Journal of Universal Computer Science, vol. 10, no. 5 (2004), 581-599
submitted: 1/4/04, accepted: 10/4/04, appeared: 28/5/04 © J.UCS

On P Systems with Promoters/Inhibitors

Mihai Ionescu
(Rovira i Virgili University, Spain
mi@fll.urv.es)

Dragoş Sburlan
(Ovidius University of Constanţa, România
dsburlan@univ-ovidius.ro)

Abstract: This article shows how the computational universality can be reached using P systems with object rewriting non-cooperative rules, promoters/inhibitors at the level of rules, and only one catalyst. Both generative and accepting cases are studied. The theoretical issues presented are illustrated by several examples.

Key Words: P Systems, Universality, Promoters, Inhibitors

Category: F.4.2, F.4.3

1 Introduction

P systems represent a class of distributed/parallel computing devices whose functioning is inspired from the behavior of molecules and living cells. There, chemical compounds are processed in a massive parallel manner inside a compartmental structure of membranes that control the substances exchanges between regions they delimit. The reactions that take place inside such a biological structure can be formally described by cooperative rules. One particular case is of catalytic rules which model the biological reactions that can take place only with the help of certain enzymatic proteins (which participate in reactions and remain unmodified after they occur). Another important type is of promoted/inhibited reactions that happen in the presence/absence of certain chemicals which are not directly implied in reactions.

In this abstract mathematical framework it is interesting to see which is the computational power when weak cooperation features are used. In this sense, as it was shown in [Freund et al. 2003], P systems with non-cooperative and catalytic rules with only two distinct catalysts are computational universal. Also, in [Bottoni et al. 2002] a model with non-cooperative rules, one catalyst and promoters at the level of rules were shown to be universal.

In this paper we explore the computational power of the systems with non-cooperative rules, catalytic rules with one catalyst and promoters/inhibitors. Both generative and accepting cases will be studied here.

Meanwhile, we introduce the regulated rewriting mechanism of regularly controlled context-free grammars as a tool for studying P systems.

2 Preliminaries

2.1 Regulated Rewriting

In any Chomsky grammar, at some given step in a derivation one can use for rewriting any applicable rule in any desired place of the sentential form. In order to restrict this non-determinism some regulating mechanisms, which can control the derivation process, were considered. Using such regulations we can arrive to computational universality even if we use context-free grammars as a core generative device. In literature there are many types of regulations which restrict the use of rules in a Chomsky grammar (see [Dassow and Păun 1989]). Here we will present only regularly controlled grammars with appearance checking and λ–rules.

A *regularly controlled context-free grammar with appearance checking* is a 6-tuple $G_{rC} = (N, T, P, S, R, F)$ where N,T,P, and S are specified as in a context-free grammar (nonterminal alphabet, terminal alphabet, set of rules, axiom, respectively), R is a regular language over P, and F is a subset of P.

For a rule $p = A \rightarrow w \in P$ and $x, y \in V_G^*$ we write $x \Longrightarrow_p^{ac} y$ if either

1. $x = x_1 A x_2$ and $y = x_1 w x_2$, or
2. $x = y$, A does not appear in x, and $p \in F$.

The language $L(G)$ generated by G with appearance checking consists of all words $w \in T^*$ such that there is a derivation

$$S \Longrightarrow_{p_1}^{ac} w_1 \Longrightarrow_{p_2}^{ac} w_2 \cdots \Longrightarrow_{p_n}^{ac} w_n = w$$

with $p_1 p_2 \cdots p_n \in R$.

$\mathcal{L}(rC_{ac})$ we denote the families of languages generated by regularly controlled grammars (without appearance checking), regularly controlled grammars with appearance checking, regularly controlled grammars without erasing rules (and without appearance checking), and regularly controlled grammars with appearance checking and without erasing rules, respectively.

By $\mathcal{L}(\lambda rC_{ac})$ we denote the families of languages generated by regularly controlled grammars with appearance checking and erasing rules. The following result stands:

$$\mathcal{L}(\lambda rC_{ac}) = \mathcal{L}(RE),$$

where by $\mathcal{L}(RE)$ we denote the family of all recursively enumerable languages.

The Family of Turing computable sets of vectors of numbers is denoted by $PsRE$ (they are the Parikh images of the recursively enumerable languages, hence the notation).

2.2 Register Machines

We will also use in our paper the Minsky's register machines [Minsky 1967], that is why we recall here this notion. Such a machine runs a program consisting of numbered instructions of several simple types. Several variants of register machines with different number of registers and different instructions sets were shown to be computationally universal (see [Minsky 1967] for some original definitions and [Khrisna and Păun 2003] for the definition we use in this paper).

An *n-register machine* is a construct $M = (n, P, i, h)$, where:

- n is the number of registers,
- P is a set of labeled instructions of the form $j : (op(r), k, l)$, where $op(r)$ is an operation on register r of M, and j, k, l are labels from the set $Lab(M)$ (which numbers the instructions in a one-to-one manner),
- i is the initial label, and
- h is the final label.

The machine is capable of executing the following instructions:

$(add(r), k, l)$: Add one to the contents of register r and proceed to instruction k or to instruction l; in the deterministic variants usually considered in the literature we demand $k = l$.

$(sub(r), k, l)$: If register r is not empty, then subtract one from its contents and go to instruction k, otherwise proceed to instruction l.

$halt$: This instruction stops the machine. This additional instruction can only be assigned to the final label h.

A deterministic m-register machine can analyze an input $(n_1, ..., n_\alpha) \in \mathbf{N}^\alpha$ in registers 1 to α, which is recognized if the register machine stops by the halt instruction with all its registers being empty (this last requirement is not necessary). If the machine does not halt, then the analysis was not successful.

2.3 P Systems Prerequisites

A P system (of degree $m \geq 1$) with symbol–objects and rewriting evolution rules is a construct

$$\Pi = (V, C, \mu, w_1, \ldots, w_m, (R_1, \rho_1), \ldots, (R_m, \rho_m), i_0),$$

where:

- V is the alphabet of Π; its elements are called *objects*;

- $C \subseteq V$ is the set of catalysts;
- μ is a membrane structure consisting of m membranes labeled $1, 2, \cdots, m$;
- w_i, $1 \leq i \leq m$, are strings over V which specify the multisets of objects present in the corresponding regions i at the beginning of a computation;
- R_i, $1 \leq i \leq m$, are finite sets of evolution rules over V associated with the regions $1, 2, \ldots, m$ of μ, and ρ_i is a partial order relation over R_i (a priority relation); these evolution rules are of the form $a \rightarrow v$ or $ca \rightarrow cv$, where a is an object from $V - C$ and v is a string over

$$(V - C) \times (\{here, out, in\})$$

  (In general, the target indications $here$, out, in are written as subscripts of objects from V.);
- i_0 is a number between 0 and m and specifies the output membrane of Π (in case of 0, the environment is used for the output).

Starting from the original model of P system, several variants were proposed (see [Păun 2002]). One of them is that of *P systems with promoters/inhibitors* and was introduced in [Bottoni et al. 2002]. In the case of promoters, the rules (reactions) are possible only in the presence of certain symbols. An object a is a *promoter* for a rule $u \rightarrow v$, and we denote this by $u \rightarrow v|_a$, if the rule is active only in the presence of object a. An object b is an *inhibitor* for a rule $u \rightarrow v$, and we denote this by $u \rightarrow v|_{\neg b}$, if the rule is active only if inhibitor b is not present in the region. In particular, promoters/inhibitors themselves can evolve according to some rules.

The difference between catalysts and promoters consists in the fact that the catalysts directly participate in rules (but are not modified by them), and they are counted as any other objects, so that the number of applications of a rule is as big as the number of copies of the catalyst, while in the case of promoters, the presence of the promoter objects makes it possible to use the associated rule as many times as possible, without any restriction; moreover, the promoting objects do not directly participate in the rules. As a consequence, one can notice that the catalysts inhibits the parallelism of the system while the promoters/inhibitors only guide the computation process.

A P system with the mentioned features starts to evolve from an *initial configuration*, by performing all operations in a parallel way, for all applicable rules, for all occurrences of objects in the region associated with the rules, for all regions at the same time and according to a universal clock; the rules to use and the objects to evolve are chosen in a non-deterministic way. A computation is *successful* if and only if it halts, meaning that no rule is applicable to the objects present in the *final configuration*. The result of a halting computation is the multiplicity of objects present in the region i_0 in the halting configuration. The set of all vectors of numbers constructed in this way by a system Π is denoted by $Ps(\Pi)$. For this kind of P systems we will use the notation

$$PsOP_m(\alpha, \beta), \alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 0\}, \beta \in \{proR, inhR\},$$

to denote the family of sets of vectors of natural numbers generated by P systems with at most m membranes, evolution rules that can be non-cooperative ($ncoo$), cooperative (coo), or catalytic (cat_k), using at most k catalysts, and promoters ($proR$) or inhibitors ($inhR$) at the level of rules.

A P system Π can also *recognize* a vector of numbers: we introduce a vector $(n_1, \ldots, n_k)$ in the form of a multiset $a_1^{n_1} \ldots a_k^{n_k}$, for specified objects $a_1, \ldots, a_k$, in a designated input membrane and the vector is recognized (we also say accepted) if the system halts; by $Ps_a(\Pi)$ we denote the set of vectors accepted by the system Π. Then, we will use the notation

$$Ps_aOP_m(\alpha, \beta), \alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 0\}, \beta \in \{proR, inhR\},$$

to denote the family of sets of vectors of natural numbers accepted by P systems with at most m membranes, evolution rules that can be non-cooperative ($ncoo$), cooperative (coo), or catalytic (cat_k), using at most k catalysts, and promoters ($proR$) or inhibitors ($inhR$) at the level of rules.

In this paper we will show how the regularly controlled context-free grammars with appearance checking can be used to prove the computational universality of such type of P systems. Also, we will study the deterministic P systems accepting sets of vectors of natural numbers.

3 Some Relevant Examples

In this section we will present some examples of P systems computing some "sensitive" tasks using above introduced types of P systems. First we will construct a P system with promoters that, having as input two values, say 0 and/or 1, computes the *and* operation (see Figure 1).

Formally, we define the following P system

$$\Pi_{AND} = (V, C, \mu, w_1, w_2, w_3, R_1, R_2, R_3, 0),$$

where:

- $V = \{0, 1, 0', 1', 1'', A, A', A'', B, B', B'', c\}$;
- $C = \{c\}$;
- $\mu = [_3[_2[_1\ ]_1]_2]_3$;

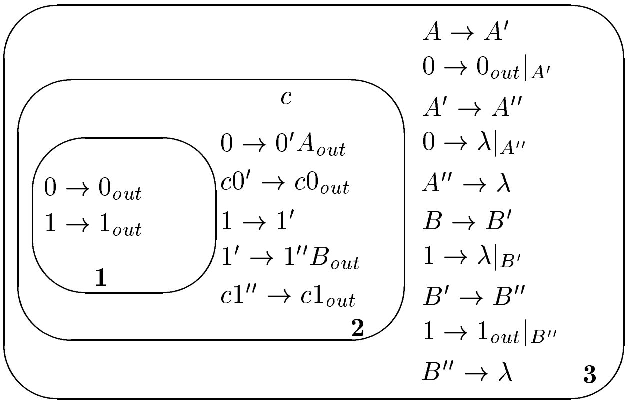

Figure 1: Simulation of the AND gate using promoters and one catalyst

– $w_1 = w_3 = \lambda$, $w_2 = c$;

– $R_1 = \{1 \to 1_{out}, 0 \to 0_{out}\}$;

$R_2 = \{0 \to 0' A_{out}, c0' \to c0_{out}, 1' \to 1'' B_{out}, 1 \to 1',$
$c1'' \to c1_{out}\}$;

$R_3 = \{A \to A', 0 \to 0_{out}|_{A'}, A' \to A'', 0 \to \lambda|_{A''},$
$A'' \to \lambda, B \to B', 1 \to \lambda|_{B'}, B' \to B'',$
$1 \to 1_{out}|_{B''}, B'' \to \lambda\}$.

The simulation of the AND gate uses the catalyst c to inhibit the parallelism and to separate the entrance time of objects 0 and 1 into region 3. According to the entrance time, objects will be either deleted, or sent out into the environment. More specifically, if we consider that initially we had two objects 0 inside region 2, the rule $0 \to 0' A_{out}$ is executed. Its role is to introduce the object A into region 3 to set up the "right" configuration of the region. Next, in region 2 the only applicable rule is $c0' \to c0_{out}$, which will introduce one object 0 into region 3. At the same time, in region 3 the rule $A \to A'$ is executed. Now, we will have in region 3 the objects A' and 0, and the rules that will be applied are $0 \to 0_{out}|_{A'}$ and $A' \to A''$. These rules guarantee that an object 0 is sent out into the environment. In the meantime, in region 2, the remaining object $0'$ reacts with the catalyst c and an object 0 will be introduced into region 3 (the rule used is again $c0' \to c0_{out}$). Here, the object 0 will find a different context since now, in region 3 there is no object A'. Therefore, the rules $0 \to \lambda|_{A''}$ and $A'' \to \lambda$ are applied, hence the initial configuration of the system is restored. Basically, a similar method stands for the other cases, with some minor changes:

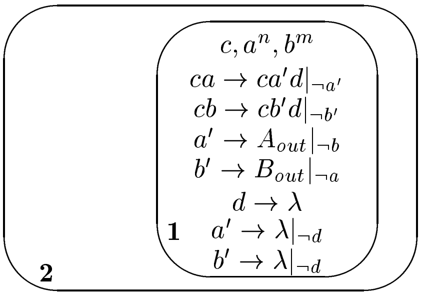

Figure 2: Integer subtraction using inhibitors and one catalyst

objects 1 enter into region 3 with one computational delay (because of the rule $1 \to 1'$ present in region 2) in order not to influence the processes executing in region 3; the first object 1 that enters into region 3 is deleted (as opposed to the above case when the first object 0 that arrives in region 3 is sent out) by using the rule $1 \to \lambda|_{B'}$.

Recall that the membrane 1 can be entirely avoided, its role being only to specify the entry point of the input. Also, the result of computation is sent out into environment even if it is actually obtained in region 3. This features are useful when we want to connect gates into circuits (see [Ceterchi and Sburlan 2003] for more details).

The second example (see Figure 2) uses context-free rules, inhibitors and one catalyst to compute the arithmetic difference between the initial multiplicity of two distinct objects, present at the beginning of computation into an input region.

Formally, we define the following P system

$$\Pi_{dif} = (V, C, \mu, w_1, w_2, R_1, R_2, 2),$$

where:

- $V = \{a, b, a', b', d, A, B, c\}$;
- $C = \{c\}$;
- $\mu = [_2[_1\ ]_1]_2$;
- $w_1 = ca^n b^m, \quad w_2 = \lambda$;
- $R_1 = \{ca \to ca'd|_{\neg a'},\ cb \to cb'd|_{\neg b'},\ a' \to A_{out}|_{\neg b},$
  $b' \to B_{out}|_{\neg a},\ d \to \lambda,\ a' \to \lambda|_{\neg d},\ b' \to \lambda|_{\neg d}\}$;

  $R_2 = \emptyset$.

The system starts the computation having into the input membrane 1 a catalyst c and the objects a^n, b^m, whose multiplicity we want to subtract. The result of computation is sent to region 2 and it is represented by:

- A^{n-m} if $n > m$;
- B^{m-n} if $m > n$;
- no object is sent to region 2 meaning that $m = n$.

The system works as follows: while there are still objects a and b, they are deleted in pairs, iteratively, up to a moment when there are no more objects a, for instance (or objects b). At that moment, the flow of computation changes and as a result, also iteratively, the remaining objects b (or objects a, respectively) are send out. During the computation, the promoters control the derivation process, while the catalyst inhibits the parallelism. For a better understanding we present the configuration table for the case when both objects a and b are present simultaneously into the input membrane.

	region 1	*region* 2
t_0	c, a^n, b^m $ca \to ca'd\|_{\neg a'}$	
t_1	c, a^{n-1}, b^m, a', d $cb \to cb'd\|_{\neg a'}$ $d \to \lambda$	
t_2	$c, a^{n-1}, b^{m-1}, a', b', d$ $d \to \lambda$	
t_3	$c, a^{n-1}, b^{m-1}, a', b'$ $a' \to \lambda\|_{\neg d}$ $b' \to \lambda\|_{\neg d}$	
t'_0	c, a^{n-1}, b^{m-1} $ca \to ca'd\|_{\neg a'}$	
...	....	....

Here we have considered only the case when at the first step an object a reacts with the catalyst c. The result of computation remains unchanged (due to symmetry reasons) even if, at the first step, an object b reacts with the catalyst c.

When in the region remain only objects a, the configuration table for the forthcoming computations is:

	region 1	*region* 2
t_p	c, a^k $ca \to ca'd\|_{\neg a'}$	
t_{p+1}	c, a^{k-1}, a', d $a' \to A_{out}\|_{\neg b'}$ $d \to \lambda$	
t'_p	c, a^{k-1} $ca \to ca'd\|_{\neg a'}$	A
...	....	....

The case when inside region 1 remain only objects b and the catalyst c is similar with the previous one, and has as result the production into region 2 of $m - n$ copies of object B.

Since in both examples we have used some context-sensing features we may conjecture that both P systems with promoters and P systems with inhibitors, using only one catalyst, are computational universal. Indeed, the following section will be dedicated to these issues and we will show how any recursively enumerable set of natural numbers can be generated/accepted by these types of P systems.

4 Universality Results

4.1 Computational Universality – The Generative Case

Here, we present two universality results concerning P systems with promoters or inhibitors at the level of rules. The proofs are based on the simulations of regularly controlled context-free grammars with appearance checking.

Theorem 1. $PsOP_2(cat_1, proR) = PsRE.$

Proof. We consider for this proof the implication $PsRE \subseteq Ps = P_2(cat_1, proR)$; the other way round can be obtained by a straightforward construction, or we can invoke the Church-Turing thesis.

Let us consider a regularly controlled grammar $G = (N, T, P, S, R, F)$ and let $G_{reg} = (N_{reg}, T_{reg}, P_{reg}, S_{reg})$ be a regular grammar generating the regular set R. We denote by r the number of rules in P_{reg}. The rules of P_{reg} are enumerated as $i : (M_i \to p_i Q_i)$ or $i : (M_i \to p_i)$ with $1 \le i \le r$, where $M_i \in N_{reg}$ and $p_i \in T_{reg}$, $1 \le i \le r$. For any such grammar G_{reg} we can construct an equivalent right–linear grammar $G' = (N', T', P', S')$ in the following way:

$T' = T_{reg}$,
$S' = S_{reg}$,
$N' = N_{reg} \cup \{M_{(i,1)}, M_{(i,2)}, M_{(i,3)} \mid 1 \le i \le r\}$.

For any rule $i : (M_i \to p_i Q_i) \in P_{reg}$ or $i : (M_i \to p_i) \in P_{reg}$, $1 \le i \le r$ we will have in P' the sequence of rules:

$$M_i \to M_{(i,1)},\ M_{(i,1)} \to M_{(i,2)},\ M_{(i,2)} \to M_{(i,3)},\ M_{(i,3)} \to p_i Q_i,$$
$$M_i \to M_{(i,1)},\ M_{(i,1)} \to M_{(i,2)},\ M_{(i,2)} \to M_{(i,3)},\ M_{(i,3)} \to p_i$$

respectively. Moreover, P' does not contain other rules excepting the rules considered above.

In other words, the only difference between the two grammars is that the production of a new terminal in grammar G' is done after each fourth step of a derivation.

Now let us construct a P system which simulates the derivation process of the regularly controlled grammar with appearance checking G. The system will use only two membranes, one catalyst and promoters. The innermost membrane will contain the generative mechanism and the results of computation will be send out to the skin membrane which will be the output membrane of the system (the reason is that the catalyst is used during the computation to inhibit the parallelism and it cannot be removed, therefore we cannot obtain the number 0 as the result of computation if we use only one membrane). In what follows we will discuss only the rules in the innermost membrane since the skin membrane does not execute any task (its role is only to collect the objects obtained during computation).

The promoters will be generated by a mechanism like the one presented above (promoters will be actually terminal symbols from T' and, therefore, they will be generated at each forth step). They will permit the execution of "context-free" rules in the "right" order – the order given by the control mechanism.

In order to correctly simulate the appearance checking mechanism we have to modify the rules in the grammar G' such that we replace each rule of type $M_{(i,3)} \to p_i Q_i$ by a rule $M_{(i,3)} \to p_i Q_i f$ or $M_{(i,3)} \to p_i Q_i a$ depending on how the object p_i indicates a rule from F (in the regularly controlled grammar definition, the set $F \subset P$ represents the appearance checking set of rules; we will use the object a to identify that a rule with the corresponding label p_i is in the appearance checking set; if not, we will produce in the rule the object f). We will consider also the same construction for the rules in G' of type $M_{(i,3)} \to p_i$, i.e., $M_{(i,3)} \to p_i f$ or $M_{(i,3)} \to p_i a$. This means that, in the definition of our P system, for the inner membrane we will have rules of the following types:

- $M_i \to M_{(i,1)},\ M_{(i,1)} \to M_{(i,2)},\ M_{(i,2)} \to M_{(i,3)},\ M_{(i,3)} \to p_i Q_i f$ if p_i is not a label in the appearance checking set;
- $M_i \to M_{(i,1)},\ M_{(i,1)} \to M_{(i,2)},\ M_{(i,2)} \to M_{(i,3)},\ M_{(i,3)} \to p_i Q_i a$ if p_i is a label in the appearance checking set;
- $M_i \to M_{(i,1)},\ M_{(i,1)} \to M_{(i,2)},\ M_{(i,2)} \to M_{(i,3)},\ M_{(i,3)} \to p_i f$ if p_i is not a label in the appearance checking set;
- $M_i \to M_{(i,1)},\ M_{(i,1)} \to M_{(i,2)},\ M_{(i,2)} \to M_{(i,3)},\ M_{(i,3)} \to p_i a$ if p_i is a label in the appearance checking set.

Up to this moment, we only have considered the control mechanism which generates labels indicating the context-free rules that should be applied. Let us denote by $G_{CF} = (N_{CF}, T_{CF}, P_{CF}, S_{CF})$ a context-free grammar with productions labeled with the elements of T_{reg}. Now we will discuss how we can simulate in the P system the application of a context-free rule $p : (A \to \alpha)$ indicated by the control mechanism.

For a context-free rule $(p : (A \to \alpha)) \in G_{CF}$ we will have in our P system the following sequence of rules:

$cA \to cD\alpha'|_p$, with $\alpha' = \alpha$ if $\alpha \in N$, and $\alpha' = (\alpha, out)$ if $\alpha \in T$,
$p \to p'$,
$p' \to \lambda|_D$,
$D \to \lambda$.

If promoter p, object A, and catalyst c are present at a certain moment together, then they will react only once in two consecutive computational steps. This is due to the fact that the promoter p is changed ($p \to p'$) in the same moment with the execution of the rule $cA \to cD\alpha'|_p$. Moreover, the presence of the catalyst c in the rule inhibits the parallelism (we want that in one "round" the rule $A \to \alpha$ to be applied only once and not for all occurrences of object A that may exist in the region). Now, in order to be sure that the rule $cA \to cD\alpha'|_p$ was executed an object D is created; it will help to delete the object p' present in membrane (which if not deleted can cause problems in further steps). The object D will be also deleted by the rule $D \to \lambda$.

This sequence of rules stands for the case when the context-free rule $A \to \alpha$ can be applied and so, it must be applied. In the case when the rule mentioned cannot be applied we have to decide if the promoter present indicates a rule with the label in the appearance checking set or not.

First, let us consider the case when the promoter is not a label in the appearance checking set. In this case, recall that we deal with the following sequences of productions (from the regular mechanism):

- $M_i \to M_{(i,1)},\ M_{(i,1)} \to M_{(i,2)},\ M_{(i,2)} \to M_{(i,3)},\ M_{(i,3)} \to p_iQ_if$, or
- $M_i \to M_{(i,1)},\ M_{(i,1)} \to M_{(i,2)},\ M_{(i,2)} \to M_{(i,3)},\ M_{(i,3)} \to p_if$.

As the result of applying these rules we will have in the inner region, among others, the objects p and f. Let us consider that, for this case, we have the rules:

$f \to f_1$,
$f_1 \to f_2$,
$p' \to \#|_{f_2}$,
$f_2 \to \lambda$,
$\# \to \#'$,
$\#' \to \#$.

The first two rules from this group are meant to delay the execution of the third rule because we are not "sure" if the rule $cA \to cD\alpha'|_p$ is or it is not applied. So, if the mentioned rule is applied, then the object f_2 will be deleted by the rule $f_2 \to \lambda$ and will not promote the rule $p' \to \#|_{f_2}$ since the object $p' \to \lambda|_D$ was "consumed" in a previous step. As a consequence, there will be no effect (in terms of objects produced) if the previous set of rules is executed. In the opposite case (when the rule $cA \to cD\alpha'|_p$ is not applied because there is no object A present in the region), then the object $\#$ will be generated. The rules $\# \rightarrow \#'$, $\#' \rightarrow \#$ will cycle forever and the computation will never halt and this will mean that the computation have failed (recall that we show the universality for the nondeterministic case).

With a similar construction like above we can solve the case when we deal with rules that have labels in the appearance checking set. This means that the rules to be applied are of the types:

- $M_i \rightarrow M_{(i,1)},\ M_{(i,1)} \rightarrow M_{(i,2)},\ M_{(i,2)} \rightarrow M_{(i,3)},\ M_{(i,3)} \rightarrow p_i Q_i a$, or
- $M_i \rightarrow M_{(i,1)},\ M_{(i,1)} \rightarrow M_{(i,2)},\ M_{(i,2)} \rightarrow M_{(i,3)},\ M_{(i,3)} \rightarrow p_i a.$

Here the difference from the previous case consists in not generating the trap symbol $\#$ if the rule $cA \rightarrow cD\alpha'|_p$ cannot be applied. We only have to delete the promoter p'. The rules below state the fact that if a rule is in the appearance checking set and it cannot be applied even if it is indicated by the control mechanism, then it can be skipped.

$a \rightarrow a_1,$
$a_1 \rightarrow a_2,$
$p' \rightarrow \lambda|_{a_2},$
$a_2 \rightarrow \lambda.$

Finally, the initial configuration of the P system is composed by the starting symbol S_{Reg} of the regulating mechanism, the starting symbol S_{CF} of the context-free mechanism and the catalyst c. The system will evolve in a maximally parallel manner its behavior being controlled by the catalyst and promoters. The set of vectors generated by the P system Π is clearly equal to the Parikh image of the language $L(G)$. □

One can notice that we arrive at the same descriptional complexity in terms of number of membranes, number of catalysts and promoters as in the original proof in [Bottoni et al. 2002], but simulating a different computational universal mechanism.

As it can be seen, the promoters combined with one catalyst are sufficient to prove the computational universal capabilities of the P systems when using only context-free object rewriting rules. A similar result stands, but based on inhibitors instead of promoters.

Theorem 2. $PsOP_2(cat_1, inhR) = PsRE.$

Proof. We follow a similar construction like the one in the previous proof, but using a sequence of rules as the one given below. Here we also will have two membranes and will discuss only the rules presented in the inner membrane, the skin membrane being used only to collect the result of computations. First let us recall that the rules from the context-free grammar are labeled with the symbols $p_1, \cdots, p_k$. Consider also that if in the regular grammar which controls the derivation of the context-free grammar we have a rule $M_i \to p_j Q_i$, then in our simulation we will have:

$M_i \to p_1 \cdots p_k M_{(i,1)}$,
$M_{(i,1)} \to p_1 \cdots p_k M_{(i,2)}$,
$M_{(i,2)} \to p_1 \cdots p_k M_{(i,3)}$,
$M_{(i,3)} \to p_1 \cdots p_{j-1} p_{j+1} \cdots p_k Q_i f r$.

Moreover, in the above construction we have considered that the label p_j indicates a context-free rule which is not in the appearance checking set. As it can be seen, instead of indicating the rule with the symbol p_j (meaning that the context-free rule with label p_j should be applied), we have used all the symbols from the complementary set, i.e., $p_1, \cdots, p_{j-1}, p_{j+1}, \cdots, p_k$.

Now, let us consider the following set of rules which simulates the behavior of the context-free mechanism:

$cA \to CD\alpha'|_{\neg p_j}$, with $\alpha' = \alpha$ if $\alpha \in N$, and $\alpha' = (\alpha, out)$ if $\alpha \in T$,
$p_j \to \lambda$,
$f \to f_1$,
$r \to r_1$,
$D \to D_1$,
$r_1 \to r_2|_{\neg D}$,
$f_1 \to f_2$,
$f_2 \to \#|_{\neg D_1}$,
$f_2 \to \lambda|_{\neg r_2}$,
$r_2 \to \lambda$,
$D_1 \to \lambda$,
$\# \to \#'$,
$\#' \to \#$.

The first three rules of the "regular controller" produce the symbols $p_1, \cdots, p_k$ and so the rule $cA \to CD\alpha'|_{\neg p_j}$ cannot be applied because always object p_j will be among the mentioned objects. This means that during the first three steps in a cycle the rule $cA \to CD\alpha'|_{\neg p_j}$ cannot be applied. Also, at each step we delete the objects $p_1, \cdots, p_k$ by rules of type $p_j \to \lambda$. In the last step in a cycle, we avoid to introduce the object p_j and so the rule $cA \to CD\alpha'|_{\neg p_j}$ can be applied. With this occasion, we also introduce the objects f and r which are used to indicate a rule which is not in the appearance checking set.

The rule $cA \to CD\alpha'|_{\neg p_j}$ will modify only one (if any) occurrence of object A present in the region because of the catalyst c. Moreover, in the next step of computation the objects $p_1, \cdots, p_k$ will be again introduced, so they will forbid the execution of any rule of type $cA \to CD\alpha'|_{\neg p_j}$.

Now, coming back, we can notice that the rule $cA \to CD\alpha'|_{\neg p_j}$ can be applied only if there is an occurrence of object A. Recall that this rule is not in the appearance checking set and so, if it cannot be applied, then the computations must cycle forever in order not to accept. If the rule can be applied, it has to be applied and moreover all the symbols that were produced during the computation must be deleted in order not to interfere in the next cycle.

Now let us see which is the result of a computation in both cases. Consider the first case, when the rule $cA \rightarrow CD\alpha'|_{\neg p_j}$ is applied. Then, in the same moment, the rules $f \rightarrow f_1$ and $r \rightarrow r_1$ are executed. In the next step the rules $D \rightarrow D_1$ and $f_1 \rightarrow f_2$ can be applied. The rule $r_1 \rightarrow r_2|_{\neg D}$ cannot be applied since the object D is present in the region. As an effect, the region will contain the objects D_1, f_2, and r_1. So, the only applicable rules are $D_1 \rightarrow \lambda$, $f_2 \rightarrow \lambda|_{\neg r_2}$, and $r_1 \rightarrow r_2|_{\neg D}$. Finally, the rule $r_2 \rightarrow \lambda$ deletes the last symbol that was created in this cycle. This means that we reestablish the initial configuration and the computation can continue.

Now, consider the opposite case, when the rule $cA \rightarrow CD\alpha'|_{\neg p_j}$ has to be applied (because is indicated by the regular control), but it cannot be applied (because, there is no symbol A present in the region). So, the symbol D is not released. As an effect we will have in the region the objects f_1 and r_1 and the rules to be applied are $r_1 \rightarrow r_2|_{\neg D}$ and $f_1 \rightarrow f_2$. Next, the rules $f_2 \rightarrow \#|_{\neg D_1}$ and $r_2 \rightarrow \lambda$ will be applied and so the # symbol will be created and the computation will never halt.

For the rules that are in the appearance checking set we can consider for the regular mechanism rules of the following types:

$M_i \rightarrow p_1 \cdots p_k M_{(i,1)}$,
$M_{(i,1)} \rightarrow p_1 \cdots p_k M_{(i,2)}$,
$M_{(i,2)} \rightarrow p_1 \cdots p_k M_{(i,3)}$,
$M_{(i,3)} \rightarrow p_1 \cdots p_{j-1} p_{j+1} \cdots p_k Q_i$,

while for the "context-free" mechanism a rule:

$cA \rightarrow C\alpha'|_{\neg p_j}$ is considered, with α' as above.

In this way, if the rule can be applied, then it will be applied and if cannot be applied, then nothing will happen and the process will continue correctly simulating the appearance checking mechanism. $\square$

4.2 Computational Universality – The Accepting Case

The following theorems illustrate the computational universality (in their accepting variants) of P systems with object rewriting non-cooperative rules and promoters/inhibitors at the level of rules. The systems we propose simulate the moves of deterministic register machines. Moreover, the obtained P systems are also deterministic.

Theorem 3. *$Ps_aOP_2(cat_1, proR) = PsRE$.*

Proof. In order to prove this assertion we will simulate an n–register machine $M = (n, P, i, h)$, which, when reaching the halt instruction has all registers empty. At each time during the computation, the current contents of register j is represented by the multiplicity of the object a_j.

Formally, we define the P system

$$\Pi = (V, C, [_1\ [_2\ ]_2\ ]_1, w_1 = \emptyset, w_2, R_1 = \emptyset, R_2, 1),$$

where:

$$\begin{aligned} V &= \{a_j, A_j, S_j \mid 1 \le j \le n\} \cup \{c, F, T\} \cup \{e, e' \mid (e : add(j), f) \in P\} \\ &\quad \cup \{e, e', e'' \mid (e : sub(j), f, z) \in P\}, \\ C &= \{c\}, \\ w_2 &= ci, \end{aligned}$$

and R_2 is defined as follows:

- for each instruction $(e : add(j), f) \in P$, we add to R_2 the rules:
  $e \to e'A_j$,
  $c \to ca_j|_{A_j}$,
  $A_j \to \lambda$,
  $e' \to f$;
- for each instruction $(e : sub(j), f, z) \in P$, we add to R_2 the rules:
  $e \to e'TS_j$,
  $ca_j \to cF|_{S_j}$,
  $S_j \to \lambda$,
  $e' \to e''$,
  $T \to T'$,
  $e'' \to f|_F$,
  $F \to \lambda$,
  $T' \to T''$,
  $e'' \to z|_{T''}$,
  $T'' \to \lambda$;
- for the instruction $(h : halt) \in P$, we add to R_2 the rule
  $h \to \lambda$;
- no other rules are added to R_2.

The system works as following. Initially the P system starts the computation having in its input region (region 2) the objects $a_1^{k_1}, \cdots, a_m^{k_m}$, the catalyst c and the label i of the first instruction of the register machine we want to simulate.

The vector $(k_1, \cdots, k_m)$ represents the vector that has to be accepted by our P system.

The P system starts the computation by simulating the first instruction of the register machine program. Let us suppose that the current instruction to be executed is of type $(e : add(j), f) \in P$. Then, in region 2 the rule $e \to e'A_j$ is executed. The object A_j indicates that the number of objects a_j has to be incremented. This will be realized, in the second computational step, by the promoted evolution rule $c \to ca_j|_{A_j}$; the rules $A_j \to \lambda$ and $e' \to f$ are executed in the same time with the previous one. The first rule assures that $c \to ca_j|_{A_j}$ is not executed again in one iteration (therefore, j is just incremented), while the second rule allows the P system to further simulate the instruction of the register machine indicated by label f.

If a subtraction instruction $(e : sub(j), f, z) \in P$ is simulated, then in region 2 the rule $e \to e'TS_j$ is executed (note that the object S_j stands for the subtraction command). As a result, the rule $ca_j \to cF|_{S_j}$ is executed and the number of objects a_j is decreased by 1 (if it is possible, i.e., the number of objects a_j is greater than 0). Meanwhile, the execution of rule $S_j \to \lambda$ guarantees that only one object a_j was deleted from the current multiset (again, if it is possible). If everything worked fine, then the sequence of rules $e' \to e''$, $e'' \to f|_F$, $F \to \lambda$ was executed, and the label of the next register machine instruction was generated. Otherwise (i.e., there is no objects a_j in region 2) the rule $ca_j \to cF|_{S_j}$ is not executed therefore the object F is not produced and the rule $e'' \to f|_F$ cannot be applied. Meanwhile, the object T evolves to T' and then to T'', in three consecutive steps. The last step of an iteration consists of simultaneously execution of the rules $e'' \to z|_{T''}$ and $T'' \to \lambda$. In conclusion, the label of the next register machine instruction is created.

The above presented simulations of the instructions, $(e : add(j), f) \in P$ and $(e : sub(j), f, z) \in P$, are iterated according to the register machine program. The simulation stops when h label (which stands for *halt* instruction) is generated (we know that in that moment no other objects a_j, $1 \le j \le n$, are present in region 2). $\square$

Theorem 4. $Ps_aOP_2(cat_1, inhR) = PsRE$.

Proof. This assertion will be proved again by simulating an n–register machine $M = (n, P, i, h)$. The contents of register j will be denoted in our simulation, as in the previous theorem, by the multiplicity of the object a_j.

Formally, we define the P system

$$\Pi = (V, C, [_1\ [_2\ ]_2\ ]_1, w_1 = \emptyset, w_2, R_1 = \emptyset, R_2, 1),$$

where:

$$V = \{a_j, A_j, S_j \mid 1 \le j \le n\} \cup \{c, P, P_1, Q, Q_1, Q_2\}$$

$$\cup \{e, e' \mid (e : add(j), f) \in P\} \cup \{e \mid (e : sub(j), f, z) \in P\}$$
$$\cup \{F_{(e,0)}, F_{(e,1)}, F_{(e,2)} \mid (e : sub(j), f, z) \in P\},$$
$$C = \{c\},$$
$$w_2 = ci,$$

and R_2 is defined as follows:

- for each instruction $(e : add(j), f) \in P$, we add to R_2 the rules:
  $e \to e' A_1 \cdots A_{j-1} A_{j+1} \cdots A_n S_1 \cdots S_n$,
  $c \to ca_j A_1 \cdots A_n S_1 \cdots S_n|_{\neg A_j}$,
  $A_i \to \lambda$, $1 \leq i \leq n$,
  $S_i \to \lambda$, $1 \leq i \leq n$,
  $e' \to f A_1 \cdots A_n S_1 \cdots S_n$;
- for each instruction $(e : sub(j), f, z) \in P$, we add to R_2 the rules:
  $e \to F_{(e,0)} Q A_1 \cdots A_n S_1 \cdots S_{j-1} S_{j+1} \cdots S_n$,
  $ca_j \to cPA_1 \cdots A_n S_1 \cdots S_n|_{\neg S_j}$,
  $A_i \to \lambda$, $1 \leq i \leq n$,
  $S_i \to \lambda$, $1 \leq i \leq n$,
  $F_{(e,0)} \to F_{(e,1)} A_1 \cdots A_n S_1 \cdots S_n$,
  $Q \to Q_1 A_1 \cdots A_n S_1 \cdots S_n$,
  $P \to P_1 A_1 \cdots A_n S_1 \cdots S_n$,
  $Q_1 \to Q_2 A_1 \cdots A_n S_1 \cdots S_n|_{\neg P}$,
  $F_{(e,1)} \to F_{(e,2)} A_1 \cdots A_n S_1 \cdots S_n$
  $F_{(e,2)} \to z A_1 \cdots A_n S_1 \cdots S_n|_{\neg P_1}$,
  $P_1 \to \lambda$,
  $F_{(e,2)} \to f A_1 \cdots A_n S_1 \cdots S_n|_{\neg Q_2}$,
  $Q_2 \to \lambda$;
- no other rules are added to R_2.

As in the previous theorem we start the computation having in region 2 the objects $a_1^{k_1}, \cdots, a_m^{k_m}$, the catalyst c and the label i of the first instruction of the register machine we want to simulate.

Let us consider that an increment instruction $(e : add(j), f) \in P$ has to be simulated. Then, the existing object e (which represents the previous instruction label) is rewritten by the rule $e \to e' A_1 \cdots A_{j-1} A_{j+1} \cdots A_n S_1 \cdots S_n$. As it can be seen only the object A_j is missing from the right hand side of the rule. The absence of this object indicates that the number of objects a_j has to be incremented. Because of this, the rule $c \to ca_j|_{\neg A_j}$ can be applied. Meanwhile, the rules $A_i \to \lambda$, $1 \leq i \leq n$, $S_i \to \lambda$, $1 \leq i \leq n$, and $e' \to f A_1 \cdots A_n S_1 \cdots S_n$ are executed. Their role is to reconfigure the system for the next instruction that has to be simulated.

Now, let us consider that the system receives the command to subtract one object a_j from the current multiset ($e \rightarrow F_{(e,0)}QA_1 \cdots A_nS_1 \cdots S_{j-1}S_{j+1} \cdots S_n$).

If $|w|_{a_j} \geq 1$ where w is the current multiset present in region 2, we can apply the rule $ca_j \rightarrow cPA_1 \cdots A_nS_1 \cdots S_n|_{\neg S_j}$ since the object S_j is missing. The rules $A_i \rightarrow \lambda$, $1 \leq i \leq n$, and $S_i \rightarrow \lambda$, $1 \leq i \leq n$, are executed at each step of computation. The objects $A_1, \cdots, A_n, S_1, \cdots, S_n$ are created always with only one exception – when we want to execute the instruction corresponding to the missing object. The computation continues until the rule $F_{(e,2)} \rightarrow fA_1 \cdots A_nS_1 \cdots S_n|_{\neg Q_2}$ is executed and the object f indicating the next instruction is generated.

In a similar way as shown in the proof of the previous theorem, if $|w|_{a_j} = 0$, then the rule $ca_j \rightarrow cPA_1 \cdots A_nS_1 \cdots S_n|_{\neg S_j}$ is not executed. Notice that objects $F_{(e,0)}$ and Q are "witnesses" that the command to simulate the subtraction scheme was made. Similarly, the presence of object P indicates that, in region 2, was at least one object a_j. In order to correctly simulate the decrement instruction we want to have the "right" missing object at the "right" time. This involves some delaying rules like $F_{(e,0)} \rightarrow F_{(e,1)}A_1 \cdots A_nS_1 \cdots S_n$, $Q \rightarrow Q_1A_1 \cdots A_nS_1 \cdots S_n$, $P \rightarrow P_1A_1 \cdots A_nS_1 \cdots S_n$, $F_{(e,1)} \rightarrow F_{(e,2)}A_1 \cdots A_nS_1 \cdots S_n$.

If everything worked well, then the rule $F_{(e,2)} \rightarrow zA_1 \cdots A_nS_1 \cdots S_n|_{\neg P_1}$ was executed and the symbol z indicating the next instruction was generated.

Finally, if the h symbol is generated then computation stops and accepts the input. □

5 Conclusion

As it can be seen from the proofs of first two theorems concerning promoters/inhibitors at the level of rules, the use of regularly controlled context-free grammars with appearance checking is useful to show the computational universality when we are not interested in minimizing the number of symbols acting as promoters/inhibitors. Practically, in both proofs we have used a number of promoters/inhibitors equal to the number of terminals in the regular grammar which controls the derivation process.

For the last two theorems we succeeded to simulate with a P system, in a deterministic manner, a deterministic register machine. There we discovered that, in case of promoted P systems, $4n$ symbols acting as promoters were enough to recognize $PsRE \cap \mathbb{N}^n$; in case of P systems with inhibitors, the number of inhibitors used to recognize $PsRE \cap \mathbb{N}^n$ was $4n + 3$.

For all theorems presented, an important aspect is that the promoters/ inhibitors may react at the same time as the rules they promote/inhibit. This fact, together with the use of one catalyst which inhibits the parallelism, makes this types of P systems computational universal.

Several problems regarding this topic still remain open. In the deterministic variant of recognizing $PsRE \cap \mathbb{N}^n$ there is not known which is the lower bound of symbols that, acting as promoters/inhibitors, make the P system model universal (when one catalyst is used). Also, there is not known which is the computational power of P systems with promoters/inhibitors at the level of rules when no catalyst is used.

Acknowledgements

The work of the first author was supported by the FPU fellowship from the Spanish Ministry of Education, Culture and Sport. The work of the second author was possible due to a doctoral grant from AECI, Spanish Ministry of Foreign Affairs.

References

[Bottoni et al. 2002] Bottoni P., Martín-Vide C., Păun Gh., Rozenberg G.: "Membrane Systems with Promoters/ Inhibitors"; Acta Informatica, 38, 10 (2002), 695–720

[Ceterchi and Sburlan 2003] Ceterchi R., Sburlan D.: "Simulating Boolean Circuits with P Systems"; "Workshop on Membrane Computing WMC2003" (A. Alhazov, C. Martín-Vide, Gh. Păun, eds.), TR 28/03, URV Tarragona (2003)

[Dassow and Păun 1989] Dassow J., Păun Gh.: "Regulated Rewriting in Formal Language Theory"; Springer, Berlin (1989)

[Freund et al. 2003] Freund R., Kari L., Oswald M., Sosik P.: "Computationally Universal P Systems without Priorities: Two Catalysts Are Sufficient"; submitted 2003

[Khrisna and Păun 2003] Khrisna S., Păun A.: "Three Universality Results on P Systems"; "Workshop on Membrane Computing WMC2003" (A. Alhazov, C. Martín-Vide, Gh. Păun, eds), TR 28/03, URV Tarragona (2003), 198–206

[Minsky 1967] Minsky M.L.: "Finite and Infinite Machines"; Prentice Hall, Englewood Cliffs (1967)

[Păun 2002] Păun Gh.: "Membrane Computing. An Introduction", Springer, Berlin (2002)

[Păun and G. Rozenberg 2002] Păun Gh., Rozenberg G.: "A Guide to Membrane Computing"; Theoretical Computer Science, 287, 1 (2002), 73–100

[Rozenberg and Salomaa 1997] Rozenberg G., Salomaa A. (eds.): "Handbook of Formal Languages", Springer, Berlin (1997)