P Systems with Limited Capacity

Artiom Alhazov[1], Rudolf Freund[2], and Sergiu Ivanov[3]

[1] Vladimir Andrunachievici Institute of Mathematics and Computer Science
Academiei 5, Chișinău, MD-2028, Moldova
artiom@math.md

[2] Faculty of Informatics, TU Wien
Favoritenstraße 9–11, 1040 Wien, Austria
rudi@emcc.at

[3] IBISC, Univ Évry, Paris-Saclay University
23, boulevard de France 91034 Évry, France
sergiu.ivanov@ibisc.univ-evry.fr

Summary. P systems are a model of compartmentalized multiset rewriting inspired by the structure and functioning of the living cell. In this paper, we focus on a variant in P systems in which membranes have limited capacity, i.e., the number of objects they may hold is statically bounded. This feature corresponds to an important physical property of cellular compartments. We propose several possible semantics of limited capacity and show that one of them allows real-time simulations of partially blind register machines, while the other one allows for obtaining computational completeness.

1 Introduction

Membrane systems were introduced in [9] as a multiset-rewriting model of computing inspired by the structure and the functioning of the living cell. Among the basic features of the original model are the hierarchical arrangement of the membranes and the parallel evolution of the objects contained in the membrane compartments. Usually a result is obtained if the computation halts, i.e., if no rule is applicable any more.

In this paper we consider an additional feature also inspired by biology, namely the limited capacity of cells to include objects – in total or of a specific kind. When the number of cells is not bounded as in P systems with active membranes, this biological feature of limited capacity can be kept for all cells below a given fixed bound. On the other hand, in the standard hierarchical model with a static number of cells, or, even if we allowed membrane dissolution, with a fixed upper bound for the number of cells, we can only limit the number of specific objects and have to allow an unbounded number of other objects when aiming at non-trivial theoretical results.

When the number of cells is not bounded because of using membrane creation and/or membrane division (together with membrane dissolution), the number of objects in one cell/membrane can even be restricted to one, still allowing for obtaining computational completeness, thereby counting the number of membranes/cells instead of the number of objects in an output membrane/cell; for example, see [1, 4, 3].

In this paper we now propose this new feature of limited capacity for (specific) objects to be contained in a cell or a membrane region and several semantics of how to treat the situation when the application of a (multiset of) rule(s) would violate this limiting condition. We will only investigate two special variants in more detail, both of them blocking or aborting computations which try to apply a multiset of rules leading to a violation of the limited capacity conditions. For the first variant we show that it allows for real-time simulations of partially blind register machines (PBRM), while the other variant allows for obtaining computational completeness.

The development of the fascinating area of membrane computing during the last two decades is documented in two textbooks, see [10] and [11]. For actual information see the P systems webpage [13] and the issues of the Bulletin of the International Membrane Computing Society and of the Journal of Membrane Computing.

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. For any $a \in V$ and any string w over A, w_a denotes the number of symbols a in w.

A *multiset* M with underlying set A is a pair (A, f) where $f : A \to \mathbb{N}$ is a mapping. For a multiset $M = (A, f)$, its *support* is defined as $supp(M) = \{x \in A \mid f(x) > 0\}$. A multiset is called *empty* or *finite* if its support is the empty set or a finite set, respectively. If $M = (A, f)$ is a finite multiset over A and $supp(M) = \{a_1, \ldots, a_k\}$, then it can also be represented by the string $a_1^{f(a_1)} \ldots a_k^{f(a_k)}$ over the alphabet $\{a_1, \ldots, a_k\}$, and, moreover, all permutations of this string precisely identify the same multiset M. For any $a \in V$ and any multiset M over A, M_a denotes the number of symbols a in w.

For further notions and results in formal language theory we refer to textbooks like [5] and [12].

2.1 Register Machines

Register machines are well-known universal devices for computing (or generating or accepting) 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,*
- P *is the set of instructions bijectively labeled by elements of* B*,*
- $l_0 \in B$ *is the initial label, and*
- $l_h \in B$ *is the final label.*

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.*

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 a 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.

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 $p : (ADD(r), q)$.

For useful results on the computational power of register machines, we refer to [8]; for example, for proving computational completeness results for specific variants of P systems, usually the following formulations 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 are helpful:

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 nondeterministic 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 state and input alone and not on the register configuration. 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,*
- *P is the set of instructions bijectively labeled by elements of B,*
- *$l_0 \in B$ is the initial label, and*

- $l_h \in B$ *is the final label.*

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* l_2, *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.

2.3 P Systems

The standard model of hierarchical P systems can be defined as follows, for example, see [11] for several variants:

Definition 3. *A (hierarchical)* P system *of degree* $m \geq 1$ *is a construct*

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

where

- O *is the alphabet of objects;*
- μ *is a membrane structure of degree* m *with membranes labeled in a one-to-one manner with the natural numbers* $1, \ldots, m$;
- $w_1, \ldots, w_m \in O^*$ *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* $1, 2, \ldots, m$ *of* μ; *these evolution rules are of the forms* $u \to v$ *where and* u *is a multiset over* O *and* v *is a string from* $((O \setminus C) \times \{here, out, in\})^*$;
- $i_0 \in \{0, 1, \ldots, m\}$ *indicates the output region of* Π.

The membrane structure and the multisets in Π constitute a *configuration* of the P system; the *initial configuration* is given by the initial multisets $w_1, \ldots, w_m$. A transition between configurations is governed by the application of the evolution rules, which is done in the maximally parallel way, 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.

The application of a rule $u \to v$ 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* and *out*.

The P system continues with applying multisets of rules in the maximally parallel way until there remain no applicable rules in any region of Π. Then the system *halts*. We consider the number of objects from O contained in the output region i_0 at the moment when the system halts as the *result* of the underlying computation of Π. The set of results of all computations possible in Π is called the set of natural numbers *generated by* Π and it is denoted by $N(\Pi)$ 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 $Ps(\Pi)$. We refer to [11] for further details and examples.

A special variant of P systems uses so-called *catalysts*, which are objects which allow other objects to evolve, but never evolve themselves.

Definition 4. *A* catalytic P system *of degree* $m \geq 1$ *is a construct*

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

where $C \subseteq O$ *is the alphabet of catalysts; the* evolution rules *are of the forms* $ca \to cv$ *or* $a \to v$*, where* c *is a catalyst,* a *is an object from* $O \setminus C$*, and* v *is a string from* $((O \setminus C) \times \{here, out, in\})^*$*; the other ingredients are defined as for hierarchical P systems in Definition 3. A catalytic P system is called* purely catalytic *if all rules are catalytic ones.*

Since the beginning, the question how many catalysts are needed in catalytic and purely catalytic P systems for obtaining computational completeness has been a challenging theoretical question. The following result was shown in [7], 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 have at least the computational power of partially blind register machines.*

Example 1. In [7] 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 some (even extended version of a) PBRM and therefore by a P system with only one catalyst and 19 rules.

As already shown in [6], register machines with $n \geq 2$ decrementable registers can be simulated by catalytic P systems with n catalysts and by purely catalytic P systems with $n + 1$ catalysts.

3 Limited Capacity

In most of the variants of P systems considered in the literature the number of objects in a membrane region is not limited. In this paper, we propose a variant in which the number of objects a membrane may contain is bounded, with the bound already being given in the definition of the system.

In this paper we consider two variants of limiting the capacity – limiting the total capacity of objects in a cell and only limiting the capacity of specific objects in a cell, respectively.

Definition 5. *A P system with* per-membrane limited capacity *is the following construct:*

$$\Pi = (O, \mu, w_1, \ldots, w_n, k_1, \ldots, k_n, R_1, \ldots R_n, i_0),$$

where $k_i \in \mathbb{N} \cup \{\infty\}$ *is the* total capacity *of membrane* i, $1 \leq i \leq n$, *meaning that, for* $|v_i|$ *denoting the contents of memrane* i *in the current configuration, the condition* $|v_i| \leq k_i$ *must always be enforced, unless* $k_i = \infty$. *The other components of the tuple are as in Subsection 2.3.*

Definition 6. *A P system with* per-symbol limited capacity *is the following construct:*

$$\Pi = (O, \mu, w_1, \ldots, w_n, K_1, \ldots, K_n, R_1, \ldots R_n, i_0),$$

where $K_i : O \to \mathbb{N} \cup \{\infty\}$ *are functions defining the* per-symbol *capacity of membrane* i. *The condition* $w_a \leq K(a)$ *must therefore be enforced at all times, for any* $a \in O$, *unless* $K(a) = \infty$.

In this paper, we will focus on P systems with per-symbol limited capacity.

Remark 1. We immediately remark that the flattening technique which is folklore in the membrane computing community can be applied in the case of P systems with per-symbol limited capacity. Without loss of generality, we therefore in Section 4 will only consider 1-membrane systems, which can be written in a simplified version as follows with omitting the trivial membrane structure and taking the skin membrane 1 as the output membrane:

$$\Pi = (O, w, K, R) \text{ and } \Pi = (O, C, w, K, R) \text{ for catalytic P systems.}$$

3.1 Semantics of Limited Capacity

What should happen if a membrane is about to exceed its capacity (total or per-symbol)? Multiple kinds of behaviors may be considered, for example the following variants:

1. *Blocking behavior:* Prohibit the application of (multisets of) rules which would produce more objects. Attempting to apply such rules blocks the system, and yields no result.

2. *Destructive behavior:* Completely remove the offending membrane from the system, together with its contents.

3. *Dissolutive behavior*: Dissolve the offending membrane, dumping its contents into its parent membrane; in this case, (one of) the parent membrane(s) must allow for more objects, as otherwise the whole system would be dissolved.

4. *Separation behavior:* Divide the offending membrane separating its contents across the child membranes. Since every child membrane only receives a part of the contents of the parent, the capacity constraints may be satisfied.

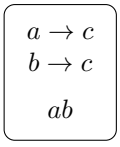
The separation behavior may be useful for P systems with active membranes, whereas the first three behaviors may also be applied for hierarchical P systems.

In this paper, we focus on the blocking behavior, see 1. Yet there are still at least two possible semantics for the blocking behavior itself under the maximally parallel derivation mode:

Semantics 1: Take all the applicable multisets of rules in the maximally parallel derivation mode, but discard all those multisets which would violate the constraints.

Semantics 2: Take all the applicable multisets of rules in the asynchronous derivation mode, discard the multisets which would violate the constraints, and then pick the non-extendable, i.e., maximal multisets out of these applicable multisets of rules.

To illustrate the difference between these two semantics, consider the following 1-membrane system with limited capacity:



It can formally be written as

$$\Pi_{ab} = (\{a, b\}, ab, K_{ab}, \{a \to c, b \to c\})$$

where $K_{ab}(c) = 1$ and $K_{ab}(a) = K_{ab}(b) = \infty$.

In the case of Semantics 1, no multisets of rules not violating the constraint of limiting the capacity of symbols c in the resulting configuration would be applicable, and the P system will block/abort this computation.

On the other hand, under Semantics 2, Π_{ab} would be allowed to apply *either* $a \to c$ *or* $b \to c$, but not both.

4 Computational Power

In this section we investigate the computational power of P systems with limited per-symbol capacity: when operating with Semantics 1, they at least can simulate partially blind register machines in real time; when operating with Semantics 2, they can simulate any register machines and therefore are computationally complete.

4.1 Semantics 1 Allows for Simulating a PBRM in Real Time

In this subsection, we will show that P systems with limited per-symbol capacity operating under Semantics 1 can simulate partially blind register machines (PBRM) in real time: an instruction of the register machine is simulated in one step of the P system. An additional cleanup procedure at the end of the computation takes 3 more steps. In comparison with the result stated in [7] showing that P systems with one catalyst can simulate partially blind register machines (without any further ingredients), we here obtain a real-time simulation, whereas the result there needs a cycle of $n+3$ for each step of the register machine, with n being the number of decrementable registers.

Theorem 1. *Catalytic P systems with one catalyst and per-symbol limited capacity operating with Semantics 1 can simulate partially blind register machines (PBRM) in real time, plus three additional cleanup steps at the end of the computation.*

Proof. Consider an arbitrary partially blind register machine

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

The following proof is given for the most general case of a partially blind 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 of decrementable registers, no matter how many of them are the first l input registers and the working registers, respectively. Moreover, we may assume that on the output registers, i.e., the last k registers, no *SUB*-instruction is ever used. On the other hand, the computation of the PBRM yields a result if and only if at the end of the computation all registers except the output registers are empty.

We now construct the P system

$$\Pi = (O, \{c\}, w_0, K, R)$$

with per-symbol limited capacity operating under Semantics 1 and simulating the PBRM M.

The set of objects of the construction includes register symbols a_r for representing the contents of register r, the catalyst c, and the state symbols $p \in B$. Moreover, we use decrement witness symbols λ_r for every decrementable register r, $1 \leq r \leq n$, as well as the catalyst c and the trap symbol $\#$ and, finally, the additional symbols $a_0, \lambda_0, l_{h'}$. As we will see later, a_0 can be interpreted as a register symbol for an additional decrementable register 0, which during the whole computation has the value 1, i.e., in every configuration we have exactly one copy of a_0, and it is only eliminated in the final cleanup procedure.

Now let $B_{SUB(r)}$ denote the set of labels of SUB-instruction $p : (SUB(r), q)$ of decrementable registers r, $B_{SUB} = \bigcup_{1 \leq r \leq n} B_{SUB(r)}$, and B_{ADD} denote the set of labels of ADD-instructions, i.e., $B = B_{ADD} \cup B_{SUB}$.

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

$$O = \{a_r \mid 0 \leq r \leq m\} \cup B \cup \{l_{h'}\} \cup D \cup \{c, \#\},$$
$$D = \{\lambda_r \mid 0 \leq r \leq n\}.$$

The capacity of the symbols in D is limited to 1, while all other symbols may appear in an unlimited number of copies:

$$K(\lambda_r) = 1, \quad 0 \leq r \leq n,$$
$$K(x) = \infty, \ \ x \in O \setminus D.$$

Moreover, let $D_\emptyset$ denote the multiset containing exactly one copy of each object in D and D_r the multiset containing exactly one copy of each object in D except λ_r.

Then the starting configuration of the P system is defined as

$$w_0 = c\, l_0\, D_\emptyset\, a_0\, \alpha_0,$$

where α_0 is the multiset encoding the initial values of the registers.

The set of rules now is going to be described in several parts below.

First, we want all symbols in D to disappear after one step:
$\lambda_r \to \lambda \in R$ for all $0 \leq r \leq n$.
We also include the traditional trap rule $\# \to \# \in R$.

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

To simulate the ADD instruction $p : (ADD(r), q, s)$ without letting the catalyst block the system or do unwanted decrements, the catalyst is forced to process the state symbol:

$$cp \to cqa_r D_\emptyset \qquad cp \to csa_r D_\emptyset \qquad p \to \#.$$

When the label of an ADD instruction is present in the configuration, the catalyst cannot act on any of the register symbols a_r, $0 \leq r \leq m$, because this would leave the state symbol p to be transformed to # due to the maximally parallel derivation mode. This evolution will not violate the capacity constraints, but introducing the trap symbol will prevent the system from ever halting. Therefore, the catalyst must be used in one of the two rules simulating the increment. Incidentally, these rules also replenish the supply of the symbols from D.

Decrement $p : (SUB(r), q)$ *(no zero test):*

Consider the configuration $c\,p\,D_\emptyset\,a_0\,\alpha$, where α is a string of register symbols describing the current contents of the registers. The following rules have to be applied in this configuration:

$$p \to qD_r \qquad ca_r \to c\lambda_r.$$

All the symbols from $D_\emptyset$ from the *current* configuration will disappear in the next configuration. The rule $p \to qD_r$ will reintroduce almost all of the symbols, except for the particular λ_r corresponding to the register to be decremented. This allows $ca_r \to c\lambda_r$ to be applied in the *current* step, because in the next configuration there is still room for λ_r. All catalytic rules involving a wrong $\lambda_{r'}$ (and therefore a wrong $a_{r'}$) cannot be applied, because they would introduce a second instance of $\lambda_{r'}$, thus blocking the system.

Therefore, the only possible evolution from the configuration $c\,p\,D_\emptyset\,a_0\,\alpha$ is to the configuration $c\,q\,D_\emptyset\,a_0\,\beta$ where $\beta = \alpha - a_r$. Note that if the expected register symbol a_r is not present in α, then there will be no non-extendable multiset of rules including the correct $ca_r \to c\lambda_p$, because then at least the rule $ca_0 \to c\lambda_0$ described below would become applicable, thus blocking (aborting) the computation without producing any result. This behavior corresponds to a crash in the PBRM when it tries to decrement a register which is already empty.

Final zero test, cleanup, and halting:

The simulation of the decrement instruction on register r only works correctly when there are still some register symbols a_r left. Indeed, as already mentioned above, in order to force the computation in the P system to abort if a decrement on an empty register would be tried, we at least would have the rule $ca_0 \to c\lambda_0$, but as long as the decrement symbol λ_0 is re-introduced by applying a rule $p \to qD_r$ simulating a decrement on register r, the computation in the P system will be forced to crash as two symbols λ_0 are not allowed in a configuration.

On the other hand, if finally, the PBRM has reached a configuration with all decrementable registers r, $1 \leq r \leq n$ being empty, we have to allow for a final zero test: in this case the rule $ca_0 \to c\lambda_0$ is welcome to be applied if we have reached the final (halting) label l_h:

$$l_h \to l_{h'} D_{\lambda_0} \qquad ca_0 \to c\lambda_0$$

The additional label $l_{h'}$ is used to check whether all decrementable registers are empty as required for a computation of the PBRM to be successful:

$$l_{h'} \to D_{\lambda_0} \qquad ca_r \to c\#\lambda_0, 1 \leq r \leq n$$

In this step, the catalyst is free to use one of the rules $ca_r \to c\#\lambda_0$ for any non-empty register r without violating the limiting condition for λ_0, hence, the trap symbol # is introduced if and only if any of the decrementable registers is not empty.

If all decrementable registers have been empty, in the final step, the system will just erase the symbols of D_{λ_0}, which will disappear and the system will halt with only symbols a_r for the output registers $n+1 \leq r \leq m$.

This final cleanup phase takes one step to erase a_0, one more step to test the presence of register symbols a_r, $1 \leq r \leq n$, and one final step to erase the last symbols of D_{λ_0}. Hence, in a successful computation this final phase takes three steps.

In the case of the simulation of a non-successful computation of the PBRM, there may be many more steps applying rules $ca_r \to c\#\lambda_0$, possibly already in the second step, but with the trap rule $\# \to \# \in R$ causing an infinite computation we need not take care about this situation in detail. □

Remark 2 (Trapping by limited capacity). Instead of having the rule $\# \to \#$ to implement the trap symbol as a guarantee for an infinite computation and thus for any computation introducing it to not be successful, we can limit its capacity to 1 and use rules of the form $u \to v\#\#$ instead of $u \to v\#$. Alternatively, we could limit the capacity of # to 0, meaning that even having to pick the rule $u \to v\#$ will already block the evolution. This means that, if all non-extendable multisets of rules contain a rule of the form $u \to v\#$, then we must discard all multisets, thereby blocking the evolution without producing any result. This blocking of computations reflects the concept of using toxic objects as introduced in [2].

4.2 Semantics 2 Allows for Computational Completeness

In this subsection, we show that (purely catalytic) P systems with limited per-symbol capacity are computationally complete when operating with Semantics 2 without any additional ingredients.

Remark 3 (Simulating catalytic rules). We first observe that when operating with Semantics 2 we can limit the parallelism of a non-cooperative rule by producing a marker symbol whose capacity is limited to one. For example, consider the rule $p : a \to u\lambda_p$ together with the rule $\lambda_p \to \lambda$ and the limiting condition $K(\lambda_p) = 1$, i.e., the symbol λ_p may not appear in more than one copy. Then, in any multiset of rules allowed to be applied λ_p may appear in at most one copy. This effectively prohibits applying p more than once in any step.

Moreover, we can ensure that the rules compete for the marker symbol just as catalytic rules would compete for a catalyst. For example, consider two catalytic rules $ca \to cu$ and $cb \to cv$. These two rules cannot be applied at the same time, even if both a and b are present, because the catalyst is only present in a single copy. We can ensure the same mutual exclusion by having the symbol λ_c with the capacity limited to 1 ($K(\lambda_c) = 1$), and the rules $a \to u\lambda_c$ and $b \to v\lambda_c$.

Remark 4 (No catalysts needed). As elaborated in Remark 3, catalytic rules can be replaced by non-cooperative rules, i.e., P systems with per-symbol limited capacity operating with Semantics 2 do not need catalysts for simulating purely catalytic P systems.

All together, these observations imply the following results:

Theorem 2. *P systems with per-symbol limited capacity operating with Semantics 2 without catalysts can simulate purely catalytic P systems.*

Since purely catalytic P systems are computationally complete, for example see [6], we immediately derive the following corollary.

Corollary 1. *P systems with per-symbol limited capacity operating with Semantics 2 are computationally complete, even without using catalysts.*

Remark 5 (Trapping by limited capacity). When following the proofs as given in [6] for simulating register machines by [purely] catalytic P systems, often rules introducing the trap symbol # as well as the rule $\# \to \#$ are used to guarantee an infinite computation and thus any computation introducing it to not be successful. As already explained in Remark 2, we can avoid these rules by limiting the capacity of the trap symbol to 1 and use rules of the form $u \to v\#\#$ instead of $u \to v\#$, or alternatively, limit the capacity of # to 0, meaning that even having to pick the rule $u \to v\#$ will already block the computation.

5 Conclusion

In this paper, we have introduced the idea of bounding the number of symbols that may appear in the membranes of a P systems. This is a quite natural restriction to consider, given that actual biological membranes are of limited capacity, too. We defined limited total and per-symbol capacities, and defined two possible semantics for handling the overflow. We then showed that Semantics 1 allows non-cooperative P systems to simulate partially blind register machines in real time, with 3 additional cleanup steps at the end of the computation. We also showed that non-cooperative P systems operating under Semantics 2 of limited capacity directly simulate purely catalytic P systems (in real time), yet without needing catalysts, and therefore are computationally complete.

This paper only scratches the surface of the study of P systems with limited capacity. One immediate open problem is that of computational completeness of (catalytic, purely catalytic) P systems with limited capacity operating with Semantics 1 or else characterizing the computational power of these systems.

Furthermore, Section 3 gives three more different behaviors which P systems may adopt when their membranes overflow. In particular, the separation and the dissolutive behaviors may even better represent the phenomena one would expect to observe in overfull membranes in biological cells.

Acknowledgements

The ideas, concepts, and results described in this paper have mainly been developed in the inspiring atmosphere of the 18^{th} Brainstorming Week on Membrane Computing during the first week of February 2020 in Sevilla.

Sergiu Ivanov is partially supported by the Paris region via the project DIM RFSI n°2018-03 "Modèles informatiques pour la reprogrammation cellulaire".

References

1. Artiom Alhazov. P systems without multiplicities of symbol-objects. *Inf. Process. Lett.*, 100(3):124–129, 2006.
2. Artiom Alhazov and Rudolf Freund. P systems with toxic objects. In Marian Gheorghe, Grzegorz Rozenberg, Arto Salomaa, Petr Sosík, and Claudio Zandron, editors, *Membrane Computing – 15th International Conference, CMC 2014, Prague, Czech Republic, August 20–22, 2014, Revised Selected Papers*, volume 8961 of *Lecture Notes in Computer Science*, pages 99–125. Springer, 2014.
3. Artiom Alhazov, Rudolf Freund, and Sergiu Ivanov. Length P systems. *Fundam. Inform.*, 134(1-2):17–37, 2014.
4. Artiom Alhazov, Rudolf Freund, and Agustin Riscos-Núñez. Membrane division, restricted membrane creation and object complexity in P systems. *Int. J. Comput. Math.*, 83(7):529–547, 2006.
5. Jürgen Dassow and Gheorghe Păun. *Regulated Rewriting in Formal Language Theory*. Springer, 1989.
6. Rudolf Freund, Lila Kari, Marion Oswald, and Petr Sosík. Computationally universal P systems without priorities: two catalysts are sufficient. *Theoretical Computer Science*, 330(2):251–266, 2005.
7. Rudolf Freund and Petr Sosík. On the power of catalytic P systems with one catalyst. In Grzegorz Rozenberg, Arto Salomaa, José M. Sempere, and Claudio Zandron, editors, *Membrane Computing – 16th International Conference, CMC 2015, Valencia, Spain, August 17–21, 2015, Revised Selected Papers*, volume 9504 of *Lecture Notes in Computer Science*, pages 137–152. Springer, 2015.
8. Marvin L. Minsky. *Computation. Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, NJ, 1967.
9. Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.
10. Gheorghe Păun. *Membrane Computing: An Introduction*. Springer, 2002.

11. Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors. *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.
12. Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages*. Springer, 1997.
13. The P Systems Website. `http://ppage.psystems.eu/`.