Inhibiting the parallelism in P systems

Michal Kováč

P systems are formal models of distributed parallel multiset processing. Many variants make use of the maximal parallelism to achieve universality. It is known that P systems with catalytic rules with only one catalyst are not universal, but when using promoters and inhibitors, the universality is achieved. The sequential variant of P system is also not universal and we will show how the computational universality can be reached by using sequential P systems with inhibitors. Both accepting and generating case are investigated. In addition, a new variant of P system is defined in this paper, where the emptiness of a region is represented by a special vacuum object. This variant is shown to be universal when operating in the sequential mode.

1 Introduction

Membrane systems (P systems) were introduced by Păun (see [12]) as distributed parallel computing devices inspired by the structure and functionality of cells. One of the objectives is to relax the condition of using the rules in a maximally parallel way in order to find more realistic P systems from a biological point of view. In sequential systems, only one rewriting rule is used in each step of computation.

We are looking for ways to obtain universality in this sequential mode.

At first, we consider inhibitors and show that using them we can simulate P systems working in a maximally parallel way.

Then, we introduce the concept of vacuum. In the common sense, vacuum represents a space with no or a little matter in it. Using vacuum in modelling frameworks can help express certain phenomena more easily. We define a new P system variant, which creates a special vacuum object in a region as soon as the region becomes empty. The vacuum is removed whenever some object interacts with it. This removal process is realized by allowing the vacuum object to be used only on the left side of rules. If we made the vacuum to be removed automatically when an object enters the region, there would be no difference with the variant without vacuum objects because of no interactions with it. We are interested in how the variant with the vacuum improves the computational power of a P system in comparison to the variant without the vacuum.

There has been lot of research that considers rewriting rules with catalysts, promoters and inhibitors. However, they were only used to overcome the limitations brought by non-cooperative rules, and still working in a maximal parallel manner.

In [9], a variant of P systems with maximal parallelism restricted to use at most one instance of any rule in a step. It is shown that for universality is sufficient non-cooperative variant where the only allowed cooperative rule with both symbols equal. For example rule $a|a \to b$. Also, catalytic rules are sufficient too.

In [14] the author formally proves equality between classes of P systems and Parikh images of classes in Chomsky hierarchy. P systems with context-free rules are equal to $PsCF$ and P systems with cooperative rules are universal (equal to $PsRE$).

Submitted to:
ECAL 2013

The gap between is filled in [10]. For P systems with context-free rules and maximal parallelism, there is a need for some context added to the rules. Cooperative or catalytic rules are sufficient for universality. The paper shows that two catalyst are enough. There was an open question whether one catalyst is enough. Then it was shown that it is not enough unless we allow promoters or inhibitors. Without them it has the power of Lindenmayer systems.

Rewriting controlled by inhibition is more thoroughly researched in [4]. P systems with context-free rules are extended with rules with option to inhibit / de-inhibit another rule. This, with one catalyst is shown to be universal. Without catalyst and with only one membrane it is shown to have at least the power of Lindenmayer systems.

However, [15] shows that even if we have maximal parallel context-free rules with inhibitors, it is only equal to the family of Parikh sets of ET0L languages (PsET0L). If we have promoters instead of inhibitors, only the inclusion of the family PsET0L was proved.

Several sequential variants of P systems were studied. In [7] membranes have been assigned energy values that can control rule rewriting. Each rule has energy that is consumed when the rule is applied. If we have rules with priorities, the system becomes universal.

However, if we extend sequential P systems with cooperative rules as in [6] and restrict them to use only one membrane, they define exactly semilinear sets and are not universal.

There is something fundamental that prevents the sequential systems to be universal. Unlike maximal parallel systems, the application of rules in sequential systems is not synchronized between membranes. Even inside one membrane an object can be rewritten multiple times while other objects are not rewritten at all. Maximal parallelism does not suffer from these issues.

There is also a notion of minimal parallelism ([5]), which forces all membranes to apply at least one rule if there is one that can be used. This approach solves the synchronization issue between membranes, but not the one inside one membrane. Nevertheless, P systems with minimal parallelism are universal.

In [1] several non-cooperative variants with either promoters or inhibitors were studied. Maximally parallel and asynchronous modes were shown to be universal. They also defined the deterministic mode (only for accepting case). The computation is accepted only if for each configuration there was at most one multiset of rules applicable. Such deterministic variant was shown to only accept the family of finite languages.

Another sequential model, but not a variant of P systems, is Calculi of Looping Sequences. It is introduced in [3] as a membrane model where objects are strings that can loop around forming a membrane. Rewriting rules are global and are executed sequentially. It is shown to be universal by directly simulating P systems.

In section 2 we recall some fundamental notions which we will use through the rest of the paper. We introduce the notion of P system and define a computation of a P system in section 3. Some existing P system variants are mentioned in section 4. There is also introduced our new variant of P system with vacuum. In section 5 we make a proof that sequential P systems with inhibitors are universal. Similar result, but for our new variant of P system with vacuum is achieved and proven in section 6.

2 Preliminaries

Here we recall several notions from the classical theory of formal languages.

An **alphabet** is a finite nonempty set of symbols. Usually it is denoted by Σ or V. A **string** over an alphabet is a finite sequence of symbols from alphabet. We denote by V^* the set of all strings over an alphabet V. By $V^+ = V^* - \{\varepsilon\}$ we denote the set of all nonempty strings over V. A **language** over the alphabet V is any subset of V^*.

The number of occurrences of a given symbol $a \in V$ in the string $w \in V^*$ is denoted by $|w|_a$. $\Psi_V(w) = (|w|_{a_1}, |w|_{a_2}, \ldots, |w|_{a_n})$ is called a Parikh vector associated with the string $w \in V^*$, where $V = \{a_1, a_2, \ldots a_n\}$. For a language $L \subseteq V^*$, $\Psi_V(L) = \{\Psi_V(w) | w \in L\}$ is the Parikh mapping associated with V. If FL is a family of languages, by PsFL is denoted the family of Parikh images of languages in FL.

A multiset over a set X is a mapping $M : X \to \mathbb{N}$. We denote by $M(x), x \in X$ the multiplicity of x in the multiset M. The **support** of a multiset M is the set $supp(M) = \{x \in X | M(x) \geq 1\}$. It is the set of items with at least one occurrence. A multiset is **empty** when its support is empty. A multiset M with finite support $X = \{x_1, x_2, \ldots, x_n\}$ can be represented by the string $x_1^{M(x_1)} x_2^{M(x_2)} \ldots x_n^{M(x_n)}$. As elements of a multiset can also be strings, we separate them with the pipe symbol, e.g. $element|element|other\_element$. We say that multiset M_1 is included in multiset M_2 if $\forall x \in X : M_1(x) \leq M_2(x)$. We denote it by $M_1 \subseteq M_2$. If $M_1 \subseteq M_2$, the **difference** of two multisets $M_2 - M_1$ is defined as a multiset where $\forall x \in X : (M_2 - M_1)(x) = M_2(x) - M_1(x)$. The **union** of two multisets $M_1 \cup M_2$ is a multiset where $\forall x \in X : (M_1 \cup M_2)(x) = M_1(x) + M_2(x)$. Product of multiset M with natural number $n \in \mathbb{N}$ is a multiset where $\forall x \in X : (n \cdot M)(x) = n \cdot M(x)$.

3 P-systems

The fundamental ingredient of a P system is the **membrane structure** (see [13]). It is a hierarchically arranged set of membranes, all contained in the **skin membrane**. A membrane without any other membrane inside is said to be **elementary**. Each membrane determines a compartment, also called region, the space delimited from above by it and from below by the membranes placed directly inside, if any exists. Clearly, the correspondence membrane – region is one-to-one, that is why we sometimes use interchangeably these terms.

Consider a finite set of symbols $V = \{a_1, a_2, \ldots, a_n\}$. An arbitrary multiset rewriting rule is a pair (u, v) of multisets over the set V where u is not empty. Such a rule is typically written as $u \to v$. For a multiset rewriting rule $r : u \to v$, let $left(r) = u$ and $right(r) = v$. Let w be a multiset of symbols over V and let $R = \{r_1, r_2, \ldots, r_k\}$ be a set of multiset rewriting rules such that $r_i = u_i \to v_i$ with u_i, v_i multisets over V. Denote by $R_w^{ap} \subseteq R$ the **set of applicable multiset rewriting rules** to w, that is, $R_w^{ap} = \{r \in R | left(r) \subseteq w\}$. Denote by $R_w^{msap} : R \to \mathbb{N}$, a multiset over R of **maximal simultaneously applicable multiset rewriting rules** to w. R_w^{msap} is any multiset such that $\bigcup_{r \in R} R_w^{msap}(r) \cdot left(r) \subseteq w$ and $\forall r' \left(\bigcup_{r \in R} R_w^{msap}(r) \cdot left(r) \right) \cup left(r') \not\subseteq w$. In other words, we use a multiset of rewriting rules such that no more rules can be applied simultaneously.

P system is a tuple $(V, \mu, w_1, w_2, \ldots, w_m, R_1, R_2, \ldots, R_m)$, where:

- V is the alphabet of symbols,
- μ is a membrane structure consisting of m membranes labeled with numbers $1, 2, \ldots, m$,

- $w_1, w_2, \dots w_m$ are multisets of symbols present in the regions $1, 2, \dots, m$ of the membrane structure,
- $R_1, R_2, \dots R_m$ are finite sets of rewriting rules associated with the regions $1, 2, \dots, m$ of the membrane structure.

Each rewriting rule may specify for each symbol on the right side, whether it stays in the current region, moves through the membrane to the parent region ($\uparrow$) or through membrane to all of the child regions ($\downarrow$) or to a specific child region ($\downarrow_m$, where m is a label of a membrane). We denote these transfers with arrows immediately after the symbol. An example of such rule is the following: $a|b|b \to a|b\downarrow|c\uparrow|c$.

A **configuration** of a P system is represented by its membrane structure and the multisets of objects in the regions.

A **computation step** of P system is a relation $\Rightarrow$ on the set of configurations such that $C_1 \Rightarrow C_2$ iff:

For every region in C_1 (suppose it contains a multiset of objects w) the multiset in corresponding region in C_2 is the result of applying a multiset of simultaneously applicable multiset rewriting rules to w.

In the default P system, which works in maximal parallel mode, a maximal multiset of these rules is applied in each step and region.

For example, let's have two regions with multisets $a|a$ and b. In the first region there is a rule $a \to b$ and in the second membrane there is a rule $b \to a|a$. The only possible result of a computation step is $b|b$, $a|a$. The first rule was applied twice and the second rule once. No more object could be consumed by rewriting rules.

Computation of a P system consists of a sequence of steps. The step S_i is applied to result of previous step S_{i-1}. So when $S_i = (C_j, C_{j+1})$, $S_{i-1} = (C_{j-1}, C_j)$.

There are two possible ways of assigning a result of a computation:

1. By considering the multiplicity of objects present in a designated membrane in a halting configuration. In this case we obtain a vector of natural numbers. We can also represent this vector as a multiset of objects or as Parikh image of a language.
2. By concatenating the symbols which leave the system, in the order they are sent out of the skin membrane (if several symbols are expelled at the same time, then any ordering of them is accepted). In this case we generate a language.

The result of a computation is clearly only one multiset or a string, but for one initial configuration there can be multiple possible computations. It follows from the fact that there exist more than one maximal multiset of rules that can be applied in each step.

4 P system variants

There are several variants of P systems. They varies in the definition of the rewriting process.

We study the power of new P system variants. Those with the power of the Turing machine are especially interesting, they are said to be computationally universal. Computational power for a model is usually proved by proving equivalence to other model with known power. The equivalence is usually proved through the simulation of a step of the computation.

Sequential P systems are variant where in each step only a single rule in a single region is applied. The membrane and the rule are chosen nondeterministically. In [8] it is shown that

the sequential P systems can be simulated by vector addition systems are are not universal. However, if we allow rules with membrane creation with unbounded number of mebranes, they become universal. **Cooperative P systems** have rules that have at most two symbols on the left side. **Context-free P systems** have only rules that have only one symbol on the left side. **Catalytic P systems** have catalysts as a specific subset of alphabet. There are two types of rules:

1. $c|a \to c|w$, where c is catalyst,
2. $a \to w$

An object a can be a **promoter** (see [10]) for a rule $u \to v$, and we denote this by $u \to v|_a$, if the rule is active only in the presence of object a in the same region. An object b is **inhibitor** for a rule $u \to v$, and we denote this by $u \to v|_{\neg b}$, if the rule is active only if the inhibitor b is not present in the region. 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 necessarily 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.

We propose a new variant: **P system with vacuum**. The notion of vacuum is new in context of P systems.

From the physical point of view, vacuum is space that is empty of matter. We will consider vacuum as state of a P system region with no objects that are defined in the P system. There still can be some other object present which are not of interest such as air, water, cytoplasm and which do not react with objects of P system.

We will allow vacuum to act as a reactant in reactions. But there is a problem: when there is a vacuum in a region, there is no other object to react with. So what does the reaction with vacuum mean?

The reaction can be also seen as "object entering the vacuum" - objects are facing low pressure and are sucked into the region. So when some object enters the region with the vacuum, the vacuum is not removed immediately. We will represent this behavior with rewriting rules that can have vacuum on the left side of the rule. The vacuum cannot be on the right side of a rule. It is created automatically when the region becomes empty.

From certain point of view, vacuum can be seen as a sort of inhibitor. For example food is often packed in vacuum packing in order to delay / inhibit food decay.

5 Trading maximal parallelism for inhibitor usage

The goal of this paper is to show that using inhibitors, the sequential P systems can achieve universality like maximal parallel variant. The proof is partially inspired by [3], where it is shown that Calculi of Looping Sequences (CLS) can simulate computation of a P system. CLS is a membrane model with string objects and sequential rules.

5.1 Context rules instead of cooperative rules

In the proof, we use cooperative rules, but for the convenience, we will write more than two objects at the left side of some rules.

Lemma 5.1 *For sequential P system, the variant with cooperative rules and the variant with context rules are equal if at most one reactant in the context rule is consumed in other rewrite rule.*

Proof 5.1 Consider rule $a_1|a_2|\dots|a_n \to v$, where $a_1,\dots a_n$ are objects and v is a product of the rule, which may include sending object up/down through the membrane. This rule can be simulated with multiple sequential steps:

- $a_1|a_2 \to a_{1,2}$
- $a_{1,2}|a_3 \to a_{1,2,3}$
- $\dots$
- $a_{1,2,\dots n-1}|a_n \to v$

There could be a problem with a case when these multiple sequential steps would be interrupted in the middle and the rewriting would be left incomplete. Therefore, we have the additional condition of having at most one reactant being used and consumed in other rule. That reactant will be referred as a_1. So after the first step is executed, there is no way to be interrupted.

5.2 Inhibitor set

Original definition of P system with inhibitors (see [10]) have only a single inhibitor object per rule. P systems with inhibitor set can have rules of type $u \to v|_{\neg B}$, where $B \subseteq V$. The rule $u \to v$ is active only if there is no occurrence of any symbol from B.

Lemma 5.2 will establish equivalence of these two definitions of rewriting rule when special precondition is fulfilled - there is no empty region.

Lemma 5.2 *If there is at least one object present in each region of a P system, rewriting step in P system with inhibitor set can be simulated by multiple consecutive steps of P system with single inhibitor.*

Proof 5.2 Consider P system with alphabet V and a set of inhibitors $B = \{b_1, b_2, \dots, b_n\}$. For each rule $u \to v|_{\neg B}$ we will have rules:

- $c \to c|GONE_b|_{\neg b}$ for all $c \in V, b \in B$
- $u|GONE_{b_1}|GONE_{b_2}|\dots|GONE_{b_n} \to v|GONE_{b_1}|GONE_{b_2}|\dots|GONE_{b_n}$

5.3 Simulation of maximal parallel P system

The last lemmas (5.1 and 5.2) will simplify the proof of the following theorem.

Theorem 5.3 *The sequential P system with inhibitors defines the same Parikh image of language as P system with maximal parallelism.*

Proof 5.3 We will prove the theorem by simulation. The proof is quite technical with some workarounds. For every maximal parallel P system Π we will find a sequential P system with inhibitors $\overline{\Pi}$ such that for every computation of Π there exists a computation of $\overline{\Pi}$ with the same result. The computation of Π is a sequence of maximal parallel steps. In each step of Π, a maximal multiset of rewriting rules are applied in each membrane. We simulate this maximal parallel step with several steps of sequential system $\overline{\Pi}$.

We must prevent rule application from the next maximal parallel step as that would be an incorrect simulation. Preventing this will be done with synchronization. Every membrane will have a state, represented as an object. The most important states are RUN and $SYNCHRONIZE$. In the RUN state, rewriting rules of Π are applied until there is no rule to be applied. Then, the membrane proceeds to the $SYNCHRONIZE$ state. All the membranes in the $SYNCHRONIZE$ state are waiting for other membranes to reach $SYNCHRONIZE$ state, so they all can proceed to the RUN state of the next maximal parallel step of Π.

Other states are just technical - we need to implement sending objects between membranes and preparing for the next maximal parallel step by clearing temporary objects.

More detailed description of states:

- RUN: Rewriting occurs. To prevent double rewriting, we mark the products of rules with a' instead of a. Objects that are to be sent to the parent membrane are directly sent because the parent membrane is already in RUN or $SYNCHRONIZE$ state (see Figure 1), so the a' symbols that are sent don't break anything.

  But objects that are to be sent down, can't be sent immediately because child membranes can be in the $RESTORE$ state restoring symbols from the previous maximal parallel step. Current symbols could interfere with them and they could be rewritten twice in this step. Such objects are only marked as "to be sent down": $a^{\downarrow\prime}$.

  When RUN phase ends (in the membrane i), the $SYNCTOKEN_i$ is sent upwards to notify the skin membrane that the membrane i is ready to be synchronized.

- $SYNCHRONIZE$: Rewriting has ended and membrane is now waiting to receive signal $SYNCED$ from the skin membrane to continue to the next state.

- $SENDDOWN$: Signal $SYNCED$ has been caught and now all descendant membranes are in the $SYNCHRONIZE$ state. So objects $a^{\downarrow\prime}$ can be sent down.

- $RESTORE$: All a' symbols are restored to a and other temporary symbols are cleared, so the next maximal parallel step can take place.

In the figure 1 the pairs of states of parent and child membrane are joined with possible transitions between two consecutive global synchronizations - after the maximal parallel steps i and $i+1$.

In the figure 2 the membrane structure is presented as hierarchical structure. Every region is in one of four states. It is obvious that the sending of the objects is performed in such states that the receiving region is in RUN or $SYNCHRONIZE$ state, so the received objects (marked a') cannot interfere with rewriting.

It is important to note that when a region is in $SENDDOWN$ state and objects are sent through the child membrane, the receiving region is in the $SYNCHRONIZE$ state waiting for the $SYNCED$ signal, which will be sent to it when $SENDDOWN$ and $RESTORE$ phases finished.

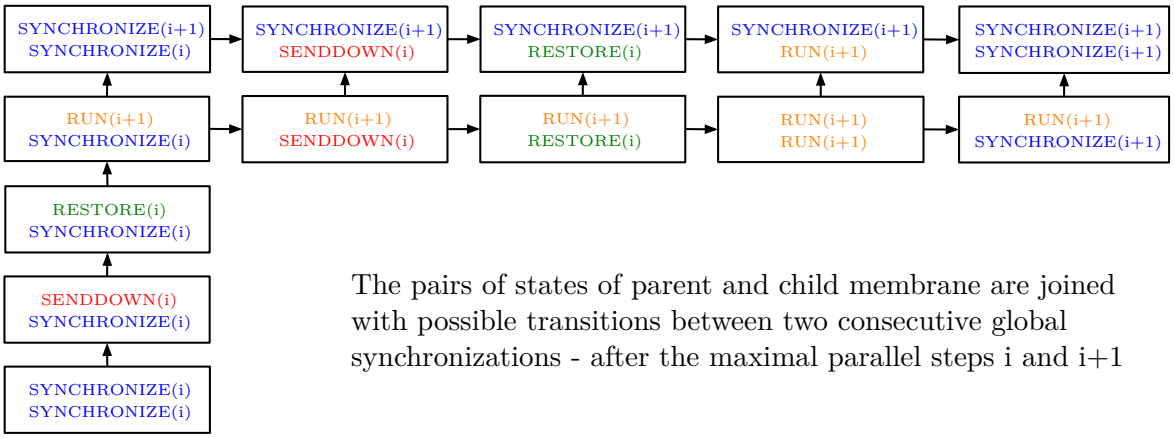

Figure 1: Possible pairs of states of parent and child membrane

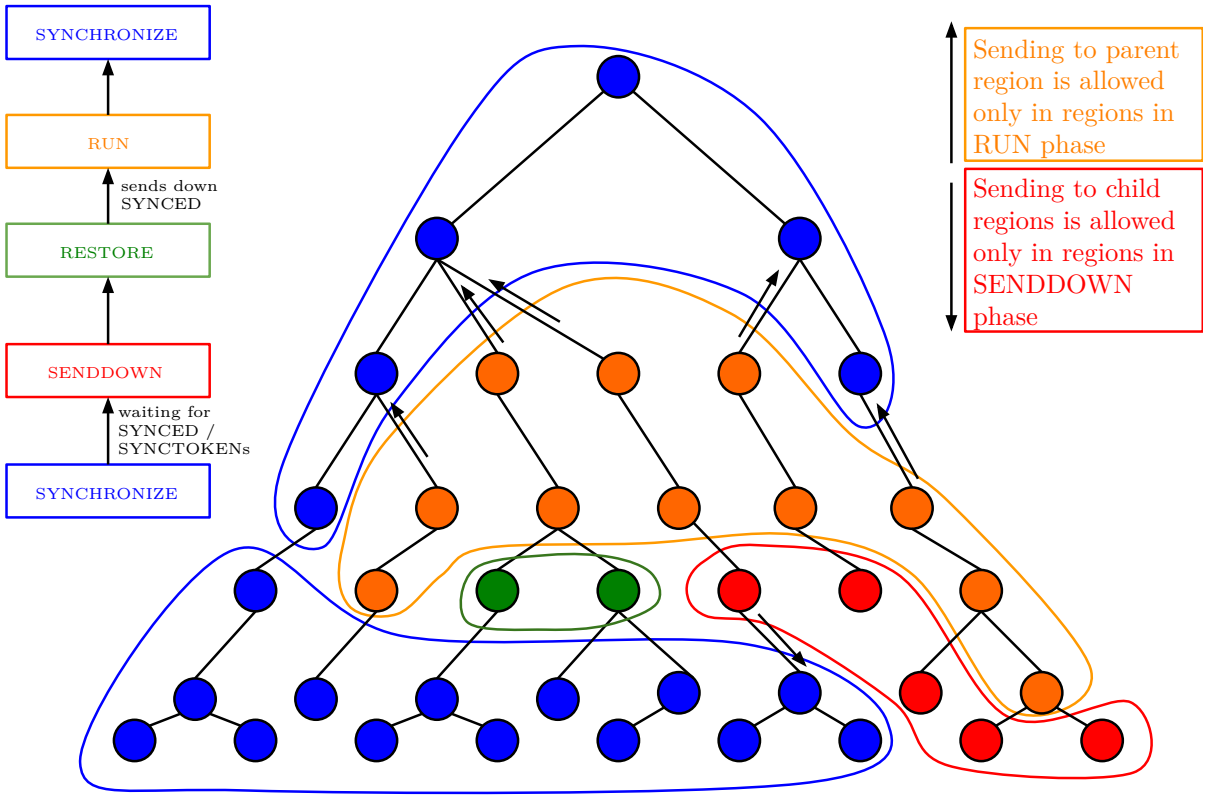

Figure 2: Snapshot of all membrane states while simulating

The rewriting rules in $\overline{\Pi}$ fire in a sequential manner. They must have at most two objects at the left side. If a rule has more than two objects at the left side, we recall the lemma 5.1 and discuss the preconditions. They may contain inhibitors. Every time it contains more than one inhibitor, we recall the lemma 5.2 and discuss the preconditions.

Here follows the list of rules in $\overline{\Pi}$.

- For every rule $r_i \in R$ such that $r_i = a_1^{M(a_1)} a_2^{M(a_2)} \dots a_n^{M(a_n)} \to a_1^{N(a_1)} a_2^{N(a_2)} \dots a_n^{N(a_n)}$ we will have the following rules:
  $a_1^{M(a_1)-m_1} \dot{a}_1^{m_1} a_2^{M(a_2)-m_2} \dot{a}_2^{m_2} \dots a_n^{M(a_n)-m_n} \dot{a}_n^{m_n} | RUN \to a_1'^{N(a_1)} a_2'^{N(a_2)} \dots a_n'^{N(a_n)} | RUN$
  There will be such rule for each $0 \leq m_j \leq M(a_j)$. Right side of the rules contains symbols a', that prevents the symbols to be rewritten again. Usage of symbols $\dot{a}$ in these rules represents the idea that they can be used in rewriting in the same way as a. Their purpose will be explained later in the proof.

- For every symbol $a \in V$ we will have the following rules:
  $a | RUN \to \dot{a} | RUN |_{\neg \dot{a}}$
  There will be max one occurrence of $\dot{a}$.

- For every rule $r_i \in R$ there will be a rule that detects if the rule r_i is not usable. According to left side of the rule r_i, symbol $UNUSABLE_i$ will be created when there is not enough objects to fire the rule r_i. It means that left side of rule r_i requires more instances of some object than are present in membrane.
  If the left side is of type:
  - a: It is a context free rule. The rule can't be used if there is no occurrence of a nor $\dot{a}$.
    $RUN \to UNUSABLE_i | RUN |_{\neg \{UNUSABLE_i, a, \dot{a}\}}$
  - ab: It is a cooperative rule with two distinct objects on the left side. The rule can't be used if there is one of them missing.
    $RUN \to UNUSABLE_i | RUN |_{\neg \{UNUSABLE_i, a, \dot{a}\}}$
    $RUN \to UNUSABLE_i | RUN |_{\neg \{UNUSABLE_i, b, \dot{b}\}}$
  - a^2: It is a cooperative rule with two same objects. The rule can't be used if there is at most one occurrence of the symbol. That happens if there is no occurrence of a. There can still be $\dot{a}$, but at most one occurrence.
    $RUN \to UNUSABLE_i | RUN |_{\neg \{UNUSABLE_i, a\}}$

- For every membrane with label i there will be rule:
  $UNUSABLE_1 | \dots | UNUSABLE_m | RUN \to SYNCHRONIZE | SYNCTOKEN_i \uparrow$
  If no rule can be used, maximal parallel step in the region is completed so it goes to synchronization phase and sends a synchronization token to the parent membrane. The objects $UNUSABLE_i$ are not consumed in other rules, so by Lemma 5.1 this rule can be written with set of cooperative rules.

- For every membrane other than the skin membrane, there will be a rule:
  $SYNCHRONIZE | SYNCTOKEN_j \to SYNCHRONIZE | SYNCTOKEN_j \uparrow$
  Membrane forwards all sync token to parent membrane.

- In the skin membrane there is a rule which collects all the synchronization tokens from all membranes $1 \dots k$ and then sends down signal that synchronization is complete. But before that, there can be some symbols that should be sent down, but they weren't, because the region below could have not started the rewriting phase that time. The result was just marked with $a^{\downarrow\prime}$.
  $SYNCTOKEN_1 | \dots | SYNCTOKEN_k | SYNCHRONIZE \to SENDDOWN$
  The objects $SYNCTOKEN_i$ are not consumed in other rules in the skin membrane, so by Lemma 5.1 this rule can be written with set of cooperative rules.
- Every membrane other than skin membrane have to receive the signal to go to senddown phase:
  $SYNCHRONIZE | SYNCED \to SENDDOWN$
- Every membrane will have rules for every symbol $a \in V$ to send down all unsent object that should have been sent down:
  $SENDDOWN | a^{\downarrow\prime} \to SENDDOWN | a' \downarrow$
- Every membrane will have rule for detecting when all such objects have been sent and it goes to restore phase:
  $SENDDOWN \to RESTORE|_{\neg\{a_i^{\downarrow\prime} | 1 \le i \le n\}}$
- In restore phase all symbols a' will be rewritten to a in order to be able to be rewritten in next maximal parallel step:
  $RESTORE | a' \to RESTORE | a$
- There may be some $GONE$ symbols left, which should be cleared too:
  $RESTORE | GONE_i \to RESTORE$
- When restore phase ends, it sends down signal that all membranes have been synchronized and next phase of rewriting has began in upper membranes:
  $RESTORE \to RUN | SYNCED \downarrow |_{\neg\{a_i' | 1 \le i \le n\} \cup \{GONE_i | 1 \le i \le n\}}$

The regions are never empty, because they contain at least the object representing the state. Thus the rules with set of inhibitors can be simulated by single inhibitors due to the Lemma 5.2.

5.4 Register machines

We will use in our paper the power of Minsky's register machine [10], 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 [8] for some original definitions and [11] for the definition used in this paper).

Definition 5.1 *A* n**-register machine** *is a tuple* $M = (n, P, i, h)$*, 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 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 $(n_1, \ldots, n_m) \in N_0^m$ in registers 1 to m, which is recognized if the register machine finally stops by the halt instruction with all its registers being empty (this last requirement is not necessary). If the machine does not halt, the analysis was not successful.

5.5 Accepting vs generating

According to [2], a P system can be used either as an acceptor or as a generator of a multiset language over Σ. In the first case, a multiset over Σ is inserted in the skin membrane of the P system and the result of its computations says whether such a multiset belongs to the multiset language accepted by the P system or not. In the second case the P system has a fixed initial configuration and can give as results (possibly in a non-deterministic way) all the possible multisets belonging to a given multiset language.

5.6 Universality results for accepting case

Theorem 5.4 *Sequential P systems with inhibitors can simulate register machines and thus equal PsRE.*

Proof 5.4 Suppose we have a n-register machine $M = (n, P, i, h)$. In our simulation we will have a membrane structure consisting of one membrane and the contents of register j will be represented by the multiplicity of the object a_j.

We will have P system (V, μ, w, R), where:

- V is an alphabet consisting of symbols that represent registers $a_1, \ldots a_n$ and instruction labels in $Lab(M)$,
- μ is a membrane structure consisting of only one membrane,
- w is initial contents of the membrane. It contains symbols for the input for the machine $a_i^{n_i}$ where n_i is initial state of register with label i and initial instruction $e \in Lab(M)$.
- R is the set of rules in the skin membrane:

  For all instructions of type $(e : add(j), f)$ we will have rule:

  · $e \to a_j | f$.

  For all instructions of type $(e : sub(j), f, z)$ we will have rules:

- $e|a_j \to f$ and
- $e \to z|_{\neg a_j}$.

And finally halting rules:

- $h|a_j \to h|\#$ for all $a \leq j \leq n$,
- $\# \to \#$,

When the halting instruction is reached, if there is an object present in the membrane, the hash symbol # is created and it will cycle forever. If there is no object present, there is no rule to apply and computation will halt. It corresponds to the condition that all registers should be empty when halting.

6 Trading maximal parallelism for vacuum

In this section we will research how the vacuum object can help achieving universality.

Theorem 6.1 *The sequential P system with Vacuum is universal.*

Proof 6.1 We can simulate the variant of P system where the only cooperative rule is of type $a|a \to b$. According to [9] the variant, where the only cooperative rule is when both objects are the same, is universal. If there is no rule $a \to b$, we can rewrite a to a' so we can mark all present symbols. a' symbols are kept in special membrane so the Vacuum can be created in main membrane and we can synchronize.

But we won't do this madness again.

Instead, we will try to prove universality by simulating the register machine. We need to detect when the current register is empty. If there was a symbol for every register as in the proof 5.4, the Vacuum would be created only if all registers are empty. But the $sub()$ instruction need to detect when one concrete register is empty.

We will have a membrane for each register. That membrane will be contained in the skin membrane. The number of objects in membrane i will correspond to the value of register i.

The alphabet will consist of instruction labels and register counter a. The skin membrane will only have an instruction label. It is sent to corresponding membrane where the instruction is executed. Then, the following instruction is sent back to the skin membrane.

We will have following rules in the skin membrane:

- $e \to e\downarrow_j$ for an instruction of type $e : add(j), f$ or $e : sub(j), f, z$ and
- $h \to h\downarrow$ for a halting instruction h.

And in non-skin membranes:

- $e \to a|f\uparrow$ instructions of type $e : add(j), f$,
- $e|a \to f\uparrow$ for instructions of type $e : sub(j), f, z$,
- $e|VACUUM \to z\uparrow$ for instructions of type $e : sub(j), f, z$, and
- $h|a \to h|a$

When halting, if there is an nonempty register, it will cycle forever with the last rule. However, if all registers are empty, the halting instruction label will stay in all membranes and the computation will halt.

7 Conclusion

We have studied several variants of sequential P systems in order to obtain universality without using maximal parallelism. A variant with rewriting rules that can use inhibitors was shown to be universal in both generating and accepting case. The generating model is able to simulate maximal parallel P system and the accepting model can simulate a register machine.

In addition, we have defined a new notion of vacuum, which is immediately created in the region that becomes empty. A new P system variant was introduced, which allowed the vacuum to be used on the left side of P system rewriting rules. The accepting case of this variant was shown to be universal by direct implementation of a register machine.

A lot of other variants deserve to be combined with the vacuum, so we suggest to research them more thoroughly(non-cooperative rules, rules with priorities, decaying objects, deterministic steps, ...).

References

[1] Artiom Alhazov & Rudolf Freund (2012): *Asynchronous and Maximally Parallel Deterministic Controlled Non-Cooperative P Systems Characterize $NFIN \cup coNFIN$*. In: *13th International Conference on Membrane Computing (CMC13)*, CMC'13, MTA SZTAKI, pp. 87–98.

[2] Roberto Barbuti, Andrea Maggiolo-Schettini, Paolo Milazzo & Simone Tini (2010): *Membrane systems working in generating and accepting modes: expressiveness and encodings*. In: *Proceedings of the 11th international conference on Membrane computing*, CMC'10, Springer-Verlag, Berlin, Heidelberg, pp. 103–118. Available at `http://dl.acm.org/citation.cfm?id=1946067.1946081`.

[3] Roberto Barbuti, Paolo Milazzo & Angelo Troina (2007): *The Calculus of Looping Sequences for Modeling Biological Membranes*. In: *8th Workshop on Membrane Computing (WMC8), LNCS 4860*, Springer, pp. 54–76.

[4] Matteo Cavaliere, Mihai Ionescu & Tseren-Onolt Ishdorj (2005): *Inhibiting/de-inhibiting rules in P systems*. In: *Proceedings of the 5th international conference on Membrane Computing*, WMC'04, Springer-Verlag, Berlin, Heidelberg, pp. 224–238, doi:10.1007/978-3-540-31837-8_13. Available at `http://dx.doi.org/10.1007/978-3-540-31837-8_13`.

[5] Gabriel Ciobanu, Linqiang Pan, Gheorghe Pun & Mario J. Pérez-Jiménez (2007): *P systems with minimal parallelism. Theor. Comput. Sci.* 378(1), pp. 117–130, doi:10.1016/j.tcs.2007.03.044. Available at `http://dx.doi.org/10.1016/j.tcs.2007.03.044`.

[6] Zhe Dang & Oscar H. Ibarra (2004): *On P systems operating in sequential mode*. In: *International Journal of Foundations of Computer Science*, pp. 164–177.

[7] Rudolf Freund, Alberto Leporati, Marion Oswald & Claudio Zandron (2005): *Sequential P systems with unit rules and energy assigned to membranes*. In: *Proceedings of the 4th international conference on Machines, Computations, and Universality*, MCU'04, Springer-Verlag, Berlin, Heidelberg, pp. 200–210, doi:10.1007/978-3-540-31834-7_16. Available at `http://dx.doi.org/10.1007/978-3-540-31834-7_16`.

[8] Oscar H. Ibarra, Sara Woodworth, Hsu-Chun Yen & Zhe Dang (2005): *On sequential and 1-deterministic P systems*. In: *Proceedings of the 11th annual international conference on Computing and Combinatorics*, COCOON'05, Springer-Verlag, Berlin, Heidelberg, pp. 905–914, doi:10.1007/11533719_91. Available at `http://dx.doi.org/10.1007/11533719_91`.

[9] Oscar H. Ibarra, Hsu chun Yen & Zhe Dang (2004): *Dang: The Power of Maximal Parallelism in P Systems*. In: *Proceedings of the Eight Conference on Developments in Language Theory*, Springer, pp. 212–224.

[10] Mihai Ionescu & Dragos Sburlan (2004): *On P Systems with Promoters/Inhibitors*. *Journal of Universal Computer Science* 10(5), pp. 581–599. Available at `http://www.jucs.org/jucs_10_5/on_p_systems_with`.

[11] S. Khrisna & A. Păun (2003): *Three Universality Results on P Systems, Workshop on Membrane Computing WMC-Tarragona 2003. A. Alhazov, C. Martín-Vide, G. Păun, eds),TR 28/03, URV Tarragona*.

[12] Gheorghe Păun (2000): *Computing with Membranes*. *Journal of Computer and System Sciences* 61(1), pp. 108 – 143, doi:10.1006/jcss.1999.1693. Available at `http://www.sciencedirect.com/science/article/pii/S0022000099916938`.

[13] Gheorghe Păun (2006): *Introduction to Membrane Computing*. In Gabriel Ciobanu, Gheorghe Păun & MarioJ. Pérez-Jiménez, editors: *Applications of Membrane Computing*, Natural Computing Series, Springer Berlin Heidelberg, pp. 1–42, doi:10.1007/3-540-29937-8_1. Available at `http://dx.doi.org/10.1007/3-540-29937-8_1`.

[14] Dragos Sburlan (2005): *Promoting and Inhibiting Contexts in Membrane Computing*. Ph.D. thesis, University of Seville.

[15] Dragoş Sburlan (2006): *Further Results on P Systems with Promoters/Inhibitors*. *International Journal of Foundations of Computer Science* 17(01), pp. 205–221, doi:10.1142/S0129054106003772. Available at `http://www.worldscientific.com/doi/abs/10.1142/S0129054106003772`.