version of the proof the simulation of the decrement case on register r still takes two steps, but as the second step of the simulation can overlap with the first step of simulating the decrement case on register $r + 1$ in the next cycle, we only need $n + 2$ steps (the first +1 comes from the second step for register n in the decrement case, the second +1 comes from the third step for register n in the zero-test case) instead of $2n$ steps as needed in the proof elaborated in [3].

Moreover, we want to emphasize that the simulation is also a specific kind of deterministic: the only non-deterministic choice happens between a rule producing a trap symbol $\#$ and another one which does not introduce $\#$. This means that the appearance of the trap symbol may immediately abort the computation, which is the concept used for toxic *P* systems as introduced in [1]. Using the trap symbol $\#$ as such a toxic object, the only successful computations are those which simulate register machines in a quasi-deterministic way with a look-ahead of one, i.e., considering all possible configurations computable from a given one, there is at most one successful continuation of the computation.

It remains a challenging question whether the length of the cycle now being $n + 2$ can still be reduced; in fact, as we will immediately show below, the length of the cycle can even be reduced to n in case we have at least three decrementable registers and to $n + 1$ in case we have at least two decrementable registers. As for n decrementable registers we only need a cycle of length n and in the version of the proof elaborated below we do not need extra steps at the beginning or the end, this proof technique itself does not seem to allow for further improvements.

Theorem 2 *For any register machine with at least three decrementable registers, we can construct a catalytic P system with only one catalyst and with weak priority of catalytic rules over the non-cooperative rules and working in the derivation mode max which can simulate every step of the register machine in n steps where n is the number of decrementable registers.*

Proof In the proof of Theorem 1, the last two steps in the cycle of length $n + 2$ were only needed to capture the second and third step in the simulation of *SUB*-instructions of the

n -th register, and to capture the third step in the simulation of *SUB*-instructions of register $n - 1$.

The new idea now is to shift the second simulation step in the case of register $n - 1$ to the first step of the next cycle and to shift the first and the second simulation step in the case of register n to the first and second step of the next cycle. Yet, in this case, we have to guarantee that after a *SUB*-instruction on register $n - 1$ or n the next instruction to be simulated is not a *SUB*-instruction on register 1 or 2, respectively.

One solution for this problem is to use a similar trick as already elaborated in Remark 1: we not only do not start with a *SUB*-instruction, but we also change the register machine program in such a way that after a *SUB*-instruction on register $n - 1$ or n two intermediate instructions are introduced, i.e., as in Remark 1, we use an *ADD*-instruction on register 1 immediately followed by a *SUB*-instruction on register 1, whose simulation will end at most in step n , as we have assumed $n \geq 3$.

We then may simulate the resulting register machine fulfilling these additional constraints $M = (m, B, l_0, l_h, P)$ by a corresponding catalytic P system with one membrane and one catalyst $\Pi = (O, \{c\}, w, R)$. Without loss of generality, we may again assume that, depending on its use as an accepting or generating or computing device, the register machine M , as stated in Proposition 1, Proposition 2, and Proposition 3, fulfills the condition that on the output registers, we never apply any *SUB*-instruction. As in the proof of Theorem 2, also here we take the most general case of a register machine computing a partial recursive function on vectors of natural numbers with l components as input and vectors of natural numbers with k components as output using n decremtable registers, where without loss of generality, we may assume that at the end of a successful computation the first n registers are empty, and, moreover, on the output registers, i.e., the last k registers, no *SUB*-instruction is ever used.

The alphabet O of symbols now reduces to the following set, as the length of the cycle now is only n :

$$\begin{aligned}
 O = & \{a_r \mid n + 1 \leq r \leq m\} \\
 & \cup \{(a_r, i) \mid 1 \leq r \leq n, 1 \leq i \leq n\} \\
 & \cup \{(p, i) \mid p \in B_{ADD}, 1 \leq i \leq n\} \\
 & \cup \{(p, i) \mid p \in B_{SUB(r)}, 1 \leq i \leq r + 1, 1 \leq r \leq n\} \\
 & \cup \{(p, i)^-, (p, i)^0 \mid p \in B_{SUB(r)}, r + 2 \leq i \leq n, 1 \leq r \leq n - 2\} \\
 & \cup \{c, e, \#\}.
 \end{aligned}$$

The construction includes the trap rule $\# \rightarrow \#$ which will always keep the system busy and prevent it from halting and thus from producing a result as soon as the trap symbol $\#$ has been introduced.

The sets of rules introduced by Eqs. 1, 2, 3, 4, 5, 6, 7, 8, and 9 read as follows:

$$(a_r, i) \rightarrow (a_r, i + 1), 1 \leq r < n; \quad (a_r, n) \rightarrow (a_r, 1). \quad (10)$$

For simulating *ADD*-instructions, we need the following rules:

Increment p : (ADD(r), q, s):

$$c(p, i) \rightarrow c(p, i + 1), \quad 1 \leq i < n. \quad (11)$$

If r is a decremtable register:

$$c(p, n) \rightarrow c(q, 1)(a_r, 1), \quad c(p, n) \rightarrow c(s, 1)(a_r, 1), \quad (12)$$

If r is an output register:

$$c(p, n) \rightarrow c(q, 1)a_r, \quad c(p, n) \rightarrow c(s, 1)a_r \quad (13)$$

Enforcing the use of the catalyst:

$$(p, i) \rightarrow \#, \quad 1 \leq i \leq n. \quad (14)$$

For simulating *SUB*-instructions, we need the following rules:

Decrement and zero-test p : (SUB(r), q, s):

$$c(p, i) \rightarrow c(p, i + 1) \quad > \quad (p, i) \rightarrow \#, \quad 1 \leq i < r. \quad (15)$$

$$(p, r) \rightarrow (p, r + 1), \quad c(a_r, r) \rightarrow ce. \quad (16)$$

If $r < n - 1$:

$$\begin{aligned}
 ce \rightarrow c \quad > \quad e \rightarrow \#, \\
 (p, r + 1) \rightarrow (p, r + 2)^- \quad < \quad c(p, r + 1) \rightarrow c(p, r + 2)^0.
 \end{aligned} \quad (17)$$

$$\begin{aligned}
 c(p, i)^- \rightarrow c(p, i + 1)^-, \quad r + 2 \leq i < n, \quad c(p, n)^- \rightarrow c(q, 1), \\
 c(p, i)^0 \rightarrow c(p, i + 1)^0, \quad r + 2 \leq i < n, \quad c(p, n)^0 \rightarrow c(s, 1), \\
 (p, i)^- \rightarrow \#, \quad (p, i)^0 \rightarrow \#, \quad r + 2 \leq i \leq n.
 \end{aligned} \quad (18)$$

If $r = n - 1$:

$$\begin{aligned}
 ce \rightarrow c \quad > \quad e \rightarrow \#, \\
 (p, n) \rightarrow (q, 1) \quad < \quad c(p, n) \rightarrow c(s, 1).
 \end{aligned} \quad (19)$$

If $r = n$:

$$\begin{aligned}
 ce \rightarrow c \quad > \quad e \rightarrow \#, \\
 (p, n + 1) \rightarrow (q, 2) \quad < \quad c(p, n + 1) \rightarrow c(s, 2).
 \end{aligned} \quad (20)$$

As we have guaranteed that in the case of $r = n - 1$ or $r = n$, the next instruction to be simulated is an *ADD*-instruction, starting with the simulation of the next instruction later in the first or second step of the next cycle makes no harm. \square

If we only have two decrementable registers, then we need one additional step at the end; we leave the details of the proof, following the constructions elaborated in the preceding two proofs, to the interested reader:

Theorem 3 *For any register machine with at least two decrementable registers, we can construct a catalytic P system with only one catalyst and with weak priority of catalytic rules over the non-cooperative rules and working in the derivation mode max which can simulate every step of the register machine in $n + 1$ steps where n is the number of decrementable registers.*

As the number of decrementable registers in generating register machines needed for generating any recursively enumerable set of (vectors of) natural numbers is only two, from Theorem 2, we obtain the following result:

Corollary 1 *For any generating register machine with two decrementable registers, we can construct a catalytic P system with only one catalyst and with weak priority of catalytic rules over the non-cooperative rules which can simulate every step of the register machine in 3 steps, and therefore such catalytic P systems with only one catalyst and with weak priority of catalytic rules over the non-cooperative rules can generate any recursively enumerable set of (vectors of) natural numbers.*

4 Catalytic P systems with only one catalyst working in the derivation mode maxobjects

In this section, we now study catalytic P systems with only one catalyst which work in the derivation mode maxobjects and show that computational completeness can be obtained with only one catalyst and no further ingredients.

Theorem 4 *For any register machine with at least three decrementable registers, we can construct a catalytic P system with only one catalyst and working in the derivation mode maxobjects which can simulate every step of the register machine in n steps where n is the number of decrementable registers.*

Proof We take over the proof of Theorem 2. The priority of catalytic rules then is somehow regained by the fact that a catalytic rule $ca \rightarrow cv$ affects two objects, whereas the corresponding non-catalytic rule $a \rightarrow v$ only affects one object.

Again we use the trick as elaborated in Remark 1 and already used in the proof of Theorem 2 for getting a specific variant of register machines: without loss of generality, we

not only do not start with a *SUB*-instruction, but we also change the register machine program in such a way that after a *SUB*-instruction on register $n - 1$ or n two intermediate instructions are introduced, i.e., as in Remark 1, we use an *ADD*-instruction on register 1 immediately followed by a *SUB*-instruction on register 1.

We then may simulate the resulting register machine fulfilling these additional constraints $M = (m, B, l_0, l_h, P)$ by a corresponding catalytic P system with one membrane and one catalyst $\Pi = (O, \{c\}, w, R)$. Without loss of generality, we may again assume that, depending on its use as an accepting or generating or computing device, the register machine M , as stated in Proposition 1, Proposition 2, and Proposition 3, fulfills the condition that on the output registers we never apply any *SUB*-instruction. As in the proof of Theorem 2, also here we take the most general case of a register machine computing a partial recursive function on vectors of natural numbers with l components as input and vectors of natural numbers with k components as output using n decrementable registers, where without loss of generality, we may assume that at the end of a successful computation the first n registers are empty, and, moreover, on the output registers, i.e., the last k registers, no *SUB*-instruction is ever used.

The alphabet O of symbols now reduces to the following set:

$$\begin{aligned} O = & \{a_r \mid n + 1 \leq r \leq m\} \\ & \cup \{(a_r, i) \mid 1 \leq r \leq n, 1 \leq i \leq n\} \\ & \cup \{(p, i) \mid p \in B_{ADD}, 1 \leq i \leq n\} \\ & \cup \{(p, i) \mid p \in B_{SUB(r)}, 1 \leq i \leq r + 1, 1 \leq r \leq n\} \\ & \cup \{(p, i)^-, (p, i)^0 \mid p \in B_{SUB(r)}, r + 2 \leq i \leq n, 1 \leq r \leq n - 2\} \\ & \cup \{c, e, \#\}. \end{aligned}$$

The construction still includes the trap rule $\# \rightarrow \#$ which will always keep the system busy and prevent it from halting and thus from producing a result as soon as the trap symbol $\#$ has been introduced.

The sets of rules introduced by Eqs. 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, and 20 read as follows, yet now omitting all trap rules except $e \rightarrow \#$:

$$(a_r, i) \rightarrow (a_r, i + 1), 1 \leq r < n; \quad (a_r, n) \rightarrow (a_r, 1). \quad (21)$$

For simulating *ADD*-instructions, we need the following rules:

$$\begin{aligned} & \text{Increment } p : (ADD(r), q, s): \\ & c(p, i) \rightarrow c(p, i + 1), \quad 1 \leq i < n. \end{aligned} \quad (22)$$

If r is a decrementable register:

$$c(p, n) \rightarrow c(q, 1)(a_r, 1), \quad c(p, n) \rightarrow c(s, 1)(a_r, 1), \quad (23)$$

If r is an output register:

$$c(p, n) \rightarrow c(q, 1)a_r, \quad c(p, n) \rightarrow c(s, 1)a_r \tag{24}$$

The catalyst has to be used with the program symbol which otherwise would stay idle when the catalyst is used with a register symbol, and the multiset of rules applied in this way would use one symbol less and thus violate the condition of using the maximal number of objects, hence, now we do not need the trap rules $(p, i) \rightarrow \#, 1 \leq i \leq n$.

For simulating *SUB*-instructions, we need the following rules:

Decrement and zero-test $p : (SUB(r), q, s)$:

$$c(p, i) \rightarrow c(p, i + 1), \quad 1 \leq i < r. \tag{25}$$

For $1 \leq i < r$, we again do not need trap rules $(p, i) \rightarrow \#$, as otherwise the program symbol (p, i) would stay idle and thus the condition of using the maximal number of objects would be violated.

$$(p, r) \rightarrow (p, r + 1), \quad c(a_r, r) \rightarrow ce. \tag{26}$$

In case that register r is empty, i.e., there is no object (a_r, r) , then the catalyst will stay idle as in this step there is no other object with which it could react.

If $r < n - 1$:

$$ce \rightarrow c, \quad e \rightarrow \#, \quad (p, r + 1) \rightarrow (p, r + 2)^-; \tag{27}$$

$$c(p, r + 1) \rightarrow c(p, r + 2)^0.$$

If in the first step of the simulation phase, the catalyst did manage to decrement the register, it produced e . Thus, in the second simulation step, the catalyst has three choices:

1. The catalyst c correctly erases e , and to the program symbol $(p, r + 1)$ the rule $(p, r + 1) \rightarrow (p, r + 2)^-$ must be applied due to the derivation mode *maxobjects*; all register symbols evolve in the usual way;
2. The catalyst c takes the program symbol $(p, r + 1)$ using the rule $c(p, r + 1) \rightarrow c(p, r + 2)^0$, thus forcing the object e to be trapped by the rule $e \rightarrow \#$, and all register symbols evolve in the usual way;
3. The catalyst c takes a register object $(a_{r+1}, r + 1)$, thus leaving the object e to be trapped by the rule $e \rightarrow \#$, the program symbol $(p, r + 1)$ evolves with the rule $(p, r + 1) \rightarrow (p, r + 2)^-$, and all other register objects evolve in the usual way.

In fact, only variant 1 now fulfills the condition given by the derivation mode *maxobjects* and therefore is the only possible continuation of the computation if register r is not empty.

On the other hand, if register r is empty, no object e is generated, and the catalyst c has only two choices:

1. The catalyst c takes the program symbol $(p, r + 1)$ using the rule $c(p, r + 1) \rightarrow c(p, r + 2)^0$, and all register symbols evolve in the usual way;
2. The catalyst c takes a register object $(a_{r+1}, r + 1)$ thereby generating e , the program symbol $(p, r + 1)$ evolves with the rule $(p, r + 1) \rightarrow (p, r + 2)^-$, and all other register objects evolve in the usual way; this variant leads to the situation that e will be trapped in step $r + 2$, as otherwise the program symbol stays idle, thus violating the condition of the derivation mode *maxobjects*. Hence, this variant in any case cannot lead to a halting computation due to the introduction of the trap symbol $\#$. We mention that in case no register object $(a_{r+1}, r + 1)$ is present we have to apply case 1 and thus have a correct computation step.

Both variants fulfill the condition for the derivation mode *maxobjects*, but only variant 1 is not introducing the trap symbol $\#$ and therefore is the only reasonable continuation of the computation if register r is empty.

$$c(p, i)^- \rightarrow c(p, i + 1)^-, \quad r + 2 \leq i < n, \quad c(p, n)^- \rightarrow c(q, 1), \tag{28}$$

$$c(p, i)^0 \rightarrow c(p, i + 1)^0, \quad r + 2 \leq i < n, \quad c(p, n)^0 \rightarrow c(s, 1).$$

Again the catalyst has to be used with the program symbol which otherwise would stay idle when the catalyst is used with a register symbol, and the multiset of rules applied in this way would use one symbol less and thus violate the condition of using the maximal number of objects.

If $r = n - 1$:

$$ce \rightarrow c, \quad e \rightarrow \#, \tag{29}$$

$$(p, n) \rightarrow (q, 1), \quad c(p, n) \rightarrow c(s, 1).$$

We observe that in this case, during the first step of the next cycle, we have to guarantee that in the zero-test case, the catalyst must be used with the program symbol, hence, we will simulate an *ADD*-instruction on register 1, as the introduction of the symbol e in the wrong variant of the zero-test case must lead to introducing the trap symbol and not allowing e to be erased by the catalytic rule $ce \rightarrow c$.

If $r = n$:

$$ce \rightarrow c, \quad e \rightarrow \#, \tag{30}$$

$$(p, n + 1) \rightarrow (q, 2), \quad c(p, n + 1) \rightarrow c(s, 2).$$

In this case, the second step of the simulation is already the first step of the next cycle, and the third step of the simulation is already the second step of the next cycle, which again means that in this case of $r = n$ the next instruction to be simulated is an *ADD*-instruction on register 1. The introduction of the symbol e in the second step of the simulation, i.e., in the first step of the next cycle, in the wrong variant of the zero-test case allows for trapping e as the program

symbol has to be used with the catalyst in the second step of the next cycle.

We finally observe that the proof construction given above is quite similar to the one elaborated in the proof of Theorem 2, but the only rule introducing the trap symbol now is the single rule $e \rightarrow \#$. \square

If we only have two decrementable registers, then we need one additional step at the end; again we leave the details of the proof, based on the constructions elaborated in the preceding proofs, to the interested reader:

Theorem 5 *For any register machine with at least two decrementable registers, we can construct a catalytic P system with only one catalyst and working in the derivation mode \max objects which can simulate every step of the register machine in $n + 1$ steps where n is the number of decrementable registers.*

As the number of decrementable registers in generating register machines needed for generating any recursively enumerable set of (vectors of) natural numbers is only two, from Theorem 5, we obtain the following result:

Corollary 2 *For any generating register machine with two decrementable registers, we can construct a catalytic P system with only one catalyst and working in the derivation mode \max objects which can simulate every step of the register machine in 3 steps, and therefore such catalytic P systems with only one catalyst and working in the derivation mode \max objects can generate any recursively enumerable set of (vectors of) natural numbers.*

As for accepting register machines, in addition to the two working registers, we have at least one input register, we immediately infer the following result from Theorem 5:

Corollary 3 *For any recursively enumerable set of d -vectors of natural numbers given by a register machine with $d + 2$ decrementable registers, we can construct an accepting catalytic P system with only one catalyst and working in the derivation mode \max objects which can simulate every step of the register machine in $d + 2$ steps, and therefore such catalytic P systems with only one catalyst and working in the derivation mode \max objects can accept any recursively enumerable set of (vectors of) natural numbers.*

The results obtained in this section are optimal with respect to the number of catalysts for catalytic P systems working in the derivation mode \max objects, as the results for P systems only using non-cooperative rules are the same for both the derivation mode \max objects and the derivation mode \max ; for example, in the generating case, P systems

only using non-cooperative rules can only generate semi-linear sets.

5 Conclusion

In this paper, we revisited a classical problem of computational complexity in membrane computing: can catalytic P systems working in the derivation mode \max with only one catalyst in the whole system already generate all recursively enumerable sets of multisets? This problem has been standing tall for many years, and nobody has yet managed to give a positive or a negative answer to this problem. In this paper, we come tantalizingly close to showing computational completeness: we give a construction that simulates an arbitrary register machine with a very weak ingredient — the weak priority of catalytic rules over non-catalytic rules. This ingredient confers very little additional power indeed, because the structure of the priority relation is very simple, as it is only constrained by the two types of rules.

We believe that a similar construction driving the symbols around in a loop but avoiding any additional ingredients altogether will not be sufficient for obtaining computational completeness, as we still conjecture that P systems working in the derivation mode \max with one catalyst and no additional control mechanisms cannot reach computational completeness. Finding an answer to the question of characterizing the computational power of P systems with one catalyst working in the derivation mode \max therefore still remains one of the biggest challenges in the theory of P systems, although the result established in our paper has made the gap between the computational power of P systems with one catalyst and computational completeness smaller again.

The second variant proved to be computationally complete in this paper comes even closer to the original question: when using the derivation mode \max objects instead of the derivation mode \max , we really obtain computational completeness with only one catalyst and no further ingredients.

The results obtained in this paper can also be extended to P systems dealing with strings, following the definitions and notions used in [8], thus showing computational completeness for computing with strings.

Acknowledgements The authors gratefully acknowledge the useful hints of the referees. Rudolf Freund acknowledges the TU Wien supporting the open access publishing of this paper.

Funding Open access funding provided by TU Wien (TUW). Artiom Alhazov acknowledges project 20.80009.5007.22 “Intelligent information systems for solving ill-structured problems, processing knowledge and big data” by the National Agency for Research and Development. Sergiu Ivanov is partially supported by the Paris region via the project DIM RFSI n°2018-03 “Modèles informatiques pour la reprogrammation cellulaire”.

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Alhazov, A., & Freund, R. (2014). P systems with toxic objects. In: Gheorghe, M., Rozenberg, G., Salomaa, A., Sosík, P., Zandron, C. (Eds.) *Membrane computing—15th International Conference, CMC 2014, Prague, Czech Republic, August 20–22, 2014, Revised Selected Papers, Lecture Notes in Computer Science*, vol. 8961, pp. 99–125. Springer. https://doi.org/10.1007/978-3-319-14370-5_7.
- Alhazov, A., Freund, R., & Ivanov, S. (2020). Catalytic P systems with weak priority of catalytic rules. In: Freund, R. (Ed.) *Proceedings ICMC 2020, September 14–18, 2020*, TU Wien, pp. 67–82.
- Alhazov, A., Freund, R., & Ivanov, S. (2020). Computational completeness of catalytic P systems with weak priority of catalytic rules over non-cooperative rules. In: Orellana-Martín, D., Păun, Gh., Riscos-Núñez, A., Pérez-Hurtado, I. (Eds.) *Proceedings 18th Brainstorming Week on Membrane Computing, Sevilla, February 4–7, 2020*. RGNC REPORT 1/2020, Research Group on Natural Computing, Universidad de Sevilla, pp. 21–32.
- Dassow, J., & Păun, Gh. (1989). *Regulated Rewriting in Formal Language Theory*. Springer. <https://www.springer.com/de/book/9783642749346>.
- Freund, R., Kari, L., Oswald, M., & Sosík, P. (2005). Computationally universal P systems without priorities: Two catalysts are sufficient. *Theoretical Computer Science*, 330(2), 251–266. <https://doi.org/10.1016/j.tcs.2004.06.029>.
- Freund, R., Laporati, A., Mauri, G., Porreca, A. E., Verlan, S., & Zandron, C. (2014). Flattening in (tissue) P systems. In: Alhazov, A., Cojocaru, S., Gheorghe, M., Rogozhin, Yu., Rozenberg, G., Salomaa, A. (eds.) *Membrane Computing, Lecture Notes in Computer Science*, vol. 8340, pp. 173–188. Springer. https://doi.org/10.1007/978-3-642-54239-8_13.
- Freund, R., Oswald, M., & Păun, Gh. (2015). Catalytic and purely catalytic P systems and P automata: Control mechanisms for obtaining computational completeness. *Fundamenta Informaticae*, 136(1–2), 59–84. <https://doi.org/10.3233/FI-2015-1144>.
- Freund, R., & Sosík, P. (2015). On the power of catalytic P systems with one catalyst. In: Rozenberg, G., Salomaa, A., Sempere, J. M., Zandron, C. (eds.) *Membrane Computing—16th International Conference, CMC 2015, Valencia, Spain, August 17–21, 2015, Revised Selected Papers, Lecture Notes in Computer Science*, vol. 9504, pp. 137–152. Springer. https://doi.org/10.1007/978-3-319-28475-0_10.
- Minsky, M. L. (1967). *Computation: Finite and infinite machines*. Englewood Cliffs, NJ: Prentice Hall.
- Păun, Gh. (2000). Computing with membranes. *Journal of Computer and System Sciences*, 61(1), 108–143. <https://doi.org/10.1006/jcss.1999.1693>.
- Păun, Gh. (2002). *Membrane Computing: An introduction*. Springer. <https://doi.org/10.1007/978-3-642-56196-2>.
- Păun, Gh., Rozenberg, G., & Salomaa, A. (Eds.). (2010). *The Oxford Handbook of Membrane Computing*. Oxford University Press.
- Rozenberg, G., & Salomaa, A. (Eds.) (1997). *Handbook of Formal Languages*. Springer. <https://doi.org/10.1007/978-3-642-59136-5>.
- The P Systems Website. <http://ppage.psysteams.eu/>.

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