Department of Applied Informatics
Faculty of Mathematics, Physics and Informatics
Comenius University, Bratislava

Biologically inspired computation models

(Dissertation thesis)

Mgr. Michal Kováč

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

....

Acknowledgments

Abstract

Author: Mgr. Michal Ková č
Title: Biologically inspired computation models
University: Comenius University in Bratislava
Faculty: Faculty of Mathematics, Physics and Informatics
Department: Department of Applied Informatics
Supervisor: doc. RNDr. Damas Gruska, PhD.

This work discusses the research in the membrane systems, an emerging field of natural computing. Many variants of membranes systems have already been studied, most of them uses parallel rewriting and are computationally complete. Various sequential models have been proposed, however, in many cases they are weaker than their parallel variant.

We propose some other sequential models, provide universality proofs for them and suggest further research.

Keywords: Computation models inspired by biology, Membrane systems, P systems

Abstrakt

Autor: Mgr. Michal Kováč
Názov dizertačnej práce: Biologically inspired computation models
Škola: Univerzita Komenského v Bratislave
Fakulta: Fakulta matematiky, fyziky a informatiky
Katedra: Katedra aplikovanej informatiky
Vedúci dizertačnej práce: doc. RNDr. Damas Gruska, PhD.
 Bratislava, 2015

V tejto práci sa zaoberáme výskumom v oblasti membránových systémov. Veľa variantov už bolo preskúmaných, väčšinou pri výpočte používajú paralelizmus a sú Turingovsky úplné. Navrhlo sa aj veľa sekvenčných modelov, ale väčšina z nich má slabšiu výpočtovú silu ako ich paralelný variant.

V druhej časti predkladáme niektoré sekvenčné modely, u ktorých dokazujeme Turingovu úplnosť a navrhujeme modely, ktoré plánujeme preskúmať v rámci dizertačnej práce.

Kľúčové slová: Výpočtové modely inšpirované biológiou, Membránové systémy, P systémy

Contents

List of Figures

List of Algorithms

Introduction

There are a lot of areas in the theoretical computer science that are motivated by other science fields. Computation models motivated by biology forms a large group of them. They include neural networks, computational models based on DNA evolutionary algorithms, which have already found their use in computer science and proved that it is worth to be inspired by biology. L-systems are specialized for describing the growth of plants, but they have also found the applications in computer graphics, especially in fractal geometry. Other emerging areas are still awaiting for their more significant uses.

One of them is the membrane computing. It is relatively young field of natural computing - in comparison: neural networks have been researched since 1943 and membrane systems since 1998 [44].

Membrane systems (P systems) are distributed parallel computing devices inspired by the structure and functionality of cells. Recently, many P system variants have been developed in order to simulate the cells more realistically or just to improve the computational power.

In chapter 2 we recall some computer science basics we will use through the work. Natural computing including models inspired by nature will be introduced in chapter 1. One of these models, P systems, with the current state of the research in their variants, is presented in chapter 3.

Chapter 1

Natural computing

Recently, an interdisciplinary research between the fields of Computer Science and Biology has been rapidly growing. Natural computing consists of three classes of methods:

- those that are based on the use of computers to simulate natural phenomena

- those that employ natural materials (e.g., molecules) to compute

- those that take inspiration from nature for the development of novel problem-solving techniques

1.1 Bioinformatics

Bioinformatics has undergone a fast evolving process, especially the areas of genomics and proteomics. Bioinformatics can be seen as the application of computing tools and techniques for the management of biological data. Just to mention a few:

- the design of efficient algorithms for DNA sequence alignment,

- the investigation of methods for prediction of the three-dimensional structure of molecules and proteins,

- the development of data structures to effectively store huge amount of structured data.

1.2 Biomolecular computing

Biomolecular computing make use of molecules such as DNA and proteins to perform computations involving storing, retrieving and processing data. It takes advantage of the many different molecules to try many different possibilities at one, so it is somewhat similar to parallel computing.

Adleman in his 1994 report [1] demonstrated a proof of concept use of DNA as a form of computation which solved the Hamiltonian path problem. Since then, various Turing machines have been proven to be constructible with DNA [34].

DNA can also be used as a digital data storage with 5.5 petabits per cubic millimeter of DNA (see [13]). The information retrieval is, however, a slow process, as the DNA needs to be sequenced in order to retrieve data, so this method is intended mainly for a long-term archival of large amounts of scientific data.

1.3 Biologically inspired computing models

On the other hand, the birth of biologically inspired frameworks started the investigation of mathematical models and their properties and technological requirements for their implementation by biological hardware. Those frameworks are inspired by the nature in the way it "computes", and has gone through the evolution for billions of years.

Neural networks, genetic algorithms and DNA computing are already well established research fields.

1.3.1 Neural networks

Inspired by the human brain, which contains on average 86 billions neurons [3], neurophysiologist Warren McCulloch in 1943 proposed a mathematical model of artificial neural network.

A single perceptron computes a function $f(x)$, where x is an input - a vector of real values.

The perceptron can learn itself by modifying parameters used to compute the function. This learning can be performed in various ways, we will mention only the supervised learning. Imagine a function with

- input: perceptron's parameters

- output: error of the computed result $f(x)$

If the perceptron receives a feedback in form of error (from the supervision, e.g. dataset used to train the perceptron), it can modify its parameters such the error will be minimized in the future. This can be done through the gradient descent method, which is used to find a local minimum of a function.

A single perceptron can only compute linear functions, so they are often connected with other perceptron to form a neural network. Often it is practically unusable to say what is the purpose of a single neuron in a more complex neural networks.

Neural networks have broad applicability to real world problems and are best if the modeled system has some tolerance to error such as:

- Image recognition: OCR, web search by image

- Music recognition by voice sample

- Time series forecast: weather, stock

- Diagnosing of illnesses

1.3.2 Evolutionary algorithms

Evolutionary algorithms are inspired by Darwin's theory of evolution. Basically, an algorithm starts with a population of random individuals (solutions to the problem). In each generation, the fitness of every individual is evaluated and the more fit individuals are replicated and mutated to form a new generation. The less fit individuals die.

Idea of evolutionary computing was introduced in the 1960's by I. Rechenberg. Nowadays they have been applied to find solution of many optimization problems. They also can be used to design or to train a neural network [41].

1.3.3 Membrane systems

Nature computes not only at the neural or genetic level, but also at the cellular level. In general, any non-trivial biological system has a hierarchical structure where objects and information flows between regions, what can be interpreted as a computation process.

The regions are typically delimited by various types of membranes at different levels from cell membranes, through skin membrane to virtual membranes which delimits different parts of an ecosystem. This hierarchical system can be seen in other field such as distributed computing, where again

Figure 1.1: The membrane structure of a P system and its associated tree [54]

well delimited computing units coexist and are hierarchically arranged in complex systems from single processors to the internet.

Membranes keep together certain chemicals or information and selectively determines which of them may pass through.

From these observations, Păun [44] introduces the notion of a membrane structure as a mathematical representation of hierarchical architectures composed of membranes. It is usually represented as a Venn diagram with all the considered sets being subsets of a unique set and not allowed to be intersected. Every two sets are either one the subset of the other, or disjoint. Outermost membrane (also called skin membrane) delimits the finite "inside" and the infinite "outside".

Hierarchical structures are usually represented by trees, but membrane structures are visually better viewed as Venn diagram as seen in the figure 1.1.

Recently, several computational models based on the membrane structure have been created.

P systems

P systems were introduced in 1998 as a system with membrane structure that contains multisets of objects and rewrite rules that are executed in a maximally parallel manner. Since then, a huge amount of variants have been created with various computation powers. We have investigated the power of several such variants. Additionally, we propose a new variant with a vacuum, where the rules can describe what will happen to an object that arrives to an empty membrane.

P systems with existing and newly proposed variants will be discussed in the chapter 3.

Aside from P systems, other models based on the membrane structure have been created such as the Calculus of Looping Sequences (CLS), which was inspired by P systems.

The Calculus of Looping Sequences

Barbuti in [6] concluded that there is a need for a formalism having a simple notation, having the ability of describing biological systems at different levels of abstractions, having some notions of compositionality and being flexible enough to allow describing new kinds of phenomena as they are discovered, without being specialized to the description of a particular class of systems. The Calculus of Looping Sequences (CLS) was introduced in [6].

The membrane structure in CLS is defined recursively, consisting of terms T and sequences S:

$$T ::= S \mid (S)^L \rfloor T \mid T \mid T$$

$$S ::= \varepsilon \mid a \mid S \cdot S$$

, where a is a symbol from the alphabet, ε represents the empty sequence, $\cdot$ is a sequencing operator, $(S)^L$ is a looping operator, $\mid$ is a parallel composition operator and $\rfloor$ is a containment operator.

Membranes are represented by a sequence that is looped around. Several extensions of the CLS have been proposed. In CLS+ the looping operator can be applied to a parallel composition of sequences, which represents a notion of a fluid membrane. CLS+ can be translated to CLS (see [6]). Milazzo in his PhD thesis [40] includes also a simulation of a P system using CLS. The major difficulty is simulating the maximal parallelism of rule application.

Chapter 2

Preliminaries

This chapter is about some basic notions of computer science which will be used through the work. We start by defining formal languages and basic models (grammars, machines) that define language families and end by defining multiset languages.

2.1 Formal languages

Our study is based on the classical theory of formal languages. We will recall some definitions:

Definition 2.1.1. *An* **alphabet** *is a finite nonempty set of symbols.*

Definition 2.1.2. *A* **string** *over an alphabet is a finite sequence of symbols from the alphabet.*

The length of the string s is denoted by $|s|$. 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.

Definition 2.1.3. *A* **language** *over the alphabet V is any subset of V^*.*

Definition 2.1.4. *A* **family of languages** *is a set of languages.*

2.2 Formal grammars

Definition 2.2.1. *A* **formal grammar** *is a tuple $G = (N, T, P, \sigma)$, where*

- N, T *are disjoint alphabets of non-terminal and terminal symbols,*

- $\sigma \in N$ *is the initial non-terminal,*

- *P is a finite set of rewriting rules of the form $u \to v$, with $u \in (N \cup T)^*N(N \cup T)^*$ and $v \in (N \cup T)^*$.*

Definition 2.2.2. *A* **rewriting step** *in the grammar G is a binary relation $\Rightarrow$ on $(N \cup T)^*$, where $x \Rightarrow y$ only if $\exists w_1, w_2 \in (N \cup T)^+$ and a rule $u \to v \in P$ such that $x = w_1 u w_2$ and $y = w_1 v w_2$.*

Definition 2.2.3. *Language defined by a grammar G is a set $L(G) = \{w \in T^* | \sigma \Rightarrow w\}$.*

Languages that can be generated by a formal grammar are the recursively enumerable languages RE.

2.3 Chomsky hierarchy

In this section we introduce several well-known families of languages.

Definition 2.3.1. *A* **regular grammar** *is a formal grammar, where the rewriting rules are of the form $u \to v$, where $u \in N$ and $v \in T^*(N \cup \{\varepsilon\})$.*

Definition 2.3.2. *A* **regular language** *is a language generated by a regular grammar. The family of regular languages is denoted R.*

Definition 2.3.3. *A* **context-free grammar** *is a formal grammar, where rewriting rules are of the form $u \to v$, where $u \in N$ and $v \in (N \cup T)^*$.*

Definition 2.3.4. *A* **context-free language** *is a language generated by a context-free grammar. The family of context-free languages is denoted CF.*

Definition 2.3.5. *A* **context-sensitive grammar** *is a formal grammar, where rewriting rules are of the form $u \to v$, where $u \in (N \cup T)^*N(N \cup T)^*$, $v \in (N \cup T)^*$ and $|u| < |v|$.*

Definition 2.3.6. *A* **context-sensitive language** *is a language generated by a context-sensitive grammar. The family of context-sensitive languages is denoted CS.*

These families of languages forms the Chomsky hierarchy by means of inclusions: $R \subset CF \subset CS \subset RE$.

2.4 Matrix grammars

Definition 2.4.1. *A* **matrix grammar** *is a tuple $G = (N, T, M, \sigma)$, where:*

- *N, T are disjoint alphabets of non-terminal and terminal symbols,*

- *$\sigma \in N$ is the initial non-terminal,*

- *M is a finite set of matrices, which are sequences of context-free rules of the form $u \to v$, where $u \in N$ and $v \in (N \cup T)^*$.*

Definition 2.4.2. *A* **rewriting step** *$x \Rightarrow y$ holds only if there is a matrix $(u_1 \to v_1, u_2 \to v_2, \ldots, u_n \to v_n) \in M$ such that for each $1 \leq i \leq n$ the following holds: $x_i = x_i' u_i x_i''$ and $x_{i+1} = x_i' v_i x_i''$, where $x_i, x_i', x_i'' \in (N \cup T)^*$ and $x_1 = x$ and $x_{n+1} = y$.*

Example 2.4.1. Consider the following matrix grammar:

$$G = (\{\sigma, X, Y\}, \{a, b, c\}, M, \sigma),$$

where M contains three matrices:

$$[S \to XY], [X \to aXb, Y \to cY], [X \to ab, Y \to c].$$

There are only context-free rules, yet the grammar generate the context-sensitive language $\{a^n b^n c^n | n \geq 1\}$.

The family of matrix grammars is denoted MAT. It is known that $CF \subset MAT \subset RE$. Interestingly, $MAT \cap a^* \subset R$ (see [7]).

2.5 Register machines

Definition 2.5.1. *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.

2.6 Lindenmayer systems

In 1968, a Hungarian botanist and theoretical biologist Aristid Lindenmayer introduced [37] a new string rewriting algorithm named Lindenmayer systems (or L-systems for short). They are used by biologists and theoretical computer scientists to mathematically model growth processes of living organisms, especially plants. The difference with Chomsky grammars is that rewriting is parallel, not sequential.

The simplest version of L-systems assumes that the development of a cell is free of influence of other cells. This type of L-systems is called $0L$ systems, where "0" stands for zero-sided communication between cells.

Definition 2.6.1. *A $0L$ system is a triple (Σ, P, ω), where Σ is an alphabet, ω is a word over Σ and P is a finite set of rewriting rules of the form $a \to x$, where $a \in \Sigma, x \in \Sigma^*$.*

It is assumed there is at least one rewriting rule for each letter of Σ. $0L$ system works in parallel way, so all the symbols are rewritten in each step.

Example 2.6.1. Consider a $0L$ system with alphabet $\Sigma = \{a, b\}$, initial word $\omega = a$ and rewriting rules $P = \{a \to b, b \to ab\}$. Since in this system there is exactly one rule for every letter of the alphabet, the rewriting is thus deterministic and the generated words will be $\{a, b, ab, bab, abbab, \ldots\}$.

$1L$ systems allows the rewriting rules to include context of size 1, so it allows for rules of type $yaz \to x$.

L-systems with tables (T) have several sets of rewriting rules instead of just one set. At one step of the rewriting process, rules belonging to the same set have to be applied. The biological motivation for introducing tables is that one may want different rules to take care of different environmental conditions (heat, light, etc.) or of different stages of development.

Definition 2.6.2. *An extended (E0L) system is a pair $G_1 = (G, \Sigma_T)$, where $G = (\Sigma, P, \omega)$ is an 0L system, where $\Sigma_T \subseteq \Sigma$, referred to as the terminal alphabet. The language generated by G_1 is defined by $L(G_1) = L(G) \cap \Sigma_T^*$.*

Such languages are called *E0L* languages. *E0L* languages with tables are called *ET0L* languages.

It is known that $CF \subset E0L \subset ET0L \subset CS$ (see section 2.3 for definitions of CF and CS).

2.7 Multisets

Definition 2.7.1. *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.

Definition 2.7.2. *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.

Definition 2.7.3. *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.

$$element | element | other\_element$$

.

Definition 2.7.4. *Multiset inclusion. 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 $M_1 \subseteq M_2$.*

Definition 2.7.5. *The* **union** *of two multisets $M_1 \cup M_2$ is a multiset where $\forall x \in X : (M_1 \cup M_2)(x) = M_1(x) + M_2(x)$.*

Definition 2.7.6. *The* **difference** *of two multisets $M_1 - M_2$ is a multiset where $\forall x \in X : (M_1 - M_2)(x) = M_1(x) - M_2(x)$.*

Definition 2.7.7. *Product of multiset M with natural number $n \in \mathbb{N}$ is a multiset where $\forall x \in X : (n \cdot M)(x) = n \cdot M(x)$.*

2.8 Semilinear sets

Definition 2.8.1. *A* **linear set** $L(c, p_1, \ldots, p_r)$ *is a subset* $\{c + \sum\limits_{i=1}^{r} |k_i \in \mathbb{N}\}$ *of* $\mathbb{N}^n$, *where* $c, p_1, \ldots, p_r \in \mathbb{N}^n$.

We call c the constant and $p_1, \ldots, p_r$ the periods of the linear set.

Example 2.8.1. For $n = 1$ the linear set is a subset of $\mathbb{N}$. For $n = 1$ and $r = 1$ we get an arithmetic progression.

Example 2.8.2. $L((0,0), (0,1), (1,0))$ contains all pairs with one zero element and one non-negative element:

$$\{(0,0), (1,0), (2,0), \ldots, (x,0), \ldots, (0,1), (0,2), \ldots, (0,y), \ldots\}$$

.

Definition 2.8.2. *A subset of* $\mathbb{N}^n$ *is called* **semilinear** *if it is a finite union of linear sets.*

2.8.1 Parikh's mapping

In this subsection we show how semilinear sets and multisets relate to the formal language theory. We will start with several basic definitions.

The number of occurrences of a given symbol $a \in \Sigma$ in the string $w \in \Sigma^*$ is denoted by $|w|_a$.

Definition 2.8.3. $\Psi_\Sigma(w) = (|w|_{a_1}, |w|_{a_2}, \ldots, |w|_{a_n})$ *is called a* **Parikh image of the string** $w \in \Sigma^*$, *where* $\Sigma = \{a_1, a_2, \ldots a_n\}$.

When referring to the Parikh mapping for a language, this should be taken to mean the mapping applied to all the words on the language. This idea is expressed in the next definition.

Definition 2.8.4. *For a language* $L \subseteq \Sigma^*$, $\Psi_\Sigma(L) = \{\Psi_\Sigma(w) | w \in L\}$ *is the* **Parikh image of the language** L.

Definition 2.8.5. *If* FL *is a family of languages, by* $PsFL$ *we denote the family of Parikh images of languages in* FL, *e.g.* $PsCF, PsRE$.

Example 2.8.3. Consider an alphabet $V = \{a, b\}$ and a language $L = \{a, ab, ba\}$. $\Psi_\Sigma(L) = \{(1,0), (1,1)\}$. Notice that Parikh image of L has only 2 element while L has 3 elements.

Example 2.8.4. Consider the context-free grammar $G = (N, T, P, \sigma)$, where $N = \{A, B\}$, $T = \{a, b\}$ and with rules $P = \{\sigma \to A\sigma B | B\sigma A | ab, A \to a, B \to b\}$.

This means that the language generated by G contains strings where as and bs can occur intermixed. But, as is clear from the language definition, the same number of as and bs will always be present. Moreover, there will always be at least one of each letters.

Consider the words $w_1 = aaaabbbb \in L(G)$ and $w_2 = babababa \in L(G)$. Both of these words have the same Parikh image: $\Psi_\Sigma(w_i) = (4, 4)$. It is interesting to note that most information embedded in a word generated with a context-free grammar is thrown away by the Parikh mapping.

This loss of information is expressed in the Parikh's theorem [42], stating that the Parikh image of a context-free language is semilinear. The biggest implication of the theorem though, is that it shows that if the order of the symbols is ignored, then it is impossible to distinguish between a regular set and a context-free language [36].

Another interesting result appeared in [33] that every semilinear set is a finite union of disjoint linear sets.

2.9 Petri nets

Petri nets [46, 53] were introduced by Carl Adam Petri in 1962 in his PhD thesis. A Petri net is a graphical and mathematical tool for the modeling of concurrent processes and analysis of system behavior. A Petri net is usually drawn as a directed bipartite graph with two kind of nodes. Places are represented by circles within which each small black dot denotes a token. Transitions are represented by bars. Each edge is either from a place to a transition or vice versa.

Definition 2.9.1. *A **Petri net** is a tuple (P, T, φ), where:*

- *P is a finite set of places,*

- *T is a finite set of transitions,*

- *$\varphi : (P \times T) \cup (T \times P) \to \mathbb{N}$ is a flow function.*

The edges of the bipartite graph are annotated by either $\varphi(p, t)$ or $\varphi(t, p)$, where $p \in P$ and $t \in T$ are two endpoints of the arc. If $\varphi(p, t) = 1$ or $\varphi(t, p) = 1$, we usually omit the label.

Definition 2.9.2. *A **marking** is a mapping $\mu : P \to \mathbb{N}$.*

The mapping μ assigns certain number of tokens to each place of the net.

Definition 2.9.3. *A marking μ_1* **covers** *marking μ_2, when $\forall p \in P : \mu_1(p) \geq \mu_2(p)$ - in each place there is no less tokens in μ_1 than in μ_2. We denote it by $\mu_1 \geq \mu_2$.*

Definition 2.9.4. *A transition $t \in T$ is* **enabled** *at a marking μ iff $\forall p \in P, \varphi(p, t) \leq \mu(p)$.*

If a transition t is enabled, it may fire by removing $\varphi(p, t)$ tokens from each input place p and putting $\varphi(t, p')$ tokens in each output place p'. We then write $\mu \xrightarrow{t} \mu'$, where $\forall p \in P : \mu'(p) = \mu(p) - \varphi(p, t) + \varphi(t, p)$.

Example 2.9.1. In the Figure 2.1 the Petri net has four places and two transitions. At the current marking the transition t_1 is enabled and the transition t_2 is not enabled. Firing the transition t_1 takes one token from the place p_1 and produces one token to places p_2, p_3 and p_4. In the resulting marking both transitions t_1 and t_2 are enabled.

Definition 2.9.5. *A* **marked Petri net** *is a tuple (P, T, φ, μ_0), where (P, T, φ) is a Petri net and μ_0 is called the initial marking.*

Definition 2.9.6. *A sequence of transitions $\sigma = t_1 \ldots t_n$ is a* **firing sequence** *from μ_0 iff $\mu_0 \xrightarrow{t_1} \mu_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} \mu_n$ for some markings $\mu_1, \ldots, \mu_n$. We also write $\mu_0 \xrightarrow{\sigma} \mu_n$.*

We write $\mu_0 \xrightarrow{\sigma}$ to denote that σ is enabled and can be fired from μ_0, i.e., $\mu_0 \xrightarrow{\sigma}$ iff there exists a marking μ such that $\mu_0 \xrightarrow{\sigma} \mu$. The notation $\mu_0 \xrightarrow{*} \mu$ is used to denote the existence of a firing sequence σ such that $\mu_0 \xrightarrow{\sigma} \mu$.

Definition 2.9.7. *A marking μ is reachable for a marked Petri net $\mathcal{P} = (P, T, \varphi, \mu_0)$ iff $\mu_0 \xrightarrow{*} \mu$.*

Definition 2.9.8. *Let $\mathcal{P} = (P, T, \varphi, \mu_0)$ be a marked Petri net. The* **reachability set** *of $\mathcal{P}$ is $R((P)) = \{\mu | \mu_0 \xrightarrow{*} \mu\}$.*

A notion of reachability graph is helpful for analyzing the behavior of a Petri net as a tool for visualisation of the structure of the reachability set.

Definition 2.9.9. *Let $\mathcal{P} = (P, T, \varphi, \mu_0)$ be a marked Petri net. The* **reachability graph** *of $\mathcal{P}$ is a labelled graph whose nodes are the reachable markings and edge from μ_1 to μ_2 is labeled with a transition $t \in T$ iff $\mu_1 \xrightarrow{t} \mu_2$.*

Figure 2.1: An example Petri net

Figure 2.2: An example reachability graph

Example 2.9.2. Consider a Petri net $\mathcal{P}$ from the Figure 2.1. Its reachability graph is in the Figure 2.2. $\mathcal{P}$ is not bounded because by alternately firing transitions t_1 and t_2 we can reach infinitely many different markings. We can also easily see that it is live, because in every marking for every transition $t \in \{t_1, t_2\}$ there is a firing sequence ending with t.

In spite of its simplicity, the applicability of the technique of reachability graph analysis is rather limited in the sense that it suffers from the state explosion phenomenon as the sizes of the reachability sets grow beyond any primitive recursive function in the worst case [53].

Coverability graph analysis offers an alternative to the techinque of reachability graph analysis by abstracting out certain details to make the graph finite. To understand the intuition behind coverability graphs, consider the Figure 2.2 which shows a part of the reachability graph of the Petri net in the Figure 2.1. Consider the path $(3, 0, 0, 0) \xrightarrow{t_1} (2, 1, 1, 1) \xrightarrow{t_2} (3, 1, 1, 0)$ along which the places p_2 and p_3 both gain an extra token in the end, i.e. $(3, 0, 0, 0) > (3, 1, 1, 0)$. Clearly they can be made to contain arbitrary large number of tokens by repeating the firing sequence $t_1 t_2$ for a sufficient number of times, as $(3, 0, 0, 0) \xrightarrow{t_1 t_2} (3, 1, 1, 0) \xrightarrow{t_1 t_2} (3, 2, 2, 0) \xrightarrow{t_1 t_2} \ldots \xrightarrow{t_1 t_2} (3, n, n, 0)$, for arbitrary n. In order to capture the notion of a place being unbounded, we short-circuit the above infinite sequence of computation as $(3, 0, 0, 0) \xrightarrow{t_1} (2, 1, 1, 1) \xrightarrow{t_2} (3, \omega, \omega, 0)$, where ω is a symbol denoting something being arbitrarily large. As it turns out, the coverability graph of a Petri net is always finite [35]. The corresponding coverability graph of the example Petri net

Figure 2.3: An example coverability graph

in the Figure 2.1 is in the Figure 2.3. The algorithm for generating the coverability graph of a Petri net [53] is shown below.

Algorithm 1 Coverability graph algorithm

1: **procedure** CoverabilityGraph(marked Petri net $\mathcal{P} = (P, T, \varphi, \mu_0)$)
2:    create a node μ_{init} such that $\mu_{init} = \mu_0$ and mark it as 'new'
3:    **while** there is a 'new' node μ **do**
4:      **for** each transition t enabled at μ **do**
5:        **if** there is a node $\mu' = \mu + \Delta t$ **then**
6:          add an edge $\mu \xrightarrow{t} \mu'$
7:        **else if** there is a path $\mu_{init} \xrightarrow{*} \mu'' \xrightarrow{*} \mu$ such that $\mu'' < \mu + \Delta t$ **then**
8:          add a 'new' node x with
9:            $x(p) = \omega$ if $\mu''(p) < (\mu + \Delta t)(p)$
10:           $x(p) = \mu''(p)$ otherwise
11:          add an edge $\mu \xrightarrow{t} x$
12:        **else**
13:          add a 'new' node x with $x = \mu + \Delta t$ and an edge $\mu \xrightarrow{t} x$
14:      mark μ with 'old'

2.9.1 Analysis of behavioral properties

Analysis of several behavioral properties is studied and following decidability problems are of special importance:

The boundedness problem

The boundedness problem is, given a marked Petri net $\mathcal{P}$, deciding whether $|R(\mathcal{P})|$ is finite. This problem was first considered by Karp and Miller [35], where it was shown to be decidable using the technique of coverability graph analysis. A Petri net is unbounded iff an ω occurs in the corresponding coverability graph. The algorithm presented there was basically an unbounded search and consequently no complexity analysis was shown. Subsequently, a lower bound of $O(2^{cm})$ space was shown by Lipton in [38], where m is the number of places in the Petri net and c is a constant. Finally, an upper bound of $O(2^{cn \log n})$ space was given by Rackoff in [49]. Here, however, n represents the size or number of bits in the problem instance and c is a constant.

The covering problem

The covering problem is, given a marked Petri net $\mathcal{P}$ and a marking μ, deciding whether there exists $\mu' \in R(\mathcal{P})$ such that $\mu' \geq \mu$. The complexity (both upper and lower bounds) of the covering problem can be derived along a similar line of that of the boundedness problem [49].

The reachability problem

The reachability problem is, given a marked Petri net $\mathcal{P}$ and a marking μ, deciding whether $\mu \in R(\mathcal{P})$. This problem has attracted the most attention in the Petri net community. One reason is that the problem has many real-world applications; furthermore, it is the key to the solutions of several other Petri net problems. Before the decidability question of the reachability problem for general Petri nets was proven by Mayr in 1981 [39], a number of attempts had been made to investigate the problem for restricted classes of Petri nets, in hope of gaining more insights and developing new tools in order to conquer the general Petri net reachability problem. It should be noted that the technique of the coverability graph analysis does not answer the reachability problem as ω abstracts out the exact number of tokens that a place can accumulate, should the place be potentially unbounded.

The containment problem

The containment problem is, given two marked Petri nets $\mathcal{P}_1$ and $\mathcal{P}_2$, deciding whether $R(\mathcal{P}_1) \subseteq R(\mathcal{P}_2)$. In the late 1960's, Rabin first showed the containment problem for Petri nets to be undecidable. Even though the original work of Rabin have never been published, a new proof based on Hilbert's Tenth Problem [16] was presented at MIT in 1972 [4].

The equivalence problem

The equivalence problem: given two marked Petri nets $\mathcal{P}_1$ and $\mathcal{P}_2$, deciding whether $R(\mathcal{P}_1) = R(\mathcal{P}_2)$. In 1975, Hack [28] extended Rabin's result of the containment problem by showing the equivalence problem to be undecidable as well. The proof was also based on Hilbert's Tenth Problem.

The liveness problem

The liveness problem: given a marked Petri net $\mathcal{P}$, deciding whether for every $t \in T, \mu \in R(\mathcal{P})$ there exists a sequence of transitions σ such that $\mu \xrightarrow{\sigma t}$, i.e. t is enabled after firing σ from μ. In [26], several variants of the reachability problem were shown to be recursively equivalent. Among them is the single-place zero reachability problem, i.e. the problem of determining whether a marking with no tokens in a designated place can be reached. Hack also showed the single-place zero reachability problem to be recursively equivalent to the liveness problem, which is then as well decidable.

2.9.2 Backwards-compatible extensions

Conventional Petri nets have several limitations such as the inability to test for zero tokens in a place. Models often tend to become large, with no support for structuring. To remedy these weaknesses, a number of extended Petri net models have been proposed in the literature. Some of these extensions only increase the modeling convenience by making the model more expressive while they do not actually increase the power of the basic model and are completely backwards-compatible, e.g. colored Petri nets.

Colored Petri nets

A colored Petri net (CPN) has each token attached with a color, indicating the identity of the token. Moreover, each place and each transition has attached a set of colors. A transition can fire with respect to each of its colors. By firing a transition, tokens are removed from the input places and added to the output places in the same way as that in original Petri nets, except that a functional dependency is specified between the color of the transition firing and the colors of the involved tokens. The color attached to a token may be changed by a transition firing and it often represents a complex data-value. CPNs lead to compact net models by using of the concept of colors.

2.9.3 Turing complete extensions

The computational power of conventional or backwards-compatible Petri nets such as CPNs is strictly weaker than that of Turing machines [53]. From a theoretical viewpoint, the limitation of Petri nets is precisely due to the lack of abilities to test potentially unbounded places for zero and then act accordingly. With zero-testing capabilities, it is fairly easy to show the equivalence to register machines, which are Turing equivalent. We will mention several extensions, which add properties that cannot be modeled in the original Petri net.

Petri nets with inhibitor arcs

The inhibitor arc connects an input place to a transition. The presence of an inhibitor arc connecting an input place to a transition changes the transition enabling conditions. In the presence of the inhibitor arc, a transition is regarded as enabled if each input place, connected to the transition by a normal arc, contains at least the number of tokens equal to the weight of the arc, and no tokens are present on each input place connected to the transition by the inhibitor arc. The transition firing rule is the same for normally connected places. The firing, however, does not change the marking in the inhibitor arc connected places.

Time petri nets

Time Petri nets (TPN) are Petri nets in which each transition t has attached two times $a \leq b$. Assuming that t was last enabled at time c, then t may fire only during the interval $[c + a, c + b]$ and must fire at time $c + b$ at the latest unless it is disabled before by the firing of another transition. Firing a transition takes no time.

Timed Petri nets

Timed Petri nets (TdPN) are obtained from Petri nets by associating a firing time (delay) to each transition of the net. Moreover, each token has an age property, so the marking of a place p is a finite multiset of ages. The precondition of a transition with input place p is an interval. A transition is enabled if for every input place there exists an appropriate token, i.e. its age is included in the interval.

Prioritized Petri nets

Transactions in prioritized Petri nets have input arcs of two types: normal and prioritized. A place with a token and several transitions enabled from this place will fire the transition with a priority arc first. If there are more than one priority arc outgoing from a place which causes that more than one transition is enabled, then the firing choice is nondeterministic.

Maximal parallel Petri nets

Under the maximal parallel semantics, maximal sets of simultaneously enabled rules are fired. They are of interest due to their close connections with the model of P systems.

2.9.4   Other extensions

While the above extended Petri nets are powerful enough to simulate Turing machines, all nontrivial behavioral properties for such Petri nets become undecidable. A natural and interesting question to ask is: are there Petri nets whose powers lie between conventional Petri nets and Turing machines? As it turns out, the quest for such weaker extensions has attracted considerable attention in recent years.

Reset nets

A reset arc does not impose a precondition on firing, and empties the place when the transition fires. This makes reachability and boundedness undecidable, while some other properties, such as termination, coverability remain decidable.

Transfer nets

Petri nets with transfer arcs (transfer nets) can contain transitions that can also consume all the tokens present in one place regardless of the actual number of tokens and move them to another place. The termination and coverability remain decidable and reachability undecidable, on the other hand, in this case the boundedness is decidable [19].

2.10   Vector addition systems

Vector addition systems were introduced by Karp and Miller [35], and were later shown by Hack [27] to be equivalent to Petri nets.

Definition 2.10.1. *A* **vector addition system** *(VAS) is a pair* $G = (x, W)$*, where* $x \in \mathbb{N}^n$ *is an initial vector and* $W \subseteq \mathbb{Z}^n$ *is a finite set of vectors, where* $n > 0$ *is called the dimension of VAS.*

The initial vector is seen as the initial values of multiple counters and the vectors in W are seen as actions that update the counters. These counters may never drop below zero.

Definition 2.10.2. *The* **reachability set** *of the VAS* $G = (x, W)$ *is the set* $R(G) = \{z | \exists v_1, \ldots, v_j \in W : z = x + v_1 + \ldots + v_j \wedge \forall 1 \le i \le j : x + v_1 + \ldots + v_i \ge 0\}$.

Definition 2.10.3. *A* **vector addition system with states** *(VASS) is a tuple* $G = (x, W, Q, T, p_0)$*, where:*

- *(x, W) is a vector addition system,*

- *Q is a finite set of states,*

- *T is a finite set of transitions of the form* $p \to (q, v)$*, where* $v \in W$ *and* $p, q \in Q$ *are states,*

- $p_0 \in Q$ *is the starting state.*

The transition $p \to (q, v)$ can be applied at vector y in state p and yields the vector $y + v$ in state q, provided that $y + v \ge 0$.

Example 2.10.1. For the Petri net in the Figure 2.1, the corresponding VAS (x, W) is:

- $x = (3, 0, 0, 0)$,

- $W = \{(-1, 1, 1, 1), (1, 0, 0, -1)\}$.

It is known [27] that Petri nets, VAS and VASS are computationally equivalent.

2.11 Büchi automaton

No need to include Buchi yet.

2.12   Calculi of looping sequences

Another tool for describing biological membranes is the formalism Calculus of Looping Sequences (CLS) [6].

In the last few years many formalisms originally developed by computer scientists to model systems of interacting components have been applied to biology. Here, we can mention Petri nets (see Section 2.9). Others, such as P systems (see Chapter 3), have been proposed as biologically inspired computational models and have been later applied to the description of biological systems.

Many of these models either offer only very low-level interaction primitives or they are specialized to the description of some particular kinds of phenomena such as membrane interactions or protein interactions. Finally, P Systems have a simple notation and are not specialized to the description of a particular class of systems, but they are still not completely general. For instance, it is possible to describe biological membranes and the movement of molecules across membranes, and there are some variants able to describe also more complex membrane activities. However, the formalism is not so flexible to allow describing easily new activities observed on membranes without extending the formalism to model such activities.

For these reasons there was a new formalism called Calculus of Looping Sequences introduced.

CLS is a formalism based on term rewriting with some features, such as a commutative parallel composition operator, and some semantic means, such as bisimulations, which are common in process calculi. This permits to combine the simplicity of notation of rewriting systems with the advantage of a form of compositionality.

A CLS model consists of a term and a set of rewrite rules. The term is intended to represent the structure of the modeled system and the rewrite rules to represent the events that may cause the system to evolve.

We start with defining the syntax of terms. We assume a possibly infinite alphabet Σ of symbols.

Definition 2.12.1. *Terms T and sequences S of CLS are given by the following grammar:*

$$T ::= S \ \Big| \ (S)^L \rfloor T \ \Big| \ T|T$$
$$S ::= \varepsilon \ \Big| \ a \ \Big| \ S \cdot S$$

where $a \in \Sigma$ and ε represents the empty sequence. We denote with $\mathcal{T}$ the infinite set of terms and with $\mathcal{S}$ the infinite set of sequences.

A representation of the term $(a \cdot b \cdot c)^L \rfloor ((d \cdot e)^L | f \cdot g)$.

Figure 2.4: Example CLS

In CLS we have a sequencing operator $\_ \cdot \_$, a looping operator $(\_)^L$, a parallel composition operator $\_|\_$ and a containment operator $\_\rfloor\_$. Sequencing can be used to concatenate elements of the alphabet Σ. The empty sequence ε denotes the concatenation of zero symbols. A term can be either a sequence or a looping sequence (that is the application of the looping operator to a sequence) containing another term, or the parallel composition of two terms. By definition, looping and containment are always applied together, hence we can consider them as a single binary operator $(\_)^L \rfloor \_$ which applies to one sequence and one term.

Example 2.12.1. In the Figure 2.4 we show an example of CLS and its visual representation. The same structure may be represented by syntactically different terms, e.g. $(b \cdot c \cdot a)^L \rfloor (f \cdot g | (e \cdot d)^L)$. We introduce a structural congruence relation to identify such terms.

Definition 2.12.2. *The structural congruence relations $\equiv_S$ and $\equiv_T$ are the least congruence relations on sequences and on terms, respectively, satisfying the following rules:*

- $S_1 \cdot (S_2 \cdot S_3) \equiv_S (S_1 \cdot S_2) \cdot S_3$

- $S \cdot \varepsilon \equiv_S \varepsilon \cdot S \equiv_S S$

- $S_1 \equiv_S S_2 \text{ implies } S_1 \equiv_T S_2 \text{ and } (S_1)^L \rfloor T \equiv_T (S_2)^L \rfloor T$

- $T_1|T_2 \equiv_T T_2|T_1$

- $T_1|(T_2|T_3) \equiv_T (T_1|T_2)|T_3$

- $T|\varepsilon \equiv_T T$

- $(\varepsilon)^L \rfloor \varepsilon \equiv_T \varepsilon$

- $(S_1 \cdot S_2)^L \rfloor T \equiv_T (S_2 \cdot S_1)^L \rfloor T$

Note that the last rule does not introduce the commutativity of sequences, but only says that looping sequences can rotate.

What could look strange in CLS is the use of looping sequences for the description of membranes, as sequencing is not a commutative operation and this do not correspond to the usual fluid representation of membrane surface in which objects can move freely. What one would expect is to have a multiset or a parallel composition of objects on a membrane. For this reason, a variant called CLS+ was introduced in [40], in which the looping operator can be applied to a parallel composition of sequences.

Definition 2.12.3. *Terms T, branes B and sequences S of CLS+ are given by the following grammar:*

$$T ::= S \,\Big|\, (B)^L \rfloor T \,\Big|\, T|T$$
$$B ::= S \,\Big|\, B|B$$
$$S ::= \varepsilon \,\Big|\, a \,\Big|\, S \cdot S$$

The structural congruence relation of CLS+ is a trivial extension of the one of CLS. The only difference is that commutativity of branes replaces rotation of looping sequences. CLS+ models can be translated into CLS models, while preserving the semantics of the model [6]. Moreover, in the same paper, a traslation of maximal parallel P system to CLS is shown.

2.13 Graph theory

Definition 2.13.1. *A **graph** is a pair $G = (V, E)$ of sets such that $E \subseteq V \times V$ and $V \cap E = \emptyset$. The elements of V are the vertices (or nodes) of the graph G, the elements of E are its edges.*

The vertex set of a graph G is denoted by $V(G)$, its edge set as $E(G)$.

Definition 2.13.2. *The **order of the graph** G is the number of its vertices, denoted by $|G|$. Graphs are (in)finite iff their order is (in)finite.*

Definition 2.13.3. *A vertex $v \in V$ is* **incident** *with an edge $e \in E$ iff $v \in e$, i.e. either $e = (v, y)$, where $y \in V$ or $e = (x, v)$, where $x \in V$.*

Definition 2.13.4. *Two vertices $x, y \in V(G)$ are* **adjacent** *iff $(x, y) \in E(G)$.*

Definition 2.13.5. *A* **path** *is a non-empty graph $P = (V, E)$, where $V = \{x_0, x_1, \ldots, x_k\}$ and $E = \{(x_i, x_{i+1}) | 0 \leq i < k\}$ and the x_i are all distinct.*

Definition 2.13.6. *If $P = (V, E)$ is a path where $V = \{x_0, x_1, \ldots, x_k\}$ and $|V| \geq 3$, then the graph $C = (V, E \cup \{(x_k, x_0)\})$ is called a* **cycle**.

Definition 2.13.7. *A graph $G = (V, E)$ is a* **subgraph** *of a graph $G' = (V', E')$ iff $V \subseteq V'$ and $E \subseteq E'$. We denote it by $G \subseteq G'$ and also say that G' contains G.*

Definition 2.13.8. *A non-empty graph G is called* **connected** *iff any two of its vertices are linked by a path in G.*

Definition 2.13.9. *A graph not containing any cycle is called a* **forest**.

Definition 2.13.10. *A connected forest is called a* **tree**.

Sometimes it is convenient to consider one vertex of a tree as special. Such a vertex is then called the root of this tree.

Definition 2.13.11. *A tree T with a fixed root is called a* **rooted tree**. *The root node of T is denoted by r_T.*

Definition 2.13.12. *Let d be a node of a non-root tree T, i.e. $d \in V(T) \setminus \{r_T\}$. As T is a tree, there is a unique path from d to r_T [18]. The node adjacent to d on that path is also unique and is called a* **parent node** *of d and is denoted by $parent_T(d)$.*

Definition 2.13.13. *Let T_1, T_2 be two rooted trees. A bijection $f : V(T_1) \to V(T_2)$ is an* **isomorphism** *iff $\forall x, y \in V(T) : (x, y) \in E(T_1) \Leftrightarrow (f(x), f(y)) \in E(T_2)$.*

Rooted trees T_1 and T_2 are called isomorphic if there exists an isomorphism on their nodes.

2.14 Bisimulations

Definition 2.14.1. *A* **state transition system** *is a pair* $(S, \rightarrow)$*, where* S *is a set of states and* $\rightarrow \subseteq S \times S$ *is a binary transition relation over* S*.*

If $p, q \in S$, then $(p, q) \in \rightarrow$ is usually written as $p \rightarrow q$. This represents the fact that there is a transition from state p to state q.

Definition 2.14.2. *A* **labeled state transition system** *(LTS) is a tuple* $(S, A, \rightarrow)$*, where* S *is a set of states,* A *is a set of labels and* $\rightarrow \subseteq S \times A \times S$ *is a ternary transition relation.*

If $p, q \in S$ and $a \in A$, then $(p, a, q) \in \rightarrow$ is usually written as $p \xrightarrow{a} q$. This represents the fact that there is a transition from state p to state q with a label a.

Definition 2.14.3. *Let* $(S_1, A, \rightarrow)$ *and* $(S_2, A, \rightarrow)$ *be two labeled transition systems. A* **simulation** *is a binary relation* $R \subseteq S_1 \times S_2$ *such that if* $(s_1, s_2) \in R$ *then for each* $s_1 \xrightarrow{a} t_1$ *there is some* $s_2 \xrightarrow{a} t_2$ *such that* $(t_1, t_2) \in R$*.*

Definition 2.14.4. *Let* $(S_1, A, \rightarrow)$ *and* $(S_2, A, \rightarrow)$ *be two labeled transition systems. A* **bisimulation** *is a binary relation* $R \subseteq S_1 \times S_2$ *such that if* $(s_1, s_2) \in R$ *then:*

1. for each $s_1 \xrightarrow{a} t_1$ *there is some* $s_2 \xrightarrow{a} t_2$ *such that* $(t_1, t_2) \in R$*,*

2. for each $s_2 \xrightarrow{a} t_2$ *there is some* $s_1 \xrightarrow{a} t_1$ *such that* $(t_1, t_2) \in R$*.*

Sometimes, we want to allows systems to perform internal (silent) steps, of which the impact is considered unobservable. There are multiple notions of the bisimulation. The definition above is called a strong bisimulation.

Definition 2.14.5. *A* **labeled transition system with silent actions** *is a triple* $(S, A, \rightarrow)$*, where* S *is a set of states,* A *is a set of actions, also a silent action* τ *is assumed that is not in* A *and* $\rightarrow \subseteq (A \cup \tau) \times S$ *is the transition relation.*

We use A_τ to denote $A \cup \{\tau\}$. By $\xrightarrow{\tau^*}$ we denote the transitive and reflexive closure of $\xrightarrow{\tau}$. For $a \in A$ we define the relation $\xRightarrow{a}$ on S by $r \xRightarrow{a} s$ iff there exists $r', s' \in S$ such that $r \xRightarrow{\tau^*} r' \xrightarrow{a} s' \xRightarrow{\tau^*} s$.

Definition 2.14.6. *A* **run** *of a labeled transition system is a finite, non-empty alternating sequence* $\rho = s_0 a_0 s_1 a_1 \ldots s_{n-1} a_{n-1} s_n$ *of states and actions, beginning and ending with a state such that for* $0 \le i < n : s_i \xrightarrow{a_i} s_{i+1}$*.*
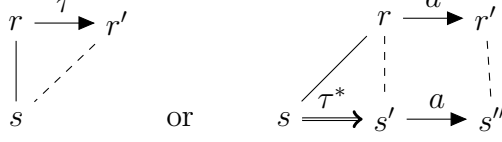
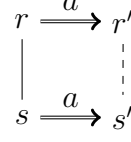Figure 2.5: Transfer diagram for branching bisimulation

Figure 2.6: Transfer diagram for weak bisimulation

We also say that ρ is a run from s_0. We denote $first(\rho) = s_0$ and $last(\rho) = s_n$.

Definition 2.14.7. *A relation $R \subseteq S \times S$ is called a* **branching bisimulation** *if it is symmetric and satisfies the following transfer property: if rRs and $r \xrightarrow{a} r'$ then either $a = \tau$ and $r'Rs$ of there exist $s', s'' \in S$ such that $s \xRightarrow{\tau^*} s' \xrightarrow{a} s''$, rRs' and $r'Rs''$.*

Example 2.14.1. The diagrams shown in the Figure 2.5 summarize the main transfer properties of branching bisimulation. We have used the dashed lines to represent the relations that have to be established in order to conclude that the two states connected by the plain line are equivalent.

We could have strengthened the above definition of branching bisimulation by requiring all intermediate states in $s \xRightarrow{\tau^*} s'$ to be related with r. However, that would lead to the same equivalence relation [17].

Definition 2.14.8. *A relation $R \subseteq S \times S$ is called a* **weak bisimulation** *if it is symmetric and satisfies the following transfer property: if rRs and $r \xRightarrow{a} r'$ then there exist $s' \in S$ such that $s \xRightarrow{a} s'$ and $r'Rs'$.*

This means that two states are considered equivalent if they lead via the same sequences of visible actions to equivalent states. The intermediate states are not questioned. The diagram in the Figure 2.6 summarizes the transfer property for the weak bisimulation. We have used the same notational conventions of the Figure 2.5.

Example 2.14.2. In the Figure 2.7 a participant in a contest opens a door and then it is decided if he wins a car or flowers. In the Figure 2.8 the price is decided beforehand so when a participant opens a door, he can directly see his outcome. We can model these events (open door, win car, win flowers) as actions of a labeled transition system.
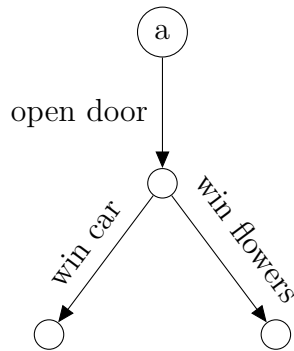
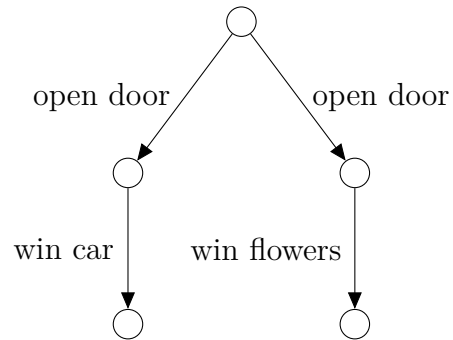Figure 2.7: Example run of a labeled transition system

Figure 2.8: Example run of a labeled transition system with hidden branching before the first action

The contestant is not interested in the inner states of the system, he only sees these events. For him, the two systems are bisimular in terms of the weak bisimulation.

In the state right after opening the door and before the price is given there is no way for him knowing the outcome. However, in the system from the Figure 2.8 some people who organize the contest already know what is the outcome. For them, the system is not bisimilar in terms of the branching bisimulations.

Chapter 3

P systems

In previous chapter we introduced the notions of membrane and membrane structure.

The next step is to place certain objects in the regions delimited by the membranes. The objects are identified by their names, mathematically symbols from a given alphabet.

Several copies of the same object can appear in a region, so we will work with multisets of objects.

In order to obtain a computing device, we will allow the objects to evolve according to evolution rules. Any object, alone or together with another objects, can be transformed in other objects, can pass through a membrane, and can dissolve the membrane in which it is placed.

All objects evolve at the same time, in parallel manner across all membranes.

The evolution rules are hierarchized by a priority relation, which is a partial order.

These aspects all together forms a P system as introduced in [44]. An example of a P system containing rules for computing the Fibonacci sequence can be seen in the figure 3.1.

In section 3.1 we will provide formal definition of a P system.

3.1 Definitions

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 $1, 2, \ldots, m$,

- $w_1, w_2, \ldots w_m$ are multisets of symbols present in the regions $1, 2, \ldots, m$ of the membrane structure,

- $R_1, R_2, \ldots R_m$ are finite sets of rewriting rules associated with the regions $1, 2, \ldots, m$ 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 or through membrane to one of the child regions. An example of such rule is the following: $abb \rightarrow (a, here)(b, in)(c, out)(c, here)$.

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 $a \rightarrow b$ and in the second membrane there is a rule $b \rightarrow a|a$. 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.
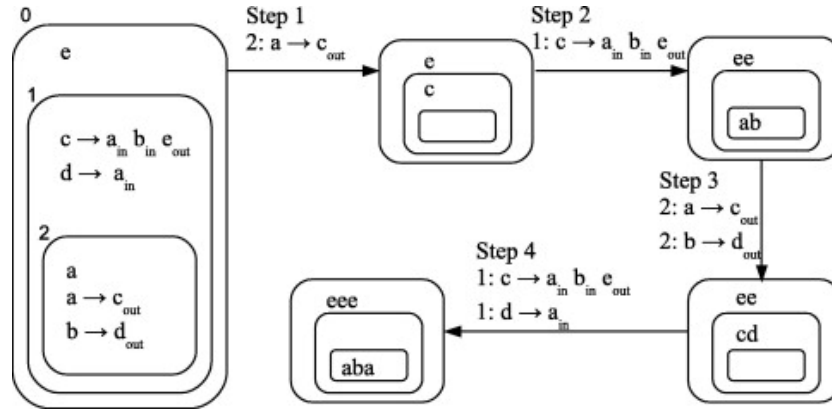
Figure 3.1: P system computing a Fibonacci sequence [9]

3.2   P system variants

Besozzi in his PhD thesis (see [7]) formulates three criteria that a good P system variant should satisfy:

1. It should be as much realistic as possible from the biological point of view, in order not to widen the distance between the inspiring cellular reality and the idealized theory.

2. It should result in computational completeness and efficiency, which would mean to obtain universal (and hence, programmable) computing devices, with a powerful and useful intrinsic parallelism;

3. It should present mathematical minimality and elegance, to the aim of proposing an alternative framework for the analysis of computational models.

In membrane computing, many models are equal in power with Turing machines. We should say they are Turing complete (or computationally complete), but because the proofs are always constructive, starting the constructions from these proofs from universal Turing machines or from equivalent devices, we obtain universal P systems (able to simulate any other P system of the given type). That is why we speak about universality results, and not about computational completeness.

3.2.1   Accepting vs generating

In the Chomsky hierarchy, there are language acceptors (finite automata, Turing machines) and language generators (formal grammars).

Bordhin in [8] extends grammars to allow for accepting languages by interchanging the left side with the right side of a rule. The mode will apply rewriting rules to an input word and accept it when it reaches the starting nonterminal. However, the input word consists of terminal symbols, which could not be rewritten when using original definition, hence they consider the pure version of various grammar types where they give up the distinction between terminal and nonterminal symbols.

The regular, context-free, context-sensitive and recursively enumerable languages were shown to have equal power in accepting and generating mode. Some other grammars (programmed grammars with appearance checking) are shown to be more powerful in accepting mode than in generating mode. For deterministic Lindenmayer systems, the generating and accepting mode are incomparable.

It can be interesting to investigate accepting and generating mode also in P system variants. Barbuti in [5] shown that in the nondeterministic case, when either promoters or cooperative rules are allowed, acceptor P systems have shown to be universal. The same in known to hold for the corresponding classes of nondeterministic generator P systems. In the deterministic case, acceptor P systems have been shown to be universal only if cooperative rules are allowed. Universality has been shown not to hold for the corresponding classes of generator P systems.

3.2.2 Active vs passive membranes

Most of the studied P system variants assumes that the number of membranes can only decrease during a computation, by dissolving membranes as a result of applying evolution rules to the objects present in the system. A natural possibility is to let the number of membranes also to increase during a computation, for instance, by division, as it is well-known in biology. Actually, the membranes from biochemistry are not at all passive, like those in the models briefly described above. For example, the passing of a chemical compound through a membrane is often done by a direct interaction with the membrane itself (with the so-called protein channels or protein gates present in the membrane); during this interaction, the chemical compound which passes through membrane can be modified, while the membrane itself can in this way be modified (at least locally).

In [47] Păun considers P systems with active membranes where the central role in the computation is played by the membranes: evolution rules are associated both with objects and membranes, while the communication through membranes is performed with the direct participation of the membranes; moreover, the membranes can not only be dissolved, but they also

can multiply by division. An elementary membrane can be divided by means of an interaction with an object from that membrane.

Each membrane is supposed to have an electrical polarization (we will say charge), one of the three possible: positive, negative, or neutral. If in a membrane we have two immediately lower membranes of opposite polarizations, one positive and one negative, then that membrane can also divide in such a way that the two membranes of opposite charge are separated; all membranes of neutral charge and all objects are duplicated and a copy of each of them is introduced in each of the two new membranes. The skin is never divided. If at the same time a membrane is divided and there are objects in this membrane which are being rewritten in the same step, then in the new copies of the membrane the result of the evolution is included.

In this way, the number of membranes can grow, even exponentially. As expected, by making use of this increased parallelism we can compute faster. For example, the SAT problem, which is NP complete, can be solved in linear time, when we consider the steps of computation as the time units. Moreover, the model is shown to be computationally universal.

3.2.3 Context in rules

Rewriting rules in P systems can be cooperative and non-cooperative, like in Chomsky's context-free and context-sensitive grammars. Non-cooperative rules are restricted to use only one object on the left side and cooperative rules do not have this restriction. P systems with cooperative rules are universal [44], while P systems with non-cooperative rules only characterize Parikh image of context-free languages ($PsCF$) [50].

Păun [44] also defines P systems with catalysts where catalysts are a specified subset of the alphabet. Rewriting rules can contain catalysts, which are not modified by applying the rule. Surprisingly, P systems with catalytic rules are universal, actually two membranes in the P system are sufficient to achieve universality.

In systems where only catalytic rules (purely catalytic systems [30]), three catalysts are enough [23].

Freund in [23] also shows that two catalysts and one membrane are enough and raised an open problem whether one catalyst is sufficient. He conjectured that for computationally universal P systems the results obtained in this paper are optimal not only with respect to the number of membranes (2), but also with respect to the number of catalysts.

From some point of view, catalysts are way too powerful in restricting the parallelism - they directly participate in the rules, hence the number of

catalytic rules that can be applied in one step, is bounded by number of catalysts.

A variant with promoters and inhibitors have been proposed (see [32]).

In the case of promoters, the rules are possible only in the presence of certain symbols. An object p is a promoter for a rule $u \to v$ and we denote this by $u \to v|_p$, if the rule is active only in the presence of object p. Note that unlike in the case with catalysts, promoters allow the associated rules to be applied as many times as possible.

An object i is inhibitor for a rule $u \to v$ and we denote this by $u \to v|_{\neg i}$, if the rule is active only if inhibitor i is not present in the region. One of our results (see section 4.1) uses inhibitors as a tool to achieve universality for sequential P systems.

Ionescu in [32] shows that P systems with non-cooperative catalytic rules with only one catalyst and with promoters / inhibitors are universal.

Non-cooperative rules with no catalysts and with inhibitors were studied in [51], the equivalence with Lindenmayer systems ($ET0L$ as defined in section 2.6) was proved.

Dang [29] proposes a simple cooperative system (SCO) as a P system where the only rules allowed are of the form $a \to v$ or of the form $aa \to v$, where a is a symbol and v is a (possibly null) string of symbols not containing a. This variant is investigated with various modes of parallelism, so their results will be mentioned in the subsection 3.2.7

3.2.4   Rules with priorities

In the original definition of a P system [44], a partial order relation over set of rewriting rules have been specified. The rule can be used only if no rule of a higher priority in the region can be applied at the same time.

Sosík in [52] showed that the priorities may be omitted from the model without loss of computational power.

3.2.5   Energy in P systems

Various notions of energy has been proposed for use in P systems. Păun in [48] considers a P system where each evolution rule "produces" or "consumes" some quantity of energy, in amounts which are expressed as integer numbers. In each moment and in each membrane the total energy involved in an evolution step should be positive, but if "Too much" energy is present in a membrane, then the membrane will be destroyed (dissolved). This variant was investigated in two cases, both were shown to be universal:

1. when using only two membranes and unbounded amount of energy,

2. when using arbitrarily many membranes and a bounded energy associated with rules

Freund in [24] introduced a new variant where the rules are assigned directly to membranes (every rule consume objects on one side of the membrane and produce objects on the other side) and every membrane carries an energy value that can be changed during a computation by objects passing through the membrane.

This variant is universal even in sequential mode if we allow priorities on the objects. When omitting the priority relation, only the family of Parikh sets generated by context-free matrix grammars ($PsMAT$ as defined in section 2.4) is obtained.

3.2.6 Symport / antiport rules

Păun in [43] proposes a new way of communicating between membranes.

Symports allow two chemicals to pass together through a membrane in the same direction using symport rules of type (ab, in) or (ab, out). Antiports allow two chemicals to pass simultaneously through a membrane in opposite directions using antiport rules of type $(a, in; b, out)$.

Surprisingly, a P system variant, where only the symport / antiport rules are used are computationally complete. Five membranes are enough for this result. If more than two chemicals may collaborate when passing through membranes, two membranes are sufficient for universality. These results are proven in [43].

3.2.7 Parallelism options

Original definition of P system (see [44]) uses maximal parallelism when doing a step of computation. There is an obvious biological motivation relying on the assumption that "if we wait long enough, then all reaction which may take place will take place". This condition is rather powerful, because it decreases the non-determinism of the system's evolution. For various reasons ranging from looking for more realistic models to just the mathematical challenge, the maximal parallelism was questioned.

Dang in [15] investigates the sequential mode. In each step, from the set of applicable rules across all membrane one is nondeterministically chosen and applied.

Definition 3.2.1. *A computation step of a sequential P system is a relation $\Rightarrow$ on the set of configurations such that $C_1 \Rightarrow C_2$ holds iff there is an applicable rule in a membrane in C_1 such that applying that rule can result in C_2.*

For purely catalytic systems with 1 membrane, the sequential mode generates only the semilinear sets and thus is strictly weaker than the maximally parallel version. Sequential version of symport / antiport systems are equivalent to vector addition systems making it strictly weaker than the original maximally parallel version.

Investigation of the sequential mode continues in [31]. Sequential P system without priorities with cooperative rules with rules for membrane dissolution are not universal by showing they can be simulated by vector addition systems with states (VASS). This holds even when the membrane creation is allowed for bounded number of created membranes. However, if any number of membranes are allowed to be created, the system becomes universal. This result was shown by simulation of the register machine (see section 2.5).

We have further investigated this variant (sequential P system without priorities with cooperative rules) in chapter 4 by allowing rules with inhibitors, which resulted in universality.

Dang in [29] proposes several restricted versions of parallelism. n-**Max-Parallel** version nondeterministically selects a maximal subset of at most n rules to apply. It is proved that 9-**Max-Parallel** SCO (defined in the subsection 3.2.3) is universal. $\leq n$-**Parallel** version is similar, but does not require the condition of a maximal subset of rules. It is shown to be weaker than n-**Max-Parallel** version. n-**Parallel** version requires the size of the subset of rules to apply to be exactly n. All three versions are equal to the sequential mode when $n = 1$. For non-universality results, Dang used the proof technique by simulation by vector addition systems. Our future research may be inspired by this technique.

Ciobanu in [14] proposes a minimal parallelism: for each region if at least one rule can be applied, then at least one rule will be applied. The symport / antiport rules variant and variant with active membranes were both shown to be universal.

Freund in [22] studied the asynchronous mode of P systems, where in each step, arbitrary many rules can be applied. The application of rules is hence done in parallel way, but are not synchronized or somewhat controlled. In many cases the sequential and asynchronous modes were shown to be equivalent.

3.3 Case studies

3.3.1 Vultures in Pyrenees

Spanish researchers in [10] presented a model of an ecosystem related to the Bearded Vulture in Pyrenees in Spain by using P systems. They have constructed a simulator to validate the designed P system, which allows them to analyze the evolution under different initial conditions.

3.3.2 Cellular Signalling Pathways

Perez in [45] proposed a model for EGFR Signalling Cascade. More than 60 proteins were included and 160 chemical reactions. Membrane structure consists of 3 regions: the environment, the cell surface and the cytoplasm.

3.3.3 Solving SAT in linear time

Polynomial time solutions to NP-complete problems by means of P systems are achieved by trading time (number of computation steps) for space (number of membranes and objects). This is inspired by the capabililty of cells to produce an exponential number of new membranes in polynomial time.

However, many simulators of P system are inefficient since they cannot handle the parallelism of these devices. Nowadays, we are witnessing the consolidation of the GPUs as a parallel framework to compute general purpose applications such as bitcoin mining. The simulation of P systems with active membranes using GPUs is analysed in [12] and an efficient linear solution to the SAT problem is illustrated.

They compared it to the classical simulator and reported up to 94x of speedup for 256 literals in the formula. The major constraint of this parallel simulation is the GPU memory size, which can be overcome with a data partition on a cluster of GPUs.

3.3.4 Implementation of P systems in vitro

"In vitro" are studies in experimental biology that uses components of an organism that have been isolated from their usual biological surroundings in order ot provide a more convenient analysis.

There was a planned experiment of computing the Fibonacci sequence using P systems in vitro (see [25]) using test tubes as membranes and DNA molecules as objects, evolving under the control of enzymes. Number of

objects in a multiset was represented by a pre-defined "mole" of the substance and synchronization was obtained by "waiting enough", such that all reactions that can take place in a test tube actually take place. Hence, the variant where reactions don't cycle is required (Local loop-free P systems). Communication was done by moving all the relevant objects to the next tube in a mechanical way. The final result was read by spectrometry.

The proposed framework has many difficulties. The notable ones are:

- Is there any chance to solve NP-complete problems in this frameworks?

- Are LL-free P systems universal?

Chapter 4

On the edge of universality of sequential P systems

In this chapter we propose various modifications of sequential P systems and study their effect on computing power and decidability of behavioral properties.

We have presented the current state of the research of the P system variants. Especially the parallelism options have been investigated very well, but there are still some gaps that should be filled. For many variants the universality has been proven only when working in the maximally parallel mode. In most cases the sequential mode is strictly weaker, but this is not always true.

We have studied several variants of sequential P systems in order to obtain universality without using maximal parallelism.

4.1 Inhibitors

(From project proposal):

A sequential variant without priorities and with cooperative rules is not universal (see [29]). They have tried modifying the variant to make it universal and showed that with rules for membrane creation with unbounded number of membranes it became universal.

We tried other approach using rules with inhibitors. This variant is universal in both generating and accepting case and we don't even have to use membrane dissolution rules. In generative case the model is able to simulate maximal parallel P system and the accepting model can simulate a register machine.

Future plans include research other more restricted variants such as omitting cooperation in the rules or restrict the power of inhibitors.

Original definition of P systems with inhibitors (see [32]) allow to use only one inhibitor per rule, e.g. $u \rightarrow v|_{\neg i}$. Alternative definition (see [2]) allow to use whole inhibitor set in the rule like $u \rightarrow v|_{\neg B}$, where B is a set of objects. Such a rule can be applied only if no element of B is present in the region.

For sequential P systems, the following lemma will show the equivalence of these definitions. However, special condition must be fulfilled, the regions with such rules cannot be empty.

Lemma 4.1.1. *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 4.1.1. Consider P system with alphabet V. For each rule $u \rightarrow v|_{\neg B}$, where $B = \{b_1, b_2, \ldots, b_n\}$ we will have rules:

$$c \rightarrow c|GONE_b|_{\neg b} \text{ for all } c \in V, b \in B$$
$$u|GONE_{b_1}|GONE_{b_2}|\ldots|GONE_{b_n} \rightarrow v|GONE_{b_1}|GONE_{b_2}|\ldots|GONE_{b_n}$$

Note that symbols $GONE_b$ are created automatically when some object c is present in the region.

Theorem 4.1.2. *The sequential P system with inhibitors defines the same Parikh image of language as P system with maximal parallelism.*

Proof 4.1.2. We show that we can simulate maximal parallel step of P system with several steps of sequential P system with inhibitors. The proof is quite technical with some workarounds.

It is important to note that in the maximal parallel step the rewriting occurs in all membranes, so we need to synchronize this process. Every membrane will have a state, represented as an object.

The RUN state represents that the rewriting still occurs. When there is no more rules to apply, the region has done its maximal parallel step and proceeds to the state $SYNCHRONIZE$. Other states are just technical - we need to implement sending objects between membranes and preparing for the next maximal parallel step by unmarking newly created objects in the current maximal parallel step, which have been marked to prevent double rewriting in one step.

- *RUN*: Rewriting occurs. Objects that are to be sent to parent membrane are directly sent because the parent membrane is already in *RUN*

or $SYNCHRONIZE$ phase, 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 previous phase waiting to restore symbols from previous step. Current symbols could interfere with them and be rewritten twice in this step. Such objects are only marked as "to be sent down": $a^{\downarrow\prime}$

- $SYNCHRONIZE$: Rewriting has ended and membrane is waiting to get signal $SYNCED$ from parent membrane to continue to next step.

- $SENDDOWN$: Signal was caught and now all descendant membranes are in $SYNCHRONIZE$ phase so $a^{\downarrow\prime}$ can be sent down.

- $RESTORE$: All a' symbols are restored to a, so the next step of rewriting can take place.

- For every rule $r_i \in R$ such that

$$r_i = a_1^{M(a_1)} a_2^{M(a_2)} \ldots a_n^{M(a_n)} \to a_1^{N(a_1)} a_2^{N(a_2)} \ldots a_n^{N(a_n)}$$

  we will have the following rules:

$$a_1^{M(a_1)-m_1} \dot{a}_1^{m_1} a_2^{M(a_2)-m_2} \dot{a}_2^{m_2} \ldots a_n^{M(a_n)-m_n} \dot{a}_n^{m_n} |RUN$$
$$\to a_1'^{N(a_1)} a_2'^{N(a_2)} \ldots a_n'^{N(a_n)} |RUN$$

  There will be such rule for each $0 \le m_i \le M(a_i)$. It represents the idea that $\dot{a}$ can be used in rewriting in the same way as a. Right side of the rules contains symbols a', that prevents the symbols to be rewritten again.

- 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 $\dot{a}$.

  $RUN \rightarrow UNUSABLE_i | RUN|_{\neg\{UNUSABLE_i, a, \dot{a}\}}$

- 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.

  $RUN \rightarrow UNUSABLE_i | RUN|_{\neg\{UNUSABLE_i, a, \dot{a}\}}$

  $RUN \rightarrow UNUSABLE_i | RUN|_{\neg\{UNUSABLE_i, b, \dot{b}\}}$

- 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.

  $RUN \rightarrow UNUSABLE_i | RUN|_{\neg\{UNUSABLE_i, a\}}$

- For every membrane with label i there will be rule:

$$UNUSABLE_1 | UNUSABLE_2 | \dots | UNUSABLE_m | RUN$$
$$\rightarrow SYNCHRONIZE | SYNCTOKEN_i \uparrow$$

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.

- For every membrane there will be a rule:

$$SYNCHRONIZE | SYNCTOKEN_j$$
$$\rightarrow SYNCHRONIZE | SYNCTOKEN_j \uparrow$$

Membrane resends all sync token to parent membrane.

- In 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 $a^{\downarrow\prime}$.

$$SYNCTOKEN_1 | \dots | SYNCTOKEN_k | SYNCHRONIZE$$
$$\rightarrow SENDDOWN$$

- Every membrane other than skin membrane have to receive the signal to go to senddown phase:

  $SYNCHRONIZE|SYNCED \rightarrow SENDDOWN$

- Every membrane will have rules for every symbol $a \in V$ to send down all unsent object that should have been sent down:

  $SENDDOWN|a^{\downarrow'} \rightarrow SENDDOWN|a' \downarrow$

- Every membrane will have rule for detecting when all such objects have been sent and it goes to restore phase:

  $SENDDOWN \rightarrow RESTORE|_{\neg\{a_i^{\downarrow'}|1 \leq i \leq n\}}$

- In restore phase all symbols a' will be rewritten to a in order to be able to be rewritten in next maximal parallel step:

  $RESTORE|a' \rightarrow RESTORE|a$

- When restore phase ends, it sends down signal that all membranes have been synchronized and next phase of rewriting has began in upper membranes:

  $RESTORE \rightarrow RUN|SYNCED \downarrow |_{\neg\{a_i'|1 \leq i \leq n\}\cup\{GONE_i|1 \leq i \leq n\}}$

Phase of membrane is represented by object so the region is never empty and by the Lemma 4.1.1 the rules with set of inhibitors can be simulated by single inhibitors.

We have also reached this result in the accepting case by simulation of register machines.

Theorem 4.1.3. *Sequential P systems with inhibitors can simulate register machines and thus equal $PsRE$.*

Proof 4.1.3. 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.

4.2 Active membranes

In this section we study a variant of sequential systems where universality can be achieved without checking for zero by allowing membranes to be created unlimited number of times [31]. Such P systems are called active P systems. Contrary, if we place a limit on the number of times a membrane is created, such active P system is only equivalent to vector addition systems, hence not universal.

In the subsection 4.2.1 we will introduce membrane structure and formally define membrane configuration and active P system because standard definitions are not convenient for our formal proofs.

The subsection 4.2.3 contains two main results. The existence of an infinite computation is surprisingly[1] shown to be decidable. On the other hand, the existence of a halting computation is shown to be undecidable.

---

[1]At first sight it seems to relate to the Rice's theorem, but it is not the case.

4.2.1 Active P systems

Definition 4.2.1. *Let Σ be a set of objects. Recall that $\mathbb{N}^{\Sigma}$ contains all multisets of objects from Σ. A* **membrane configuration** *is a tuple (T, l, c), where:*

- *T is a rooted tree,*

- *$l \in \mathbb{N}^{V(T)}$ is a mapping that assigns for each node of T a number (label), where $l(r_T) = 1$, so the skin membrane is always labeled with 1,*

- *$c \in (\mathbb{N}^{\Sigma})^{V(T)}$ is a mapping that assigns for each node of T a multiset of objects from Σ, so it represents the contents of the membrane.*

Definition 4.2.2. *An* **active P system** *is a tuple $(\Sigma, C_0, R_1, R_2, \ldots, R_m)$, where:*

- *Σ is a set of objects,*

- *C_0 is initial membrane configuration,*

- *$R_1, R_2, \ldots R_m$ are finite sets of rewriting rules associated with the labels $1, 2, \ldots, m$ and can be of forms:*

  - *$u \to w$, where $u \in \Sigma^+$, $w \in (\Sigma \times \{\cdot, \uparrow, \downarrow_j\})^*$ and $1 \leq j \leq m$,*
  - *$u \to w\delta$, where $u \in \Sigma^+$, $w \in (\Sigma \times \{\cdot, \uparrow, \downarrow_j\})^*$ and $1 \leq j \leq m$,*
  - *$u \to [_j v]_j$, where $u \in \Sigma^+, v \in \Sigma^*$ and $1 \leq j \leq m$.*

Although rewriting rules are defined as strings, u, v and w represent multisets of objects from Σ. For the first two forms, each rewriting rule may specify for each object on the right side, whether it stays in the current region (we will omit the symbol $\cdot$), moves through the membrane to the parent region ($\uparrow$) or to a specific child region ($\downarrow_j$, where j is a label of a membrane). If there are more child membranes with the same label, one is chosen nondeterministically. We denote these transfers with an arrow immediately after the symbol. An example of such rule is the following: $abb \to ab \downarrow_2 c \uparrow c\delta$. Symbol δ at the end of the rule means that after the application of the rule, the membrane is dissolved and its contents (objects, child membranes) are propagated to the parent membrane. Active P systems differs from classic (passive) P systems in ability to create new membranes by rules of the third form. Such rule will create new child membrane with a given label j and a given multiset of objects v as its contents.

Definition 4.2.3. *For active P system $(\Sigma, C_0, R_1, R_2, \ldots, R_m)$, configuration $C = (T, l, c)$, membrane $d \in V(T)$ the rule $r \in R_{l(d)}$ is* **applicable** *iff:*

- $r = u \rightarrow w$ *and* $u \subseteq c(d)$ *and* $\forall (a, \downarrow_k) \in w \exists d_2 \in V(T) : l(d_2) = k \wedge parent(d_2) = d$,

- $r = u \rightarrow w\delta$ *and* $u \subseteq c(d)$ *and* $\forall (a, \downarrow_k) \in w \exists d_2 \in V(T) : l(d_2) = k \wedge parent(d_2) = d$ *and* $d \neq r_T$,

- $r = u \rightarrow [_j v]_j$ *and* $u \subseteq c(d)$.

In this section we assume only sequential systems, so in each step of the computation, there is one rule nondeterministically chosen among all applicable rules in all membranes to be applied as already stated in the definition 3.2.1.

4.2.2 Active membranes with a limit on the total number of membranes

For the simplicity of proofs it is convenient to introduce a variant with a global limit upon the membrane structure. We achieve this by restricting the rule application such that if the rule would result in a structure exceeding the limit, the rule will not be applicable.

Definition 4.2.4. *An* **active P system with a limit on the total number of membranes** *is a tuple* $\Pi = (\Sigma, L, C_0, R_1, R_2, \ldots, R_m)$, *where:*

- $(\Sigma, C_0, R_1, R_2, \ldots, R_m)$ *is an active P system from the definition 4.2.2,*

- $L \in \mathbb{N}$ *is a limit on thetotal number of membranes.*

Anytime during the computation, a configuration (T, l, c) is not allowed to have more than L membranes, so the following invariant holds: $|V(T)| \leq L$. This is achieved by adding a constraint for rule of the form $r = u \rightarrow [_k v]_k$, which is defined to be applicable iff $u \subseteq c(d)$ and $|V(T)| < L$. If the number of membranes is equal to L, there is no space for newly created membrane, so in that case such rule is not applicable.

Definition 4.2.5. *For active P system with a limit on the total number of membranes* $(\Sigma, L, C_0, R_1, R_2, \ldots, R_m)$, *configuration* $C = (T, l, c)$, *membrane* $d \in V(T)$ *the rule* $r \in R_{l(d)}$ *is* **applicable** *iff:*

- $r = u \rightarrow w$ *and* $u \subseteq c(d)$ *and* $\forall (a, \downarrow_k) \in w \exists d_2 \in V(T) : l(d_2) = k \wedge parent(d_2) = d$,

- $r = u \rightarrow w\delta$ *and* $u \subseteq c(d)$ *and* $\forall (a, \downarrow_k) \in w \exists d_2 \in V(T) : l(d_2) = k \wedge parent(d_2) = d$ *and* $d \neq r_T$,

- $r = u \rightarrow [_j v]_j$ *and* $u \subseteq c(d)$ *and* $|V(T)| < L$.
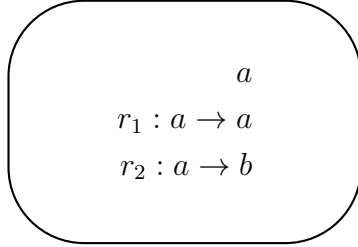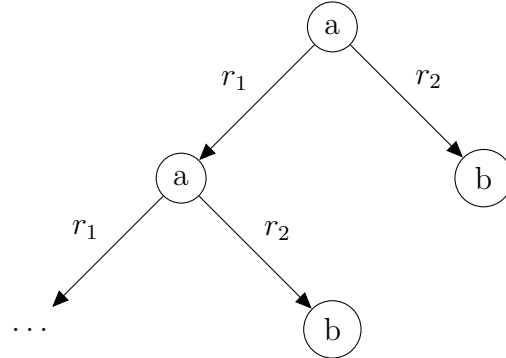
Figure 4.1: A single-membrane sample P system

Figure 4.2: The computation tree of the P system from the figure 4.1

4.2.3   Termination problems

In this subsection we recall the halting problem for Turing machines. The problem is to determine, given a deterministic Turing machine and an input, whether the Turing machine running on that input will halt. It is one of the first known undecidable problems. On the other hand, for non-deterministic machines, there are two possible meanings for halting. We could be interested either in:

- whether there exists an infinite computation (the machine can run for-ever), or

- whether there exists a finite computation (the machine can halt).

We can ask the same questions for non-deterministic P systems. For example we can look at the P system from the figure 4.1. Its computation tree (figure 4.2) contains an infinite branch, so there exists an infinite computation with rule r_1 applied in each step. There is also a finite branch, so there exists a finite computation, e.g. applying rule r_2 results in a halting configuration.

We will prove the (un)decidability of these problems on active P systems with limit on the total number of membranes. The results are quite interesting, because:

Theorem 4.2.1. *Sequential active P systems with limit on the total number of membranes are universal.*

Proof 4.2.1. The proof of this theorem for sequential active P systems in [31] uses simulation of register machines and during the simulation, every configuration has at most three membranes. Hence the active P system with limit on the total number of membranes exists (e.g. with $L = 3$), so the universality holds. $\square$
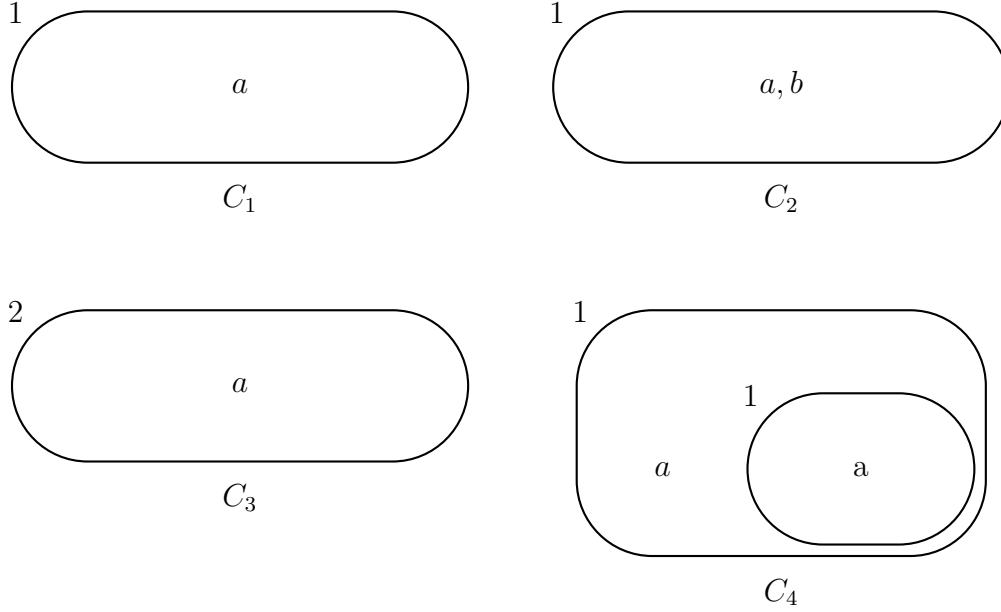
Figure 4.3: Sample membrane configurations

Existence of infinite computation

We will propose an algorithm for deciding existence of infinite computation. Basic idea is to consider the minimal coverability graph ([21]), where nodes are configurations and an edge leads from the configuration C_1 to the configuration C_2, whenever there is a rule applicable in C_1, which results in C_2. The construction in [21] is performed on Petri nets, where the configuration consists just of a vector of natural numbers. The situation is the same for single-membrane sequential P systems. We need to modify the construction for active P systems.

Definition 4.2.6. *A configuration $C_2 = (T_2, l_2, c_2)$ **covers** configuration $C_1 = (T_1, l_1, c_1)$ iff $\exists$ isomorphism $f : T_1 \to T_2$ preserving membrane labels and contents: $\forall d \in T_1$ the following properties hold: $l_1(d) = l_2(f(d)) \wedge c_1(d) \subseteq c_2(f(d))$. We will denote this with $C_1 \leq C_2$.*

Example 4.2.1. In the figure 4.3 there are four membrane configuration. $C_1 \leq C_2$, because the membrane structures consists of one membrane, so the corresponding trees are isomorphic. The label is the same and the contents of the membrane in C_1 is a subset of the contents of the membrane in C_2. It does not have to be a proper subset, i.e. $C_1 \leq C_1$. C_1 and C_3 are incomparable, because the label is different, so neither $C_1 \leq C_3$ nor $C_3 \leq C_1$ holds. C_1 and

C_4 are also incomparable, because the trees of their membrane structure are not isomorphic.

We will now follow with a proof of an important property of the covering relation - that it maintains rule applicability.

Lemma 4.2.2. *For sequential active P system with limit on the total number of membranes, if $C_2 = (T_2, l_2, c_2)$* **covers** *configuration $C_1 = (T_1, l_1, c_1)$, then there is an isomorphism $f : T_1 \rightarrow T_2$ such that if a rule r is applicable in membrane $d \in T_1$, then r is applicable in $f(d)$.*

Proof 4.2.2. Suppose r is applicable in d. Then the left side u of the rule r is contained within the contents of the membrane $u \subseteq c_1(d)$. Because $C_1 \leq C_2$, then there is an isomorphism $f : T_1 \rightarrow T_2$ such that $c_1(d) \subseteq c_2(f(d))$ and then $u \subseteq c_2(f(d))$.

There are three possible forms of the rule r.

- If $r = u \rightarrow w$, then because r is applicable in d, $\forall (a, \downarrow_k) \in w \exists d_2 \in V(T_1) : l_1(d_2) = k \land parent_{T_1}(d_2) = d$. Because $C_1 \leq C_2$, then for $f(d_2) \in V(T_2)$ the following holds: $l_2(f(d_2)) = l_1(d_2) = k$ and $parent_{T_2}(f(d_2)) = f(d)$. Hence r is applicable in $f(d)$.

- If $r = u \rightarrow w\delta$, then $d \neq r_{T_1}$. Since f is an isomorphism, then also $f(d) \neq r_{T_2}$. Other properties follows from the previous case.

- If $r = u \rightarrow [_k v]_k$, then $|V(T_1)| < L$. Isomorphism preserves number of nodes, hence $|V(T_2)| = |V(T_1)| < L$ and r is applicable in $f(d)$. $\square$

Now, we will define the encoding of a configuration $C = (T, l, c)$ into a tuple of integers.

A membrane $d \in T$ will be encoded as $(n + m)$-tuple $enc(d) \in \mathbb{N}^{(n+m)}$, where first n numbers will be actual counts of objects and next m numbers will encode the membrane label:

$$enc(d)_i = \begin{cases} c(d)(a_i) & \text{if } i \leq n \\ 0 & \text{if } n < i \leq m \land i - n \neq l(d) \\ 1 & \text{if } n < i \leq m \land i - n = l(d) \end{cases}$$

.

The entire tree will be encoded into concatenated sequences of encoded nodes in the preorder traversal order. This sequence is then padded with zeroes to have length $(n + m)L$ as that is the maximal length of encoded tree.

Since there are only finitely many non-isomorphic trees with at most L nodes ([11]), there is a constant z such that we can uniquely assign the tree an order number $o(T) \leq z$.

The entire configuration will be encoded in tuple which consists of z parts. All but the part with index $o(T)$ will contain just zeros. The part with index $o(T)$ will contain the encoding of the tree.

Example 4.2.2. Consider $L = 2$. There are just two rooted trees with at most 2 nodes. We can define $o(T) = 1$ for the single-node tree and $o(T) = 2$ for the tree with a root and one child. So the encodings of configurations from figure 4.3 contain two parts. Configurations C_1, C_2, and C_3 consists of one membrane, so their $o(T) = 1$ and the second part is filled with zeroes. Configuration C_4 contains two membranes, so its $o(T) = 2$ and the first part of its encoding will be filled with zeroes.

- $enc(C_1) = \overbrace{\underbrace{1}_{c(a)} \underbrace{0}_{c(b)} \underbrace{1}_{l=1?} \underbrace{0}_{l=2?}}^{\text{skin membrane encoded}} \underbrace{0000}_{\text{padding to fit } (n+m)L} \overbrace{00000000}^{\text{second part filled with zeroes}},$

- $end(C_2) = 1110000000000000,$

- $enc(C_3) = 1001000000000000,$

- $enc(C_4) = 00000000 \overbrace{1010}^{\text{skin membrane encoded}} \underbrace{1010}_{\text{child membrane encoded}}$

We will now show that comparing two encodings corresponds to covering of two configurations. Recall that configurations are encoded into tuples of integers, so the comparison is performed position by position.

Lemma 4.2.3. *For configurations $C_1 = (T_1, l_1, c_1)$ and $C_2 = (T_2, l_2, c_2)$, $enc(C_1) \leq enc(C_2) \Rightarrow C_1 \leq C_2$.*

Proof 4.2.3. Both $enc(C_1)$ and $enc(C_2)$ contains z parts and exactly one part contains non-zero values. Non-zero part of $enc(C_1)$ must be non-zero also in $enc(C_2)$, because $enc(C_1) \leq enc(C_2)$. Then $o(T_1) = o(T_2)$, so the trees are isomorphic. Suppose there is an isomorphism $f : T_1 \to T_2$. For every membrane $d \in T_1$, $l_1(d) = l_2(f(d))$ and $c_1(d) \subseteq c_2(f(d))$. Hence, $C_1 \leq C_2$. $\square$

Lemma 4.2.4. *For sequential active P system with limit on the total number of membranes L for every infinite sequence of configurations $\{C_i\}_{i=0}^{\infty} \exists i < j : C_i \leq C_j$.*

Proof 4.2.4. Suppose an infinite sequence $\{enc(C_i)\}_{i=0}^{\infty}$. We use a variation of Dickson's lemma ([20]): Every infinite sequence of tuples from $\mathbb{N}^k$ contains an increasing pair. Applied to our sequence, there are two positions $i < j$: $enc(C_i) \leq enc(C_j)$. From lemma 4.2.3, $C_i \leq C_j$. $\square$

Theorem 4.2.5. *Existence of infinite computation for active P systems with limit on the total number of membranes is decidable.*

Proof 4.2.5. The algorithm for deciding the problem will traverse the reachability graph. When it encounters a configuration that covers another configuration, from lemma 4.2.2 follows that the same rules can be applied repeatedly, so the algorithm will halt with the answer YES. Otherwise, the algorithm will answer NO. Algorithm will always halt, because if there was an infinite computation, from lemma 4.2.4 there would be two increasing configurations which is already covered in the YES case. $\square$

Existence of halting computation

In this subsection we will focus on the opposite problem: whether there is a computation that is halting. Recall that halting computation has no applicable rule in the last configuration. First, we will reduce this problem to the reachability. It is a problem of determining, for a given configuration C, whether there exists a computation from C_0 to C. The reachability of active P systems can be then reduced to the reachabililty of register machines, which is undecidable.

For a given P system Π and a target configuration C we will construct a P system Π' such that there is a halting computation of $\Pi' \Leftrightarrow C$ is reachable for Π. Suppose $\Pi = (\Sigma, C_0, R_1, \ldots R_m)$ and $C = (T, l, c)$. Then we will construct $\Pi' = (\Sigma', C_0', R_1', \ldots R_m')$, where:

- $\Sigma' = \Sigma \cup \{\xi_d | d \in V(T)\}$,

- $C_0' = (T, l, c')$, where $\forall d \in V(T) \setminus r_T : c'(d) = c(d)$ and $c'(r_T) = c(r_T) \cup \{\xi_{r_T}\}$,

- $\forall i \in \{1 \ldots m\} : R_i' = R_i \cup \{\xi_d c(d) \rightarrow \xi_{d'} \downarrow_{l(d')} | d, d' \in V(T), l(d) = i, parent(d') = d\}$.

The ξ_d objects are called verifiers, they are intended to verify if the contents of the membrane corresponds to the contents in the target configuration C. After this verification it descends down into child membranes for the verification of other parts of the membrane structure. Initially, there is an object ξ_{r_T} in the skin membrane. Verification is performed in the rule
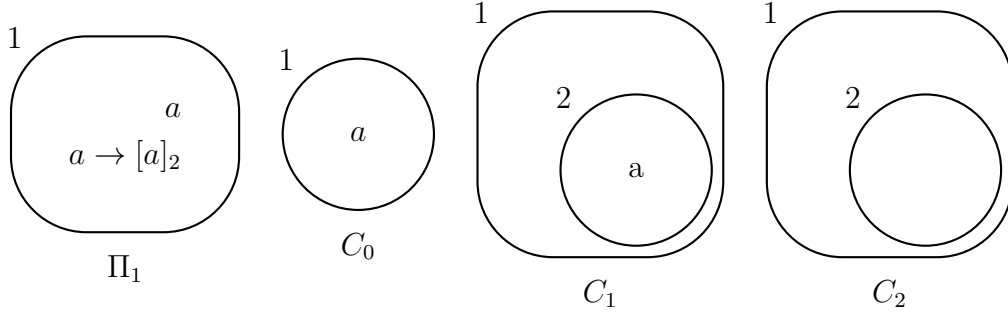
Figure 4.4: Example P system Π_1
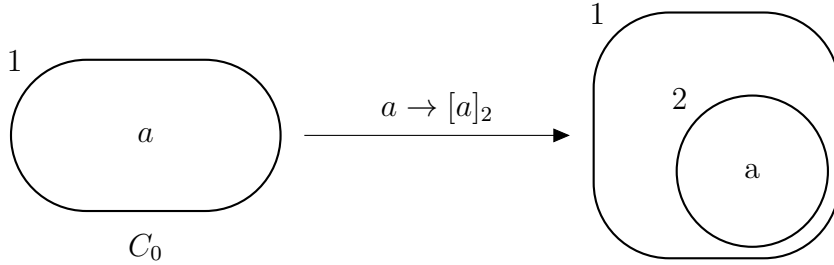
Figure 4.5: Computation of Π_1

$\xi_d c(d) \to \xi_{d'} \downarrow_{l(d')}$, where on the right side there is $\xi_{d'}$ object for every child membrane d' in the target configuration C.

The construction is not complete. The system should not be able to halt unless the verification takes place. That is why we introduce a new object ω to each membrane with a rule $\omega \to \omega$ and the verifier will erase them with rule $\xi_d \omega c(d) \to \xi_{d'} \downarrow_{l(d')}$. We also need to ensure that newly created membranes contain the ω object, so we replace every rule for membrane creation $u \to [_k v]_k$ with $u \to [_k v \omega]_k$.

Example 4.2.3. An example P system $\Pi_1 = (\{a, b\}, C_0, \{a \to [a]_2\}, \{\})$ is depicted in the Figure 4.4. The target configuration C_1 with the membrane structure consisting of the root membrane d_1 with label 1 and its child membrane d_2 with label 2 can be easily reached by applying the rule $a \to [a]_2$ in the skin membrane. This computation is shown in the Figure 4.5.

We will construct a corresponding P system $\Pi_1' = (\{a, b, \xi_{d_1}, \xi_{d_2}, \omega\}, C_0', \{a \to [a]_2, \xi_{d_1} \omega \to \xi_{d_2}, \omega \to \omega\}, \{\xi_{d_2} a \omega \to \varepsilon, \omega \to \omega\})$. The initial configuration C_0' is shown in the Figure 4.6. There is also a halting computation which corresponds to the computation of Π_1 that reaches the target configuration C_1.

Example 4.2.4. There is still a problem that we actually check, whether
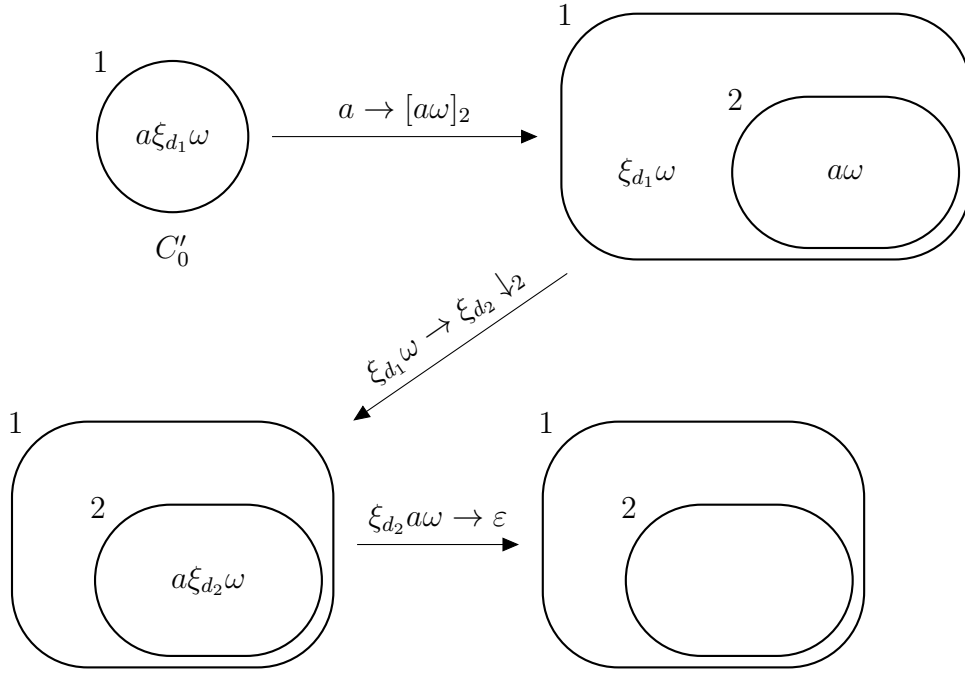
Figure 4.6: Computation of Π'_1
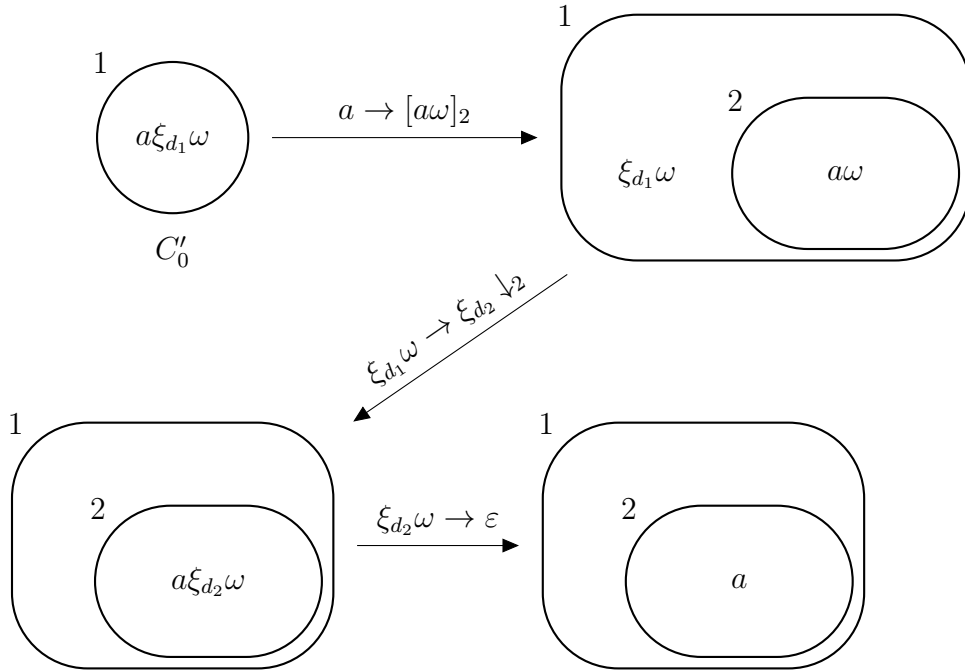
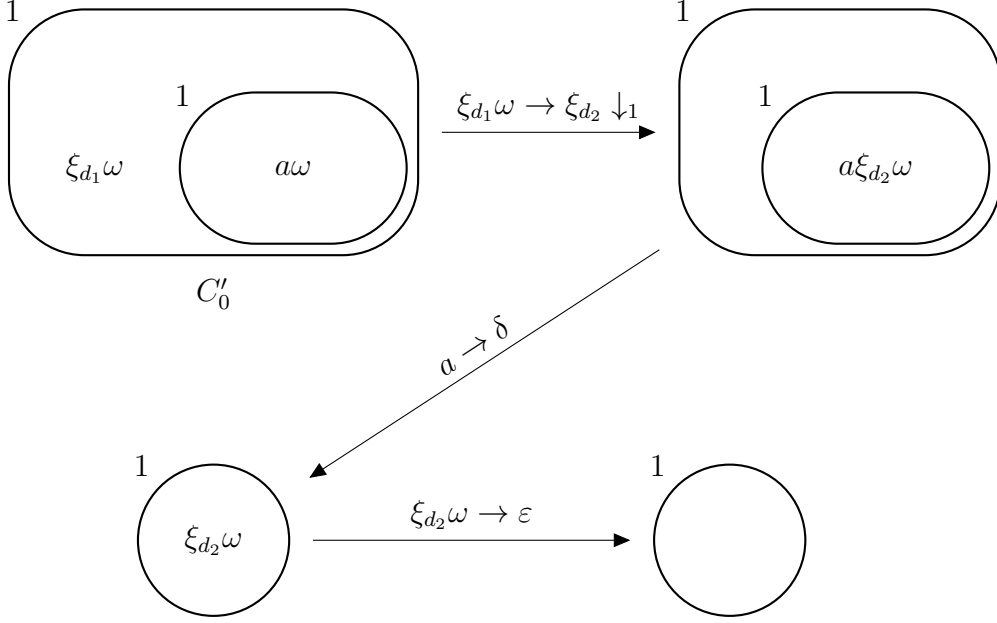

Figure 4.7: Computation of Π'_2
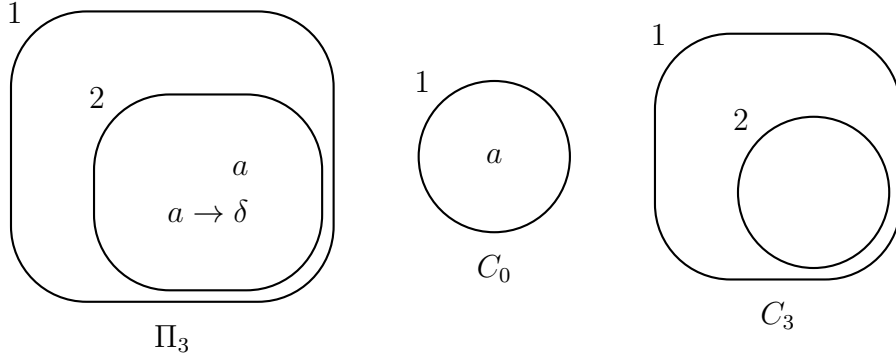
Figure 4.8: Example P system Π_3
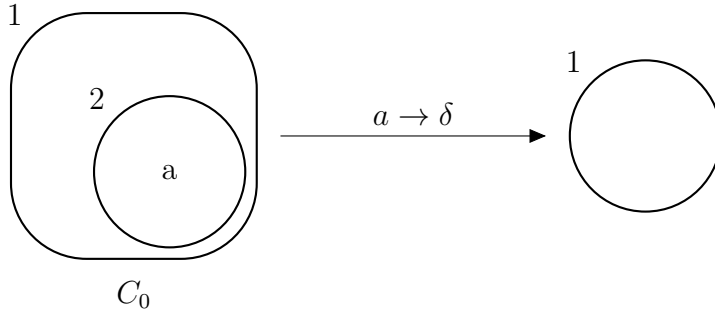
Figure 4.9: Computation of Π_3

a target configuration is contained within the current configuration. If our target configuration is C_2 from the Figure 4.4, the resulting P system $\Pi_2' = (\{a, b, \xi_{d_1}, \xi_{d_2}, \omega\}, C_0', \{a \rightarrow [a]_2, \xi_{d_1}\omega \rightarrow \xi_{d_2}, \omega \rightarrow \omega\}, \{\xi_{d_2}\omega \rightarrow \varepsilon, \omega \rightarrow \omega\})$ will have the same halting computation shown in the Figure 4.7 although the configuration C_2 cannot be reached.

If there are additional objects which cannot be erased, so the target configuration cannot be reached, we need to ensure Π' will not halt. We will add a rule $a \rightarrow a$ to each membrane for each object $a \in \Sigma$, so Π' can halt only if all objects are erased. This modification renders the computation in the Figure 4.7 as not halting.

The last issue to solve is that it is still possible that in the middle of the verifying, some of already verified membranes got dissolved and all yet unverified membranes will be successfully verified causing Π' to halt, although without that dissolution it would be unable to reach C.

Example 4.2.5. In the figure 4.8 there is a P system $\Pi_3 = (\{a\}, C_0, \{a \rightarrow \delta\})$. The target configuration C_3 cannot be reached, because erasing a will

Figure 4.10: Computation of Π_3'

also dissolve the child membrane. We cannot keep the membrane while getting rid of a. The only possible computation is depicted in the Figure 4.9.

The resulting P system will be $\Pi_3' = (\{a, \xi_{d_1}, \xi_{d_2}, \omega\}, C_0', \{a \to \delta, a \to a, \xi_{d_1}\omega \to \xi_{d_2}, \xi_{d_2}\omega \to \varepsilon, \omega \to \omega\})$. Its initial configuration C_0' and the halting computation are shown in the Figure 4.10.

We need to ensure all dissolution happen before the verification takes place. We will add a new object σ, which stands as a footprint object. It will be created as a result of the verification rule $\xi_d\omega c(d) \to \sigma\xi_{d'} \downarrow_{l(d')}$. If a membrane is dissolved after it was verified, then two σs will meet in the same membrane, because the parent membrane also contains σ as it had been verified before. We will add a rule $\sigma\sigma \to \sigma\sigma$ to prevent Π' from halting.

The final construct is:

- $\Sigma' = \Sigma \cup \{\omega, \sigma\} \cup \{\xi_d | d \in V(T)\}$,

- $C_0' = (T, l, c')$, where $\forall d \in V(T) \setminus r_T : c'(d) = c(d) \cup \{\omega\}$ and $c'(r_T) = c(r_T) \cup \{\omega, \xi_{r_T}\}$,

- $\forall i \in \{1 \ldots m\} : R_i' = \{r | r \in R_i, r = u \to w \vee r = u \to w\delta\} \cup \{u \to [_k v\omega]_k | u \to [_k v]_k \in R_i\} \cup \{a \to a | a \in \Sigma\} \cup \{\sigma\sigma \to \sigma\sigma, \omega \to \omega\} \cup \{\xi_d\omega c(d) \to \sigma\xi_{d'} \downarrow_{l(d')} | d, d' \in V(T), l(d) = i, parent(d') = d\}$.

We need to prove two implications in order to formally prove correctness of this construction.

Lemma 4.2.6. *If C is reachable for Π then there is a halting computation of Π'.*

Proof 4.2.6. Consider computation Π with C as the last configuration. For Π' there is a corresponding computation as the rules of Π are included in Π' with an exception of the rule $u \to [_k v]_k$, which has a corresponding rule with the same left side: $u \to [_k v\omega]_k$. The corresponding computation of Π' will result in a configuration, where in every membrane d, the contents will be $\omega c(d)$, and the skin membrane will contain an additional ξ_{r_T}. Then the cascade of applications of rule $\xi_d \omega c(d) \to \sigma \xi_{d'} \downarrow_{l(d')}$ can be applied, starting in the skin membrane and cascading down to children until all the membranes applied that rule. The objects $\omega c(d)$ will be replaced by σ and the computation will halt. $\square$

The other direction of the implication is more complicated, so following lemmas will state some properties about computations of Π and Π'.

Lemma 4.2.7. *For all halting computations of Π' there is no rule of form $u \to w\delta$ applied in the membrane d' after the application of rule $\xi_d \omega c(d) \to \sigma \xi_{d'} \downarrow_{l(d')}$ in membrane $d = parent(d')$.*

Proof 4.2.7. After the application of the rule $\xi_d \omega c(d) \to \sigma \xi_{d'} \downarrow_{l(d')}$, the object σ remains in the membrane. There is no possible interaction of σ with other objects, only with another σ. If a child membrane d' is dissolved with a rule $u \to w\delta$, then two objects σ will meet in the membrane d and the computation will not halt, which contradicts the fact that the computation is halting. $\square$

Lemma 4.2.8. *For all halting computations of Π' there is no rule of form $u \to [_k v\omega]_k$ applied in the membrane d' after the application of rule $\xi_d \omega c(d) \to \sigma \xi_{d'} \downarrow_{l(d')}$.*

Proof 4.2.8. After the application of rule $r = \xi_d \omega c(d) \to \sigma \xi_{d'} \downarrow_{l(d')}$ in membrane d, there will not be any of ξ objects present in the membrane, because they are only sent into child membranes, which cannot be dissolved because of lemma 4.2.7. So there will be no more of rule r applied in membrane d. The newly created membrane will never receive any of ξ objects, so the object ω will never be erased and the computation will not halt, which contradicts the fact that the computation is halting. $\square$

Lemma 4.2.9. *If a halting computation of Π' exists then there is a halting computation, where for every membrane $d \in V(T)$ in the target configuration $C = (T, l, c)$ the last rule used is $r = \xi_d \omega c(d) \to \sigma \xi_{d'} \downarrow_{l(d')}$.*

Proof 4.2.9. Consider membrane d: let C_1 be the configuration before the application of r, C_2 the configuration after the application of r and C_3 the halting configuration.

First consider elementary membranes, lets have a membrane d. As it has no child membranes, C_2 is contained within C_1 (with the exception of the σ object). Therefore the sequence of steps from C_2 to C_3 can be applied starting from C_1 instead of C_2. It would result in a configuration C_4, where contents of d is exactly $\xi_d \omega c(d)$, assuming d has not been dissolved, what is ensured by lemma 4.2.7. So the rule $r = \xi_d \omega c(d) \to \sigma$ can be applied, resulting in configuration, where d contains exactly one σ and nothing else, so no more rule will be applied in d.

Consider a membrane d, which is not elementary, so it has child membranes. In C_1 the verifier is in d and no child membrane contains the verifier. In C_2 every child membrane contains the verifier. We cannot use the same argument, because C_2 is not contained within C_1.

We will proceed by induction, starting from the elementary membranes to parent membranes. Assume the lemma holds for child membranes. Then we have a computation, which contains a configuration C_5, after which only verification rules are applied in child membranes. So the sequence of steps from C_2 to C_5 does not contain the verification rule in child membranes. Hence this sequence can be used starting from C_1 instead of C_2, resulting in a configuration C_6, where application of verification rule in d results in C_5. So again, after C_6 only verification rules are applied. □

Lemma 4.2.10. *If there is a halting computation of Π' then C is reachable for Π.*

Proof 4.2.10. According to lemma 4.2.9 there is also a halting computation where in every membrane d the last used rule is $r = \xi_d \omega c(d) \to \sigma \xi_{d'} \downarrow_{l(d')}$. So the corresponding computation in Π will result in the configuration C_5, where every membrane d contains exactly $c(d)$. If C_5 contains a membrane not present in C, it will contain the object ω, which will not be reached by any of objects ξ so the computation will not halt. If C contains a membrane d' not present in C_5, then the membrane $d = parent(d')$ will never get rid of ω because the rule $r = \xi_d \omega c(d) \to \sigma \xi_{d'} \downarrow_{l(d')}$ cannot be applied due to lack of the child membrane d'. Hence $C = C_5$ and there is a computation of Π that will result in C. □

Theorem 4.2.11. *Existence of halting computation for active P systems with limit on the total number of membranes is undecidable.*

Proof 4.2.11. For a given P system Π and a target configuration C we have constructed a P system Π' such that there is a halting computation of Π' iff the Π can reach configuration C. The two directions of the equivalence have been proven in lemmas 4.2.6 and 4.2.10. Using this construction, we can reduce the existence of halting computation to reachability of register machines [31], which is known to be undecidable. □

4.2.4 Concluding remarks

We have studied the termination problems for active sequential P systems. Unlike deterministic systems, the termination problems cannot be simply reduced to the halting problem. We have shown that active P systems with limit on the number of membranes have decidable existence of infinite computation and undecidable existence of halting computation. It is currently unknown whether the same results apply also for a variant without the limit on the number of membranes, so it could be a subject for the future study.

Regarding the open problem stated in [31] about sequential active P systems with hard membranes (without communication between membranes), it could be interesting to find a connection between the universality and decidability of these termination problems.

4.3 Membrane capacity

4.4 Vacuum

(from project proposal):

We propose a new variant of P system with vacuum in accordance with the criteria formulated in [7] (see beginning of the section 3.2). In the common sense, vacuum represents a state of 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. After the interaction, there is vacuum no longer. 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 computation power of a sequential P system in comparison to the variant without using the vacuum. This case have been shown to be universal.

In the future we will research other restriction for this variant such as non-cooperative rules, decaying objects, deterministic steps and will use techniques to show even non-universality results.

Theorem 4.4.1. *The sequential P system with Vacuum is universal.*

Proof 4.4.1. We can simulate the variant of P system where the only cooperative rule is of type $a|a \rightarrow b$. According to [29] the variant, where the only cooperative rule is when both objects are the same, is universal. If there is no rule $a \rightarrow b$, 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 4.1.3, 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:

- $e \rightarrow e \downarrow_j$ for an instruction of type $e : add(j), f$ or $e : sub(j), f, z$ and

- $h \rightarrow h \downarrow$ for a halting instruction h.

And in non-skin membranes:

- $e \rightarrow a|f \uparrow$ instructions of type $e : add(j), f$,

- $e|a \rightarrow f \uparrow$ for instructions of type $e : sub(j), f, z$,

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

- $h|a \rightarrow h|a$

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.

4.5   Emptyness detection

4.6   Reaction systems

Chapter 5

Generating membrane structures

CLS, CLS+, CLS++, CLS-

Chapter 6

Maximal parallel P systems with context-free rules

Notes about how inhibitors help, how determinism may help, how allowing only rules $a|a \rightarrow \_$ may help, how inhibiting other rules may help.

Conclusions

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, . . .).

Bibliography

[1] LM Adleman (1994): *Molecular computation of solutions to combinatorial problems.* Science 266(5187), pp. 1021–1024, doi:10.1126/science.7973651. Available at http://www.sciencemag.org/content/266/5187/1021.abstract.

[2] Oana Agrigoroaiei & Gabriel Ciobanu (2010): *Flattening the transition P systems with dissolution.* In: *Proceedings of the 11th international conference on Membrane computing*, CMC'10, Springer-Verlag, Berlin, Heidelberg, pp. 53–64. Available at http://dl.acm.org/citation.cfm?id=1946067.1946077.

[3] FAC Azevedo, LRB Carvalho, LT Grinberg, JM Farfel, REL Ferreti, REP Leite, WJ Filho, R Lent & S Herculano-Houzel (2009): *Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain.* Journal of Comparative Neurology 513(5), pp. 532–541, doi:10.1002/cne.21974.

[4] Henry Givens Baker (1973): *Rabin's proof of the undecidability of the reachability set inclusion problem of vector addition systems.* Massachusetts Institute of Technology, Project MAC.

[5] 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.

[6] Roberto Barbuti, Andrea Maggiolo–Schettini, Paolo Milazzo & Angelo Troina (2007): *The Calculus of Looping Sequences for Modeling Biological Membranes.* In George Eleftherakis, Petros Kefalas, Gheorghe Păun, Grzegorz Rozenberg & Arto Salomaa, editors: *Membrane Computing, Lecture Notes in Computer Science* 4860,

Springer Berlin Heidelberg, pp. 54–76,
doi:10.1007/978-3-540-77312-2_4. Available at
`http://dx.doi.org/10.1007/978-3-540-77312-2_4`.

[7] Daniela Besozzi (2004): *Computational and modelling power of P systems.* Ph.D. thesis, Universita' degli Studi di Milano, Milano, Italy.

[8] Henning Bordihn, Henning Fernau & Markus Holzer (1999): *Accepting Pure Grammars and Systems.* In: *Otto-von-Guericke-Universitat Magdeburg, Fakultat fur Informatik, Preprint Nr.*

[9] Catalin Buiu, Cristian Vasile & Octavian Arsene (2012): *Development of membrane controllers for mobile robots. Information Sciences* 187(0), pp. 33 – 51, doi:10.1016/j.ins.2011.10.007. Available at `http://www.sciencedirect.com/science/article/pii/S0020025511005421`.

[10] Mónica Cardona, M. Angels Colomer, Mario J. Pérez-Jiménez, Delfí Sanuy & Antoni Margalida (2009): *Membrane Computing.* chapter Modeling Ecosystems Using P Systems: The Bearded Vulture, a Case Study, Springer-Verlag, Berlin, Heidelberg, pp. 137–156, doi:10.1007/978-3-540-95885-7_11. Available at `http://dx.doi.org/10.1007/978-3-540-95885-7_11`.

[11] Professor Cayley (1881): *On the Analytical Forms Called Trees. American Journal of Mathematics* 4(1), pp. pp. 266–268. Available at `http://www.jstor.org/stable/2369158`.

[12] José M. Cecilia, José M. García, Ginés D. Guerrero, Miguel A. Martínez del Amor, Ignacio Pérez-Hurtado & Mario J. Pérez-Jiménez (2010): *Simulating a P system based efficient solution to SAT by using GPUs. Journal of Logic and Algebraic Programming* 79, pp. 317–325, doi:dx.doi.org/10.1016/j.jlap.2010.03.008. Available at `http://dx.doi.org/10.1016/j.jlap.2010.03.008`. Membrane computing and programming.

[13] George M. Church, Yuan Gao & Sriram Kosuri (2012): *Next-Generation Digital Information Storage in DNA. Science* 337(6102), p. 1628, doi:10.1126/science.1226355. Available at `http://www.sciencemag.org/content/337/6102/1628.abstract`.

[14] 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`.

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

[16] Martin Davis (1973): *Hilbert's tenth problem is unsolvable.* American *Mathematical Monthly*, pp. 233–269.

[17] Rocco De Nicola & Frits Vaandrager (1995): *Three Logics for Branching Bisimulation. J. ACM* 42(2), pp. 458–487, doi:10.1145/201019.201032. Available at `http://doi.acm.org/10.1145/201019.201032`.

[18] Reinhard Diestel (1997): *Graph Theory. Graduate Texts in Mathematics* 173, Springer.

[19] Catherine Dufourd, Alain Finkel & Ph. Schnoebelen (1998): *Reset Nets Between Decidability and Undecidability.* In: *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, ICALP '98, Springer-Verlag, London, UK, UK, pp. 103–115. Available at `http://dl.acm.org/citation.cfm?id=646252.686157`.

[20] Diego Figueira, Santiago Figueira, Sylvain Schmitz & Philippe Schnoebelen (2011): *Ackermannian and Primitive-Recursive Bounds with Dickson's Lemma.* In: *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*, LICS '11, IEEE Computer Society, Washington, DC, USA, pp. 269–278, doi:10.1109/LICS.2011.39. Available at `http://dx.doi.org/10.1109/LICS.2011.39`.

[21] Alain Finkel (1993): *The minimal coverability graph for Petri nets.* In Grzegorz Rozenberg, editor: *Advances in Petri Nets 1993, Lecture Notes in Computer Science* 674, Springer Berlin Heidelberg, pp. 210–243, doi:10.1007/3-540-56689-9_45. Available at `http://dx.doi.org/10.1007/3-540-56689-9_45`.

[22] Rudolf Freund (2005): *Asynchronous P systems and P systems working in the sequential mode.* In: *Proceedings of the 5th international conference on Membrane Computing*, WMC'04, Springer-Verlag, Berlin, Heidelberg, pp. 36–62, doi:10.1007/978-3-540-31837-8_3. Available at `http://dx.doi.org/10.1007/978-3-540-31837-8_3`.

[23] Rudolf Freund, Lila Kari, Marion Oswald & Petr Sosík (2005): *Computationally universal P systems without priorities: two catalysts*

are sufficient. Theoretical Computer Science 330(2), pp. 251 – 266, doi:10.1016/j.tcs.2004.06.029. Available at `http://www.sciencedirect.com/science/article/pii/S0304397504006553`. Descriptional Complexity of Formal Systems.

[24] 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`.

[25] Renana Gershoni, Ehud Keinan, Gheorghe Păun, Ron Piran, Tamar Ratner & Sivan Shoshani (2008): *Research topics arising from the (planned) P systems implementation experiment in Technion.* In Daniel Díaz-Pernil, Carmen Graciani, Miguel Angel Gutiérrez-Naranjo, Gheorghe Păun, Ignacio Pérez-Hurtado & Agustín Riscos-Nú nez, editors: *Sixth Brainstorming Week on Membrane Computing*, pp. 183–192.

[26] Michel Hack (1974): *The Recursive Equivalence of the Reachability Problem and the Liveness Problem for Petri Nets and Vector Addition Systems.* In: *Proceedings of the 15th Annual Symposium on Switching and Automata Theory (Swat 1974)*, SWAT '74, IEEE Computer Society, Washington, DC, USA, pp. 156–164, doi:10.1109/SWAT.1974.28. Available at `http://dx.doi.org/10.1109/SWAT.1974.28`.

[27] Michel Hack (1974): *The Recursive Equivalence of the Reachability Problem and the Liveness Problem for Petri Nets and Vector Addition Systems.* In: *Proceedings of the 15th Annual Symposium on Switching and Automata Theory (Swat 1974)*, SWAT '74, IEEE Computer Society, Washington, DC, USA, pp. 156–164, doi:10.1109/SWAT.1974.28. Available at `http://dx.doi.org/10.1109/SWAT.1974.28`.

[28] Michel Hack (1976): *The equality problem for vector addition systems is undecidable. Theoretical Computer Science* 2(1), pp. 77 – 95, doi:http://dx.doi.org/10.1016/0304-3975(76)90008-6. Available at `http://www.sciencedirect.com/science/article/pii/0304397576900086`.

[29] 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.

[30] OscarH. Ibarra, Zhe Dang, Omer Egecioglu & Gaurav Saxena (2003): *Characterizations of Catalytic Membrane Computing Systems*. In Branislav Rovan & Peter Vojtáš, editors: *Mathematical Foundations of Computer Science 2003*, *Lecture Notes in Computer Science* 2747, Springer Berlin Heidelberg, pp. 480–489, doi:10.1007/978-3-540-45138-9_42. Available at http://dx.doi.org/10.1007/978-3-540-45138-9_42.

[31] OscarH. Ibarra, Sara Woodworth, Hsu-Chun Yen & Zhe Dang (2005): *On Sequential and 1-Deterministic P Systems*. In Lusheng Wang, editor: *Proceedings of the 11th annual international conference on Computing and Combinatorics*, *COCOON'05* 3595, Springer-Verlag, Berlin, Heidelberg, pp. 905–914, doi:10.1007/11533719_91. Available at http://dx.doi.org/10.1007/11533719_91.

[32] 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.

[33] Ryuichi Ito (1969): *Every semilinear set is a finite union of disjoint linear sets*. *Journal of Computer and System Sciences* 3(2), pp. 221 – 231, doi:http://dx.doi.org/10.1016/S0022-0000(69)80014-0. Available at http://www.sciencedirect.com/science/article/pii/S0022000069800140.

[34] Lila Kari, Greg Gloor & Sheng Yu (2000): *Using DNA to solve the Bounded Post Correspondence Problem*. *Theoretical Computer Science* 231(2), pp. 193 – 203, doi:10.1016/S0304-3975(99)00100-0. Available at http://www.sciencedirect.com/science/article/pii/S0304397599001000.

[35] Richard M. Karp & Raymond E. Miller (1969): *Parallel Program Schemata*. *J. Comput. Syst. Sci.* 3(2), pp. 147–195, doi:10.1016/S0022-0000(69)80011-5. Available at http://dx.doi.org/10.1016/S0022-0000(69)80011-5.

[36] Dexter C. Kozen (1997): *Automata and Computability*, 1st edition. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[37] Aristid Lindenmayer (1968): *Mathematical Models for Cellular Interactions in Development, II. Simple and Branching Filaments with Two-sided Inputs. J. Theoretical Biology*, pp. 280–315.

[38] Richard J. Lipton (1976): *The reachability problem requires exponential space.* Research report (Yale University. Department of Computer Science), Department of Computer Science, Yale University. Available at `http://books.google.sk/books?id=7iSbGwAACAAJ`.

[39] Ernst W. Mayr (1981): *An Algorithm for the General Petri Net Reachability Problem.* In: *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, ACM, New York, NY, USA, pp. 238–246, doi:10.1145/800076.802477. Available at `http://doi.acm.org/10.1145/800076.802477`.

[40] Paolo Milazzo (2007): *Qualitative and Quantitative Formal Modeling of Biological Systems.* Ph.D. thesis, Universita di Pisa.

[41] David J. Montana & Lawrence Davis (1989): *Training feedforward neural networks using genetic algorithms.* In: *Proceedings of the 11th international joint conference on Artificial intelligence - Volume 1*, IJCAI'89, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 762–767. Available at `http://dl.acm.org/citation.cfm?id=1623755.1623876`.

[42] Rohit J. Parikh (1966): *On Context-Free Languages. J. ACM* 13(4), pp. 570–581, doi:10.1145/321356.321364. Available at `http://doi.acm.org/10.1145/321356.321364`.

[43] Andrei Paun & Gheorghe Paun (2002): *The power of communication: P systems with symport/antiport. New Gen. Comput.* 20(3), pp. 295–305, doi:10.1007/BF03037362. Available at `http://dx.doi.org/10.1007/BF03037362`.

[44] Gheorghe Păun (1998): *Computing with Membranes.* Technical Report 208, Turku Center for Computer Science-TUCS. (www.tucs.fi).

[45] Mario J. Pérez-Jiménez (2006): *P systems-based modelling of cellular signalling pathways. Workshop on Membrane Computing, WMC7*, pp. 54–73.

[46] Carl Adam Petri (1962): *Kommunikation mit Automaten.* Ph.D. thesis, Universität Hamburg.

[47] Gheorghe Păun (1999): *P Systems with Active Membranes: Attacking NP Complete Problems.* Journal of Automata, Languages and Combinatorics 6, pp. 75–90.

[48] Gheorghe Păun, Yasuhiro Suzuki & Hiroshi Tanaka (2001): *P systems with energy accounting.* International Journal of Computer Mathematics 78(3), pp. 343–364, doi:10.1080/00207160108805116. Available at `http://www.tandfonline.com/doi/abs/10.1080/00207160108805116`.

[49] Charles Rackoff (1978): *The covering and boundedness problems for vector addition systems.* Theoretical Computer Science 6(2), pp. 223 – 231, doi:http://dx.doi.org/10.1016/0304-3975(78)90036-1. Available at `http://www.sciencedirect.com/science/article/pii/0304397578900361`.

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

[51] 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`.

[52] Petr Sosík & Rudolf Freund (2003): *P Systems without Priorities Are Computationally Universal.* In: *Revised Papers from the International Workshop on Membrane Computing*, WMC-CdeA '02, Springer-Verlag, London, UK, UK, pp. 400–409. Available at `http://dl.acm.org/citation.cfm?id=647270.721876`.

[53] Hsu-Chun Yen (2006): *Introduction to Petri Net Theory.* In Zoltán Esik, Carlos Martín-Vide & Victor Mitrana, editors: *Recent Advances in Formal Languages and Applications*, *Studies in Computational Intelligence* 25, Springer Berlin Heidelberg, pp. 343–373, doi:10.1007/978-3-540-33461-3_14. Available at `http://dx.doi.org/10.1007/978-3-540-33461-3_14`.

[54] Ge-Xiang Zhang, Chun-Xiu Liu & Hai-Na Rong (2010): *Analyzing radar emitter signals with membrane algorithms.* Mathematical and Computer Modelling 52(11–12), pp. 1997 – 2010, doi:10.1016/j.mcm.2010.06.002. Available at `http://www.sciencedirect.com/science/article/pii/S0895717710002736`.

The BIC-TA 2009 Special Issue, International Conference on
Bio-Inspired Computing: Theory and Applications.