Universiteit Leiden
Opleiding Informatica

Transformation of Membrane Systems

Name: Dennis Roos

Date: 28/08/2016

1st supervisor: Hendrik Jan Hoogeboom

2nd supervisor: Jetty Kleijn

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)

Leiden University

Niels Bohrweg 1

2333 CA Leiden

The Netherlands

Transformation of Membrane Systems

Dennis Roos

Abstract

We study transformations between different types of membrane systems. Membrane systems are a model of computation inspired by biological cells. For many classes of membrane systems, also known as P systems, it is known that they are computationally complete. Therefore, we know that these P systems are equally computationally powerful, so they should be able to emulate each other. We have algorithms for simulating Turing machines using these different classes of membrane systems. We can also simulate P systems using Turing machines, so we could always convert a P system of a certain type to a P system of another type by first converting to a Turing machine, then simulating that Turing machine. However, this process is slow and inefficient.

In this paper we search for algorithms and procedures to directly convert P systems of different types to each other. This will also illustrate the similarities and differences between different classes of P systems. We present algorithms to transform minimally and bounded parallel P systems, and communication P systems.

Contents

Chapter 1

Introduction

Computer scientists have often been inspired by nature and biological phenomena to create new algorithms and techniques. From DNA computing to evolutionary algorithms and ant-colony pathfinding: animals and plants have often found clever algorithms we can use in computing.

Recently, another biological phenomenon has come into focus in the field of theoretical computer science. In the relatively new field of *membrane computing* biological cells are the inspiration for a very different way of computing. In cell biology, a cell contains small rooms or reactors, in which the conditions are optimal for allowing specific chemical reactions to take place. For example, cells contain mitochondria, which convert sugars and fats into 'fuel molecules', which are then sent to other parts of the cell that need energy. A cell can contain multiple nested membranes, and the entire cell itself can also be seen as a membrane. Cells interact with the rest of the body by absorbing nutrients from the outside and ejecting produced molecules.

Membrane computing was introduced by Gheorge Păun [6] in 2000. A complete overview of the field can be found in [7]. In membrane computing, we emulate biological systems by creating a series of membranes. Each membrane can apply specific transformation rules, representing the chemical reactions, to the objects contained in the membrane, representing molecules. These membranes are ordered in the form of a hierarchical tree with the root membrane called a skin membrane. The skin is the outer wall of our little 'cell computer'. We can import and output objects from the skin membrane to the environment, which represents the outside body.

The full details of how membrane systems work are explained in Chapter 2, but for now we can say that these systems are computationally complete [4]. There are various different types of membrane systems, such as dynamic systems which can create or dissolve membranes during computation, or communication systems, which are very limited in the rules they can apply. These different classes of membrane systems are similar to the distinction between different classes of Turing machines, for instance. There are Turing

machines with multiple tapes, nondeterministic Turing machines, universal Turing machines, etc. Each class is computationally complete, yet they are all distinct.

In this paper we look into direct ways of having different classes of membrane systems simulate each other. Given a membrane system of class A, we want to create a system of class B which functions in the same way. We already know this is possible, because each class is, by itself, computationally complete. We know that these different classes of membrane systems are computationally complete because researchers have managed to simulate Turing machines with them [7]. Or, more commonly, they manage to simulate a register machine, which has been proven to be computationally complete. So we have algorithms that allow us to create a membrane system to simulate a Turing machine, by simulating a register machine which simulates a Turing machine. Now, because we can design a Turing machine which can simulate a membrane system, we can already have membrane system A simulate system B, simply by simulating the Turing machine simulating B.

However, this is a hard and inefficient process. Membrane systems function very differently from Turing machines, with membranes working in applying rules in parallel, using different rulesets and objects. This is in stark contrast to the sequential single-tape Turing machine. Membrane systems are inherently nondeterministic, so we need a nondeterministic Turing machine to simulate them. Then we need to simulate this nondeterministic Turing machine using a multi-tape Turing machine to make it deterministic. Then we need to create a single-tape Turing machine that simulates the multi-tape Turing machine. Then we need to create a register machine that simulates this Turing machine, and finally we can create our membrane system of class A that simulates the register machine. So we have 5 levels of simulation, and each level increases the size of the resulting program exponentially. Membrane system A might be only a few membranes big, but the resulting system B could be thousands or even millions of membranes big, making it useless for practical purposes.

In this paper we present algorithms for letting different classes of membrane systems simulate each other directly in a process we call *transformation*. Because different classes of membrane systems function much more like each other than they do like Turing machines, this direct process creates systems that are much smaller, more efficient, and more closely resemble the original structure than anything the Turing machine method would create.

In Chapter 3 we present ways of simulating different levels of *parallelism*, a concept explained in Chapter 2. In Chapter 4 we present algorithms that link two distinct classes of membrane systems: Regular P systems and Communication Systems.

Finally, in Chapter 5 we summarize our findings and discuss other avenues of research into this topic.

Chapter 2

Definitions

2.1 Multisets

For a set S, we define a *multiset* over S as a mapping $U : S \to \mathbb{N}$, where $\mathbb{N}$ is the set of nonnegative integers.

If the set S is finite, $S = \{a_1, a_2, ..., a_n\}$, we can describe this multiset U using a string $a_1^{U(a_1)} a_2^{U(a_2)} ... a_n^{U(a_n)}$. For instance, if $S = \{a, b, c\}$, and a occurs twice in the multiset U, b four times, and c once, we can write $U = a^2b^4c$. Note that a multiset is unordered, and this representation as an ordered string is simply used for ease of reference. In the rest of this paper, we refer to individual elements of multisets as objects.

A multiset U is a *subset* of a multiset U', written as $U \subseteq U'$, if for all $a \in S$: $U(a) \leq U'(a)$.

2.2 Membrane Structures

In membrane computing the concept of a *membrane* is central. In cell biology, a membrane can be described as a sealed-off room or reactor inside a cell, in which conditions are suitable for chemical reactions that may not occur elsewhere in the cell. Membranes themselves may contain one or more other membranes, and we can see a cell itself as a membrane.

Mathematically, we define a *membrane structure* as a node-labeled tree of membranes, where each child node is a membrane contained in the membrane denoted by the parent node. We typically draw membranes as nested rectangles, where each rectangle is a membrane (See Figure 2.1).

More precisely, we say a membrane m is a *child membrane* of membrane m', or is a *lower neighbour* of m', if the node m is a child node of m'. We call m' the *upper neighbour* of m.

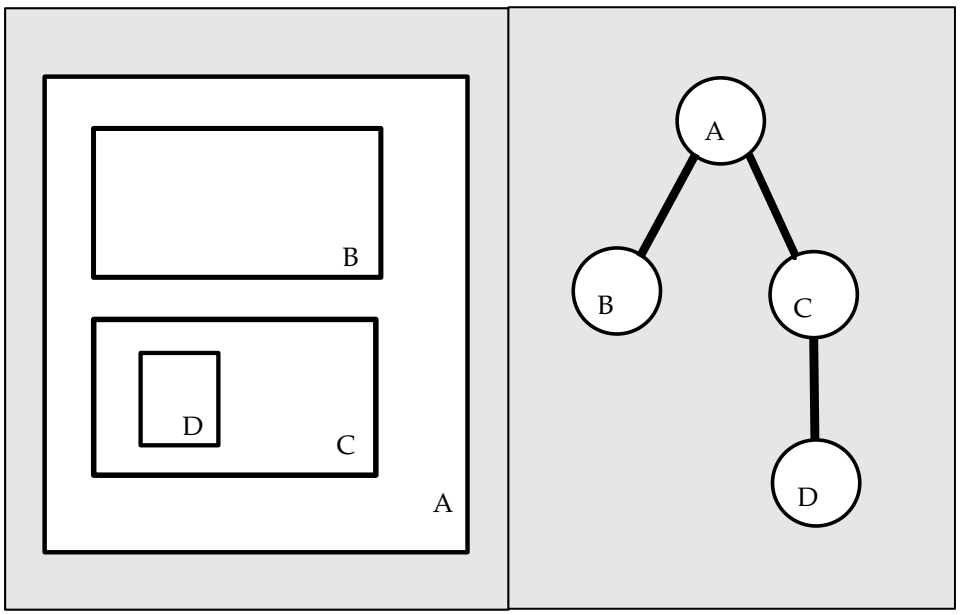

Figure 2.1: A diagram of a membrane structure, both in graphical form (left) and in tree form (right). A is the skin membrane, and the upper neighbour of B and C, which are lower neighbours of A. B and D are elementary membranes. B and C are sibling membranes. A and C are on the path to D.

We say different membranes m and m' are *sibling membranes* if there exists a membrane m'' such that m'' is the upper neighbour for both m and m'. A membrane with no lower neighbours is called an *elementary membrane*, and a membrane with no upper neighbour is called a *skin membrane*. A membrane structure contains exactly one skin membrane, the root of the tree. The space 'outside' of the skin membrane is called *the environment*, which can have some special rules for different types of membrane systems.

Given any membrane m, we define the *path* from the skin membrane to i as the set of membranes $\{m_1, m_2, ..., m_n\}$
where m_1 is the skin membrane,
m_i is the upper neighbour of m_{i+1} for $1 \leq i \leq n-1$ and
m_n is the upper neighbour of m.

We define the *degree* of a membrane structure as the number of membranes it contains.

2.3 Multiset Rewriting Rules

We define a *multiset rewriting rule* (hereafter simply referred to as a *'rule'*) as a construct
$x \to y$
where x and y are strings representing multisets over a given alphabet O. Every membrane m_i has a multiset

s_i associated with it during any point of the computation.

We say we can *apply* a rule $x \to y$ to a membrane m_i if $x \subseteq s_i$. Each membrane has a finite set of rules associated with that membrane.

In most types of P systems, every object that is produced has a target indication. For a rule $x \to y$, we define y as a string over $O \times \{here, out, in\}$.
Thus, each element of y is of the form $(a, target)$ where a is an element of O, and $target$ is either $here$, in, or out.

When a rule $x \to y$ is applied in membrane m with associated multiset s, the multiset x is removed from s, and the following actions happen:
For all objects a in y with $tar = here$, a is added to s.
For all objects a in y with $tar = out$, a is added to the multiset of m's upper neighbour. If the upper neighbour is the environment, the outcome depends on the type of membrane system. In the next sections this will be explained further.
For all objects a in y with $tar = in$, a lower neighbour of m is nondeterministically chosen, and a is added to the multiset of this lower neighbour.

As an example, we can have the rule $a^2bc^3 \to (ab^2, here)(c, out)(a^2c^5, in)$
When this rule is applied, a^2bc^3 is removed from the membrane, but ab^2 is immediately returned to the membrane. The object c is sent to the upper neighbour, and for each of the elements in a^2c^5, a lower neighbour is randomly selected as a target. Note that this means that if there are multiple lower neighbours, all elements can end up in the same membrane, in different ones, or some combination of the two.

For compactness' sake, we often omit the *here* part of this rule, and we will write the rule above as follows:
$a^2bc^3 \to ab^2(c, out)(a^2c^5, in)$

Note that elementary membranes can have no rules sending objects in, since they have no lower neighbours. If a skin membrane sends objects out, they are send to the environment, where special rules apply. These rules are explained further in the sections on specific Membrane Systems.

We say a multiset of rules $R = \{x_1 \to y_1, x_2 \to y_2, ..., x_n \to y_n, \}$ for some positive integer n can be *applied* to a membrane m containing the multiset s if $\bigcup_{i=1}^{n} x_i \subseteq s$. In other words, if there are enough objects in m to satisfy all rule combined. Note that since we defined R as a multiset, the same rule can occur multiple times.

2.4 Computational Steps: Maximal and Bounded Parallelism

We have two different systems for applying rewriting rules to membrane systems: *maximal* and *bounded parallelism*.

All membrane systems run on an global clock consisting of an infinite number of time cycles, called *steps*. During each step rules are applied. When a rule $x \to y$ is applied during step t, the objects x are immediately removed from the membrane, and at the start of step $t+1$, the objects from y are deposited in their respective membrane targets, and can be used for rules during step $t+1$. Each element can only be used for at most one rule per step.

In other words, during each computation step membranes choose a set of rules to apply to the objects they contain. The left-hand side of each rule is consumed, and the right-hand side is added to the targeted membranes at the start of the next step.

Each membrane chooses a set of rules to apply during each step in the following way: Let $R = \{r_1, r_2, ..., r_n\}$ be the set of rules associated with the membrane m containing the multiset s. There are a finite number of multisets of rules from R that can be applied to s. Now we define a *maximal ruleset* as an applicable ruleset M over R where for all $r_i \in R, M \cup r_i$ can not be applied to s. In other words, we define a ruleset as maximal if no other rule can be added to it without invalidating its applicability.
In a *maximally parallel* system, each step, each membrane nondeterministically chooses a maximal ruleset to apply.

In a *bounded parallel* system [1], each rule r_i has a multiplicity $t_i \in \mathbb{N}$ associated with it. Under this system, a ruleset M has the extra constraint that for all rules r_i, $M(r_i) \leq t_i$ to be applicable. In other words, the system can still choose to apply as many rules as possible, but each rule cannot be applied more times per step than its multiplicity.

Whereas in maximally parallel systems the rules can be applied as long as there are objects to consume, in a bounded parallel system there is a hard limit on the size of the rulesets, no matter how many objects there are in the system.

2.5 Generic Membrane Systems

In this section we present the *generic membrane system*, hereafter referred to as a *P system*. These generic membrane systems were the first versions of membrane systems, and are still the most commonly studied systems today.

We define a P system of degree $d \geq 1$ as a construct

$\Pi = (O, T, \mu, w_1, ..., w_d, R_1, ..., R_d, i_0)$

where:

O is a finite alphabet of *objects*.

$T \subseteq O$ is the set of *terminal objects*.

μ is a *membrane structure* $m_1, m_2, ..., m_d$ of degree d.

$w_1, ..., w_d \in O^*$ are the multisets of objects contained in the initial configuration of Π in membranes $m_1, ...m_d$.

$R_1, ..., R_d$ are the finite sets of multiset rewriting rules associated with the d membranes of μ.

$i_0 \in \{1, 2, ..., d, e\}$ is the *output region*.

The system halts when, during a single step, none of the membranes could apply at least one rule. The system's output is defined as the multiset of objects from T present in the designated membrane at the time of halting. If $i_0 = e$, the output region is the environment, and the *output* is defined as the multiset of all objects in T sent out to the environment over the course of the computation.

Each step, the system applies rules as outlined in the previous section. A system is either maximally or bounded parallel, and behaves accordingly in each step. Each application of rules results in a new set of multisets of objects in the membranes. Such a set of multisets is called a *configuration*. Each step, the system's application of rules creates a new configuration. Note that due to the nondeterminism of the application of rules, a single configuration can have a large (but finite) number of possible configurations the systems can reach in the next step.

2.6 Communication Membrane Systems

Communication systems, also known as symport/antiport systems, are a variation on normal P systems, introduced in 2002 [4]. While a P system can only apply rules to its own contents, a communication system's rules can be applied to the contents of both a membrane and its upper neighbour. However, a communication system can not discard or create objects when sending objects between membranes, It can only attain new objects by importing them from the environment, where a specific subset of all the objects are in unlimited supply.

Communication systems have the advantage that they are more modular and more closely resemble the behaviour of actual cells.

We define a *Communication Membrane System* of degree $d \geq 1$ as a construct:

$\Pi = (O, E, T, \mu, w_1, ..., w_d, R_1, ..., R_d, i_0)$

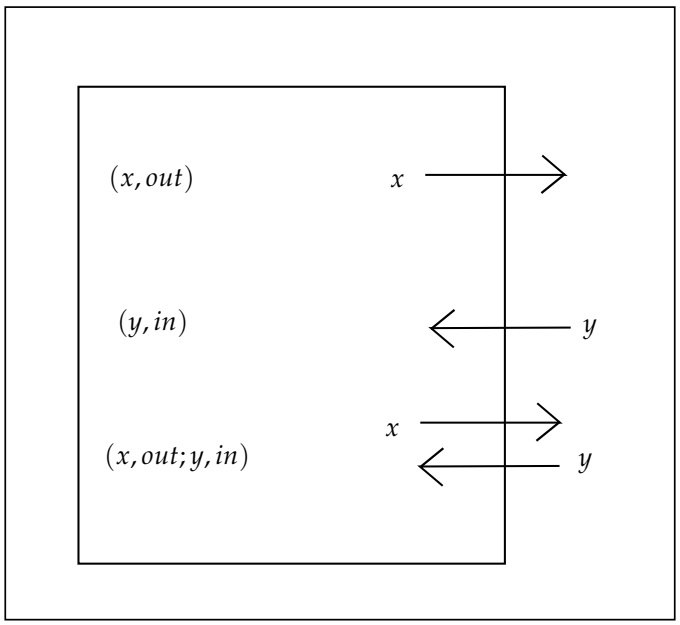

Figure 2.2: The different rules in a communication system, and how they send objects between a membrane and its upper neighbour

where:

O is a finite alphabet of *objects*.

$E \subseteq O$ is the set of *environment objects* that are in unlimited supply in the environment.

$T \subseteq O$ is the set of *terminal objects*. μ is a *membrane structure* of degree d.

$w_1, ..., w_d \in O^*$ are the multisets of objects contained in the initial configuration of Π in membranes $m_1, ...m_d$.

$R_i, 1 \leq i \leq d$ are finite sets of rules of the form $(x, out; y, in)$, with $x, y \neq \lambda$ (antiport rule), (x, out) or (x, in) with $x \neq \lambda$ (symport rules).

i_0 is the *output region*.

An antiport rule $(x, out; y, in)$ in R_i exchanges the multiset x inside membrane i with the multiset y in the upper neighbour of i.

The symport rule (x, out) sends the multiset x from membrane i into the upper neighbour of i, and the symport rule (x, in) retrieves the multiset x from the upper neighbour of i and sends it to i. A rule can be applied if all the membranes involved contain enough objects necessary. For instance, a rule $(x, out; y, in)$ can only be applied if the membrane contains x and its upper neighbour contains y. A visual explanation of the different rules can be seen in Figure 2.2.

In regular P systems, the environment contains no elements, and any objects send out are considered 'lost'. In communication systems, however, the environment does contain elements, and has its own multiset of the objects in $(O \setminus E)$. All objects in E occur in an unbounded number and are not specified. Initially, the environment contains no elements other than those in E. The environment has no rules associated with it, and the elements can only be accessed through the application of rules in the skin membrane.

If i is the skin membrane, the rule (y, in) requires y to contain at least one element not in E, to prevent the rule being applied an infinite amount of times each step in a maximally parallel system, and to prevent eternal looping in a bounded parallel system.

Communication systems use either maximal or bounded parallelism. The system halts when no further rules can be applied anywhere in the system during a single step. If the environment is chosen as the output region, T must be disjoint from E, since it is impossible to count the objects from E present in the environment.

2.7 Equivalency

We say two membrane systems Π and Π' are *equivalent* when they have the exact same effective behaviour: Π has configuration c and can only progress to one of configurations $c_1, ..., c_n$ in the next computational step $\leftrightarrow$ if Π' has configuration $f(c)$ there is at least one computation in which it will progress to one of the configurations $f(c_1), ..., f(c_n)$ in a finite number of computational steps, for a given standard mapping of configurations f. It is possible that the system will loop forever trying to reach the next configuration, but there will always be at least one computation in which it does not do so. f keeps all objects in T intact, so the output is not affected. Because of the behaviour of membrane systems, equivalent systems will halt for the same input and produce the same output as well.

Chapter 3

Parallelism

3.1 Introduction

Although maximally parallel systems are studied the most, bounded parallel systems can be useful as well, and have their own advantages. For example, it can be easier to simulate certain problems or algorithms in a bounded parallel system, because there is greater control over the type and amount of rules applied during a computation step.

Therefore it can be useful to be able to be able to switch between the two, turning a maximally parallel system into an equivalent bounded one, and vice versa. In this chapter we present algorithms for doing this. The algorithm for turning a bounded system into a maximal system is fairly trivial, but still requires new objects.

The algorithm for simulating a maximal system using bounded parallelism is more complicated. This is because any bounded parallel system has a finite number of rules it can apply each step, whereas a maximal system is only limited by the number of objects it contains. Therefore it will take multiple steps to simulate a single step in the original system, during which we have to make sure that all the membranes remain synchronized and do not start the next step before all the membranes are done with the current step.

3.2 From Bounded to Maximal

In this section we will show a method for creating a P system using maximally parallel rules that functions identically to any given P system using bounded parallel rules. This is accomplished by introducing new objects necessary for the application of each rule, and in that way limit the amount of times a rule can be applied.

Given any bounded parallel P System of degree m

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

we first define the following sets:

for $1 \leq i \leq m$: $C_i = \{c_j | r_j \in R_i\}$.

In other words, for each rule r_j in a given membrane i we define an object c_j. All objects for a given membrane are then combined into the set C_i. Since these objects will not move between membranes, the C_i are not required to be disjoint.

Using this we define the following equivalent maximally parallel P system of degree m:

$\Pi' = (O', T, \mu, w'_1, ..., w'_m, R'_1, ..., R'_m, i_0)$ where:

$O' = O \cup C_1 \cup C_2 \cup ... \cup C_m$.

For $1 \leq i \leq m$, $w'_i = w_i c_1^{m_1} c_2^{m_2} ... c_k^{m_k}$,

where $r_1, ..., r_k$ are the rules associated with membrane i, and m_j is the multiplicity of the rule r_j. In other words, For each rule, we add objects associated with that rule, the number of which is equal to the multiplicity of the rule.

For each R_i in Π we define R'_i as follows. For each rule $r_j : x \to y \in R_i$, we add a rule $r'_j : c_j x \to (c_j, here)y$.

The new system Π' is equivalent to Π, because it directly simulates Π. For any given rule r_i in Π, Π' can only apply it m_i times, because it only has m_i amount of c_i objects in the membrane, and one is needed for every application of the rule. The addition of the c_i does not change the behaviour of the system in any other way, since they don't apply to any other rules, and don't leave the membrane they are associated with.

Halting is obviously done at the same time, and the output remains the same: The newly introduced symbols c_i are not a part of T, so they do not contaminate the output.

Complexity-wise, Π' is just as efficient as Π. It takes the exact same number of steps to complete a computation. It requires the same number of membranes and rules. The only change is to the size of the alphabet O and the initial configuration $w_1, ..., w_m$. This size increase depends on the amount of rules and their multiplicity, but overall this is a small price to pay.

If Π is context-free(the left-hand side of every rule consists of only one object), Π' does not preserve this quality. Therefore this transformation may not always be useful for transforming systems that depend on being context-free.

3.3 From Maximal to Bounded

In this section we do the opposite of the previous section: We take a maximally parallel system and create an equivalent bounded parallel system. This is more complicated than the other way around, since it is relatively easy to limit the amount of times a rule can be applied in a single step. However, when we want to simulate maximal parallelism we need to apply more rules than the bounded system would apply in a single step, all the while ensuring that the different membranes, which may need to apply a different amount of rules, stay in sync and do not start applying new rules before other membranes have finished their previous step.

In this section we call the sequence of steps that our new system has to take to simulate a single step in the old system a cycle.

First of all we define some specific sets we will use in this section.
Given an alphabet $O = \{\sigma_1, \sigma_2, ..., \sigma_n\}$ we define the alphabet $O_2 = \{\sigma'_1, \sigma'_2, ..., \sigma'_n\}$ as a disjoint copy of O.
For any multiset $x = x_1x_2...x_m \in O^*$ we define $x' \in O_2^*$ as $x' = x'_1x'_2...x'_m$

Given any maximally parallel P System of degree m
$\Pi = (O, T, \mu, w_1, ..., w_m, R_1, ...R_m, i_0)$, with the skin membrane having label 1,

we define the equivalent bounded parallel P system of degree m
$\Pi' = (O', T, \mu, w'_1, ..., w'_m, R'_1, ..., R'_m, i_0)$ where:

$O' = O \cup \{a_1, a_2, ..., a_m, b, b', c, d, d', e, f, g_1, g_2, ..., g_m, h, h', k, l\}$.
For $w'_i, 1 \leq i \leq m, w'_i = w_i a_i$.

The multiplicity of all rules in Π' is 1.
For $2 \leq i \leq m, R'_i$ is defined as follows:
For all rules $r_j : w \to x(y, out)(z, in) \in R_i$, we add the rules $r'_j : cw \to d'x'(y', out)(z', in)$ and
$r''_j : dw \to d'x'(y', out)(z', in)$.

The skin membrane gets the following rules for each rule $r_j : w \to x(y, out)(z, in) \in R_1$:
$r'_j : cw \to d'x'(y, out)(z', in)$ and
$r''_j : dw \to d'x'(y, out)(z', in)$.

In addition to that, we add the following rules to all R'_i:
$r_a : a_i \to bc$
$r_b : b \to b'$

$r_d : b'd' \to bd$

$r_e : b'c \to (e, out)$

$r_f : b'd \to (f, out)$

$r_g : g_i \to bh$

$r_h : h\sigma' \to h'\sigma$ for all $\sigma \in O$

$r_{bh} : b'h' \to bh$

$r_k b'h \to k$

All membranes apart from the skin membrane additionally get the rules:

$r'_e : e \to (e, out)$

$r'_f : f \to (f, out)$

$r'_k : k \to (k, out)$

R'_1 (the skin membrane) gets the following rules:

$r_{continue} : e^i f^{m-i} \to hg_1g_2...g_m$ for all $0 \leq i \leq m-1$.

$r_{end} : e^m \to g_{i_0}$

$r_{new} : hk^m \to a_1a_2...a_m$

All membranes i that are on the path between the skin membrane and a membrane j get the following rules:

$r_{path,a} : a_j \to (a_j, in)$

$r_{path,g} : g_j \to (g_j, in)$

All membranes i that are not on the path between the skin membrane and a membrane j get the following rules:

$r_{\neg path,a} : a_j \to (a_j, out)$

$r_{\neg path,g} : g_j \to (g_j, out)$

The new system Π' works in the following way: Each cycle it starts by transforming the objects a_i into the objects b and c.

The new objects c and d are used to apply the original rules. Every even step one applicable rule is applied using c or d. In the same step, b is transformed into b'. If both b' and d' are present in the following step (i.e., a rule was applied), both are transformed back into b and d. When rules are applied, all objects $\sigma \in O$ used

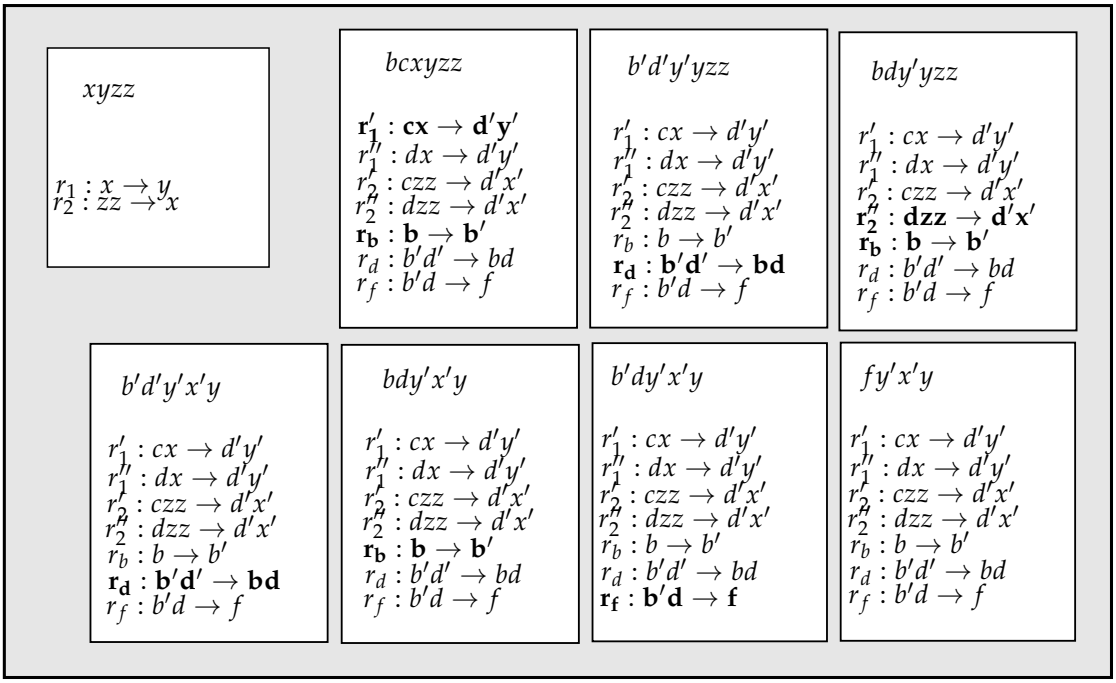

Figure 3.1: An example of the first part of this transformation in progress. In the top left we have the original membrane, containing two rules and the objects $xyzz$. In this step, it would apply each rule once. The new system adds the objects b and c to the membrane, and applies the bolded rules to the objects. Each square is one step in the computation, and at the end all rules have been applied, and the object f is ready to be sent back to the skin membrane

are temporarily transformed into σ', to signify that they have been used this cycle. This process can be seen in Figure 3.1.

This continues until inevitably a membrane has applied all the rules it can. In other words, when it has done all the rules in a maximally parallel fashion. When this happens, d can not transform into d' during the even step, so b' and d are present at the same time. They then apply rule r_f during the next step to create an object f, which is then sent up to the skin membrane, signaling the completion of a maximal parallel step in this membrane.

In order to make the new system halt when the old system would halt, we use the object c. If no rule could be applied in this membrane at the start of this cycle, c would not transform into d, and rule r_e is applied to create an object e that is sent to the skin membrane.

When all membranes have finished applying their rules, m objects, either e or f, will have been sent to the skin membrane. If at least one f object exists (in other words, if at least one of the membranes applied at least one rule), the system continues for another cycle. But it first has to convert all the σ' back into σ.

It does this in a similar manner to the first part: It sends objects g_i to every membrane i, where b and h use the same two-step cycles b and d used to convert all the σ' back to σ. When no more σ' exist, h will not convert into h'. The membrane will then create an object k and send it back up to the skin membrane, signaling that

all σ' in that membrane have been converted back. When all membranes are done cleaning up, there will be m k objects in the skin membrane, which will be converted back into a_i objects. It then sends the a_i objects from the skin membrane to the different membranes using the rules $r_{path,a}$ and $r_{\neg path,a}$. Note that the a_i might arrive at different times in their respective membranes, but this is no problem, because they will eventually arrive, and the system will not move forward until they do and apply all the rules there. This entire process can be seen in Figure 3.2.

If all objects created at the end of the first part were e, then none of the membranes in the original system could apply any rules, so the original system Π has halted. In this case, the rule r_{end} is applied to only create a g object for the output membrane, which is then sent to the membrane i_0. There it converts all the σ' back to σ. When it is done with this, and sends a single k object to the skin membrane. But when it arrives there, the h object necessary to start the next step will not exist, so Π' will halt as well.

The path rules are used to send objects a_i and g_i down to the membranes where they are needed. When a membrane has multiple lower neighbours, this process becomes inherently nondeterministic. Therefore we add rules to keep the objects on the path to the membrane. If they leave the path, because they enter the wrong lower neighbour, they are immediately sent back up to try it again. In a way, this is similar to repeatedly flipping a coin until it lands on heads. This makes it impossible to determine how many steps the system will take until it halts. But we can be sure that if Π halts, there is at least one computation which leads to Π' halting in a given number of steps. If the object would repeatedly fail to move into the correct membrane, the system will simply keep looping instead of halting, so there is no difference in the language the system accepts or produces.

Output remains the same, since the σ objects in O, which are the only objects that can be in T, are all present in the output membrane as they would be in the original system Π. None of the newly introduced objects are elements of T, so they will not contaminate the output. Every cycle, rules will be applied in each membrane until there are no more applicable rules, just like in a maximally parallel system. Therefore, the behaviour on output, computation, and halting are identical between Π and Π', so they are equivalent.

The greatest drawback to this transformation is its loss of efficiency. Π' will take a large number of steps to simulate a single step in Π. This amount S can be estimated as the largest amount of rules C that have to be applied in a single membrane times 2, plus a small and constant number of steps to transform the newly introduced objects, plus a varying number of steps V depending on how long it takes the a_i and g_i objects to reach the correct membranes. So we can describe
$S = 2 * max(R) + C + V$, where $max(R)$ is the largest amount of rules that have to be applied during this step.

If Π is context-free(the left-hand side of every rule consists of only one object), Π' does not preserve this

quality. Therefore this transformation may not always be useful for transforming systems that depend on being context-free.

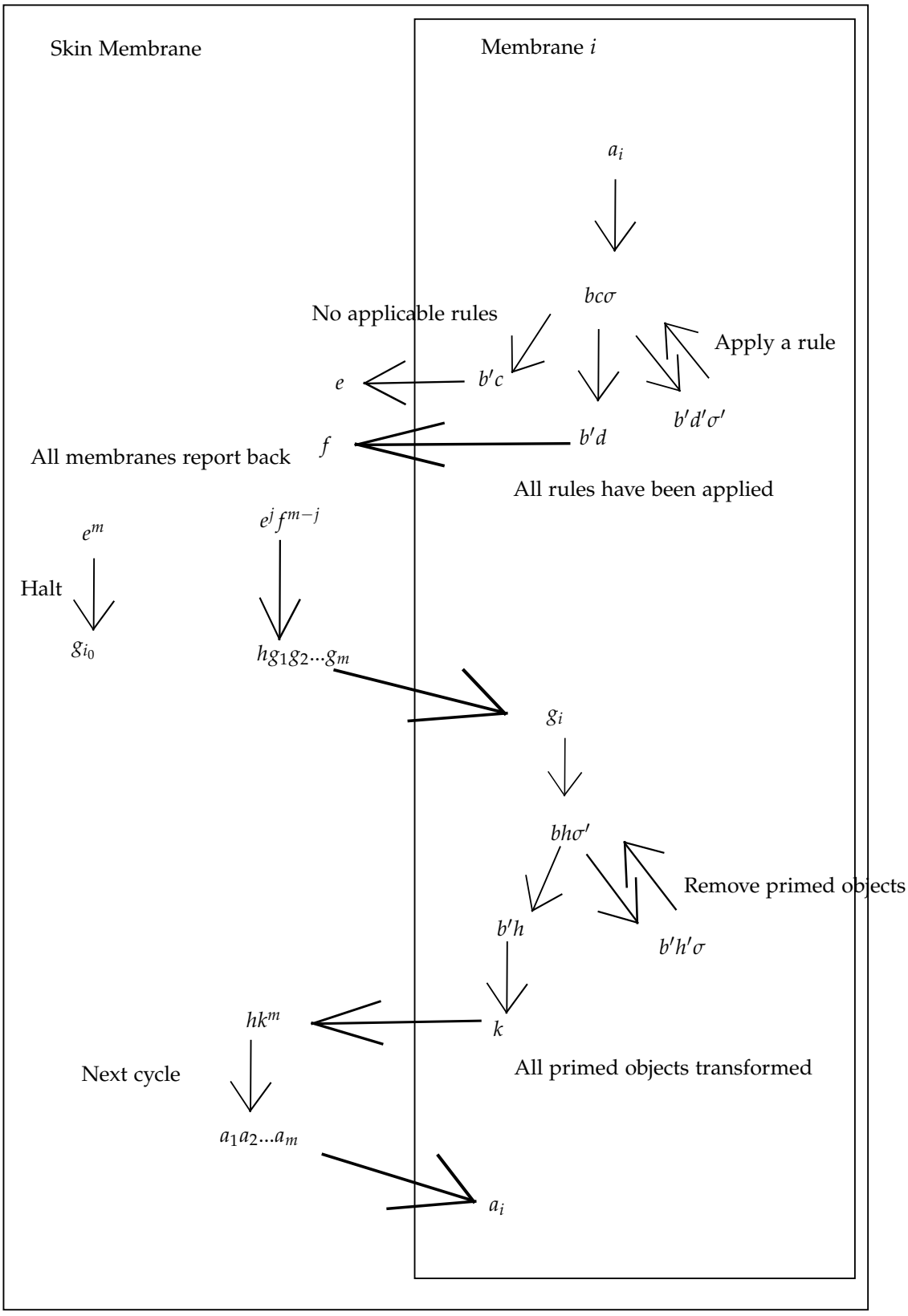

Figure 3.2: The process of simulating a single step. Each arrow represents one or more rules being applied

Chapter 4

Communication

4.1 Introduction

Communication systems work quite differently from regular P systems, and if we want the systems to emulate each other, we have to overcome these differences.

One big difference is that communication systems can only create new objects by importing them from the environment. A regular P system can create new objects 'out of thin air', so to speak, from any membrane in a single step.

Another big difference is that in a regular P system, the rules a membrane can apply depend solely on the objects present in that membrane. In communication systems, however, many rules depend on the contents of both the membrane and its upper neighbour. Because a membrane's contents might also be altered by a lower neighbour applying a rule, each membrane is effectively dependent on the contents of the entire system in choosing which rules it can and will apply. While communication systems do this automatically, in regular P systems a membrane has no direct way of knowing what the contents of another membrane are. When transforming between communication systems and membrane systems, this is another challenge we have to overcome.

In this chapter we present algorithms to create equivalent communication systems for any regular P system, and vice versa.

We will use a similar technique for both simulations: The new systems will have only one membrane, but will contain disjoint sets of objects representing the contents of each membrane in the original system. After finishing this section we learned of a similar single-membrane approach [5].

For any alphabet O and membrane m we define the set $O_m = \{\sigma_m | \sigma \in O\}$ as a disjoint copy of O. We will use the set O_m as a representation of the objects present in membrane m.

We define the set $O_e = \{\sigma_e | \sigma \in O\}$ as a representation of the objects present in the environment.

Given a multiset $x = x_1 x_2 ... x_n \; in O^*$, we define the multiset $x_i = x_{1_i} x_{2_i} ... x_{n_i} \in O_i^*$.

4.2 Communication to Normal

In this section we present a way to create an equivalent P system for any given communication system. If the communication system uses maximal parallelism, the P system will also be maximally parallel. If the communication system uses bounded parallelism, the resulting P system will be bounded too.

For any communication system of degree m
$\Pi = (O, T, E, \mu, w_1, w_2, ..., w_m, R_1, R_2, ..., R_m, i_0)$

we define the equivalent P system $\Pi' = (O', T, \mu', w_1', R_1', 1)$ where:

$O' = (O \cup O_1 \cup O_2 \cup ... \cup O_m \cup O_e) \setminus O_{i_0}$.
μ' is a membrane structure of degree 1. In other words, Π' consists of a single membrane.
$w_1' = \{\sigma_i | \sigma \in w_i$ where $1 \leq i \leq m$ and $i \neq i_0\} \cup w_{i_0}$

If Π is bounded parallel, then Π' is bounded parallel as well, and each new rule r' has the same multiplicity as r.

We define R_1' as follows:
Given a membrane $a \in \mu$ with upper neighbour b:

For any symport rule $r : (x, out) \in R_a$, we add the rule:
$r' : x_a \rightarrow x_b$
to R_1'. For any symport rule $r : (y, in) \in R_a$, we add the rule:
$r' : y_b \rightarrow y_a$
to R_1'. For any antiport rule $r : (x, out; y, in) \in R_a$, we add the rule:
$r' : x_a y_b \rightarrow x_b y_a$
to R_1'.

If a is the skin membrane, i.e., $a = 1$, then the rules are as follows. Remember that for any symport rule which imports objects from the environment, at least one imported object has to be in finite supply in the environment. So we can write any rule (y, in) as (fg, in) where $f \in (O \setminus E)^+$ and $g \in E^*$.

The antiport rule $(x, out; y, in)$ can be rewritten as $(x, out; fg, in)$ where $f \in (O \setminus E)^*$ and $g \in E^*$ (f may be the empty string).

Now, for the skin membrane 1 we do the following:
For any symport rule $r : (x, out) \in R_1$, we add the rule:
$r' : x_1 \to x_e$
to R'_1.
For any symport rule $r : (fg, in) \in R_1$, we add the rule:
$r' : f_e \to f_1 g_1$
to R'_1.
For any antiport rule $r : (x, out; fg, in) \in R_1$, we add the rule:
$r' : x_1 f_e \to x_e f_1 g_1$
to R'_1.

Once these rules are created, we take all objects labeled as σ_{i_0} in R'_1 and replace it with a simple σ. This way the output will remain the same. For example, if a is the output membrane, then the rule
$r \in R_a : (x, out; y, in)$
is transformed into:
$r' \in R'_1 : xy_b \to x_b y$.

In short, Π' works by representing all the different membranes of Π in one membrane, by giving each object a subscript indicating in which membrane of Π it is located at that point in time. We represent the objects in the environment by using the σ_e objects.

By putting all objects in the same membrane, we have full knowledge of the contents of each membrane. This way, we can apply the rules for every membrane of Π at the same time.

By letting the output membrane use the original objects instead of new objects, we ensure that the output remains the same at the time of halting. None of the new σ_i objects are elements of T, so they will not influence the output. Because Π' will apply the same rules as Π, computation and halting will occur the same way as well. Therefore, Π is equivalent to Π'.

The new system has the exact same complexity as the old system: It has the same amount of rules, applies the same amount of rules during each step, and at any point in the computation the entire system contains the same number of objects as the original system, with the exception of the objects in the environment. The only increase is in the size of the alphabet, with $|O'| = m * |O|$. But we make up for this increased size by making the membrane structure as small and simple as possible. In fact, while O' is m times bigger than O,

μ' is exactly m times smaller than μ.

4.3 Normal to Communication

A notable difference between normal P systems and communication systems is that a normal P system can create objects out of thin air. Any membrane in a P system can apply a rule $x \to y$ to remove the x objects and create a new y object. A communication system, on the other hand, has to import these new objects from the environment, and can only remove unneeded objects by ejecting them to the environment.

Therefore, if we want to simulate a P system using a communication system, we will need to use the environment to create new objects. Because only the skin membrane can interact with the environment, the solution we present here is by creating a new communication system with only one membrane, where we give each object a subscript designating the membrane in the original system it belongs to. This system is quite similar to the one presented in the previous section.

Another difference is that a normal P system automatically chooses lower neighbours as targets for its rules randomly. Because a communication system never has to make this nondeterministic choice, we are forced to add a large amount of rules to represent all the options.

For any regular P system of degree d
$\Pi = (O, T, \mu, w_1, w_2, ..., w_d, R_1, R_2, ..., R_d, i_0)$

We create the equivalent communication system of degree 1:
$\Pi' = (O', T, E, \mu, w_1, R_1, 1)$, where:

$O' = (O \cup O_1 \cup O_2 \cup ... \cup O_d \cup O_e) \setminus O_{i_0}$
$E = O'$
μ' is a membrane structure of degree 1. In other words, Π' consists of a single membrane.
$w'_1 = \{\sigma_i | \sigma \in w_i$ where $1 \leq i \leq d$ and $i \neq i_0\} \cup w_{i_0}$.

Before we define R'_1 we need to define a set of possible target membranes. If a membrane m in a normal P system has a rule $x \to (a_1 a_2 ... a_n, in)$ where a_i are individual objects in O, every a_i nondeterministically chooses a lower neighbour of m as a target membrane. If m has k lower neighbours, there are n^k different ways all the objects can choose a target.
Given a membrane m with lower membranes $l_1, l_2, ..., l_k$, and a rule $r : w \to x(y, out)(a_1, a_2 ... a_n, in) \in R_m$, where $w, x, y, a_1 a_2 ... a_n \in O^*$, we define the set
$L_r = \{a_{1_{m_1}} a_{2_{m_2}} ... a_{n_{m_n}} | m_i \in \{l_1, l_2, ..., l_k\}\}$.

So L_r is the set of all potential targets of the lower membranes of the rule r. For example, if membrane m has two lower membranes, p and q, and a rule $r : a \to (bc, in)$, then $L_r = \{b_p c_p, b_p c_q, b_q c_p, b_q c_q\}$.

The new set of rules R_1' is created as follows. For any rule $r : w \to x(y, out)(z, in) \in R_m$ for a membrane m with upper neighbour n we add the following set of antiport rules to R_1:
$R_r = \{(w_m, out; x_m y_n l) | l \in L_r\}$.

In other words, for a rule we create a new set of rules, one for each possible mapping of the lower neighbours the rule could apply.

If Π is bounded parallel, then Π' is bounded parallel as well, and each new rule in R_r has the same multiplicity as r. If Π is maximally parallel, then Π' is maximally parallel as well.

Once these rules are created, we take any object labeled as σ_{i_0} in R_1' and replace it with a simple σ. We do not represent the objects in the output membrane i_0 using newly introduced symbols, but instead use the original symbols from O to preserve the output. All the new objects we introduce are not elements of T, so they do not contaminate the output.

We can easily see that Π' can only apply rules if the original system Π could apply them. Although we now perform all computational actions in a single membrane, the actions remain the same. Using the system of targeting sets emulates the nondeterministic choosing of lower membranes in P systems. The systems halt at the same time, and produce the same output. Therefore, the systems Π and Π' are equivalent.

The systems mostly have the same complexity: Over the course of a computation, they apply the same number of rules, take the same number of steps to halt, and contain the same number of objects. The size of O' is increased by a factor m, $|O'| = m * |O|$. The biggest increase in complexity is the amount of rules. For every rule $r : w \to x(y, out)(a_1 a_2 ... a_n, in)$ in a membrane with k lower neighbours we have to add n^k new rules to our new system.

Chapter 5

Conclusions

In this paper we have presented several algorithms for simulating different membrane systems without having to use inefficient Turing machines or register machines. We have shown ways of transforming different types of parallelism in regular P systems into the other type. We have also shown ways of transforming both bounded and maximally parallel communication systems into regular P systems, and the inverse operation. Almost all transformations preserve the complexity of the original system, introducing at most a polynomial amount of new objects or rules, and often compensating for the new objects by requiring fewer membranes. This is in stark contrast to using Turing machines, which would most certainly require an exponential complexity increase. These transformations also tend to preserve the original structure and behaviour of the systems, which creates more elegant and easier to understand systems. Unfortunately the introduction of non-cooperating rules means that transformed P systems are no longer context-free.

By combining the results we have also given a way to directly transform a bounded parallel communication system into a maximal communication system, and vice versa, by first transforming into a regular P system, transforming the P system into one with the correct parallelism, and then transforming the result back into a communication system. Unfortunately this creates a system with a very different structure from the original, with only one membrane.

Undoubtedly more research could be put into other transformations and simulations using other classes of membrane systems. Dynamic membrane systems [3], string membrane systems [6], P automata [2], and others could almost certainly be transformed directly into other classes. Other systems of halting or producing output could also be researched. Establishing such a network of transformation algorithms would allow researchers to incorporate different types of membrane systems into a single system or program with negligible

loss of efficiency.

Bibliography

[1] Francesco Bernardini, Francisco J Romero-Campero, Marian Gheorghe, Mario J Perez-Jimenez, Maurice Margenstern, Sergey Verlan, and Natalio Krasnogor. On p systems with bounded parallelism. In *Pre-Proc. of First International Workshop on Theory and Application of P Systems, Timisoara, Romania*, pages 31–36, 2005.

[2] Rudolf Freund, Marion Oswald, and Ludwig Staiger. *ω -P Automata with Communication Rules*, pages 203–217. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[3] Carlos Martın-Vide, Gheorghe Păun, and Alfonso Rodrıguez-Patón. On p systems with membrane creation. *Computer Science*, 9(2):26, 2001.

[4] Andrei Păun and Gheorghe Păun. The power of communication: P systems with symport/antiport. *New Generation Computing*, 20(3):295–305, 2002.

[5] G. Paun. *Membrane Computing: An Introduction*. Natural Computing Series. Springer Berlin Heidelberg, 2002.

[6] Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108 – 143, 2000.

[7] Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa. *The Oxford handbook of membrane computing*. Oxford University Press, 2010.