Parallel approach of algorithms

Herendi, Tamás
Nagy, Benedek

Parallel approach of algorithms

írta Herendi, Tamás és Nagy, Benedek

Kivonat

Summary: Nowadays the parallelization of various computations becomes more and more important. In this book the theoretical models of parallel computing are presented. Problems that can be solved and problems that cannot be solved in these models are described. The parallel extensions of the traditional computing models, formal languages and automata are presented, as well as properties and semi-automatic verifications and generations of parallel programs. The concepts of theory of parallel algorithms and their complexity measures are introduced by the help of the (parallel) Super Turing-machine. The parallel extensions of context-free grammars, such as L-systems, CD and PC grammar systems; multihead automata (including various Watson-Crick automata), traces and trace languages are described. The theory and applications of Petri nets are presented by their basic concepts, special and extended models. From the practical, programming point of view, parallelization of (sequential) programs is investigated based on discovering of dependencies.

Reviewer: Iván Szabolcs

ISBN 978 963 279 338 2

Prepared under the editorship of Typotex Kiadó

Responsible manager: Zsuzsa Votisky

Made within the framework of the project Nr. TÁMOP-4.1.2/A/1-11/1-2011-0063, entitled „Sokprocesszoros rendszerek a mérnöki gyakorlatban”.

Tartalom

Az ábrák listája

A táblázatok listája

Introduction

The classical computing paradigms (Turing machine, Chomsky-type generative grammars, Markov normal algorithms, etc.) work entirely in a sequential manner as the Neumann-type computers do, too. Among the theoretical models parallel computing models have appeared in the middle of the last century, but their importance were mostly theoretical since in practice only very few special architecture machines worked in a parallel way (and their parallelism was also very limited, in some cases the sequential simulation was faster than the parallel execution of the algorithm). In the past 10-15 years, however, Moore's law about the increasing computational powers still works notwithstanding that the producing techniques are very close to their physical limits. This is done by a new way of developments. Instead of the earlier trends, increasing the speed of the CPU, the new trend uses parallel computing. Nowadays almost everywhere only parallel architectures are used and we can meet almost only them in the market. Desktop, portable, laptop, netbook computers have several processors and/or cores, they are equipped with special (parallel) GPU's; moreover smart phones and tablets also use parallel processors. We can use FPGA's, supercomputers, GRID's and cloud computing technologies. Therefore it is necessary for a programmer, a software engineer or a electronic engineer to know the basic parallel paradigms. Besides the (traditional) Silicon-based computers, several new computing paradigms have emerged in the past decades including the quantum computing (that is based on some phenomena of quantum mechanics), the DNA computing (the theoretical models using DNA's and also the experimental computations), the membrane computing (P-systems) or the interval-valued computing paradigm. These new paradigms can be found in various (text)books (we can refer to the book *Benedek Nagy: Új számítási paradigmák (Bevezetés az intervallum-értékű, a DNS-, a membrán- és a kvantumszámítógépek elméletébe); New computing paradigms (Introduction to the interval-valued, DNA, membrane and quantum computing), Typotex, 2013, in Hungarian);* in the present book we consider mostly the traditional models.

There are several ways to define and describe traditional computing methods, algorithms and their models. Traditonal forms to give algorithms are, e.g, automata, rewriting systems (such as grammars)…

The structure of the book is as follows. In the first part we give the basic methods to analyse algorithms and basic concepts of complexity theory. Then we analyse the parallel extensions of the traditional formal grammars and automata: starting with the simplest Indian parallel grammars and D0L-systems, through on the Russen parallel grammars and the scattered context grammars to ET0L and IL systems. The cooperative distributed (CD) and parallel communicating (PC) grammar systems are also presented. Part III is about parallel automata models. We will show the 2-head finite automata (WK-automata), both the variants in which both heads go to the same direction and the variants in which the 2 heads are going to opposite direction starting from the two extreme of the input. At m-head finite automata we distinguish the one-way models, where every head can go only from the beginning of the input to its end, and the two-way variants where each head can move in both directions. We also present the simplest P-automata and give an overview on cellular automata. In part IV traces and trace languages are shown, we use them to describe parallel events. Then, in part V the Petri-nets are detailed: we start with elementary (binary) nets and continue with place-transition nets. Finally some more complex nets are recalled, shortly. Part VI is about parallel programming: we show some entirely parallel algorithms and also analyse how sequential algorithms can be changed to their parallel form. Finally, instead of a summary, the two basic forms of the parallelism are shown.

In this way the reader may see several parallel models of algorithms. However the list of them is not complete, there are some other parallel models that we cannot detail due to lack of space, some of them are the neural networks, the genetic algorithms, the networks of language processors, the automata networks, the (higher-dimensional) array grammars, etc. Some of them are closely related to the models described in this book, some of them can be found in distinct textbooks.

This book is written for advanced BSc (undergraduate) students and for MSc (graduate) students and therefore it assumes some initial knowledge. However some chapters can also be useful for PhD students to extend their knowledge. There is a wide list of literature given in the book and thus the reader has a good chance to get a deep theoretical and practical knowledge about the parallel models of the algorithms.

Parts I and VI are written by Tamás Herendi, while parts II, III, IV, V and VII are written by Benedek Nagy. The authors are grateful to Szabolcs Iván for his help by reading and reviewing the book in details. They also thank the TÁMOP project, the Synalorg Kft and the publisher (Typotex Kiadó) their help and support.

Debrecen, 2013.

The authors

1. fejezet - Methods for analysis of algorithms

1.1. Mathematical background

1.1.1.1.1. The growth order of functions

In the following chapters, according to complexity theory considerations, we will have to express the order of growth of functions (asymptotic behavior) in a uniform way. We will use a formula to express the most important, the most steeply growing component of the function.

Let $\mathbb{N}$ and $\mathbb{R}$ denote the set of natural and real numbers, respectively, and let $\mathbb{R}^+$ denote the set of positive real numbers.

Definition: Let f: $\mathbb{N} \to \mathbb{R}^+$ be a function.

The set $O(f) = \{g \dashv \mid \exists\, c > 0, N > 0, g(n) < c \cdot f(n), \forall\, n > N\}$

is called the **class of functions which belongs to the order of growth of function** *f*.

If g $\in$ $O(f)$, then we say that "*g* **is a big O** f" function.

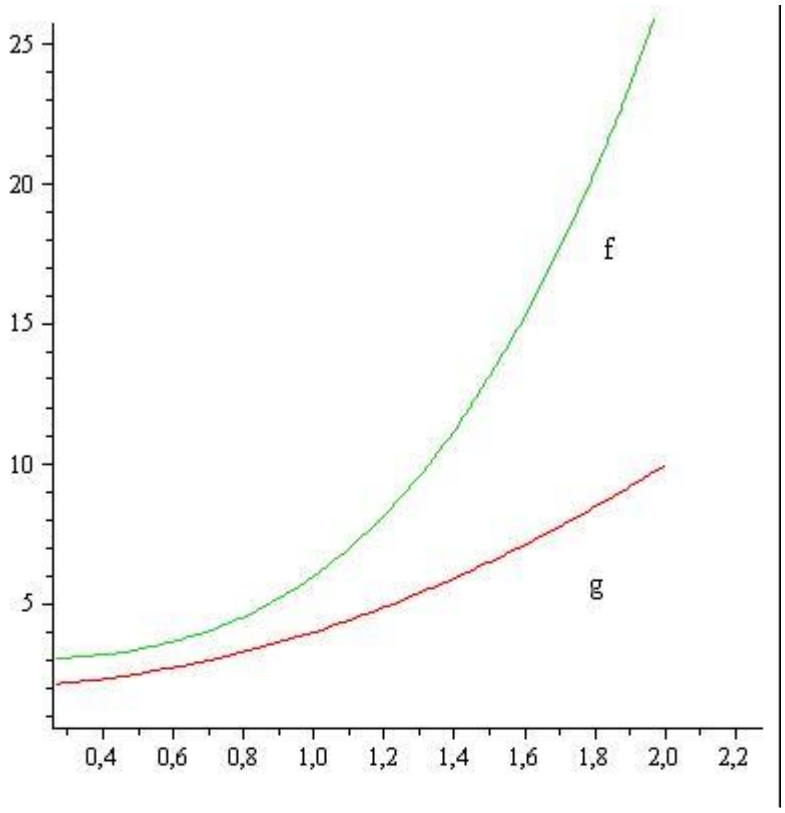
Remark: In order to make the "order of growth" explicit, we may suppose that f is monotone increasing, but it is not necessary. This assumption would make a lot of observations unnecessarily complicated, thus it would become harder to understand.

Based on the definition, the most evident example is that a function bounds from above the other. However, it is not necessary. In the following we will observe through several example graphs what the growth order expresses, at least for at first sight.

Example (the order of growth; g(n) < f(n)**)**

Supposing that $g(n) < f(n)$, $\forall\, n \in \mathbb{N}$, the conditions of the definition are satisfied with the choice of $c = 1$, $N = 1$.

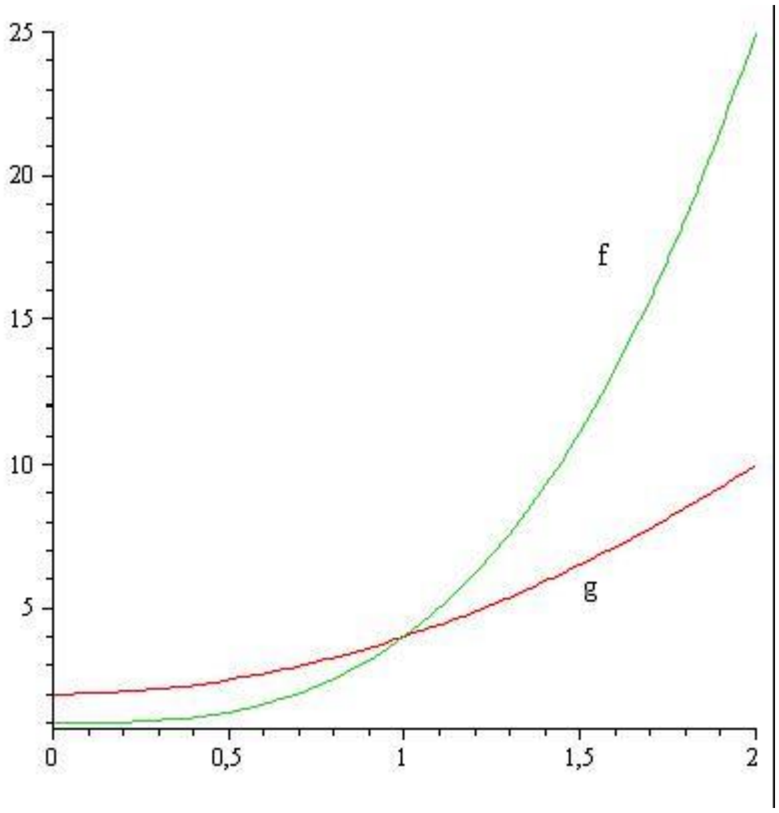
1.1. ábra - The orders of growth *g*(*n*) < *f*(*n*).



Example (the order of growth; ; g(n) < f(n), if n > N**)**

Supposing that *g*(*n*) < *f*(*n*), ∀ *n* > *N*, for small *n*'s we do not care about the relation of the functions.

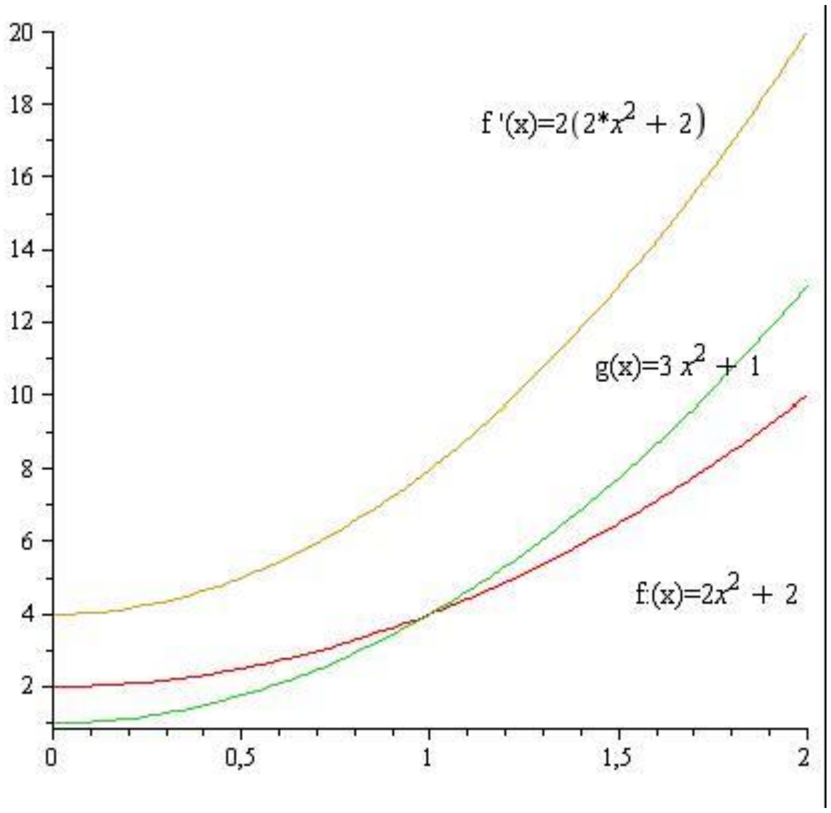
1.2. ábra - The orders of growth *g*(*n*) < *f*(*n*), ha *n* > *N*.



Example (the order of growth; g(n) ∈ O(f(n)) are general**)**

In the example $g(n) > f(n)$, $\forall\ n \in \mathbb{N}$, but if we multiply f by a constant, it is not smaller than g anywhere. g(n) ∈ O(f(n)). for the general case.

1.3. ábra - The orders of growth; $g(n) \in O(f(n))$ for the general case..



Remark: Instead of $g \in O(f)$ we often use the traditional $g = O(f)$ notation.

Properties

1. $f(n) \in O(f(n))$ (reflexivity)

2. Let $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$. Thus $f(n) \in O(h(n))$

(transitivity)

Proof:

1. Let $c = 2$ and $N = 0$. Since $f(n) > 0$ for all $n \in \mathbb{N}$, thus $f(n) < 2f(n)$ for all $n > 0$. √

2. Let c_1 and N_1 be such that $f(n) < c_1 \cdot g(n)$, *if* $n > N_1$ and let c_2 and N_2 be such that $g(n) < c2 \cdot h(n)$, if $n > N_2$. If we set $c = c_1 \cdot c_2$ and $N = \max\{N_1, N_2\}$, then we get $f(n) < c_1 \cdot g(n) < c_1(c_2 \cdot h(n)) = c \cdot h(n)$, if $n > N_1$ and $n > N_2$, which means $n > N$. √

Remark:

1. The notion of transitivity expresses our expectation that if the function f is increasing faster than *g*, then it is increasing faster than any function which is slower than *g*.

2. The order of growth is neither a symmetric nor an antisymmetric relation.

3. For the symmetry ($g \in O(f) \Rightarrow f \in O(g)$) the functions $f = n$ and $g = n^2$ and for the antisymmetry ($g \in O(f)$, $f \in O(g) \Rightarrow f = g$) the functions $f = n$ és $g = 2n$ are counterexamples.

Further properties:

5. Let $g_1 \in O(f_1)$ and $g_2 \in O(f_2)$. Then $g_1 + g_2 \in O(f_1 + f_2)$.

6. Let $g_1 \in O(f_1)$ and $g_2 \in O(f_2)$. Then $g_1 \cdot g_2 \in O(f_1 \cdot f_2)$.

7. Let f be monotone increasing and $f(n) > 1 \ \forall \ n \in \mathbb{N}$ and $\in O(f)$. Then $\log(g) \in O(\log(f))$.

Proof:

5. Assume that c_1, c_2 and N_1, N_2 are such that

$\forall \ n > N_1 : g_1(n) < c_1 \cdot f_1(n)$

and

$\forall \ n > N_2 : g_2(n) < c_2 \cdot f_2(n)$.

Let $c = \max\{c_1, c_2\}$ and $N = \max\{N_1, N_2\}$. Then

$\forall \ n > N: g_1(n) + g_2(n) < c_1 \cdot f_1(n) + c_2 \cdot f_2(n) < c \cdot (f_1(n) + f_2(n))$. √

6. As before, assume that c_1, c_2 and N_1, N_2 are such that

$\forall \ n > N_1 : g_1(n) < c_1 \cdot f_1(n)$

and

$\forall \ N_2 : g_2(n) < c_2 \cdot f_2(n)$.

Let $c = c_1 \cdot c_2$ and $N = \max\{N_1, N_2\}$. Then

$\forall \ n > N: g_1(n) \cdot g_2(n) < c_1 \cdot f_1(n) \cdot c_2 \cdot f_2(n) < c \cdot f_1(n) \cdot f_2(n)$. √

7. Assume that c and N are such that $\forall \ n > N: g(n) < c \cdot f(n)$. Without loss of generality we may assume that $c > 1$. Since the function log(.) is strictly monotone increasing,

$\forall \ n > N: \log(g(n)) < \log(g(n) < \log(c \cdot f(n)) = \log(c) + \log(f(n))$.

Since c, $f(1) > 1$, thus $\log(c)$, $\log(f(1)) > 0$. Let

$$c' = 1 + \frac{\log(c)}{\log(f(1))}$$

Because of the monotonity of f, we have $f(1) < f(n) \ \forall \ n > 1$ and

$\log(c) + \log(f(n)) =$

$$\left(\frac{\log(c)}{\log(f(n))} + 1\right) \cdot \log(f(n)) <$$

$$\left(\frac{\log(c)}{\log(f(1))} + 1\right) \cdot \log(f(n)) =$$

$c' \cdot \log(f(n))$

$\forall\ n \in \mathbb{N}$. √

Corollary:

1. Let $k_1, k_2 \in \mathbb{R}^+$! Then

$n^{k_1} \in O\left(n^{k_2}\right)$

if and only if $k_1 \leq k_2$.

2. Let $k1$, $k_2 \in \mathbb{R}^+$! Then $k_1^n \in O(k_2^n)$ if and only if $k_1 \leq k_2$.

3. Let $k \in \mathbb{R}^+$! Then $n^k \in O(2^n)$ and $2^n \notin O(n^k)$.

4. Let $k \in \mathbb{R}^+$! Then $\log(n) \in O(n^k)$.

Proof:

The Corollaries 1, 2, 3 and 4 can be simply derived from Properties 5, 6 and 7. √

Remark:

Some particular cases of Corollary 1:

1. $n \in O(n^2)$

and more generally

2. $n^k \in O(n^{k+1})$ for all $k > 0$

To express the order of growth more precisely, we may use other definitions as well. Some of these will be given in the next few pages, together with the observation some of their basic properties.

Definition: Let f: $\mathbb{N} \to \mathbb{R}^+$ be a function. The set $\Theta(f) = \{g | \exists\ c_1, c_2 > 0, N > 0, c_1 \cdot f(n) < g(n) < c_2 \cdot f(n)$ if $n > N\}$ is called the class of functions belonging to the exact growth order of f.

Properties:

Let g, f: $\mathbb{N} \to \mathbb{R}^+$ be two functions. Then

1. $g \in \Theta(f)$ if and only if $g \in O(f)$ and $f \in O(g)$ and

2. g $\in \Theta(f)$ if and only if $f \in \Theta(g)$.

Proof:

1. By the definition of the relations $g \in O(f)$ and $f \in O(g)$ we know that $\exists\ c_1 > 0$ és $N_1 > 0$, such that $g(n) < c_1 \cdot f(n)$ if $n > N_1$ and $f(n) < c_2 \cdot\ g(n)$ ifn $> N_2$.

Let = $\max\{N_1, N_2\}$,

$$c_1^{'} = \frac{1}{c_2}$$

and $c'_2 = c_1$.

By the inequalities above we get

$c_1' \cdot f(n) < g(n) < c_2' \cdot f(n)$, if $n > N$. √

2. By 1, $g \in \Theta(f)$ if and only if $g \in O(f)$ and $f \in O(g)$. Swapping the roles of f and g, we get $f \in \Theta(g)$ if and only if $g \in O(f)$ and $f \in O(g)$. The two equivalences imply the statement of the theorem. √

Definition: Let $f: \mathbb{N} \to \mathbb{R}^+$ be a function. The set $o(f) = \{g | \forall\ c > 0, \exists\ N > 0, g(n) < c \cdot f(n)$, ha $n > N\}$ is called the class of funtions having strictly smaller growth order than f.

If $g \in o(f)$, we say that "g is a small o f" function.

Properties:

Let $g, f: \mathbb{N} \to \mathbb{R}^+$ be two functions. Then

1. $g(n) \in o(f(n))$ if and only if

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

2. $O(f(n)) = \Theta(f(n)) \cup o(f)$ and

3. $\Theta(f(n)) \cap o(f(n)) = \emptyset$.

Proof:

1. Let c > 0 and $N_c > 0$ be such that $g(n) < c \cdot f(n)$, if $n < N_c$.

Then

$$0 < \frac{g(n)}{f(n)} < c,$$

if $n > N_c$, thus for all small $c > 0$ the N_c bound can be given to satisfy

$$\frac{g(n)}{f(n)} < c.$$

for all $n > N_c$. By the definition of limit this means exactly that

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0.$$

Conversely,

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

yields that $\forall\ c > 0\ \exists\ N > 0$, such that

$$0 < \frac{g(n)}{f(n)} < c,$$

if $n > N$. Multiplying this by $f(n)$ we get the statement √

2. By the definitions it is clear that $o(f(n)) \subseteq O(f(n))$ and $\Theta(f(n)) \subseteq O(f(n))$.

Let $(n) \in O(f(n)) \setminus \Theta(f(n))$.

Since $g(n) \in O(f(n))$, thus by definition $\exists\ c_2 > 0$ and $N_2 > 0$, such that

$g(n) < c_2 < c_2 \cdot f(n)$ if $n > N_2$.

Since $g(n) \notin \Theta(f(n))$, by the previous inequality and the definition $\forall\ c_1 > 0\ \exists\ N_{c1} > 0$ such that $c_1 \cdot f(n) \geq g(n)$ if $n > N_{c1}$, which is the statement to prove.√

3. In an indirect way, assume that $\Theta(f(n)) \cap o(f(n)) \neq \emptyset$. Then $\exists\ g(n) \in \Theta(f(n)) \cap o(f(n))$. For this g by definition $\exists\ c_1 > 0$ and $N_1 > 0$, such that

$c_1 \cdot f(n) < g(n)$ if $n > N_1$ and $\forall\ c > 0,\ \exists\ N_c > 0$ such that $g(n) < c \cdot f(n)$, if $n > N$.

Let $N' = \max\{N_1, N_{c1}\}$.

For this N'

$c_1 \cdot f(n) < g(n)$ and $g(n) < c_1 \cdot f(n)$, if $n > N'$, which is a contradiction. √

In the follwing we need to describe the input of a problem and thus we define the following concepts:

We start by the alphabet and by the formal languages (over the alphabet).

Definition: An arbitrary nonempty, finite set of symbols is called an alphabet, and denoted by T; the elements of T are the letters (of the alphabet). The set of words obtained from the letters of the alphabet including the empty word λ is denoted by T^*. L is a formal language over the alphabet T if $L \subseteq T^*$. The length of a word in the form $w = a_1 \dots a_n$ (where $a_1, \dots, a_n \in T$) is the number of letters the word contains (with multiplicity), formally: $|w| = n$. Further, $|\lambda| = 0$.

Further we will denote the aphabet(s) by ∑, T and N in this book.

1.1.1.1.2. Complexity concepts

When we are looking for the algorithmic solution of a particular problem, in general it is not fully defined. It involves some free arguments and thus we deal with a whole class of problems instead. If, for example, the task is to add two numbers, then there are so many free parameters that we cannot find a unique solution. (For such a general task we may have for example to compute the sum $\sqrt{2} + \sqrt{}$ or $\pi + e$.)

To make more strict observations, we have to tell more precisely what the problem is we want to solve. In the simplest way (and that could be the most common way), we represent the problems to be solved by words (sentences), thus the class of problems can be interpreted as a language containing the given sentences.

The solver algorithm during the solution may use different resources and the quantity of these resources, of course, depends on the input data. For the unique approach it is usual to distinguish one of the properties of the input - perhaps the most expressive -, the quantity or the size. If we know this information, we may define the quantity of the necessary resources as a function of it. After all, we may use further refinements and observe the need for these resources in the worst case, at average or with some other particular assumption. For more simplicity (and to make the calculation simpler), usually we do not describe the exact dependency, only the order of growth of the dependency function. (Of course the phrase "exception proves the rule" stands here as well: sometimes we can define quite exact connections.)

Definition: Resource complexity of an algorithm:

Let Σ be a finite alphabet, $\mathcal{A}$ an algorithm and e a given resource. Assume that $\mathcal{A}$ executes some well defined sequence of steps on an arbitrary $w \in \Sigma^*$ input that needs $E(w)$ unit of the resource type e.

Then the complexity of the $\mathcal{A}$ algorithm that belongs to resource e:

$f_e(n) = \max\{E(w) | w \in \Sigma^*, |w| \leq n\}$.

In the following we will look over the complexity concepts that belong to the most important resources. The exact definitions will be given later.

1. Time complexity

This is one - and the most important - property of the algorithms. With the help of time complexity we can express how a given implementation changes the running time with the size of the input. In general it is the number of operations we need during the execution of an algorithm.

Remark: We have to be careful with the notion of time complexity as the different definitions can give different values.

One of the most precise models for the description of algorithms is the Turing machine. The time complexity can be expressed precisely by it, but the description of the algorithms are rather complicated.

The RAM machine model is more clear and is closer to programming, but gives the opportunity to represent arbitrary big numbers in the memory cells and we can reach an arbitrary far point of the memory in the same time as the closer ones.

In practice the most common algorithm description method is the pseudo code. The problem is that in this case we consider every command having a unit cost and this can change the real value of the time complexity.

2. Space complexity

It expresses the data storage needs of the algorithm while solving the problem. We express that with a basic unit as well.

Remark: The Turing machine model gives the most exact definition in this case, too. However, the RAM machine still has the problem that in one cell we can represent an arbitrary big number. If not every memory cell contains valuable data, then the space needed could be either the number of the used cells or the maximal address of an accessed cell. (An upper bound on the space complexity of a Turing machine can be obtained from the largest address multiplied by the length of the largest number used by the corresponding RAM machine.) In other models with network features, the real space complexity can be hidden because of the communication delay.

3. Program complexity

It is noteworthy, but in the case of sequential algorithms it does not have that big importance. It expresses the size of the solving algorithm, by some elementary unit.

Remark: Contrary to these two former properties, the program complexity is independent from the size of the problem. In practice the size of program, due to physical bounds, is not independent from the input size. This is because of the finiteness of computers. If the size of the input data is exceeding the size of the memory, then the program should be extended by proper storage management modules.

We can represent the nonsequential algorithms in several models as well. In each model new complexity measurements can be defined as a consequence of parallelity. Just as before, we will give the number of necessary resources in terms of an elementary unit.

1. Total time complexity

Basically this is the time complexity of sequential algorithms, we express it with the number of all the executed operation and data movement.

2. Absolute time complexity

It gives the "time" that has passed from the beginning to the end of the execution of the algorithm, expressed by the number of commands. The operations executed simultaneously are not counted as different operations.

3. Thread time complexity

If it can be interpreted in the given algorithm model, it gives the different threads' time complexity. This is typically used in models describing systems with multiple processors.

4. Total space complexity

Similarly to the total time complexity, it is the complete data storage resource need of an algorithm.

5. Thread space complexity

It is the equivalent of thread time complexity, if independent computation threads and independent space usage can be defined.

6. Processor complexity (~number of threads)

If in the model we can define some computing unit, processor complexity expresses the quantity of it.

7. Communication complexity

If the computation described by the model can be distributed among different computing units, each of them may need the results given by the other units. The measure of this data is the communication complexity.

A property, related to the complexity concepts is the parallel efficiency of an algorithm. We can describe by the efficiency how the algorithm speeds up the computation by parallelizing.

Definition: Let $\mathcal{A}$ be an algorithm, $T(n)$ be the total time complexity and let $t(n)$ be the absolute time complexity of it. The value

$$\frac{T(n)}{t(n)} \leq P(n),$$

is the algorithm's parallel efficiency.

Between the parallel efficiency and the processor complexity the following relation holds.

Theorem: Let $\mathcal{A}$ be an algorithm, $H(n)$ be the parallel efficiency of the algorithm, $P(n)$ be the processor complexity. Then $(n) \leq P(n)$.

Proof: Because of the absence of the exact definitions we will only see the principle of the proof.

Let $T(n)$ be the total time complexity and let $t(n)$ be the absolute time complexity of the algorithm.

If the algorithm works with $P(n)$ processor for $t(n)$ time, the total time complexity cannot be more than $P(n) \cdot t(n)$.

Transforming these we get

$T(n) \leq P(n) \cdot t(n)$

$$F(n) = \frac{P(n)}{H(n)}$$

which was to be proved. √

Definition: Let $\mathcal{A}$ be an algorithm, $P(n)$ be its processor complexity and let $H(n)$ be its parallel efficiency. The value

$$H(n) = \frac{T(n)}{t(n)}$$

is the algorithm's processor efficiency.

Remark: The processor efficiency expresses how much of the operating time of the processors uses the algorithm during the operation. The value can be at most 1.

Limited complexity concepts:

As extensions of theoretical complexities, we can define practical or limited complexities. Then we keep one or more properties of the algorithm between bounds – typical example is the limitation of the number of processors or the used memory – and we investigate only on the key parameters. This is proportional to the observation of complexity of programs used in everyday life, as the implemented algorithms only work on a finite computer.

1.2. Parallel computing models

Between the different complexity concepts there is some relation. How can we express all these in a theoretic way?

The well known Space Time theorem gives the relation between space and time complexity in the case of deterministic single tape Turing machine.

Previously we set up a relation between the total time complexity, the absolute time complexity and the processor complexity of the algorithms.

In this chapter, with a special Turing machine model, we will concretize the complexity concepts defined in chapter 2 and we also define the relation between them.

The Turing machine is one of the oldest models of computing. This model can be used very efficiently during theoretical observations and for determining complexity values in case of certain sequential algorithms. What about parallel algorithms? The Turing machine model, because of its nature, is perfect for describing parallel algorithms, but from a practical point of view (programming) cannot necessarily solve the problems or analyze some particular complexity issues.

Those who have some experience in informatics, have some idea what an algorithm is. In most cases, we can decide whether the observed object is an algorithm or not. At least in "everyday life". It is almost clear that a refrigerator is not an algorithm. A lot of people associate to a computer program when it hears about algorithms and this association is not completely irrational. However it is not obvious that a medical examination or even walking are not algorithms.

The more we want to refine the concept, the more we arrive to difficulties. Anyway, we have the following expectations of algorithms:

An algorithm has to solve a well determined problem or sort of problem.

An algorithm consists of finitely many steps, the number of these steps is finite. We would like to get a solution in a finite time.

Each step of the algorithm has to be defined exactly.

The algorithm may need some incoming data that means a special part of the problem to be solved. The amount of the data can only be finite.

The algorithm has to answer the input. The answer has to be well interpretable and finite of course.

Apparently, the more we want to define the notion of an algorithm, the more uncertainty we have to leave in it. The most problematic part is the notion of "step". What does it mean "exactly defined"? If we give an exact definition, we restrict the possibilities.

People have tried to find an exact definition to the algorithm in the first half of the last century as well. As a result several different notions were born, such as lambda functions, recursive functions, the Markov machine and the Turing machine. As they are well defined models, we can have the impression that they are not appropriate for replacing the notion of algorithm.

One may assume that the above models are equivalent, which means that every problem that can be solved in one of the models, can be solved in the others as well.

To replace the notion of algorithm, until now, no one has found any models that are more complete then the previous ones. In this section we will pay attention to the Turing machine model as it is the clearest, easily interpretable and very expressive. However it is not very useful while modeling the recently popular computer architectures.

With Turing machines we can easily express the computability, the notion of solving the problem with an algorithm and we can determine a precise measure of the complexity of algorithms or of the description of the difficulty of each problem.

The name Turing machine is referring to the inventor of the model, A. Turing (1912-1954). The actual definition has plenty of variations with the same or similar meaning. We will use the clearest one which is one of the most expressive.

For even trying to describe the algorithm with mathematical tools, we have to restrict the repertoire of the problems to be solved. We have to exclude for example the mechanical operations done on physical objects of the possibilities. It does not mean that this kind of problems cannot be solved in an algorithmic way, it is just that instead of the physical algorithm we can only handle the mathematical model, and turn them into physical operations with the necessary interface. In practice, basically it always works like that, since we cannot observe the executed algorithms and the results of the corresponding programs themselves.

Thus we can assume that the problem, its parameters and incoming data can be described with some finite representation. Regarding this, in the following we will give the input of an algorithm (i.e. the description of the problem) as a word on a given finite alphabet and we expect the answer in the same way as well. So we can consider an algorithm as a mapping that transforms a word to another. It is clear that we can specify algorithms that do not have any reactions to certain inputs, which means that they do not specify an output (the so called partial functions). We can imagine some particular algorithms that despite the infinite number of the incoming words they can only give a finite number of possible answers. For example there is the classical accepting (recognizing) algorithm, which answers with yes or no depending on the value of the input (accepts or rejects the input).

In the following we will see that this accepting problem may seem easy, but it is not. There exist such kinds of problems that we cannot even decide whether they can be solved or not.

Definition. (Turing machine)

The object represented by $T = (Q, \Sigma, s, \delta, H)$ is called a Turing machine, where

Q: is a finite, not empty set; set of states

Σ: is a finite, not empty set ($* \in \Sigma$); tape alphabet

$s \in Q$; initial state

$\delta: Q \setminus H \times \Sigma \to Q \times (\Sigma \cup \{\leftarrow, \rightarrow\})$; state transition function

$H \subseteq Q$, not empty; set of final states

This formal definition above has to be completed with the corresponding interpretation. The following "physical" model will be in our minds:

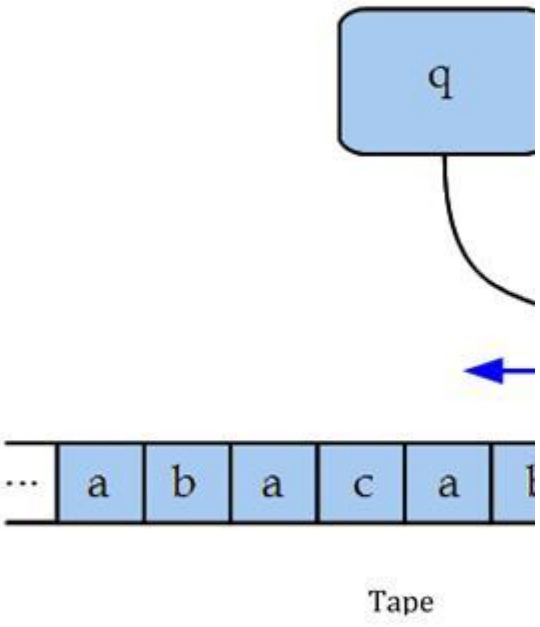
In the imaginary model the Turing machine has 3 main components:

1. State register: It consists of one element of Q, we can determine the behavior of the Turing machine with it. The stored value is one of the arguments of the state transition function.

2. Tape: It is infinite in both directions (in certain interpretations it can be expanded infinitely), it is a sequence of discrete cells (storing elements of the tape alphabet) of the same size. Each cell contains an element of the alphabet Σ. It does not influence the interpretation of the Turing machine that consider the tape infinite in one or both directions (or infinitely expandable) as in all cases we assume that on the tape we only store a single word of finite length. On the rest of the tape the cells contain the dummy symbol: $*$. (In certain interpretations this $*$ symbol is the letter denoting the empty word.)

3. Read write head: It sets up the relation between the tape and the status register. On the tape it always points to a particular cell. The Turing machine can read this cell and write in it. During the operation the read write head can move on the tape.

1.4. ábra - Turing machine.



The operation of the Turing machine: The Turing machine, during its operation, executes discrete steps. Let's assume that it implements every step in a unit time (and at the same time).

a.) At the start, the status register contains the initial state *s*, while on the tape the input word *w* can be found. The read write head points to the cell preceding the first letter.

b.) During a step, it reads the sign under the R/W head, with the transition function δ and the read sign it defines the new state and the new sign to be written on the tape or movement value of the R/W head. (Or it writes on the tape or it moves its head.) It writes the new internal state in the status register, the tape sign in the cell of the tape which is under the R/W head. Then it owes the R/W head to the next cell.

c.) If the content of the status register is a final state, then it stops and the output is the word written on the tape. (There is a definition of the Turing machine which says it stops as well when the transition function is not defined on the given state and read letter, but this case can be eliminated easily, if we assume that the value of state transition function is a final state every time we do not tell it exactly.)

According to the interpretation kept in mind we need an "operation" that is exact and mathematically well determined. The following definitions will describe it.

Definition:

Let T = (Q, Σ, s, δ, H) be a Turing machine. Any element $K \in Q \times ((\Sigma^* \times \Sigma \times \Sigma^*) \backslash (\Sigma^+ \times \{*\} \times \Sigma^+))$ is a configuration of T .

According to the definition a configuration is a quadruple: $K = (q, w_1, a, w_2)$, where$q$ is a state, $w_1$is the space on the tape before the R/W head, a is the letter under the R/W head and w_2 is the space on the tape after the R/W head. Although, by the definition of transition function it would be possible, but since we have excluded the elements of $Q \times \Sigma^+ \times \{*\} \times \Sigma^+$, thus the configurations with empty cells under the R/W head are not valid if both before and after nonempty cells can be found.

Remark: If it does not cause any problem in the interpretation, for more simplicity in the configuration we won't tell the exact place of the R/W head. Then its notation will be the following:

$K = (q, w) \in Q \times \Sigma^*$,

where q is the state of the Turing machine and w is the content of the tape.

Definition: Let T = (Q, Σ, s, δ, H) be a Turing machine and $K = (q, ,w_1, a, w_2)$ be a configuration of T.

We say that T goes in one step (or directly) from K to configuration K' (with signs $K \vdash K'$), if $K' = (q', w_1', a', w_2')$ and exactly one of the following is true:

1) $w_1' = w_1$

$w'_2 = w_2$

$\delta(q, a) = (q', a')$, where $a, a' \in \Sigma$, $q\ \Sigma\ Q \backslash H$ and $q'\ \Sigma$ Q.

(overwrite mode)

2) $w_1' = w_1 \cdot a$

$w_2 = a' \cdot w_2'$

$\delta(q, a) = (q', \rightarrow)$, where $a \in \Sigma$, $q\ \Sigma\ Q \backslash H$ and $q' \in Q$.

(right shift mode)

3) $w_1 = w_1' \cdot a'$

$w_2' = a \cdot w_2$

$\delta(q, a) = (q', \leftarrow)$, where $a \in \Sigma$, $q \in Q \setminus H$ and $q' \in Q$.

(left shift mode)

Definition: Let $T = (Q, \Sigma, s, \delta, H)$ be a Turing machine and $C = K_0, K_1, \ldots, K_n, \ldots$, be a sequence of configurations of T.

We say that C is a computation of the Turing machine T on the input word w, if

1. $K_0 = (s, \lambda, a, w')$, where $w = aw'$;

2. $K_i \vdash K_{i+1}$, if there exists a configuration that can be reached from K_i directly (and because of uniqueness, it is K_{i+1});

3. $K_i = K_{i+1}$, if there is no configuration that could be reached directly from K_i .

If $h \in H$ for some $K_n = (h, w_1, a, w_2)$, then we say that the computation is finite, T stops in the final state h. Then we call the word $w' = w_1 \cdot a \cdot w_2$ the output of T. Notation: $w' = T(w)$.

If in case of a $w \in \Sigma^*$ the Turing machine T has an infinite computation, then we do not interpret any outputs. Notation: $(w) = \emptyset$. (Must not confuse with the output $(w) = \lambda$.)

Definition (Recursive function):

We call the function $f: \Sigma^* \rightarrow \Sigma^*$ recursive, if $\exists$ T Turing machine such that $\forall$ $w \in \Sigma^*$ $f(w) = T(w)$.

Definition: The object represented by the quintuple $T = (Q,, \Sigma, s, \Delta, H)$ is called a Super Turing machine, if

Q: finite, nonempty set; (set of states)

Σ finite, nonempty set, $* \in \Sigma$; (tape alphabet)

$s \in Q$; (initial state)

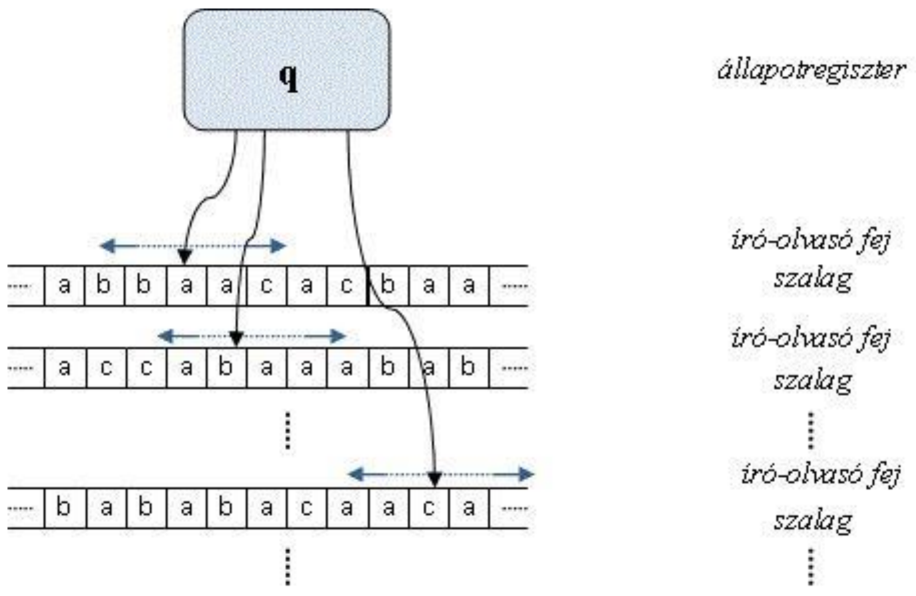
Δ: $Q \setminus H \times \Sigma^* \rightarrow Q \times (\Sigma \cup \{\leftarrow, \rightarrow\})^*$; (transition function)

recursive, i.e. is computable by a simple Turing machine

$H \subseteq Q$, nonempty set; (set of final states)

The Super Turing machine has an infinite number of tapes and among these, there are a finite number of tapes containing data, moreover each such tape contains a finite amount of data only. All of them has their own a R/W head which move independently from each other.

1.5. ábra - Super Turing machine



We define the notion of configuration and computation analogously to the case of Turing machines.

Definition: Let $T = (Q, \Sigma, s, \Delta, H)$ be a Super Turing machine. K is a configuration of T such that

$K = Q \times ((\Sigma^* \times \Sigma \times \Sigma^*) \setminus (\Sigma^+ \times \{* \} \times \Sigma^+))^*$.

Remark: If it does not cause any interpretation problems, for more simplicity in the configuration we don't mark the exact place of the R/W head. In this case the notation will be the following:

$K = (q, w_1, \ldots, w_k) \in Q \times (\Sigma^*)^*$, where $1 \leq k$. Here q is the state of the Turing machine and $w_1, \ldots, w_k$ are the contents of the first k tapes.

Definition: Let $T = (Q, \Sigma, s, \Delta, H)$ be a Super Turing machine and $K = (q, (w_{1,1}, a_1, w_{2,1}), \ldots, (w_{1,k}, a_k, w_{1,k}))$ be a configuration of T. (Only the first kk tapes contain any information.)

We say that T goes in one step (or directly) from K to K' (notation $K \vdash K'$), if

$K' = (q', (w'_{1,1}, a'_1, w'_{2,1}), \ldots, (w'_{1,l}, a'_l, w'_{2,1}))$,

$q \in Q \setminus H$ and $q' \in Q$,

$A = a_1 \ldots a_k$,

$\Delta(q, A) = (q', B)$, where $B = b_1 \ldots b_l$ and

exactly one of the following is true, for all$\forall$ $1 \leq i \leq \max \{k, l\}$:

1) $w'_{1,i} = w_{1,i}$

$w_{2,i} = w_{2,i}$

$b_i = a'_i$.

2) $w'_{1,i} = w_{1,i} \cdot a_i$

$w_{2,i} = a'_i \cdot w_{2,i}$

$b_i = \rightarrow$

3) $w_{1,i} = w'_{1,i} \cdot a'_i$

$w'_{2,i} = a_i \cdot w_{2,i}$

$b_i = \leftarrow$.

Definition: Let $T = (q, \Sigma, s, \Delta, H)$ be a Super Turing machine and

$C = K_0, K_1, \ldots, K_n, \ldots$ be a sequence of configurations of T.

We say that C is a computation of the Super Turing machine T on the input word w, if

1. $K_0 = (s, \lambda, a, w')$, where $w = aw'$ (in case $w = \lambda$ we have $a = *$ and $w' = \lambda$);

2. $K_i \vdash K_{i+1}$, if there exists a configuration directly reachable from K_i (and because of uniqueness K_{i+1});

3. $K_i = K_{i+1}$, if there is no configuration directly reachable from K_i .

If $K_n = (h, w_1, w_2, \ldots, w_k)$ with $h \in H$, then we say that the computation is finite and T stops in the final state h . By default, w_1 is the output of T. Notation: $w_1 = T(w)$. In other cases, the content of the output tape(s) is the result of the computation.

If the Super Turing machine T has an infinite computation on $w \in \Sigma^*$, we do not interpret the output. Notation: $T(w) = \emptyset$. (Must not be confused with the output $T(w) = \lambda$.)

Example (positioning)

$T = (Q, \Sigma, s, \Delta, H)$

$Q = \{s, h\}$

$\Sigma = \{0,1, * \}$

$H = \{h\}$

In case of = $a_1 \ldots a_k : \Delta(s, A) = (s, B)$,

where $B = b_1 \ldots b_k$ and $b_i = *$, if $a_i = *$, and $b_i = \Rightarrow$, if $a_i \neq *$.

$\Delta(s, A) = (h, A)$, if $A = \lambda$, namely on each tapes there is only $*$ under the R/W head;

The Turing machine in the above example does only one thing: on each tape it goes to the first empty cell following a nonempty word.

Example (selection of the greatest element)

$T = (Q, \Sigma, s, \Delta, H)$

$Q = \{s, a, h\}$

$\Sigma = \{0, 1, x_0, x_1, x_*, x^* \}$

$H = \{h\}$

Input: $w_1, \ldots, w_k \in \{0, 1\}^*$.

State transition function:

In case of $A = a_1 \ldots a_k$ $\Delta(s, A) = (a, Ax)$.

$\Delta(a, Ax) = (h, A)$, ha $A = \lambda$, e.g. on all the tapes there is $*$ under the R/W head;

$\Delta(a, Ax) = (b, Bx)$, where ahol $B = b_1 \dots b_k$ and $b_i = *$, if $a_i = *$, and $b_i = \Rightarrow$, if $a_i \neq *$ ..

$\Delta(b, Ax) = (b, B \Rightarrow)$, where $B = b_1 \dots b_k$ és $b_i = x_{ai}$. (The R/W head moves only on the last used tape.)

$\Delta(b, A) = (a, Bx)$, where $B = b_1 \dots b_k$ és $b_i = c_i$, if $a_i = x_{ci}$.

The Super Turing machine in the example writes on the first unused tape as many x signs as the lenght of the input word.

Definition: Let T be an arbitrary (Super or simple) Turing machine, $k \geq 1$ and w = $(w_1, \dots, w_k)$, where $w_1, \dots, w_k \in \Sigma^*$.

The length of the computation of T on input w is the least n, for which the computation on the input word w, the state belonging to K_n is the final state (if it exists).

If the computation is not finite, its length is infinite.

Notation: $\tau_T(w) = n$, and $\tau_T(w) = \infty$ respectively.

Remark: We represent the length of the input word $w = (w_1, \dots w_k)$ with the function $l(w)$. Its value is $l(w) = l(w_1) + \dots + l(w_k)$.

Definition: Let T be a (Super or simple) Turing machine.

The time complexity of T is the following function:

$t_T(n) = \max\{\tau_T(w) | w = (w_1, \dots, w_k)$, where $w_1, \dots, w_k \in \Sigma^*$ and

$l(w) \leq n$.

Definition: Let T be an arbitrary (Super or simple) Turing machine, $k \geq 1$ and $= (w_1, \dots, w_k)$, where $w_1, \dots, w_k \in \Sigma^*$.

Also let $K_0, K_1, \dots, K_n$ be the computation of T on input w. (i.e. $K_0 = (s, w)$.)

The space used by the Turing machine T on input w

$\sigma_T(w) = \max\{|w_i|\ K_i = (q_i,w_i),\ i = 0, \dots, n\}$.

If the computation of T on input w is infinite, then

$\sigma_T(w) = \lim_{n\to\infty} \max\{|w_i|\ |K_i = (q_i, w_i),\ i = 0, \dots, n\}$.

Remark: Even if the computation of a Turing machine is not finite, the space demand can be. This is typically an infinite loop.

Definition: Let T be a (Super or simple) Turing machine.

The space complexity of T is the following function:

$s_T(n) = \max\{\sigma_T(w)|\ w = (w_1, \dots, w_k)$, where $w_1, \dots, w_k \in \Sigma^*$ and $(w) \leq n\}$.

Definition: Let T be an arbitrary (Super or simple) Turing machine, $k \geq 1$ and $w = (w_1, \ldots, w_k)$, where $w_1, \ldots, w_k \in \Sigma^*$.

Also let $K_0, K_1, \ldots, K_n$ be the computation of T on input w. (i.e. $K_0 = (s, w)$.) The processor demand of Turing machine T on input w is

$$\pi_T(w) = \max\{k_i | K_i = (q_i, w^i),\ w^i = (w_1^i, \ldots w_{k_i}^i), i = 0, \ldots, n\}$$

If the computation of T on input w is infinite, then

$$\lim_{n\to\infty} \max\{k_i | K_i = (q_i, w^i),\ w^i = (w_1^i, \ldots, w_{k_i}^i),\ i = 0, \ldots, n\}$$

Definition: Let T be a (Super or simple) Turing machine.

The processor complexity of T is the following function:

$p_T(n) = \max\{\pi_T(w) |\ w = (w_1, \ldots, w_k)\}$, where $w_1, \ldots, w_k \in \Sigma^*$ and $(w) \leq n\}$.

Definition: Let T be a Super Turing machine and $K = (q,w)$ be a configuration of T, where $w = (w_1, \ldots, w_k)$, $k \geq 1$ and $w_1, \ldots, w_k \in \Sigma^*$.

Furthermore, let $A = (a_1, \ldots, a_k)$ be the word formed by the currently scanned symbols by the R/W heads in K and $1 \leq i, j \leq k$.

We say that the data flows from tape j to the direction of tape i if $\exists\ A' = (a'_1, \ldots, a'_k)$, with $a_j \neq a'_j$ and $a_l = a'_l$ $\forall\ 1 \in \{1, \ldots, k\} \setminus \{j\}$ and if $B = \Delta(q, A)$ and $B' = \Delta(q, A')$, then $b_i \neq b'_i$.

Accordingly$\zeta_{j,i}(K) = 0$, if there is no data stream and $\zeta_{j,i}(K) = 1$, if there is.

The amount of the streamed data belonging to the configuration:

$$\zeta_{j,i}(K) = \sum_{\substack{1\leq i,\ j\leq k\\ i\neq j}} \zeta_{j,i}(K).$$

According to the definition there is data stream from tape j to the direction of tape i, if we can change the letter under the R/W head on tape j such that during the configuration transition belonging to this letter the content of tape i will not be the same as in the original case.

Definition: Let T be a Super Turing machine, $k \geq 1$ and $w = (w_1, \ldots, w_k)$ where $w_1, \ldots, w_k \in \Sigma^*$.

Also let $K_0, K_1, \ldots, K_n$ be the computation of T on input w. (Namely $K_0 = (s, w)$.)

The communicational demand of Turing machine T on input w is $\zeta_T(w) = \sum_{i=0\ldots n} \zeta(K_i)$.

The communicational bandwidth demand of Turing machine T on input w is $\zeta_T(w) = \max\{\zeta(K_i) | K_i, i = 0, \ldots, n\}$.

Definition: Let T be a Super Turing machine.

The communicational complexity of T is the following function:

$z_T(n) = \max\{\zeta_T(w)\ |\ w = (w_1, \ldots, w_k)$, where $w_1, \ldots, w_k \in \Sigma^*$ and $(w) \leq n\}$.

The absolute bandwidth demand of T is the following function:

$d_T(n) = \max\{\eta_T(w) \mid w = (w_1, \dots, w_k)$, where $w_1, \dots, w_k \in \Sigma^*$ and $l(w) \leq n\}$

The above definitions of the time, space and processor complexity are the clarifications of the notions of chapter Complexity concepts, in the Turing machine model.

Those notions' equivalents for the Super Turing machine:

General - Super Turing machine

Absolute time complexity ↔ Time complexity

Total space complexity ↔ Space complexity

Processor complexity ↔ Processor complexity

Communicational complexity ↔ Communicational complexity

Thread time complexity

The time complexity belonging to tape (thread) *i* can be spawned from the number of steps between the first and last significant operations, just like the general time complexity. Significant operation is the rewrite of a sign on the tape and moving the R/W head.

Total time complexity

The sum of the time complexities of threads.

Thread space complexity

The space complexity observed tapewise.

The restrictions referring to the transition function of the Super Turing machine give the different computing models and architectures.

It is clear that with the Super Turing machine model the Turing machines with one or more tapes can easily be modeled.

1.2.1.2.1. Simulation

Definition: Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be algorithms possibly in different models.

We say that $\mathcal{A}_1$ simulates $\mathcal{A}_2$ if for every possible input w of $\mathcal{A}_2$, $\mathcal{A}_1(w) = \mathcal{A}_2(w)$, which means that it gives the same output (and it does not give an answer, if $\mathcal{A}_2$ does not give it).

Let M_1 and M_2 be two models of computation. We say that M_1 is at least as powerful as M_2 if for all $\mathcal{A}_2 \in M_2$ exists $\mathcal{A}_1 \in M_1$ such that $\mathcal{A}_1$ simulates $\mathcal{A}_2$.

The Super Turing machine can be considered as a quite general model. With the restrictions put on the state transition function it can simulate the other models.

Examples (for each restriction)

1. If we reduce the domain of Δ from $Q \backslash H \times \Sigma^*$ to $Q \backslash H \times \Sigma^k$ we will get to the Turing – machine with the k tape. In this case it is not necessary to reduce the range as well, since we can simply not use the data written on other tapes.

(Mathematically it can be defined by a projection.)

2. If Δ can be written as δ^∞ where δ: $Q \backslash H \times \Sigma^k \rightarrow Q \times (\Sigma \cup \{\leftarrow \rightarrow\})^k$. Here, basically, the Super Turing machine can be interpreted as the vector of Turing machines with k tapes with the same behavior. In this case we store the state that is necessary for the component Turing machines on one of the tapes.

The different use of the mutual space can be modeled as well.

3. If we define the transition function such that the data to the tape $i + 1$ flows only from the tape i and the sign that was there before does not influence the symbol written on the tape $i + 1$, then the tape $i + 1$ can be considered as an external storage of the tape i.

4. Furthermore, if we assume the Super Turing machine in point 3 such that the data flows from the direction of the tape $i + 1$ to the $i + 2$ (but the value of the tape $i + 2$ depends only on $i + 1$, then we can model such an operation as the tape $i + 1$ is a buffer storage between the tapes i and $i + 2$.

5. Similarly to the previous ones the concurrent use of memory can be modeled as well. With a well defined transition function, to the direction of the tape $i + 1$ both from the ith and the i + 2nd tape there can be a data stream. In this case the tape $i + 1$ is a shared memory of tapes i and $i + 2$.

Dropping the finiteness condition of the state space would cause the problem that a new model of algorithms would be born and with that we could solve more problems, even the ones that cannot be solved with traditional algorithms. It is because then the transition function cannot be computed by a Turing machine anymore and this way a non-computable transition function can be given.

(Example of the implementation of a nonrecursive function in case of infinite state space:

$f(n)$ = to the longest sequence of 1 as an answer that can be given on an empty input by a Turing machine of state n.)

It is important to see that with the use of the Super Turing machine the set of solvable problems is not expanding, only the time complexity of the solution can be reduced.

Theorem: For each Super Turing machine T there exists a simple Turing machine T' which simulates the first one.

Proof:

The proof is a constructive one, but here we only give the sketch of it.

First we have to define a simulating Turing machine, and then verify that it really solves the problem.

Let T' be the simulating Turing machine.

We will use the first tape of T' to store the necessary information for the proper operation. More precisely, T' uses tape 1 to execute the operations required by the simulation of the transition function.

On the second tape there will be the inputs, then at an internal step of the computation the auxialary results will get back here.

The structure of the tapes of T' and the operation can be described as follows:

The structure of the content of tape 2 is the following:

If the actual configuration of the simulated T is

$K = (q, w_{1,1}, a_1, w_{1,2}, \ldots, w_{k,1}, a_k, w_{k,2})$, then on the tape the word

$\#w_{1,1} \,\&\, a_1 \,\&\, w_{1,2} \,\#\, \dots \,\#\, w_{k,1} \,\&\, a_k \,\&\, w_{k,2}\#$

can be found.

The set of states of *T'* is $Q' = Q \times Q_1$, where Q is the set of states of T and Q_1 can be used to describe the simulating steps.

T' is iterating the following sequence of steps:

1. Reads the whole 2nd tape and copies the content of the tape between the symbols & … & to the first tape; we assign an appropriate subset of Q_1 to these steps.

2. After finishing the copy operation, it moves the head to the beginning of the 1st tape and switches to executive state; we assign now another appropriate subset of Q_1 to these steps.

3. In the executive state it behaves like Turing machine according to $\Delta(q, \dots)$. Here, obviously, another subset of Q_1 is necessary.

4. After the implementation of the change of sate, it changes the content of the 2nd tape, corresponding to the result. If it finds a letter to be written, it will write it, if it finds a character that means head movement, then it will move the corresponding & symbol.

5. If the new state *q'* is a final state, the simulation stops, otherwise it starts over again with the first point.

The execution of the 4th point is a little bit more complicated, since writing after the last character of the tape or erasing the last character requires an appropriate shift of the successive words.

The second part of the proof is to verify that *T'* actually simulates *T*.

It can be done with the usual simulation proving method:

We have to prove that the computation $K_0, K_1, \dots$ of T can be injectively embedded to the computation $K'_0, K'_1, \dots$ of \$*T'*\$.

Formally: $\exists\ i_0 < i_1 < i_2 < \dots,$, such that K_j and K'_{i_j} are corresponding uniquely to each other. This sequence of indices may be for example the indices of configurations at the start of the first sequence in the simulation of each step.

Essentially, this proves the theorem. √

We have to require the Super Turing machine's state transition function to be recursive, otherwise we would get a model that is too general, which would not fit to the notion of the algorithm. If the transition function is arbitrary, then it can be used for solving any problem. Even those that are not thought to be solvable in any manner.

Refining the definition, we may demand Δ to be implemented with a limited number of states on a single tape Turing machine. This does not affect the set of problems that can be solved by the Super Turing machine, but the different complexity theoretic observations will reflect more the real complexity of the problem. Otherwise, there can be a solution that gives answer to each input in 1 step.

State transition function bounded by 1 state:

Substantially, on every tape, only the content of the tape determines the behavior. It can be considered as a vector of Turing machines that are not fully independent and are operating simultaneously. Between the tapes there is no direct data stream.

State transition function bounded by 2 states:

Between the tapes there is a minimal data stream (a maximum of 1 bit). The tapes can be considered as they are locally the same.

1.2.1.2.2. Complexity classes

Using the definition of the model of single or more generally the fixed k-tape Turing machines we can define the complexities of problems and complexity classes of problems as well.

Definition: Let T be a Turing machine, $h \in H$ and $L \subseteq \Sigma^*$ be a language.

We say that T accepts the word $w \in \Sigma^*$, if the input of T is w, then T stops in the state h.

T recognizes the language $L \subseteq \Sigma^*$, if $L = \{w \mid T \text{ accepts } w\}$. Notation: $= L(T)$.

Furthermore, if T terminates on every input, then we say that T decides L.

Definition: Let $f\colon \mathbb{N} \to \mathbb{R}^+$ be a monotonically increasing function. The time complexity class corresponding to the growth rate of f is

$TIME(f(n)) = \{L | \exists$ T Turing machine with $L = L(T)$ and the time complexity $t(n)$ of T is $O(f(n))\}$.

Definition: Let $f\colon \mathbb{N} \to \mathbb{R}^+$ be a monotonically increasing function. The space complexity class corresponding to the growth rate of f is

$SPACE(f(n)) = \{L | \exists$ T Turing machine with $L = L(T)$, and the space complexity $s(n)$ of T is $O(f(n))\}$.

We can give a similar definition for the case of Super Turing machines.

Definition: Let $f\colon \mathbb{N} \to \mathbb{R}^+$ be a monotonically increasing function. The parallel time complexity class corresponding to the growth rate of f is

$PTIME(f(n)) = \{L |$ T Super Turing machine with $L = L(T)$, and the time complexity $t(n)$ of T is $O(f(n))\}$.

Definition: Let $f\colon \mathbb{N} \to \mathbb{R}^+$ be a monotonically increasing function. The parallel space complexity class corresponding to the growth rate of f is

$PSPACE(f(n)) = \{L | \exists$ T Super Turing Machine with $L = L(T)$, and the space complexity $s(n)$ of T is $O(f(n))\}$.

In case of parallel algorithms it is important to know how many processors are necessary to solve the problem in a given period of time. This is why the following combined class of complexity has a distinguished role.

Definition: Let $f, g\colon \mathbb{N} \to \mathbb{R}^+$ be monotonically increasing functions.

The combined time-processor complexity class is

$TIMEPROC(f(n,g(n)) = \{L | \exists$ T Super Turing machine with $L = L(T)$, $\{L | \exists$ T and the time complexity $t(n)$ of T is $O(f(n))$ while the processor complexity $p(n)$ is$O(g(n))\}$.

Important complexity classes:

1. Class of languages that can be decided in polynomial time:

$$P = \bigcup_{i=0}^{\infty} TIME\left(n^i\right).$$

Remark: One can define the so-called nondeterministic Turing machine. Essentially, the nondeterministic Turing machine can go from a configuration to others, but not only to a single one. And if it can go to a configuration, it will. This can be pictured like the Turing machine is splitting continuously and more and more machines are trying to solve the problem. If at least one of them accepts the input, we accept it. The number of its steps is the shortest solution. If there are no accepting computations and all computations are finite, then the longest computation defines the necessary number of steps. Furthermore, if there is an infinite computation, then the number of steps are infinite.

The complexity classes (that play an important role in the observation of algorithms) can be defined with this Turing machine model as well. Similarly to the class *P* a class *NP* can be defined which shows basically the class of problems that can be verified in polynomial time.

A very important research question whether the two classes are the same. (*P = NP?*)

2. The class of languages that can be recognized in a polylogarithmic time with a polynomial number of processor.

$$NC = \bigcup_{i=0}^{\infty} TIMEPROC\left(\log^i(n),\ n^i\right)$$

The problems belonging to this class of complexity can be effectively parallelized (or they can be solved effectively in a sequential way, too).

3. Questions and problems

1. Show that $n^2 - 2n + 3 \in O(n^2)$!

Hint: Prove that $\exists$ c > 0, such that

$$\frac{n^2 - 2n + 3}{n^2} < c,$$

if

$n > 0$, then let us multiply each side by n^2.)

2. Let $p(x)$ and $q(x)$ be two polynomials with positive leading coefficients. Assume that $\deg(p) \leq \deg(q)$.

a. Then $p(n) \in O(q(n))$.

b. If $\deg(p) < \deg(q)$, then $q(n) \notin O(p(n))$.

(Hint: Let $k_1 = \deg(p)$ and $k_2 = \deg(q)$. Similarly to the previous problem, let us prove that $p(n) \in O(n^{k1})$ és $n^{k2} \in O(q(n))$. The statement a) is the consequence of property 4 and consequence 1. Following the consequence 1 statement b) can be proved.

3. Give a Super Turing machine, which determines two words longest common subsequence in linear time! (Subsequence: a sequence of letters such that it appears in the given word in the same order, but not necessarily as consecutive members.)

4. Show that P ⊆ NC!

5. The number of processors and the number of states in the definition of transition function is an important argument.

Show that all algorithmically solvable problem can be solved by a (non limited) Super Turing machine in linear time. (Then communicational complexity can be very big)

4. Literature

T. H. Cormen, C. E. Leiserson, R. L. Rivest: *Algoritmusok* (Introduction to Algorithms) Műszaki Könyvkiadó, 1999 (The MIT Press, 1990)

D. E. Knuth: *A számítógép-programozás művészete* (The Art of Computer Programming); Első kötet: Alapvető algoritmusok (Fundamental Algorithms), második kiadás, Műszaki Könyvkiadó, 1994 (Addison-Wesley Professional)

L. Lovász: Algoritmusok bonyolultsága; ELTE TTK jegyzet, Tankönyvkiadó, 1989 [in Hungarian]

C. H. Papadimitriu: *Számítási bonyolultság* (Computational Complexity); Novadat, 1999 (Addison-Wesley, 1993)

L. Rónyai, G. Ivanyos, R. Szabó: *Algoritmusok*; Typotex, 1998 [in Hungarian]

2. fejezet - Parallel grammar models

2.1. An overview: Generative grammars, Chomsky-hierarchy

In the next chapters we present some parallel forms and extensions of the generative grammars that are well-known from classical formal language theory.

To understand the definitions of these chapters more easily, first we recall some more basic definitions of formal language theory and show our notations.

A basic concept is the rewriting system; we will use its various forms in the next chapters.

Definition: A formal system (or rewriting system) is a pair $W = (T,H)$, where T is an alphabet and H is a finite subset of the Cartesian product $T^*\times T^*$. The elements of H are the (rewriting) rules or productions. If $(p,q)\in H$, then we usually write it in the form $p\to q$. Let $r,s\in T^*$, we say that s can be directly derived (or derived in one step) from r (formally: $r\Rightarrow s$) if there are words $p,p',p'',q\in T^*$ such that $r = p'pp''$, $s = p'qp''$ and $p\to q\in H$. A derivation step can be understood as follows: the word s can be obtained from the word r in such a way that in s the right-hand-side q of a production of H is written instead of the left-hand-side p of the same production in r. We say that s can be derived from r (formally: $r\Rightarrow^*s$) if there is a sequence of words $r_0,r_1,\ldots,r_k$ ($k\geq 1$) such that $r_0 = r$, $r_k = s$ and $r_i\Rightarrow r_{i+1}$ holds for every $0\leq i<k$. Moreover, as it is usual, we consider the relation $r\Rightarrow^*r$ for every word r. Actually, the relation $\Rightarrow^*$is the reflexive and transitive closure of the relation $\Rightarrow$.

In the generative systems, the languages that can be derived from a given, usually finite set of axioms (words), are analysed. In the classical formal language theory and in (theoretical) computer science the generative grammars and the language families defined by them play a fundamental role. These families were introduced by N. Chomsky in the 1950's. In the generative grammars the alphabet is divided to two disjoint groups: the terminals (or in some terminology, constants) and the nonterminals (i.e., variables). The most important concepts are the following.

Definition: The ordered quadruple $G = (N,T,S,H)$ is a (generative) grammar if N and T are two disjoint finite alphabets, $S\in N$ and $H\subseteq(N\cup T)^*N(N\cup T)^*\times(N\cup T)^*$. The elements of N are the nonterminals (variables), and they are usually denoted by capital letters. The elements of T are the terminals (terminal letters, or constants) and they are usually denoted by lower case letters. The elements of H are called productions or (derivation) rules; a rule (p,q) is usually written as $p\to q$. The symbol S is a special nonterminal letter, which is the startsymbol of the derivations (it is also called sentence symbol or axiom).

In a generative grammar the word $t\in(N\cup T)^*$ can be derived in one step (or directly derivable) from the word $r\in(N\cup T)^*$ (formally: $r\Rightarrow t$) if there is a rule $p\to q$ in H and words $p',p''\in(N\cup T)^*$ such that $r = p'pp''$ and $t = p'qp''$. The derivation relation can be given in the same way from the direct derivation as at the rewriting systems, i.e., it is the transitive and reflexive closure of the direct derivations, it is denoted by $\Rightarrow^*$.

The words of $(N\cup T)^*$ that can be derived from the startsymbol S are called sentential forms. The language generated by the generative grammar G consists of all words of T^* that can be derived from S, formally: $L(G) = \{w|\ S\Rightarrow^*w\}\cap T^*$.

By the definition of the grammar and the generated language, for every grammar there is a language assigned uniquely. However it does not hold in the other way, a language can be generated by various grammars. Two grammars are called equivalent if they generate the same language modulo the empty word, i.e., the difference of the two generated languages can be at most the empty word λ.

We may observe from the definition that the central concept, the derivation works in a sequential way, at every time instant only one rule is used in an appropriate place of the sentential form.

By restricting the forms of the usable derivation rules the possible types of the grammars are classified by Chomsky. Consequently the classes of the generated languages are also defined and widely used.

Definition: The grammar $G = (N,T,S,H)$ is type i grammar (for $i = 0,1,2,2.5,3$) if the i-th restriction holds:

(0) Phrase-structured grammar: There is no additional restriction on the general definition of generative grammars.

(1) Context-sensitive grammar: Every rule in H must have a form $qAr \to qpr$, where $A \in N$, $p \in (N \cup T)^* \setminus \{\lambda\}$, $q,r \in (N \cup T)^*$; with only one possible exception: the rule $S \to \lambda$, but if $S \to \lambda \in H$, then S does not occur on the right-hand-side of any productions.

(2) Context-free grammar: Every rule is of the form $A \to p$, where $A \in N$, $p \in (N \cup T)^*$.

(2.5) Linear grammar: Every rule is in one of the following forms: $A \to uBv$ or $A \to u$, where $A,B \in N$, $u,v \in T^*$.

(3) Regular grammar: Every rule is in one of the following forms: $A \to uB$ or $A \to u$, where $A,B \in N$, $u \in T^*$.

A language is recursively enumerable (RE)/ context-sensitive (CS)/ context-free (CF)/ linear (LIN)/ regular (REG) if it can be generated by a phrase-structured/ context-sensitive/ context-free/ linear/ regular grammar. Further at the values $i = 0,1,2,3$ we say that a language L is of type i if there is a grammar of type i that generates it.

The Chomsky-type language classes form a strict hierarchy: FIN⊂REG⊂LIN⊂CF⊂CS⊂RE⊂ALL, where FIN stands for the set of finite languages and ALL stands for the set of all languages (over a given alphabet T that contains at least two letters). We note here that over a one-letter alphabet the hierarchy becomes FIN⊂REG = LIN = CF⊂CS⊂RE⊂ALL.

We note here that there is an equivalent grammar for every generative grammar such that its startsymbol does not occur on the right-hand-side of any of its productions. Furthermore we may have stronger restrictions such that every grammar of the given class has an equivalent grammar fulfilling the stronger restriction. Now we show examples for restrictions for two of the language classes.

There is an equivalent grammar for every context-free grammar such that its rules are of the forms $A \to a$, $A \to BC$ where ($A,B,C \in N$, $a \in T$). A grammar fulfilling this restriction is in Chomsky normal form.

Every linear language can be generated by a grammar having only rules that contain at most one terminal (it is a normal form for linear grammars).

We say that a language class is closed under a given language operation if applying the operation on any elements of the language class, the resulting language will also be a member of the given class. It is well-known that the classes of Chomsky type 0, 1, 2, and 3 all closed under the operations union, concatenation and (Kleene) iteration. The class of linear languages is closed under union, but it is not closed under concatenation and iteration. The classes of regular and the context-sensitive languages are closed under intersection and complementation. The classes of linear and context-free languages are not closed under intersection and complementation. However the intersection of a context-free and a regular language is context-free, i.e., the class of context-free languages are closed under intersection with regular languages. The class of recursively enumerable languages is closed under intersection, but it is not closed under complementation. We summarize the closure properties of these language classes in Table 2.1 (+ means closure, - means non-closure of the class under the given operation).

2.1. táblázat - Closure properties of the language classes of the Chomsky hierarchy.

language class	operation				
	union	concatenation	Kleene-star	intersection	complement
regular	+	+	+	+	+

linear	+	-	-	-	-
context-free	+	+	+	-	-
context-sensitive	+	+	+	+	+
recursively enumerable	+	+	+	+	-

The three language operations: union, concatenation and iteration are the regular operations. By their help, based on the letters of the alphabet, the empty word and the sign of the empty set (that is standing for the empty language) the regular expressions can be built (brackets can also be used). (Usually + denotes union, the usually dropped · stands for the concatenation and * stands for the iteration.) The regular expressions describe exactly the regular languages. It is also usual to define the Kleene-plus iteration for languages: $^+$ stands for it, and it is defined as $L^+ = LL^*$ for any language L.

For more details about the classical formal language theory we recommend, e.g., the book *Géza Horváth, Benedek Nagy: Formal Languages and Automata Theory, Typotex, 2014*.

The main motivation of the investigation of parallel grammars is to model, characterize and get a better view of the parallel computing. The next chapters are about these parallel variations of rewriting systems and grammars.

2.2. Indian, Russian and bounded parallel grammars

In the traditional formal languages, the generative grammars use sequential derivations. Even these systems are non-deterministic, at each derivation step only one rewriting rule is applied on exactly one place, i.e., on one subword of the actual sentential form. In this section we show an overview on entirely parallel systems in which the derivation steps use parallel rewriting.

The most simple parallel grammars are the Indian and Russian parallel grammars, we deal with them in the next subsections.

2.2.2.2.1. Indian parallel grammars

When applying derivation rules in a parallel manner we can have several types of restrictions what could happen with various occurrences of the same letter. In Indian parallel grammars in every step of the derivation exactly one letter is being rewritten; and all of its occurrences are rewritten by the same rewriting rule. Formally we can define these systems in the following way.

Definition: Let $IG = (T,s,H)$ be a triplet, where T is a nonempty, finite alphabet, $s\in T^*$ (axiom) and H is a finite set of rewriting rules (productions) of the form $a\to r$ (where $a\in T$, $r\in T^*$). Then IG is an Indian parallel grammar. The word v is directly derived from the word u (formally: $u\Rightarrow v$) if there is a letter $a\in T$ and derivation rule $a\to r\in H$ such that v is obtained from u by changing all occurrences of the letter a to r. By the transitive and reflexive closure of direct derivation the concept of derivation is obtained ($\Rightarrow^*$). The language generated by IG consists of all the words that can be derived from the axiom s, formally: $L(IG) = \{w| s\Rightarrow^* w\}$.

Notice that for every Indian parallel grammar IG its axiom s is contained in the generated language. These grammars are pure rewriting systems, since there is no distinction of terminal and nonterminal letters. In this way every word obtained during a derivation is also an element of the language. The rules of these grammars are closely related to the possible rules of the context-free grammars, since the left-hand-side of each rule contains exactly one letter in each rule. The Indian parallel grammars are partial parallel systems in the sense that not every letter of a sentential form (i.e., actual word) is rewritten in a derivation step, only (all) the occurrences of a chosen letter are rewritten. Although these grammars can be considered very simple generating devices, it can be shown that some sophisticated (i.e., non context-free) languages can be generated by their help.

Example (language of words with length power of three)

Let $IG = (\{a\},a,\{a\to aaa\})$ an Indian parallel grammar. Then there is the following derivation (moreover all longer derivations start with these steps): $a\Rightarrow aaa\Rightarrow aaaaaaaaa\Rightarrow a^{27}$. $L(IG)$ contains exactly those words over the unary alphabet $\{a\}$ that have length power of 3 (with a nonnegative integer as exponent).

In the next example a nondeterministic Indian parallel grammar is shown.

Example (a regular language generated by a nondeterministic Indian parallel grammar)

Let IG = $(\{a,b,c\},b,\{a\to cc,b\to cc,b\to ab\})$ an Indian parallel grammar. It is easy to show that the possible derivations give words of the following form, i.e., the generated language $L(IG)$ is described by the regular expression $(cc)^*a^*b+(cc)^*a^*cc$.

In our next example we show that these parallel rewriting systems are very limited in the other side.

Example (a finite language that cannot be generated by Indian parallel grammar)

Let $L = \{a,aaa\}$ be a finite language. We show that there is no such an Indian parallel grammar that generates L. The proof is indirect: let us assume that IG is an Indian parallel grammar that generates L. Since the axiom is in the generated language, one of the words must be the axiom of IG. There are two options:

If a is the axiom, then we must generate the other word from this without deriving other words (during the derivation). In this way our system must have the rule $a\to aaa$. However, in this case the derivation can be continued, and in the next step the word $aaaaaaaaa$ is obtained which is not in L. This is a contradiction. Let us see the other case:

If aaa is the axiom, then we must generate the other word, a, from this word in a derivation step. However, we can decrease the length only by the rule $a\to\lambda$. But in case we have this rule in our system the empty word λ will also be derivable from the axiom in one derivation step. Since the empty word is not in the language a contradiction is obtained in this case also. Thus L is not a language that can be generated by Indian parallel grammar.

The widespread application of the Indian parallel grammars is blocked by the facts that: although a kind of parallelism is used, the same rule must be applied for every occurrence of the given letter; moreover these systems have only "terminal" letters. To have a more usable system one usual way is to allow to use nonterminal letters. In this way the generating power of the systems are increasing. Instead of this step we present the more powerful Russian parallel grammars in the next subsection.

2.2.2.2.2. Russian parallel grammars

The Russian parallel grammars mix some features of the Indian parallel grammars and the traditional (sequential) context-free grammars. In this way they could be used in more applications than the Indian parallel grammars (even the extended versions where nonterminals are allowed).

Definition: Let $RG = (N,T,S,H)$, where N and T are two nonempty, finite and disjoint alphabets (the nonterminal and the terminal alphabets, respectively), $S\in N$, H is the finite set of derivation (or rewriting) rules of the form $(A\to r,i)$ where $A\to r$ ($A\in N$, $r\in(N\cup T)^*\setminus\{\lambda\}$) is a rule and $i\in\{1,2\}$ gives the way of the application of the rule: in case $i = 1$ the rule is applied only at one (it can be any) occurrence of the nonterminal A at a derivation step, while in case $i = 2$ the rule must be applied at every occurrence of the nonterminal A in a derivation step.

The word v directly derives from the word u (formally: $u\Rightarrow v$) if

- there is an $A\in N$ and $(A\to r,1)\in$ H, such that v is obtained from u by replacing one occurrence of the nonterminal A with r, i.e., $u = pAq$ and $v = prq$ for some $p,q\in(N\cup T)^*$ (these derivation steps are sequential); or

- there is an $A\in N$ and $(A\to r,2)\in H$, such that v is obtained from u such that all the occurrences of the nonterminal A are rewritten to r, i.e. $u = p_1Ap_2\ldots p_{n-1}Ap_n$ and $v = p_1rp_2\ldots p_{n-1}rp_n$ where $p_1,p_2,\ldots,p_n\in((N\setminus\{A\})\cup T)^*$ (these are the parallel derivation steps).

By the reflexive and transitive closure of direct derivation the derivation relation is obtained ($\Rightarrow^*$). The language generated by the Russian parallel grammar RG consists of all terminal words that can be derived from S: $L(RG) = \{w| S\Rightarrow^* w\}\cap T^*$.

As we can see, the Russian parallel grammars solve the problems listed at the Indian parallel grammars: There are terminals and nonterminals. Moreover it is allowed to apply a rewriting rule only at one occurrence of the given nonterminal in some derivation steps. It is obvious that for every context-free grammar there is a Russian parallel grammar that generates the same language. To obtain such an equivalent grammar, assume that the context-free grammar is given in Chomsky normal form. Then for every context-free rule $A\rightarrow r$ one needs to use the rule $(A\rightarrow r,1)$ in the equivalent Russian parallel grammar. (Actually, if all the rules are used with value 1, then the grammar is not really parallel, but works in the same way as a traditional context-free grammar). Furthermore, it can be seen that Russian parallel grammars are more powerful than Indian parallel grammars.

Theorem: Let $IG = (T,s,H)$ be an Indian parallel grammar. One can effectively construct a Russian parallel grammar $RG = (N,T,S,H')$ such that $L(IG) = L(RG)$.

Proof: let us construct RG: let $N = \{S\}\cup\{X_a| a\in T\}$ (where N contains new symbols, none of them is an element of T, X_a is introduced to play the role of the terminal a in continuing derivations), further let $H' = \{(S\rightarrow s,2),(S\rightarrow s',2)\}\cup \{(X_a\rightarrow r',2) \mid a\rightarrow r\in H\}\cup\{(X_a\rightarrow a,2)\}$, where $s'\in N^*$ and $r'\in N^*$ are constructed from $s\in T^*$ and $r\in T^*$ by replacing every (terminal) letter to its newly introduced nonterminal pair, respectively. Obviously the derivation can start in two way in RG: in case $S\Rightarrow s$ is used the axiom of IG is derived, otherwise the derivation starts with the application of rule $S\Rightarrow s'$. In this second case, the sentential form s' contains only nonterminals, the derivation can be continued by rules of the form $(X_a\rightarrow r',2)$. In this way we simulate the original derivation steps of the Indian parallel grammar by the new rewriting rules $X_a\rightarrow r$. When the derivation is not continued in the Indian parallel grammar, i.e., we obtained the word that we want, then in the grammar RG for all terminal $a\in T$ we need to apply the rules $(X_a\rightarrow a,2)$ for each occurring nonterminal X_a to obtain the derived terminal word. In this way all the words derived in IG can be derived in RG also, and for each terminating derivation of RG there is a derivation in IG obtaining the same word. √

In this way the Russian parallel grammars can be seen as such extensions of the context-free grammars that allow parallel derivation steps defined in the same way as at the Indian parallel grammars. Moreover the sequential and the parallel steps can be combined during a derivation.

Example (the copy language)

The copy language $L = \{ww| w\in\{a,b\}^*,w\neq\lambda\}$ is a well-known non context-free language. Let $RG = (\{S,A\},\{a,b\},S,\{(S\rightarrow AA,1),(A\rightarrow aA,2),(A\rightarrow bA,2),(A\rightarrow a,2),(A\rightarrow b,2)\})$. Then the first step of the derivation is $S\Rightarrow AA$. After this we can build the same word twice parallely step by step, and finishing their construction at the same time. Therefore $L = L(RG)$.

In the next example we obtain also a well-known non context-free language by a parallel grammar.

Example (the language of cross-dependencies)

Let $RG = (\{S,A,B\},\{a,b\},S,\{(S\rightarrow ABAB,1),(A\rightarrow aA,2),(B\rightarrow bB,2),(A\rightarrow a,2), (B\rightarrow b,2)\})$ be a Russian parallel grammar. Then, the first derivation step is $S\Rightarrow ABAB$. Then the blocks of a's and the block of b's are constructed independently of each other, but the blocks of the same letters step by step together in a parallel manner. Thus $L(RG) = \{a^n b^m a^n b^m| m,n\geq 1\}$.

At the Russian parallel grammars we rewrite either only one or each occurrence of the given nonterminal by the right hand side of the chosen rule. In the next subsection we show systems that use parallelism but in a bounded way.

2.2.2.2.3. Grammars with bounded parallelism

2.2.2.2.3.2.2.3.1. k-parallel grammars

We call k-parallel the grammars that are 1-parallel, 2-parallel, 3-parallel, …, n-parallel, etc. In an n-parallel grammar in all derivation steps, but the first one, exactly n nonterminals are rewritten.

Definition: Let $kG = (N,T,S,H)$, where N,T,S and H are the same as at context-free grammars such that the startsymbol S does not occur in the right-hand-side of any rule, then kG is an n-parallel grammar ($n \geq 1$). The word v is directly derived from the word u (formally $u \Rightarrow v$) if

- $u = S$ and $S \rightarrow v \in H$, (first step of a derivation); or

- $u = p_1A_1p_2A_2p_3 \ldots p_nA_np_{n+1}$ and $v = p_1r_1p_2r_2p_3 \ldots p_nr_np_{n+1}$ where $p_1,p_2,\ldots,p_{n+1} \in (N \cup T)^*$ and $A_i \rightarrow r_i \in H$, for all $1 \leq i \leq n$ (n-parallel derivation step).

The reflexive and transitive closure of the direct derivation gives the derivation relation ($\Rightarrow^*$). The language generated by kG consists of all terminal words that can be derived from S: $L(kG) = \{w \mid S \Rightarrow^* w\} \cap T^*$.

In an n-parallel grammar the degree of parallelism is tuned to n. It cannot be less or more in any steps of the derivation (excepting the very first step when there is no way to use parallelism, since the sentential form contains only one letter, the startsymbol). In these grammars there is a very efficient synchronization tool, since if the number of nonterminals decreases below n, then the derivation cannot be continued even it is not terminated.

From the definition we can see that the 1-parallel grammars are exactly the traditional, sequential, context-free grammars (parallelism with degree 1 is not a real parallelism). In this way we could view the k-parallel grammars as a kind of parallel extensions of the context-free grammars. However this type of extension entirely differs from the type of the extension seen at Russian parallel grammars (and from the way of parallelism used Indian and Russian parallel grammars). In k-parallel grammars it is exactly given how many rewriting is done in a derivation step, however various rules can be used in this rewriting, moreover the rewritten nonterminal is not fixed, and so various nonterminals can be rewritten at the same derivation step.

Example (language of multiple agreements)

Let $kG = (\{S,A,B,C\},\{a,b,c\},S,H)$ be a 3-parallel grammar with $H = \{S \rightarrow ABC, A \rightarrow aA, B \rightarrow bB, C \rightarrow cC, A \rightarrow a, B \rightarrow b, C \rightarrow c\}$. All derivations start with $S \Rightarrow ABC$. Then in each step exactly 3 nonterminals must be rewritten: the obtained sentential forms are $a^nAb^nBc^nC$ ($n \geq 0$ integer) such that the derivation is continuable. In the final step all the three nonterminals are replaced by a terminal, and thus the generated language: $L(kG) = \{a^nb^nc^n \mid n \geq 1\}$.

One of the weaknesses of the k-parallel grammars is that the degree of parallelism is fixed forehead. This could be a very good choice if the degree of the parallelism of the device is given. One may use those grammars that allow as many rule applications in a derivation step as the number of operations that the architecture allows to run parallely. To one direction the restriction can be eliminated easily. For some nonterminals A allowing the rule $A \rightarrow A$ the system can make less number of real rewriting if the sentential form contains (enough number of) the given nonterminals. In this way we may refine the control during the derivation process.

2.2.2.2.3.2.2.3.2. Scattered context grammars

The scattered context grammars are closely related to the previously shown k-parallel grammars. They can be seen as more developed versions, where the number of rule applications in a derivation step is not fixed for the whole derivation, therefore the degree of parallelism can be changed during a derivation and can be varied in various derivations. On the other side, it allows us to fix the rules that could be applied in a derivation step; moreover the order of the places where these rules are applied can also be given. Let us see the formal definition.

Definition: Let $SCG = (N,T,S,H)$, where N,T,S are the same as at context-free or k-parallel grammars and the finite set H contains finite (ordered) lists (so-called matrices) of context-free productions: $(A_1 \rightarrow r_1, A_2 \rightarrow r_2, \ldots, A_k \rightarrow r_k)$, where $k \geq 1$ integer, and its value can be different for various matrices.

The word v can be directly derived from the word u (formally $u \Rightarrow v$) if there is a matrix $(A_1 \rightarrow r_1, A_2 \rightarrow r_2, \ldots, A_k \rightarrow r_k)$ in H such that $u = p_1A_1p_2A_2p_3 \ldots p_kA_kp_{k+1}$ and $v = p_1r_1p_2r_2p_3 \ldots p_kr_kp_{k+1}$ for some $p_1,p_2,\ldots,p_{n+1} \in (N \cup T)^*$.

The reflexive and transitive closure of the direct derivation gives the derivation relation ($\Rightarrow^*$). The language generated by SCG consists of all terminal words that can be derived from S:

$L(SCG) = \{w \mid S \Rightarrow^* w\} \cap T^*$.

As we can see the degree of parallelism is fixed in each matrix, but it is (usually) not fixed for the whole grammar. On the other side, the number of rules in a matrix is fixed therefore these systems are partially parallel. Moreover the rules and their order are fixed in a matrix: the order of the left-hand-side of the rules, i.e., the order of the places of the nonterminals being rewritten is also fixed in each matrix.

Example (a language of multiple agreements)

Let $SCG = (\{S,A,B\},\{a,b\},S,\{(S\to ABA),(A\to aA,B\to bB,A\to aA),(A\to\lambda,\ B\to\lambda,A\to\lambda)\})$ be a scattered context grammar. The derivation starts with the step $S\Rightarrow ABA$. Then, by applying the rules of the second matrix the terminals are introduced in a parallel manner at the appropriate places of the sentential form. Finally, the last matrix gives the rules to finish the derivation. In this way the obtained language: $L(SCG) = \{a^n b^n a^n \mid n\geq 0\}$.

We have the following statement about the generating power of these systems, that we claim without proof.

Theorem: The language class generated by scattered context grammars is the class of recursively enumerable languages.

The next definition is closely related to definition of scattered context grammars.

Definition: Let $APG = (N,T,S,H)$, where N,T,S and H are the same as at scattered context grammars, is an absolute parallel grammar. The word v is directly derivable from the word u (formally $u\Rightarrow v$) if there is a matrix $(A_1\to r_1,A_2\to r_2,\ldots,A_k\to r_k)\in H$ such that $u = p_1A_1\mathrm{p}_2A_2p_3\ldots p_kA_kp_{k+1}$ and $v = p_1r_1p_2r_2p_3\ldots p_kr_kp_{k+1}$ for some $p_1,p_2,\ldots,p_{k+1}\in T^*$.

The reflexive and transitive closure of the direct derivation gives the derivation relation ($\Rightarrow^*$). The language generated by APG consists of all terminal words that can be derived from S:

$L(APG) = \{w \mid S\Rightarrow^* w\}\cap T^*$.

The only, but very important difference between the scattered context grammars and the absolute parallel grammars is the following: at absolute parallel grammars in each derivation step every nonterminal must be rewritten by an applicable rule of the given matrix, between two rewritten nonterminals there can only be terminals.

Before closing this section, we mention that there is an alternative version of the scattered context grammars. In these "unordered scattered context grammars" the rules of a matrix form an (unordered multi)set. Therefore applying a matrix in a derivation step, the only important thing is the following: each rule of the matrix is applied exactly one place. The original condition on the order of the application of the rules of the matrix is dropped in these grammars. It can be shown that these grammars cannot have more generative power than the scattered context grammars, since it is easy to construct a scattered context grammar that contains matrices with all the possible orders of the rules of the matrices of the original unordered scattered context grammar. In this way the same derivations are available in the constructed scattered context grammar as in the original unordered grammar.

There are additional versions in the literature (see, the end of the section).

2.2.2.2.3.2.2.3.3. Generalised parallel grammars

By mixing the features of the Russian parallel grammars and k-parallel grammars we can obtain the general parallel grammars.

Definition: Let $PG = (N,T,S,H)$, where N,T, and S are the same as at Russian parallel grammars and the finite set H contains pairs of the form $(A\to r,i)$, where the first element of a pair is a rewriting rule ($A\in N$, $\mathrm{r}\in(N\cup T)^*$) and $i\in\{\ = n,\geq n,\leq n \mid n$ is a positive integer$\}\cup\{t\}$ gives the mode of the application of the rewriting rule: the nonterminal A must be rewritten at exactly n occurrences (= n), at least n occurrences ($\geq n$), at most n occurrences ($\leq n$) or at all occurrences (t). Then PG is a generalised parallel grammar.

The word v is directly derivable from the word u (formally $u \Rightarrow v$) if one of the following cases is fulfilled:

- there is a nonterminal A and $(A \to r, = n) \in H$ such that the word v is obtained from u by replacing exactly n occurrences of the nonterminal A by r, formally: $u = p_1Ap_2Ap_3\ldots p_nAp_{n+1}$ and $v = p_1rp_2rp_3\ldots p_nrp_{n+1}$ for some $p_1,p_2,\ldots,p_{n+1} \in (N \cup T)^*$;

- there is a nonterminal A and $(A \to r, \geq n) \in H$ such that the word v is obtained from u by replacing exactly m occurrences of the nonterminal A by r, with some $m \geq n$;

- there is a nonterminal A and $(A \to r, \leq n) \in H$ such that the word v is obtained from u by replacing exactly m occurrences of the nonterminal A by r, with some $1 \leq m \leq n$;

- there is a nonterminal A and $(A \to r, t) \in H$ such that the word v is obtained from u by replacing all occurrences of the nonterminal A by r, formally: there is a natural number m such that $u = p_1Ap_2\ldots p_mAp_{m+1}$ and $v = p_1rp_2\ldots p_mrp_{m+1}$ where $p_1,p_2,\ldots,p_m,p_{m+1} \in ((N \setminus \{A\}) \cup T)^*$.

The reflexive and transitive closure of the direct derivation gives the derivation relation ($\Rightarrow^*$). The language generated by PG consists of all terminal words that can be derived from S:

$L(PG) = \{w \mid S \Rightarrow^* w\} \cap T^*$.

These systems can be seen as a generalisation of the Russian parallel grammars, since they can easily be simulated by using only rules of type $(A \to r, = 1)$ and $(A \to r, t)$. These two modes of rule applications are exactly the same as the two types of derivation modes at Russian parallel grammars. The simulation of k-parallel grammars cannot be done by generalised parallel grammars in arbitrary case, since in k-parallel grammars various nonterminals can be rewritten (by various rules) in the same derivation step in a parallel manner. The derivations in the generalised parallel grammars can be bounded in various ways that gives some power to these grammars.

Example (generalised parallel grammar)

Let $PG = (\{S\},\{a,b,c\},S,H)$ be a generalised parallel grammar with $H = \{(S \to SS, \leq 50), (S \to \text{a}, = 50), (S \to b, \geq 50), (S \to c, t)\}$.

Then $L(PG)$ contains every word w*$\{a,b,c\}^*$ with the following conditions:

- the number of a's is divisible by 50 in w;

- if there is a letter b in w, then there are at least 50 occurrences of b in w.

The previous example shows that the generalised parallel grammars can be used to form various constraints in a very condensed way.

We note here that there are various other formal models where one can control the derivation in various ways, e.g., programmed grammars, indexed grammars, matrix grammars, ordered grammars. New types of systems can also be defined by combining various features of these systems depending on the aim.

2.2.4. Questions and exercises

1. What are the main differences of the generative grammars and the Indian parallel grammars?

2. Give an Indian parallel grammar that generates the languages $\{a^nba^m \mid n,m$ are natural numbers$\}$ and $\{a^nba^n \mid n$ is a natural number$\}$.

3. Give an example language that is

a) context-free, but cannot be generated by an Indian parallel grammar;

b) can be generated by an Indian parallel grammar, but it is not context-free.

4. Give a Russian parallel grammar that generates the language $\{wwww|w\in\{0,1\}^*\setminus\{\lambda\}\}$. Can you generate this language by a k-parallel grammar? How? Which system(s) is(are) appropriate to generate the language $\{ww^{-1}ww \mid w\in\{0,1\}^*\setminus\{\lambda\}\}$, where w^{1} is the reverse of the word w (i.e., obtained from w by spelling it from the end to the beginning)?

5. Give a 2-parallel grammar that generates the language $\{a^nb^nc^n \mid n\geq 1\}$.

6. Give a k-paralel grammar that generates $\{a^nb^nc^nd^n \mid n\geq 1\}$. Which value of k could work?

7. Give a scattered context grammar that generates the language $\{www| w\in\{a,b\}^*\}$.

8. Give an absolute parallel grammar that generates the language $\{a^nb^nc^n| n\geq 1\}$.

9. Give a scattered context grammar that generates *a* different language if one considers it as an absolute parallel grammar.

10. Are there values k such that there is a k-parallel grammar that generates $\{a^nb^mc^nd^m| m,n\geq 1\}$ or $\{a^nb^ma^nb^m| m,n\geq 1\}$? Can these languages be generated by Russian parallel or by generalised parallel grammars?

11. What is the relation among language classes that can be generated by Indian, Russian and generalised parallel grammars? Give a language that can be generated by each of these systems. Give a language that can be generated by only two of these systems. Can you give a language that can be generated by only one of these systems? And a language that cannot be generated by any of these systems?

12. Give examples for generalised parallel grammars that differ only in the mode of the applications of some rules, but they generate different languages.

2.2.5. Literature

J. Dassow, Gh. Paun: *Regulated rewriting in formal language theory*. EATCS Monographs in Theoretical Computer Science 18, Springer, 1989.

J. Dassow, Gh. Paun, A. Salomaa: *Grammars with controlled derivations*, Chapter 3, in: **G. Rozenberg, A. Salomaa (eds.):** *Handbook of formal languages, vol. 2, Springer, pp. 101-154, 1997.*

J. Dassow: *Grammars with regulated rewriting*, in: **Z. Ésik, C. Martín-Vide, V. Mitrana (eds.):** *Recent Advances in Formal Languages and Applications, Springer 2006. pp. 249-273.*

H. Fernau: *Parallel grammars: a phenomenology*, Grammars 6 (2003), 25-87.

H. Fernau: *Parallel grammars: a short phenomenology*, in: **Z. Ésik, C. Martín-Vide, V. Mitrana (eds.):** *Recent Advances in Formal Languages and Applications, Springer 2006. pp. 175-182.*

M. Kudlek: *Languages defined by Indian parallel systems*, in: **G. Rozenberg, A. Salomaa (eds.):** *The Book of L, pages 233-244. Berlin: Springer, 1985.*

R. D. Rosebrugh, D. Wood: *Image theorems for simple matrix languages and n-parallel languages.* Mathematical Systems Theory, 8 (1974), 150-155.

2.3. Lindenmayer-systems

In this section we present grammars that are designed for parallel derivations. Aristid Lindenmayer (1925-1989) was a student of the famous grammar school of Budapest (Budapesti Evangélikus Gimnázium, also known as Fasori Gimnázium) such as some other well-known Hungarian scientists, e.g., Eugene Wigner (who received Nobel prize in physics), John von Neumann. As most of the famous Hungarian scientists he became a well-known scientist abroad: he worked in the Netherlands. As a biologist he examined various patterns and formations of growing of some algae. To describe these patterns he used formal systems in 1968. Later on these systems were used to describe fractal properties of some higher level plants. We are starting by the simplest such systems.

2.3.2.3.1. 0L and D0L-system

The most simple Lindenmayer systems are very similar to the Indian parallel grammars.

Definition: A 0L-system (interactionless Lindenmayer-system, *L*-system with zero interaction) is a pure parallel rewriting system $LS = (T,s,H)$, where T is a finite alphabet, $s\in T^*$ (axiom), the finite set of rewriting rules H contains rules of the form $a\to r$ ($a\in T$, $r\in T^*$). H must contain at least one rule for each $a\in T$ such that a is on the left-hand side of the rule (that rule can be of the form $a\to a$).

In a deterministic 0L-system, i.e., in a D0L-system, for every $a\in T$ there is exactly one rewriting rule having a at the left-hand-side. In a rewriting step (derivation) every letter of the actual word (sentential form) is rewritten according to a rule of H: the next word is obtained in this way.

The language generated by LS contains all the words (including the axiom itself) that can be derived in finitely many steps from the axiom. The generated words in a D0L system form a (finite or infinite) sequence of words.

The zero in the name of 0L- and D0L-systems means the zero interaction, i.e., the context-freeness of the rules (apart from the distinction of the terminals and nonterminals the rewriting rules are similar to the rules of a context-free grammar). However by reading the type 0L and other Lindenmayer-systems usually we read the 0 together with the other letters and pronounce it as the letter "O".

The main difference between the Indian parallel grammars and 0L-systems is that in the latter systems the full word is rewritten in every derivation step, while in the former systems only all occurrences of a chosen $a\in T$ letter. The other important difference is that while in Indian parallel grammars all occurrences of the chosen letter is rewritten by the same rewriting rule, in the (nondeterministic) 0L-system various rules can be applied for rewriting various occurrences of the same letter.

The languages that can be obtained by a given type of L-systems are called (the given type of) L-languages, so far we can use the terms 0L- and D0L-languages..

Example (a simple regular language)

Let $LS = (\{a,b\},ab,\{a\to a,a\to aa,b\to b\})$ be an 0L-system. Then there are two letters for the letter a, the second one enlarging the length of the word (by inserting an a), the first rule do nothing. One can easily prove that the generated language $L(LS) = \{a^nb \mid n\geq 1\}$.

In the previous example there could be derivation steps without changing the actual word: if every letter is replaced by itself. Now we show that the same language can be generated by a deterministic system also.

Example (the simple language in a deterministic way)

Let $LS = (\{a,b\},ab,\{a\to a,b\to ab\})$ be a D0L-system. One may easily show that the generated language $L(LS) = \{a^nb \mid n\geq 1\}$.

In the next examples we present some more complicated (non context-free) languages.

Example (words with square length)

Let $LS = (\{a,b,c\},a,\{a\to abcc,b\to bcc,c\to c\})$ be a D0L-system. Then the first derivation steps give the next words: *a*, *abcc*, *abccbcccc*, *abccbccccbcccccc*… Since the difference sequence of two neighbour squares are the sequence of odd numbers and the number of *b*'s is growing by 1 in every step, it is easy to show that the lengths of the words generated by *LS* is exactly the sequence of squares: 1, 4, 9, 16, 25, 36, 49…

In the next example the sequence of lengths of the derived words is also interesting.

Example (Fibonacci words)

Let $LS = (\{a,b\},a,\{a\to b,b\to ba\})$ be a D0L-system. Then one may easily check that the first words obtained by this system: *a*, *b*, *ba*, *bab*, *babba*, *babbabab*…

This is the language of Fibonacci words. Observe that the lengths of the words form the sequence of the Fibonacci numbers. The Fibonacci numbers are defined by the following recursive formula:

$f(0) = 1$,

$f(1) = 1$,

$f(i) = f(i\text{-}1)+f(i\text{-}2)$, for $i>1$.

The structure of the Fibonacci words are closely related to the formula used to obtain the Fibonacci numbers. Similar formula can be used for the words:

$w(0) = a$,

$w(1) = b$,

$w(i) = w(i\text{-}1)w(i\text{-}2)$, for $i>1$.

We note here that the structure of these words infers the statement about their lenghts.

Another famous example from the area of combinatorics of words, the set of Thue-Morse words that is shown in the next example.

Example (Thue-Morse words)

Let $LS = (\{a,b\},a,\{a\rightarrow ab,b\rightarrow ba\})$ be a D0L-system. One could obtain the first words generated by this system: *a*, *ab*, *abba*, *abbabaab*…

This is the language of Thue-Morse words. Observe that the length of the words is doubled in each derivation step. Several interesting properties of these words were described by A. Thue in 1906. The sequence has a self-similar property: reading only every second letter of a word the previous word is obtained. The sequence can also be defined in the following way: concatenate a word with its inverse (i.e., the word obtained by changing *a*'s to *b*'s and vice versa, e.g., the inverse of *abba* is *baab*) to get the next element of the sequence.

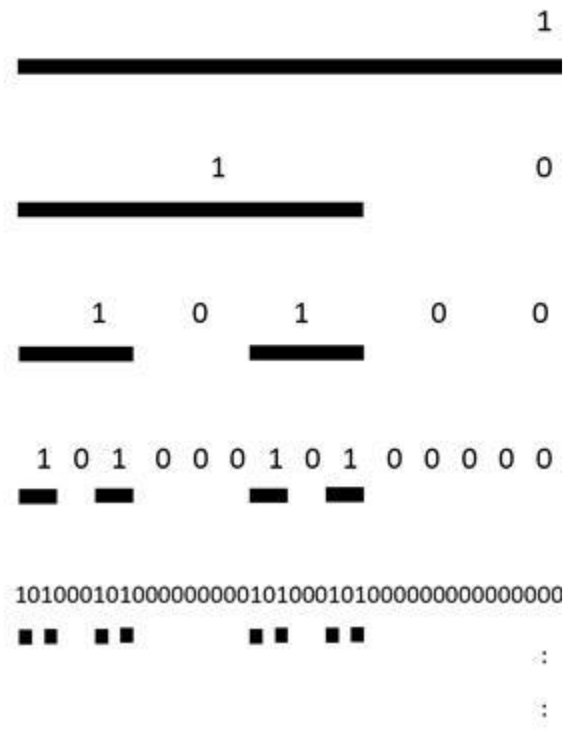
Both the Fibonacci and Thue-Morse words start with the previous element(s) of the sequence. This phenomenon gives the possibility to define the infinite word that is obtained in the limit. These infinite words are the fixed points of the previous systems, i.e., applying the rewriting step to them they won't change.

The next two examples use geometric interpretation, first fractal generation is shown.

Example (generating the Cantor-set)

The Cantor-set is one of the well-known fractals. It can be obtained (in the limit) by the following D0L-system: $(\{0,1\}, 1, \{1 \rightarrow 101, 0 \rightarrow 000\})$. The derivation goes in the following way: 1, 101, 101000101, 101000101000000000101000101… It can be more easily followed in a figure, when lines represent 1's and the absence of lines represent 0's (see Figure 2.1, that is rescaled in the usual way to represent each word in the same length). By continuing the derivation the result will be closer and closer to the Cantor-set, and after infinitely many steps the fractal of the Cantor-set is obtained.

2.1. ábra - Generation of the Cantor-set by an L-system.

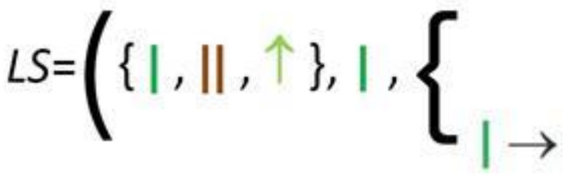


In the next example, the system itself is defined by two-dimensional figures.

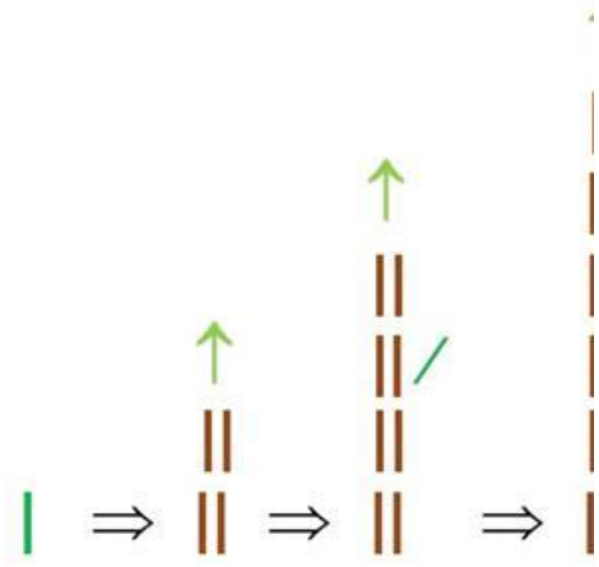
Example (simulation of the development of a plant)

Let *LS* be a D0L-system that is given in Figure 2.2. Figure 2.3 shows the first steps of the development of a plant.

2.2. ábra - D0L-system for simple simulation of the development of a plant.



2.3. ábra - The first steps of the simulation of the growing of the plant.



We have seen that some complex languages can easily and effectively be defined by 0L-systems. On the other side there are very simple languages that cannot be defined by these systems. A very simple example follows.

Example (finite language that is not 0L language)

Let $L = \{a,aaa\}$. This finite language cannot be obtained by 0L-system. To prove this fact we use indirect technique. Let us assume that *LS* is an 0L-system that generates *L*. Since the axiom must be in the generated language, one of the words must be the axiom of *LS*. There are two cases:

If the axiom is *a*, then we must generate the other word in a derivation step without obtaining any further words: there must be a rewriting rule $a \rightarrow aaa$ in the systems. But in this case the derivation can be continued obtaining the word *aaaaaaaaa* in the next step. This word is not in *L*, therefore we have got a contradiction. There can be only the other case:

If the axiom is *aaa*, then we must generate the other word, a, in a derivation step. However, this can be done, only if we have both rules $a \rightarrow \lambda$ and $a \rightarrow a$. In this case the word *aa* can also be derived from *aaa*, and since $aa \notin L$, it is a contradiction again.

We can conclude that *L* is not an 0L language, i.e., it cannot be generated by a 0L-system.

One of the most important theoretical results about D0L-systems is the decidability of their equivalence. We can state it as follows.

Lemma: Let LS_1 and LS_2 be two D0L-systems. The question if the sequence of words obtained by LS_1 is the same as the sequence obtained by LS_2 is decidable: there is an algorithm that gives the answer after finitely many steps.

Two D0L-systems can generate the same language even if the sequences of words are not the same. The order of the generated words may be different. However the next result holds.

Theorem: Let LS_1 and LS_2 be two D0L-systems. There is an algorithm that gives the answer after finitely many steps for the question if the systems LS_1 and LS_2 generate the same language.

The same result does not hold for nondeterministic systems.

Theorem: Let two arbitrary 0L-systems be given. It is not decidable (in algorithmic way) if the two systems generate the same language or not.

Note that in 0L-systems the rules of type $a \to a$ have special meaning and importance. In a nondeterministic system, if there is a rule $a \to a$ in the system for every letter $a \in T$, then sequential derivations can be simulated (using only one rule that is not in this form at every derivation step). In this way it is easy to prove that exactly the same language can be generated as it can be generated by a similar, but sequential system (without the rules of type $a \to a$).

The closure properties of the language classes obtained by 0L-systems are interesting and important.

Theorem: The language classes obtained by 0L and D0L-systems are not closed under union, concatenation, Kleene iteration, homomorphism and intersection by regular languages.

Proof: We show only the non-closure under union. Every singleton language (a language containing only one word) is a 0L, moreover a D0L language (having only rules of type $a \to a$ for every letter a). On the other side, as we could see, for instance, the language $\{a,aaa\}$ cannot be generated by any 0L (and so by any D0L) system. This shows that the language classes generated by 0L and D0L-systems, i.e., the 0L and D0L languages are not closed under the set-theoretical operation union. √

Finally, closing this section, some further decidability results are presented.

Theorem: For any D0L-system it is decidable if the generated language is regular, furthermore it is decidable if the generated language is context-free. Moreover for every context-free language it is decidable if it can be generated by a D0L-system (i.e., if it is a D0L language). Even further, it is decidable if an arbitrarily given context-free grammar is equivalent to an arbitrarily given D0L-system.

2.3.2.3.2. Extended (E0L and ED0L) systems

As we have already seen the simplest L-systems, the 0L-systems has many interesting properties. However all of them have the unpleasant property that in every step of the derivation a word of the generated language is obtained. This fact is a strong constraint. By generating a complex word (using a complex derivation) all the intermediate words are in the generated language. At the traditional generative grammars the nonterminals play give the possibility of further derivation. When a derived word contains only terminals, i.e., we have got a word of the generated language, the derivation is terminated, it cannot be continued. By the next definition we generalise 0L-system to that direction.

Definition: An E0L-system is a parallel rewriting system $ELS = (V,T,s,H)$, where V and T are finite alphabets, $T \subseteq V$, $s \in V^*$ (axiom) and the finite set of rewriting rules H consists of rules of type $a \to r$ (with $a \in V$, $r \in V^*$). (The system $LS = (V,s,H)$ is a 0L-system and it is called the base system of ELS).

In a derivation step every letter of the sentential form/actual word (the axiom at the beginning) is rewritten according to the rewriting rules generating the next sentential form/word.

The language generated by the system ELS contains all the words that have only terminal letters and can be generated by finitely many derivation steps from the axiom, i.e., $L(ELS) = L(LS) \cap T^*$.

One of the immediate consequences of the extension is that the problem with the finite languages disappears.

Theorem: The finite languages are E0L languages.

Proof: The proof of the theorem is constructive. Let $L = \{w_1,\ldots,w_n\}$ be an arbitrary finite language over an alphabet T. Then we will construct an E0L-system that generates L. Let $S \notin T$ be a new symbol. Then let

ELS=($\{S\}\cup T,T,S,S\to w_i \mid 1\le i\le n\}\cup \{a\to a \mid a\in T\}$). Then it can be easily seen that $L(ELS) = L$, since by the first step of the derivation nondeterministically the words of L are obtained, and they will not change by the further steps of the derivation. The axiom contains a nonterminal, and so it is not an element of the generated language. √

The next example shows a typical application of the extended alphabet.

Example (the language of multiple agreements $\{a^nb^nc^n \mid n\ge 1\}$ **)**

Let ELS = $(\{S,A,B,C,A',B',C',a,b,c,F\},\{a,b,c\},S,\{S\to ABC,A\to AA',B\to BB',C\to CC',A\to a,B\to b,C\to c,A'\to A',B'\to B',C'\to C',A'\to a,B'\to b,C'\to c,a\to F, b\to b,c\to c,F\to F\})$. Then all the terminating derivations start with $S\Rightarrow ABC$. Then from the letters A,B,C their primed versions can be derived parallely, and finally all the nonterminals (i.e., all the capitals including the primed ones) can be replaced by appropriate elements of T. Otherwise the letters F appear in the sentential form and they block the successful derivation. In this way, $L(ELS) = \{a^nb^nc^n \mid n\ge 1\}$.

Letter F of the example is called failure symbol and it shows that the derivation is already wrong, it cannot be continued to terminate, i.e., to get a word of the generated language. In the example we can synchronize some parts of the derivation even the derivation rules are applied nondeterministically. The next result is a formal description of this fact; and it is closely related to the normal forms used in the traditional formal language theory.

Definition: Let $ELS = (V,T,s,H)$ be an E0L-system. The system ELS is synchronized if for every $a\in T$ for all $w\in V^*$, if $a\Rightarrow^* w$ and at least one derivation step is used, then $w\notin T^*$.

A system is synchronized if all the terminal letters appear at the last step of the derivation of the given word and by continuing the derivation there are no more words of the language obtained. The generating power of the synchronized systems is the same as the generating power of these systems without this constraint, formally we have:

Theorem: Let $ELS = (V,T,s,H)$ be an E0L-system. Then there is an algorithm that produces a synchronized system ELS' such that $L(ELS) = L(ELS')$.

Proof: The proof of the theorem is constructive. Let $ELS = (V,T,s,H)$ be given. Let S be a new letter, i.e., $S\notin V$. Let $ELS'' = (V\cup\{S\},T,S,\{S\to s\}\cup H)$ be an E0L-system. Then it is obvious that the new rule can be applied only at the first step of the derivation. In other steps the old rules can be applied and thus, $L(ELS'') = L(ELS)$. Let us define a new symbol for every terminal: let $T' = \{X_a \mid a\in T\}$ such that the sets $V\cup\{S\}$ and T' are disjoint. Further, let F be a new symbol, i.e., $F\notin V\cup\{S\}\cup T'$. Let $ELS' = (V\cup\{S,F\}\cup T',T,S, \{S\to s,F\to F\}\cup\{b\to r,b\to r' \mid b\in V\setminus T$ and $b\to r\in H$ and r' can be obtained from r by changing all its terminal letters to their new symbols from $T'\}\cup\{X_a\to r,X_a\to r' \mid a\to r\in H$ and r' can be obtained from r by changing all the letters from T to their new symbol from $T'\}\cup\{a\to F \mid a\in T\})$. It can be seen that the roles of the terminals in the derivations in the new system are played by the elements of T'. This is used until the "last step" of the derivation, when all these new letters are replaced by the original terminals and in this way a word of the generated language is obtained. Further derivation steps obtain words of the form F^*. √

From the previous proof we could see that the axiom of the system ELS' can be a one-letter-long word, i.e., an element of the set $V\setminus T$. Furthermore the obtained ELS' is not only synchronized, but it uses only one failure symbol (F) and there is exactly one rule for each terminals and it is of the form $a\to F$, and there is only one rule for the failure symbol: $F\to F$.

Example (an E0L language)

Let ELS = $(\{S,A,B,C,D,a,b,F\},\{a,b\},S,\{S\to ABA,A\to AC,B\to B,B\to DB,C\to C,D\to D,A\to a,C\to a,B\to b,D\to b,a\to F,b\to F,F\to F\})$. In this system every derivation that produces a terminal word starts with $S\Rightarrow ABA$. One can see that F is the failure symbol, and therefore the terminals a and b can be introduced to the derived word only at the same time, in the "last step". In this way in the intermediate derivation steps only the rules $A\to AC,B\to B,B\to DB,C\to C,D\to D$ can be applied. Therefore the generated language is $L(ELS) = \{a^nb^ma^n \mid n\ge m\ge 1\}$.

In E0L-systems the rules of the form $A \to A$ is also usable, moreover they are usually necessary when some complex language are generated. The next example, a non E0L language, is interesting knowing the previous example.

Example (a non E0L language)

The language $\{a^n b^m a^n \mid m \geq n \geq 1\}$ cannot be generated by any E0L-system.

Now we list some theoretical results.

Theorem: For E0L-systems the emptiness and finiteness problems are decidable, i.e., for any E0L-system ELS it is decidable if the generated language is empty and/or finite.

The next result is also important.

Theorem: The word problem for E0L-systems is decidable: for the question whether a given word w can be generated by a given E0L-system ELS, there is a deterministic algorithm that gives the answer. The algorithm makes the decision on the number of steps proportional to the fourth power of the length of w.

At deterministic systems, i.e., at ED0L-systems, there is a deterministic algorithm that makes the decision on the number of steps proportional to the square of the length of w.

The closure properties are entirely changed by the extension (with letters that can be used for parts of the computation, but they cannot occur in the derived words of the language) of the systems.

Theorem: The language class generated by E0L-systems is closed under union, concatenation, Kleene-iteration, homomorphism and under intersection by regular languages.

Proof: We are proving only closure under union and concatenation. The proofs are constructive. Let $ELS' = (V',T,s',H')$ and $ELS'' = (V'',T,s'',H'')$ be given. We may assume that both ELS' and ELS'' are synchronized E0L-systems and $V' \cap V'' = T$. Let $L' = L(ELS')$ and $L'' = L(ELS'')$ over the alphabet T. Now, let S, S' and S'' be three pairwise different letters such that $\{S,S',S''\} \cap (V' \cup V'') = \{\}$. Now, let us define $ELS_u = (V' \cup V'' \cup \{S\}, T, S, H' \cup H'' \cup \{S \to s', S \to s''\})$. Then it can be easily seen that $L(ELS_u) = L' \cup L''$. Further, let us define $ELS_k = (V' \cup V'' \cup \{S,S',S''\}, T, S, H' \cup H'' \cup \{S \to S'S'', S' \to S', S' \to s', S'' \to S'', S'' \to s''\})$. Then $L(ELS_u) = L'L''$. √

Finally we place the class of E0L languages into the Chomsky-hierarchy.

Theorem: Every context-free language can be generated by an E0L-system, further every E0L language is context-sensitive.

2.3.2.3.3. Other variants

In this subsection we show some alternative variants of L-systems that allow to produce all the finite languages without extending the alphabet.

2.3.2.3.3.2.3.3.1. L-systems with finite sets of axiom (F0L)

The simplest solution to the problem with the finite languages is the following. The problem that there are finite languages that are not 0L languages is resolved by allowing more axioms in the system. F0L systems allow any finite set of axioms instead of the original case where the axiom is only one word. Formally:

Definition: An $FLS = (T,F,H)$ is a finite axiom 0L, an F0L-system, where T is a finite alphabet, $F \subset T^*$ finite set of axioms, and the finite set H contains rewriting rules of the form $a \to r$ (where $a \in T$, $r \in T^*$), as at 0L-systems. Then the language generated by the system FLS contains every word that can be derived from an element of F, i.e., it is the union of the languages generated from the elements of F: $L(FLS) = \{w \mid s \Rightarrow^* w, s \in F\}$.

For an F0L-system $FLS = (T,F,H)$, a 0L-system (T,s,H) with $s \in F$ is called a component system of FLS. In this way the generated language of a finite axiom 0L-system is the union of the generated language of its components.

Example (a finite axiom 0L-system)

Let $FLS = (\{a\},\{a,aaa\},\{a \to aa\})$ be a DF0L-system. One can easily check that $L(FLS) = \{a^n \mid n$ is a power of 2 or triple of it$\}$.

2.3.2.3.3.2.3.3.2. Adult languages

There is another way to include all finite and many more languages: to exclude the intermediate sentential forms from the generated language. It can be done by using "adult languages". The adult language of a 0L-system LS contains those words that are fixed points of the system LS, i.e., those words that cannot be changed in the system by rewriting steps. Formally:

Definition (adult language): Let $LS = (T,s,H)$ be an 0L-system. Then the adult language generated by LS is $L(A\text{-}LS) = \{w \mid s \Rightarrow^* w$, and $\{w\} = \{u \mid w \Rightarrow^* u\}\}$. When we are interested in the adult language of LS, then we call LS an A0L-system.

It can be proven that the adult language is closely connected to a so-called adult alphabet T', and this $T' \subseteq T$ adult alphabet can be computed for any arbitrarily given 0L-system. Furthermore, for every A0L-system there is an equivalent A0L-system such that it has only rules that do not change the adult alphabet, i.e., the only rule of the system for $a \in T'$ is $a \to a$.

The most important result about the adult languages is that they are closely connected to the Chomsky-hierarchy.

Theorem: The 0L-systems, as A0L-systems generate exactly the context-free languages, i.e., the class of A0L languages coincides with the class CF. For any 0L-system LS a context-free grammar G can be constructed such that $L(A\text{-}LS) = L(G)$. And, other way around, for any context-free grammar G an 0L-system LS can be constructed such that $L(A\text{-}LS) = L(G)$.

Proof: The proof is constructive, as it is stated in the theorem. For the first direction, let $LS = (T,s,H)$ be given. Without loss of generality we may assume, that the axiom s of the system is a one-letter-long word S that is not element of the adult alphabet. Further we may assume that there is only one rewriting rule for the letters of the adult alphabet: if $a \in T'$, then $a \to a$ is in H. Then, let us define $G = (T \setminus T', T', S, H \setminus \{a \to a \mid a \in T'\})$. It is clear that $L(G) = L(A\text{-}LS)$.

For the proof of the other direction, we may assume that $G = (N,T,S,H)$ is given in Chomsky normal form, and it does not contain useless nonterminals (nonterminals that cannot be reached from the startsymbol and nonterminals from that there is no terminal word that can be derived). Now, let $LS = (N \cup T, S, H \cup \{a \to a \mid a \in T\})$. It is clear that LS is an 0L-system. A derivation in LS goes in a way to build the derivation tree in a derivation of G by levels: from every nonterminal of the sentential form the derivation is continued at the same time in a parallel manner. The terminals cannot be changed, and therefore the words containing only terminals will be the fixed points of the derivations in LS. In this way, it can be seen that $L(A\text{-}LS) = L(G)$. √

2.3.2.3.4. Tabled systems

In this section some more complex, but widely used Lindenmayer systems are shown. The 0L systems can be generalised in other ways, moreover the E0L-systems can be generalised further: using the biological analogy, various sets of rewriting rules can be used in one system, mimicking the changing environment. In biology a good example is the annual rings of the trees.

Definition: Let *TLS* = (*T*,*s*,*H*) be a (pure) parallel rewriting system, where *T* is a finite alphabet, $s\in T^*$ (axiom) and *H* is a finite set of tables (sets of rewriting rules) *h* such that the triplets (*T*,*s*,*h*) are 0L-systems called component systems of *TLS*. *TLS* is a tabled L-system. If every component of *TLS* is a D0L-system, then *TLS* is a DT0L-system.

The derivation relation u⇒v is fulfilled if it there is a component in which it is true. The generated language is defined in the usual way (the words that can be derived from the axiom).

By rewriting the sentential form by various tables we can simulate the change of the environment in a tabled system.

Example (a finite language)

Let $TLS = (\{a,b,c,d\},ababa,\{\{a\to ddd,b\to b,c\to c,d\to d\},\{a\to cc,b\to b,c\to c, d\to d\}\})$. Then $L(TLS) = \{ababa,dddbdddbdddb,ccbccbcc\}$. For curiosity we note that this finite language is not a D0L, moreover it is not an 0L language.

However DT0L-systems cannot generate all the finite languages.

Example (a finite language that is not DT0L language)

Let $L = \{00,1111,11111,111111\}$. Then only 00 could be the axiom, but then all the words of the language must have even length.

In the next example we generate an infinite regular language in a nondeterministic manner.

Example (the $0^+\cup 1^+$ language)

Let $TLS = (\{0,1\},0,\{\{0\to 0,0\to 00,1\to 1\},\{0\to 1,1\to 1\}\})$. Then the first two rules of the first table can be applied to obtain any words containing only 0's. The rules of the second table allow to change all the 0's to 1's. Thus $L(TLS) = 0^+ + 1^+$.

In the systems defined in the next definition there is no problem with the finite languages, since the alphabet is extended.

Definition: The system *ETLS* = (*V*,*T*,*s*,*H*) is a rewriting system, where *V*,*T* and *s* are the same as at E0L-systems, and *H* contains finitely many sets (tables) *h* of rewriting rules such that the quadruplets (*V*,*T*,*s*,*h*) are E0L-systems, they are called the component systems of *ETLS*. Then *ETLS* is an extended tabled L-system, shortly an ET0L-system. If every component of *ETLS* is an ED0L-system, then *ETLS* is an EDT0L-system. The *T0L* (or DT0L) system *TLS* = (*V*,*s*,*H*) is called the base system of *ETLS*.

If *u*⇒*v* in a component system, then *u*⇒*v* in the whole system. Furthermore, the generated language contains all the words having only terminal letters and can be obtained from the axiom by the derivation relation.

Example (an EDT0L system)

Let $ETLS = (\{S,A,B,a,b,c\},\{a,b,c\},SSS,\{\{S\to A,A\to Sa,B\to c,a\to a,b\to b,c\to c\}, \{S\to B,A\to c,B\to Sb,a\to a,b\to b,c\to c\}\})$. Then it can be verified that $L(ETLS) = \{cwcwcw \mid w\in \{a,b\}^*\}$.

The next result shows that the power of ET0L-systems remains even some restrictions are applied.

Theorem: For every ET0L-system there is an equivalent ET0L-system that contains exactly two tables.

Furthermore, the following result is true.

Theorem: For every ET0L-system there is an equivalent ET0L-system having only rewriting rules of the form $a\to a$ for terminals.

The synchronization can also be analysed and defined for ET0L-systems.

Definition: An ET0L-system is synchronized if for every terminal letter $a \in T$ there can no terminal word be derived, i.e., for any word $w \in V^*$ if $a \Rightarrow\in {}^*w$ and at least one derivation step is applied, then $w \notin T^*$.

In synchronized systems the terminal letters must be introduced at the same time in the sentential form as the last step of a successful (terminating) derivation. Actually, the derivation is not terminating in this step, but a terminal word can be obtained. Similarly to the E0L-systems there are some results for ET0L-systems.

Theorem: Let $ETLS = (V,T,s,H)$ be an ET0L-system. Then there is an algorithm that constructs the synchronized ET0L-system $ETLS' = (V',T,s',H')$ such that $L(ELS) = L(ELS')$. Further, the axiom of $ETLS'$ is $s' \in V \backslash T$, and there is exactly one failure symbol F and $F \in V \backslash T$, $F \neq s'$. Furthermore, for each terminal $a \in T$ there is only one rule in the system, its form is $a \to F$ in every table. Similarly the only one rule for the failure symbol is $F \to F$ in every table.

Now we claim the closure properties of extended tabled L-systems without proof.

Theorem: The language classes defined by ET0L and EDT0L-systems are closed under the union, concatenation, Kleene-iteration, homomorphism and under intersection by regular languages.

The word problem for the ET0L and EDT0L-systems are decidable but with different complexity.

Theorem: The word problem for ET0L-systems is an NP-complete problem.

Proof: The proof is based on the fact that we can construct an ET0L-system that generates a well-known NP-complete problem (language). The problem of shortest common supersequence is the following: there are n words over the alphabet T, and the task is to find the shortest word (supersequence) such that every of the n words can be obtained from the supersequence by deleting some (maybe 0) letters. This problem is NP-complete over the binary alphabet, if there are at least 3 words given. The problem is still NP-complete if besides the n words a value k is also given, and the question is to decide if there exists a common supersequence of length at most k. Now we give an ET0L system that generates words coding the instances of this problem. Let $ETLS = (\{A,B,C,X,F,0,1,\$\},\{0,1,\$\},XA,\{h_1,h_2\})$, where Table 1 is $h_1 = \{A \to AA, A \to A, B \to C, C \to B, \$ \to F\} \cup \{x \to x \mid x \in \{F,X,0,1\}\}$ and Table 2 is $h_2 = \{A \to B, B \to B, B \to 0B, B \to \$, C \to C, C \to 1C, C \to \$, X \to 1X, X \to \$, \$ \to F\} \cup \{x \to x \mid x \in \{F,0,1\}\}$. Then this system generates the instances of the shortest common supersequence problem in the following form: $w = 1^{k+1}\$w_1\$w_2\$...\$w_n\$$ $(w_1,w_2,...,w_n \in \{0,1\}^*)$, where $w \in L$ if and only if the set $\{w_1,w_2,...,w_n\}$ of words has a common supersequence of length at most k. To see this fact, let us see the how the system works. The generation starts from the axiom XA and letter F is the failure symbol (the derivation cannot terminate when such a letter appears, and therefore we need to introduce all the signs \$ at the same derivation step by deriving a word of the language). By Table 1 one can obtain the word XA^n, and then by Table 2, XB^n is derived. Then, depending on the fact that B's or C's appear in the actual word, one can concatenate a letter 0 or a letter 1, respectively, by any of the number n words (to change from B's to C's and back Table 1 can be used). Then in every step the length of any word is increased, the first part (the length of the common supersequence) of the actual word is increased also by 1 letter. Finally all B's or C's (the letters that are in the actual word) and letter X is rewritten to \$. It is obvious that, in the beginning, the empty word is a supersequence of the number n empty words. Then in every step when a letter (0 or 1) is concatenated to a/some word(s) (the same letter is concatenated for every word that length is increased in this step), a supersequence can be obtained by concatenating the same letter to the supersequence obtained before this step. Since the word problem (or membership problem) for this language is NP-complete, the word problem for ET0L systems cannot be less complex. The other part of the proof that there is a nondeterministic polynomial time algorithm that solves the word problem we do not detail here. √

We note here that the original proof of the theorem (**Jan Van Leeuwen:** *The membership question for ET0L-languages is polynomially complete*, Information Processing Letters 3 (1975), 138-143. and that is copied also in some textbooks on this topic) is based on the satisfiabilty problem, however it uses *SAT* formulae with variables that are coded in a unary way, and therefore the size of the input is also very large, and thus the problem formed in this way is not NP-complete. Since there is no known correction for that way of proof, here we used the proof of Middendorf (see in the literature).

Opposite to the previous theorem, the word problem for extended deterministic tabled interactionless *L*-systems can be solved with a deterministic algorithm in polynomial time. However in the known algorithms the exponent depends on the number of nonterminals of the given EDT0L system. It is a long standing open problem (Rozenberg-Salomaa, 1980) to find such an algorithm for this problem where the exponent is fixed and independent of the used EDT0L system.

The next result is interesting and shows the limits of these systems.

Example (the language of prime-long words)

The language $\{w \mid$ the length of w is a prime number$\}$ is not an ET0L language.

The previous example shows that the language class of ET0L languages is a strict subset of the class *CS* of context-sensitive languages. However the class of ET0L languages is the largest family of languages that are obtained by interactionless L-systems.

We note here that the adult languages of DT0L and T0L-systems are usually denoted as ADT0L and AT0L (systems) languages, respectively. In the literature there are interesting results and applications of these classes also.

2.3.2.3.5. L-systems with interactions (IL)

For the sake of completeness we recall briefly that there are L-systems with interactions. The systems presented in this book so far, are interactionless, i.e., are with 0 interaction; they are context-free. At L-systems with nonzero interaction the neighbour letters of the sentential form (actual word) can interact with each other, these systems can be viewed as the context-sensitive counterpart of the previous systems.

Definition: Let $IL = (T,s,H)$, then IL is an (m,n)L-system, where T is a finite alphabet, $s\in T^*$ is the axiom of the system and the finite set of rewriting rules H contains rules of the form $(u,a,v)\to w$, where $a\in T$, $u\in T^*$, $|u|\le m$ and $v\in T^*$, $|v|\le n$. A word $w_1w_2\dots w_k$ can be directly derived from a word $a_1a_2\dots a_k$ if for every i $(1\le i\le k)$ the following statement holds for some values of m' and n' $(m'\le m, n'\le n)$:

$(a_{i-m'}a_{i-m'+1}\dots a_{i-1},a_i,a_{i+1}\dots a_{i+n'})\to w_i\in H$. The concepts of derivation and generated language can be defined in the usual way based on the direct derivation.

For historical point of view it is interesting that (0,1)L- and (1,0)L-systems together are also called 1L-systems. In these systems at most one letter context is used and it can be used only on the left-side or only at the right-side of the entirely system. The (1,1)L-systems allow to use context in both sides (even at the same time in a rule) or any of the left- and right-sides in one system. These systems are also called 2L-systems. Further, the (m,n)L-systems are called, in general, IL-systems. The $(m,0)$L-systems are called left-sided, the $(0,n)$L-systems are called right-sided IL-systems, and these special cases are also called one-sided IL-systems. When context are used in both sides, the term two-sided IL-systems is also used.

In IL-systems the subword written instead of a letter may depend on the context of the (occurrence) of the rewritten letter.

Theorem: The set of 1L-languages strictly includes the set of 0L-languages. The set of 2L-languages strictly includes the set of 1L-languages.

Proof: The first statement of the theorem can be proven by the language $\{a,aa\}$. That is not a 0L-language, but it can be generated by a 1L-system. The second statement can be proven by the language $\{a,aa\}\cup\{aaab^*\}\cup\{b^*aaa\}$. It is a 2L-language, but it is not a 1L-language. √

Further, the hierarchy of the language classes generated by IL-systems is given in the next theorems.

Theorem: Let $m,m',n,n'\geq 1$ be integers. Then the class of (m,n)L-languages strictly includes the class of (m',n')L-languages if and only if $m+n>m'+n'$. Further, the class of (m,n)L-languages is exactly the same as the class of (m',n')L-languages if and only if $m+n = m'+n'$.

Based on the previous theorem we could say that in the case of two-sided IL-systems it does not matter how large is the context in left and right side, only the whole size does matter. In the previous theorem the one-sided IL-systems are missing. The next result is about them.

Theorem: Let $m,m',n'\geq 1$. Then the class of (m',n')L-languages strictly includes the classes of $(m,0)$L- and $(0,m)$L-languages if and only if $m'+n'\geq$m. Further, if $m>m'+n'$ then the classes of (m',n')L-languages, $(m,0)$L-languages and $(0,m)$L-languages are pairwise incomparable under set theoretical inclusion.

Based on the previous theorem, we can state that by 2L-systems (i.e., by (1,1)L-systems) strictly a larger class of languages can be generated than by (0,2)L- and (2,0)L-systems. Moreover the classes of (2,0)L-languages and (0,2)-languages are incomparable under set theoretic inclusion. Figure 2.4 shows the generating power of various L-systems by a Hasse diagram.

Theorem: The language class generated by IL-systems is not closed under the union, concatenation, Kleene-iteration and intersection by regular languages.

It is possible to extend the L-systems with interactions by nonterminals, similarly as we used in other extended systems. In this way, the EIL-systems are obtained. They can be seen as E(m,n)L-systems ($m,n\geq 0$) depending on their definition. Formally:

Definition: Let $EIL = (V,T,s,H)$, where (V,s,H) is an (m,n)L-system and $T\subseteq V$, then EIL is an extended L-system with interaction. The concept of derivation can similarly be defined as at IL-systems. The generated language, as usual in extended systems, contains only those words that can be derived from the axiom and element of T^*.

At EIL-systems the one-sided and two-sided context-sensitivity gives the same generating power, moreover the following result is true.

Theorem: The class of languages generated by EIL-systems coincides with the class RE of recursively enumerable languages, moreover the class of E(0,1)L-languages (and similarly the class of E(1,0)L-languages) equals to the class RE.

About the deterministic versions it is very interesting that in one side they can generate very complex languages, on the other side some very simple languages cannot be generated by them.

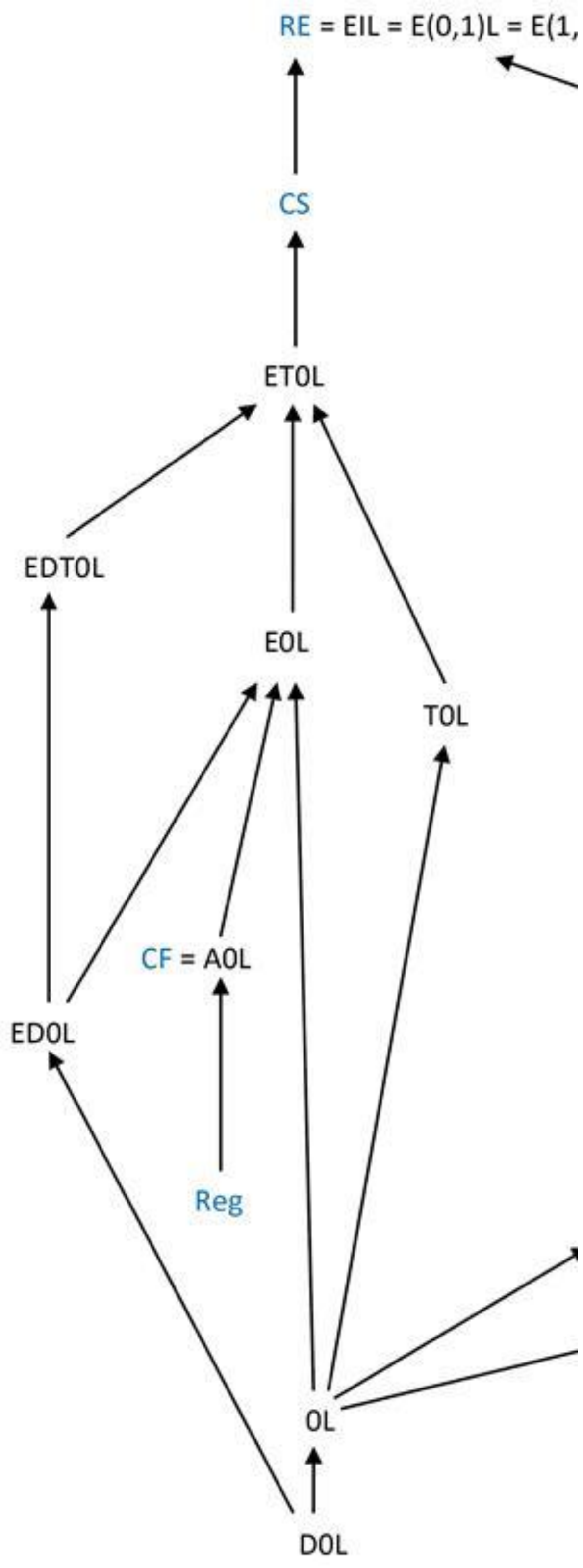
Theorem: There are non recursive D1L-langauges, i.e., there is a (0,1)L-system that is deterministic and the word problem is undecidable (there is no algorithm that decides for an arbitrary word if it can be generated by this system or not).

There are regular languages that are not ED1L-languages, e.g., the language $\{a,aa\}\cup\{bc^*b\}$ cannot be generated by any ED1L-system.

2.3.2.3.6. Further remarks

Figure 2.4 shows the hierarchy of the L-languages considered in this book. (The name of a type of systems represents the class of languages generated by those systems.) A class of languages contains the other class of languages if there is a directed path between them. The direction of the arrows is from the subset to the superset.

2.4. ábra - The classes of L-languages, their hierarchy and their relation to the Chomsky-hierarchy (blue colour) in a Hasse diagram.



The possible application of the L-systems is very wide, it is started by simulations of developments of plants (actually it was the main motivation of Lindenmayer creating these systems). These systems are appropriate for other biological simulations, but they can be effectively used in computer graphics to create fractals. IL-systems are also used to analyse sentences of natural languages.

One of the most important areas about L-systems that we have not touched so far because of the limited space is the growing function of these systems. The growing function gives the length of the obtained words by the number of derivation steps. We could see that simple D0L-systems can give examples for exponential growing, we have also seen examples for polynomial growth.

In this section of the book we analysed the L-systems mostly by the languages that can be obtained by them and we have seen their connection to traditional formal languages (string languages). However there are several variants that we could not analyse because of our limited space: there are L-systems with limited parallelism, e.g., only *k* occurrences (or if the actual word contains less, then all occurrences) of a given letter is rewritten in a derivation step, or a similar condition is applied not for each type of letters but for all of them (i.e., there are exactly *k* letters, or in case the actual word is shorter, all letters, are rewritten in a derivation step). The list of references contains literature on these and further variants of L-systems, where the generated language is obtained by a coding, or by homomorphism, etc. Further variants contain L-systems on graphs and other two- and higher-dimensional systems, some of them are closely related to the cellular automata.

2.3.7. Questions and exercises

1. What are the differences between the Indian parallel grammar and the 0L-systems?

2. Prove that the language class generated by 0L-systems is not close under concatenation. (Give two finite languages such that they are 0L-languages but their concatenation is not a 0L-language.)

3. Give examples for 0L-languages that are not D0L-languages.

4. Give some biological examples where Lindenmayer-systems can be used. What can be modelled and simulated by them? Why?

5. What are the meanings of the letters 0, I, L, D, E, T, A, F in the names of various L-systems?

6. Why are IL-systems called L-systems with interactions?

7. Define the 1L- and 2L-systems. What is the difference between one-sided and two-sided IL-systems?

8. What is the relation among the language classes obtained by IL- and EIL-systems?

9. Give various L-systems of various types that generate the language L = $\{ww \mid w \in \{1,2\}^+\}$. Can you give such type of L-systems that cannot generate the language L?

10. Show that the language $\{a^n b^m a^n \mid n,m \geq 1\}$ is an E0L-language, i.e., give an E0L-system that generates this language.

11. How can an E0L-system generate $\{a^n b^m a^n \mid n \geq m \geq 1\}$? Why does the method fail at the language $\{a^n b^m a^n \mid m \geq n \geq 1\}$?

12. Let $ELS = (\{S,a,b\},\{a,b\},S,\{S \to a, S \to aa, S \to abb, S \to aabb, a \to a, b \to bb\})$ and $ELS' = (\{A,B,a,b\},\{a,b\},A,\{A \to aB, A \to bb, B \to bB, a \to a, b \to b\})$ be two E0L-systems. Which languages are generated by them? Give an E0L-system that generates

(a) the language $L(ELS) \cup L(ELS')$;

(b) the language $L(ELS)L(ELS')$.

13. What is the generated language of the F0L-system $(\{a,b,c\},\{aaa,baba,abbabba\},\{a \to a, b \to aa, b \to c, c \to cc\})$?

14. Give an F0L-system that generates the language $\{ab^*\} \cup \{c,ccc,cac\}$.

15. Give an 0L-system such that its adult language is the Dyck language (the language of correct bracket expressions).

16. Give an A0L-system that generates the language $\{a^n b^m c^n \mid n,m > 0\}$.

17. Give a context-free grammar that generates the adult language of the 0L-system $(\{A,B,0,1\},ABA,\{A\to 0A1,A\to\lambda,B\to 1A0,0\to 0,1\to 1\})$.

18. Give an A0L-system that generates the context-free language $\{a^nb^mc^md^*a^n \mid n,m>0\}$.

19. Give an example for ED0L-systems. What is the generated language?

20. Give a D0L-system that generates the Sierpinski triangle (this fractal can be seen in Figure 2.5).

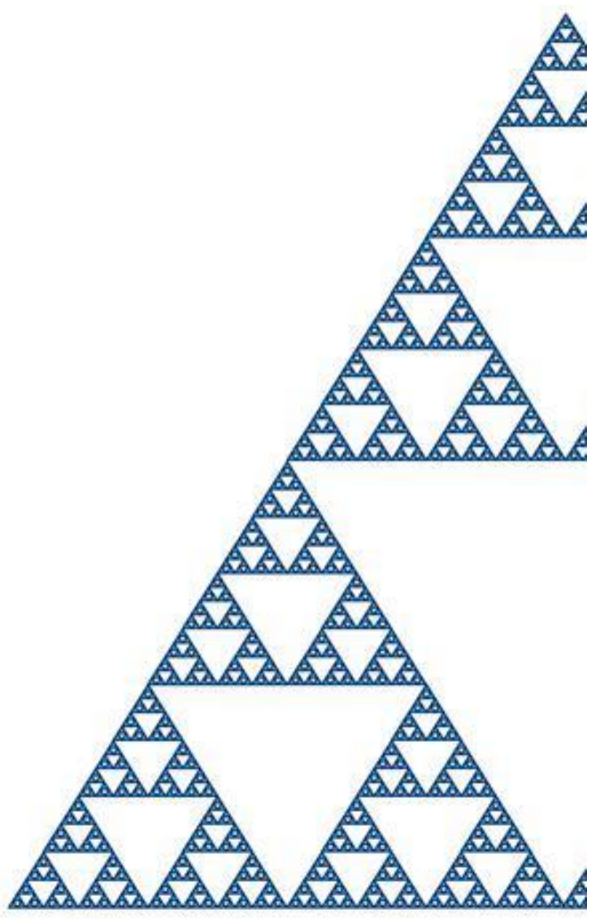
21. Generate some SAT formulae by the ET0L-system ETLS used in the proof that the word problem of ET0L-systems is NP-complete.

22. Give an example for synchronized ET0L-systems. What is the generated language?

23. Give an AT0L-system that generates an infinite language.

24. Let $IL = (\{a,b,c\},aaa,\{(a,a,a)\to b,(\lambda,b,\lambda)\to cc,(a,c,\lambda)\to a,(\lambda,a,\lambda)\to a,\ (aba,a,cc)\to bab,(\lambda,c,\lambda)\to c\})$. What is the type of this system? Show the first steps of the derivations.

2.5. ábra - Give a D0L-system that generates the fractal shown here.



2.3.8. Literature

H. Feranu: *Parallel grammars: a phenomenology*, Grammars 6 (2003), 25-87.

H. Fernau: *Parallel grammars: a short phenomenology*, in: **Z. Ésik, C. Martín-Vide, V. Mitrana (eds.):** *Recent Advances in Formal Languages and Applications, Springer 2006. pp. 175-182.*

L. Kari, G. Rozenberg, A. Salomaa: *L systems*, Chapter 5, in: **G. Rozenberg, A. Salomaa (eds.):** *Handbook of formal languages, vol. 1, Springer, pp. 253-328, 1997.*

M. Middendorf: *Supersequences, runs and CD grammar systems*, Developments in Theoretical Computer Science: Proceedings of the 7th IMYCS'92, Gordon and Breach Science Publishers, Amsterdam, pp. 101-114, 1994.

P. Prusinkiewicz, M. Hammel, J. Hanan, R. Mech: *Visual models of plant development*, Chapter 9, in: **G. Rozenberg, A. Salomaa (eds.):** *Handbook of formal languages, vol. 3, Springer, pp. 535-597, 1997.*

P. Prusinkiewicz, A. Lindenmayer: *The Algorithmic Beauty of Plants*, Springer-Verlag, 1990, 1996. (electronic version: http://algorithmicbotany.org/papers/#abop)

G. Rozenberg, A. Salomaa: *The mathematical theory of L systems*, Academic Press, New York, 1980.

G. Rozenberg, A. Salomaa (eds.): *The book of L*, Springer-Verlag, New York, 1986.

2.4. CD and PC grammar systems

There is a more direct extension of the generative grammar in which there are more grammars working simultaneously, or one after the other. In this section these types of systems will be presented.

In the classical formal languages and automata theory the languages and automata are modelling and refer to various computing devices and tools. These devices were centralised: the computation was done by a central agent (unit). In this way, by the classical theory, a language is generated by a grammar, or a language is accepted (recognized) by an automaton. In the modern computer science the distributed computing got important roles. The analysis of distributed systems in networks, in distributed databases, etc., give such concepts as parallelism, concurrency, communication. The syntactical and formal models of distributed systems are analysed in the theory of grammar systems, and thus the above concepts can be defined and analysed.

A grammar system is a group of grammars, in which the grammars are working together via a given protocol to create a language. There are several reasons to use this type of generative mechanism, such as, to model the distribution, to increase the generating power, to decrease the (descriptional) complexity. The critical part is the protocol that describes and controls the cooperation. The theory of grammar systems can be seen as a theory of formal cooperation protocols. The central problem is to describe the work of the systems using given cooperation protocols and analysis of the influence of various protocols to various properties of the system.

There are entirely two types of grammar systems: sequential and parallel ones; according to this fact we will see cooperating distributed, shortly CD and parallel communicating, shortly PC systems in the next subsections.

2.4.2.4.1. Cooperating Distributed (CD) grammar systems

The cooperating distributed systems of grammars work in a sequential manner: it is based on steps that follow each other. All the grammars of the system work on a common sentential form. At every time instant there is only one active grammar that rewrites the sentential form. The questions that which component (grammar) of the system is active in a time instant and when this active grammar becomes inactive by passing the rewritten sentential form to another component are answered by the cooperation protocol.

Examples to stopping condition, i.e., to make the actual active component inactive: (a step means an application of rewriting rule of the given grammar)

 The active component has to do exactly *k* steps.

 The active component has to do at least *k* steps.

 The active component has to do at most *k* steps.

 The active component has to do as many steps as possible.

 The active component can do any number of steps.

The language generated by the system is the set of produced terminal words. The structure of the CD grammar systems is closely related to the workmodel called "blackboard", when it is used to solve a problem in a classroom. The common sentential form is the content of the blackboard (it is the common datastructure containing the actual state of the problem to be solved). The grammars are the sources of the knowledge (agents, processing devices, procedures, students with various abilities, etc.) that help to solve the problem by modifying the content of the blackboard according to their abilities. The cooperation protocol codes the control over the sources of knowledge (e.g., the order how they should work to provide a/the solution).

Definition: A cooperative distributed system of (context-free) grammars is a system of order n ($n \geq 1$), its structure is $CD = (N,T,S,H_1,\ldots,H_n)$, where N and T are finite disjoint alphabets, $V = N \in T$, $S \in N$ and $H_1,\ldots,H_n$ are finite sets of context-free derivation rules. The elements of N are the nonterminals, the elements of T are the terminals; S is the startsymbol and $H_1,\ldots,H_n$ are the components of the system.

Remaining at the blackboard example, the components are the agents who solve the problem. The rewriting rules are the operations that the agents can do; the result of these operations is the change of the content of the blackboard, i.e., the sentential form.

- The axiom S is the initial state of the solvable problem that can be found on the blackboard.

- The terminal alphabet T contains those letters that are parts of the knowledge that can be accepted as (partial) solutions.

- The nonterminals can be interpreted as "questions" whose answers we are looking for. The questions asked by a component and rewritten by another component can be seen as questions answered by the other component. In this way the components can communicate each other by messages coded in the actual sentential form (actual state of the solution on the blackboard).

If one wants to define the components of a CD system as generative grammars, then the system CD can be written in the following form: $CD = (N,T,S,G_1,\ldots,G_n)$ where $G_i = (N,T,S,H_i)$ for every $1 \leq i \leq n$.

Definition: Let $CD = (N,T,S,H_1,\ldots,H_n)$ be a CD grammar system. Then the work of the system can be defined in the following modes:

1. For every $i \in \{1,\ldots,n\}$ the derivation of the i-th component in *-mode is denoted by $\Rightarrow_i^*$ and defined as follows p $\Rightarrow_i^*$ q if and only if p $\Rightarrow_{G_i}^*$ q.

2. For every $i \in \{1,\ldots,n\}$ the derivation of the i-th component in terminating mode is denoted by $\Rightarrow_i^t$ and defined as follows: $p \Rightarrow_i^t q$ if and only if $p \Rightarrow_{G_i}^* q$ and there is no $r \in (N \cup T)^*$ such that $q \Rightarrow_{G_i}^* r$ and $q \neq r$.

3. For every $i \in \{1,\ldots,n\}$ the derivation of the i-th component in k steps (k is a positive integer) is denoted by $\Rightarrow_i^{=l}$ and defined as follows: $p \Rightarrow_i^{=l} q$ if and only if there are $r_0,r_1,\ldots r_k \in (N \cup T)^*$ such that $p = r_0$, $q = r_k$ and for every j ($0 \leq j \leq k$-1) $r_j \Rightarrow_i r_{j+1}$ is fulfilled.

4. For every $i \in \{1,\ldots,n\}$ the derivation of the i-th component in at most k steps (k is a positive integer) is denoted by $\Rightarrow_i^{\leq k}$ and defined as follows: $p \Rightarrow_i^{=k} q$ if and only if $p \Rightarrow_i^{=j} q$ for some value $j \leq k$.

5. For every $i \in \{1,\ldots,n\}$ the derivation of the i-th component in at least k steps (k is a positive integer) is denoted by $\Rightarrow_i^{\geq k}$ and defined as follows: $p \Rightarrow_i^{\geq k} q$ if and only if $p \Rightarrow_i^{=j} q$ for some value $j \geq k$.

The derivation in *-mode means a working mode in which the agents work at the blackboard as long as they want. The derivations in t-mode give a strategy such that an agent must work at the blackboard as long as it can (maximally exploit its competence). The derivation mode = k is used if the rules of the i-th component is used in k consecutive derivation steps, this means exactly k operations for an agent at the blackboard. The derivation mode $\leq k$ refers for a timelimit, since an agent can do at most k steps. The derivation mode $\geq k$ requires a minimal competence, since at least k steps must be done by the agent. As we have seen the derivation modes requires various competences from the agents.

Definition: Let $D = \{*, t\} \cup \{ = k, \le k, \ge k \mid k \text{ is a positive integer}\}$. The generated language of a $CD = (N,T,S,H_1,\ldots,H_n)$ grammar system in the derivation mode $f \in D$ is defined as

$$L_f(CD) = \{w \in T^* \mid S \Rightarrow^f_{i_1} w_1 \Rightarrow^f_{i_2} w_2 \ldots \Rightarrow^f_{i_m} w_m = w, m \ge 1, 1 \le i_j \le n, 1 \le j \le m\}$$

In the previous definition we assign several languages to a CD system by using various stopping conditions listed in D. A component H_i of a CD system may start to work (its run is allowed) on a sentential form p, when p contains the left-hand-side of a rewriting rule of H_i. If there are more than one allowed components, than there is a nondeterministic choice. There can also be various starting conditions, e.g., a component can start its work only if certain conditions are fulfilled. In some cases an exterior control (e.g., a graph or a pushdown stack) controls the order of working components.

In the next part we highlight the generating power of the CD systems of context-free grammars: we present some well-known non context-free languages that are generated by CD systems.

Example (a typical non context-free language)

Let $CD = (\{S,A,B,C,D\},\{a,b,c\},S,\{S\to S,S\to AC\},\{A\to aBb,C\to cD\},\{B\to aAb, D\to cC\},\{A\to ab,C\to c,B\to ab,D\to c\})$. It can be easily seen that this grammar system generates the language $\{a^nb^nc^n \mid n>0\}$ in derivation mode $= 2$, ≥ 2 and t-mode. In modes $\le k$ (for any $k \ge 1$), $= 1$ and in $*$-mode the language $\{a^nb^nc^m \mid n,m>0\}$ is generated. Finally, if $k>2$, then the derivation modes $\ge k$ and $= k$ produce the empty language $\{\ \}$.

In the next example a language is generated which does not have the property of constant growth and therefore it is not semi-linear.

Example (language of words with lengths power of 2)

Let $CD = (\{S,A\},\{a\},S,\{S\to AA\},\{A\to S\},\{S\to a\})$. It can be easily seen that in t-mode CD generates the language

$$\left\{a^{2^n} \mid n \ge 0\right\}$$

Notice that we have defined the CD systems of context-free grammars, however the definition could go in analogous way, and therefore CD systems of regular/linear/context-sensitive/phrase-structure grammars can be defined. The next theorem shows why our main focus is on the context-free case.

Theorem: The CD systems of regular/linear/context-sensitive/phrase-structure grammars can generate exactly the languages of their base types, i.e, the class of regular/linear/context-sensitive/recursively enumerable languages, respectively.

The next results show that by restricting the mode of the cooperation in the CD system of context-free grammars they lose their power.

Theorem: The CD systems of (context-free grammars) in derivation modes $= 1$, ≥ 1, $*$, and $\le k$ (for any $k \ge 1$) generates exactly the class CF of context-free languages.

It is also true, moreover it is trivial, that CD systems of order 1, i.e., systems using only one (context-free) component, can generate exactly the context-free languages.

Let us analyse CD grammar systems working in t-mode.

Theorem: The (context-free) CD systems of order 1 and 2 working in t-mode generates exactly the class of context-free languages. The CD systems of order at least 3 generates more languages, they generate exactly the class of ET0L languages (systems of order 3 already generates all ET0L languages).

2.4.2.4.1.2.4.1.1. Fairness of CD systems

At CD systems it can be analysed if the system is fair, i.e., each component is done approximately the same amount of work in a derivation. To compute the fairness of the system we may count for each component how many times it was at the blackboard, i.e., how many times it worked on the sentential form according to the used derivation mode. On the other side it can also be counted how many derivation steps is made by each component during the derivation. In these measures the largest difference for any two components give the weak and strong fairness for the derivation or for the derived word. The components of the system are used in a weakly fair way, if each of them was at the blackboard at (around) the same time during the derivation. We say that the CD system works in a strong fair way, if each component is used (around) the same number of derivation steps in the derivation. In this way we can define which languages can be obtained in a fair way by CD systems. Formally we can define them in the following way.

Definition: Let CD be a CD grammar system of order n. Let $S = p_0 \Rightarrow p_1 \Rightarrow p_2 \Rightarrow \ldots \Rightarrow p_{n-1} \Rightarrow w = p_n$ ($w \in T^*$) be a derivation in some mode f.

Then let $W(i) = m$, if the i-th component became active m times during the derivation, i.e., there is exactly m values of j between 0 and n-1 such that the derivation step $p_{j-1} \Rightarrow p_j$ was not due to the i-th component (for $j = 0$ we consider it true for every component), but the derivation step $p_j \Rightarrow p_{j+1}$ is done by the i-th component. Let $dW = \max\{W(i)-W(k) \mid 1 \le i,k \le n\}$, i.e., the maximal difference between the value W of any two components in the given derivation.

Let $V(i) = m$, if the i-th component was used in exactly m derivation steps during the derivation, i.e., there are exactly m values j between 0 and n-1 such that the derivation step $p_j \Rightarrow p_{j+1}$ is done by the i-th component. Let $dV = \max\{V(i)-V(k) \mid 1 \le i,k \le n\}$, i.e., the maximal difference of the value V of two components in the derivation.

Let us fix the integer $z \ge 0$. Then the language generated by the system CD in mode f by weakly z-fair way is: $L_f(WCD_z) = \{w \in L_f(CD) \mid$ there is a derivation of w in CD in mode f such that for this derivation $dW \le z\}$.

The language generated by the system CD in mode f by strongly z-fair way is: $L_f(VCD_z) = \{w \in L_f(CD) \mid$ there is a derivation of w in CD in mode f such that for this derivation $dV \le z\}$.

Example (fair language generation)

Let $CD = (\{S,A,B,C,D\},\{a,b,c,d\},S,\{S \to aAd, D \to aAd\},\{A \to D\},\{D \to bBc,\ C \to bBc\},\{B \to C, B \to \lambda\})$. Then the generated language in modes = 1, ≥1, *, ≤k (for any $k \ge 1$) and in t-mode is $\{a^n b^m c^m d^n \mid n,m \ge 1\}$. For a given value of $z \ge 0$, for any of the previously listed derivation modes f, the generated language is $L_f(WCD_z) = L_f(VCD_z) = \{a^n b^m c^m d^n \mid n,m \ge 1, |n-m| \le z\}$. All the rules of the system are linear, but the resulted languages are not context-free, for instance, for $z = 0$ the language $\{a^n b^n c^n d^n \mid n \ge 1\}$ is obtained.

The fairness condition is an additional way to control the derivations and as we have seen the generating power of the CD systems is increasing in this way. Actually, conditions based on counting can be added to the system.

The power of CD systems can be increased in other ways too, we present such a method in the next subsubsection.

2.4.2.4.1.2.4.1.2. Hybrid CD systems

Further variants of CD grammar systems are the external and internal hybrid systems. In the external hybrid systems various components may work in various modes of computation (derivation).

Definition: An external hybrid CD grammar system of order n ($n \ge 1$) is a system $HCD = (N,T,S,(H_1,f_1),\ldots,(H_n,f_n))$, where N, T, S, $H_1,\ldots,H_n$ are the same as at CD systems of context-free grammars, and $f_i \in \{*,t\} \cup \{=k, \le k, \ge k \mid k$ positive integer$\}$ for every $1 \le i \le n$ is the mode of derivation for the i-th component. The derivation relation and the generated language are defined in the usual straightforward way.

Let us see an interesting example.

Example (external hybrid CD system)

Let $HCD = (\{S,A,B,C,X,Y\},\{a,b,c\},S,(\{S\to ABS,S\to ABX,C\to B,Y\to X\},t),(\{X\to X,B\to bC\},t),(\{X\to c,B\to c\},t),(\{X\to Y,A\to a\},=2),\{(A\to a\},=1))$. Then the derivations start by the first component in t-mode: $S(\Rightarrow ABS)\Rightarrow^*(AB)^mX$. The third component can be used if we do not want to introduce more b's since this component rewrites all B's and the X in the end of the sentential form. The derivation can be terminated by the fifth (and/or by the fourth) component by rewriting A's to a's. If the sentential form contains an X, then the second component cannot be used (having X in the sentential form this component never finishes its task, since in t-mode the rule $X\to X$ can be applied forever). In this way from the sentential form $(AB)^mX$ the derivation can be continued by two derivation steps of the fourth component (replacing any A to a, since this is the only rule in the system to rewrite an A, by simplicity we rewrite the first occurrence): $aB(AB)^{m-1}Y$. Then the second component is allowed, (since there is no X in the sentential form) and $abC(AbC)^{m-1}Y$ is obtained. The derivation can be continued by the first component (or one can use the fourth or fifth component to rewrite some of the A's to a's, but if we want to terminate the derivation we need to apply the first component to rewrite the C's and the Y). One can see that we need to apply the fourth component (and so increase the number of a's) every time before we want to use the second component (i.e., we want to introduce b's). In this way the number of a's cannot be less than the number of b's in block. Thus, the generated language is $L(HCD) = \{(ab^nc)^mc \mid 0\le n,1\le m,n\le m\}$. Note that this language is not an ET0L language.

We can see that in the previous example the second component does an occurrence check that fits very well for computations in t-mode by rules of the form $A\to A$ if there are no more rules with the same left-hand-side in this component. One needs to apply this rule until A is in the sentential form. This type of occurrence check gives an efficient control of the derivation, whether all the occurrences of the letter A were already rewritten by some previously activated components. In other cases, i.e., the usage of such component in a wrong time, results a non terminating derivation.

Theorem: For every external hybrid CD grammar system there is an equivalent external hybrid CD grammar system such that there are at most three components working in t-mode.

The next result is also interesting.

Theorem: For every external hybrid CD grammar system there is an equivalent external hybrid CD grammar system such that all the components work in $= k$ and $\ge k$ derivation mode for the same value of k.

The next result is related to the previous ones.

Theorem: For every external hybrid CD grammar system there is an equivalent external hybrid CD grammar system such that every component works in $= k$ or in t-mode of derivation.

There are internal hybrid CD systems in which the stop condition can be complex, e.g., a component must work in t-mode and in mode ≤ 5 at the same time: it must work for at most 5 derivation steps but in a way there is no applicable rule remains for the produced sentential form. As we have seen the generating power and the applicability of the CD systems are increasing by the help of hybrid systems.

2.4.2.4.2. Parallel Communicating (PC) grammar systems

In this subsection the PC systems of generative grammars will be presented. First PC systems communicating by queries are shown.

Definition: A PC grammar system of order n ($n\ge 1$) is a construct $PC = (N,K,T,(S_1,H_1),\dots,(S_n,H_n))$, where N and T are the nonterminal and terminal alphabets, $K = \{Q_1,\dots,Q_n\}$ is the set of query symbols, Q_i is the query symbol for the i-th component, (N,T and K are pairwise disjoint sets), the finite sets H_i contains rewriting rules over the alphabet ($N\cup T\cup$K) such that no elements of K occur on the left-hand-side of any of the rules. The symbols $S_i\in N$ are the startsymbols of the components for every $1\le$i$\le n$.

Two types of derivation steps can be defined for these PC systems:

Definition: Let $PC = (N,K,T,(S_1,H_1),\ldots,(S_n,H_n))$ be a PC grammar system. Let $(p_1,\ldots,p_n)$ and $(q_1,\ldots,q_n)$ be two n-tuplets such that $p_i,q_i \in (N\cup T\cup K)^*$ for every i $(1\leq i\leq n)$ and $p_1 \in T^*$. The n-tuplets $(p_1,\ldots,p_n)$ and $(q_1,\ldots,q_n)$ are in the direct derivation relation in returning and non-returning mode, i.e., $(p_1,\ldots,p_n)\Rightarrow(q_1,\ldots,q_n)$ if one of the following two cases is fulfilled:

1. For every i $(1\leq i\leq n)$, $p_i \in (N\cup T)^*$ and there is a rewriting rule in H_i such that $p_i \Rightarrow q_i$ or $p_i = q_i$, if $p_i \in T^*$ (componentwise derivation step).

2. If there is an i such that p_i contains an element of K, then for every i for which p_i contains query symbol(s), let $p_i = r_{i,1}Q_{i,1}r_{i,2}Q_{i,2}\ldots r_{i,k}Q_{i,k}r_{i,k+1}$ (for some value $k\geq 1$), where $r_{i,j} \in (N\cup T)^*$ for all j $(1\leq j\leq k+1)$. If $p_{i,j} \in (N\cup T)^*$ for every j, then let $q_i = r_{i,1}p_{i,1}r_{i,2}p_{i,2}\ldots r_{i,k}p_{i,k}r_{i,k+1}$ and in returning mode let $q_{i,j} = S_j$ for every j, in non returning mode the values $q_{i,j}$ are not changing in this step (communication step).

For every other case let $q_i = p_i$.

The n-tuple $(p_1,\ldots,p_n)$ is the configuration of the PC system. The configuration $(q_1,\ldots,q_n)$ can be directly derived from the configuration $(p_1,\ldots,p_n)$ in the following two cases:

In the first case none of the actual sentential forms (elements of the configuration) contains symbols from K, the derivation goes componentwise, in every component one derivation step is done except the sentential forms that contain only terminals, these remain without any change.

In the second case there is at least one sentential form that contains a query symbol (i.e., a query symbol appears in the configuration). Then a communication step will follow: for every occurrence of a symbol Q_i the sentential form p_i is substituted (if it does not contain any query symbol). More precisely, a sentential form containing query symbol(s) will change only if all the query symbols it contains refer for a sentential form that does not contain any query symbols. In a communication step the symbol Q_j is rewritten by p_j (the question Q_j is answered), and the j-th component starts its computation from the beginning (i.e., form its axiom S_j) in case of returning mode. In communication steps there is no derivation step in any of the components, rewriting rules are not applied and cannot be applied if any of the sentential forms contain query symbol(s). It is very important to fix if the system is used in returning mode or non returning mode.

It can be seen that communication steps have priority over derivation steps. A derivation step can be applied only if there is no way to use a communication step.

If a query cannot be answered in a step, it may be answered in the next step.

It is important that there is no rule in the system with query symbol in its left-hand-side, therefore the query symbols can be resolved only by communication steps, the derivation in the system can be continued only in this way. It is also important that there is no further derivation in the case when $p_1 \in T^*$. The reflexive and transitive closure of the direct derivation $\Rightarrow$ is the derivation relation (denoted by $\Rightarrow^*$).

In a PC system the following deadlock situation may occur:

1. There is no query symbol in the configuration, but for some sentential form $p_i \notin T^*$ (with $i\neq 1$) there is no rewriting rule to be applied.

2. A circular query happens, i.e., the symbol $Q_{i,2}$ is in $p_{i,1}$, $Q_{i,3}$ is in $p_{i,2}$ and so on, till there is a $p_{i,k}$ that contains $Q_{i,1}$. In this situation neither a communication step, nor a componentwise derivation step is applicable.

Definition: Let $PC = (N,K,T,(S_1,H_1),\ldots,(S_n,H_n))$ be a PC grammar system. Then the language generated by PC is

$L(PC) = \{w \in T^* \mid (S_1,\ldots,S_n)\Rightarrow^*(w,\ldots,p_n)\}$ for some sentential forms $p_i \in (N\cup T\cup K)^*$ $(2\leq i\leq n)$.

The derivation starts from the startsymbols and goes by componentwise derivation steps and communication steps till the first component terminates its derivation (or deadlock occurs). As we can see the first component has a special role, it is called the master component of the system.

Definition: Let $PC = (N,K,T,(S_1,H_1),\ldots,(S_n,H_n))$ be a PC grammar system. If only the first component, i.e., the master component, can introduce query symbols, then PC is a centralised PC system, otherwise PC is a non centralised PC system.

In centralised PC systems the second type of deadlock cannot occur.

A PC system is called linear, context-free, etc., if every one of its components is linear, context-free, etc.

Returning to the blackboard model of computation, the PC systems communicating by queries can be seen as problem solving methods in the following way: there is a group leader (teacher, master) who is working at the blackboard, and all the other agents (group members, students) work on some subproblems at their own places (in their exercise-notes). Opposite to the "equal" members of the CD systems, in PC systems there is a hierarchical structure. This hierarchy is more clear in the case of centralised systems: only the teacher (master) has right to ask by queries and then, he/she substitutes (places) the partial results into the central computation made by him/her (on the blackboard).

Now some examples follow to show the generating power of these parallel systems.

Example (a context-free PC system generating the copy language)

Let $PC = (\{S_1,S_2\},\{Q_1,Q_2\},\{a,b\},(S_1,\{S_1\to S_1,S_1\to Q_2Q_2\}),(S_2,\{S_2\to aS_2,\ S_2\to bS_2,S_2\to a,S_2\to b\}))$. It can be easily proven that both in returning and non returning mode PC generates the language $L(PC) = \{ww|\ w\in\{a,b\}^*, |w|>0\}$. This system is centralised.

We can see that the generating power of a PC system can also be more than the generating power of its components: in the previous example we used context-free rules to generate a non context-free language. In the next example we use regular rules.

Example (a regular PC system generating the language of multiple agreements)

Let $PC = (\{S_1,S_2,S_3\},\{Q_1,Q_2,Q_3\},\{a,b,c\},(S_1,\{S_1\to aS_1,S_1\to aaaQ_2,\ S_2\to bbQ_3,S_3\to c\}),(S_2,\{S_2\to bS_2\}),(S_3,\{S_3\to cS_3\}))$. A derivation (in non returning mode) is as follows: $(S_1,S_2,S_3)\Rightarrow^*(a^kS_1,\mathrm{b}^kS_2,\mathrm{c}^kS_3)\Rightarrow(a^{k+3}Q_2,b^{k+1}S_2,c^{k+1}S_3)\Rightarrow(a^{k+3}b^{k+1}S_2,b^{k+1}S_2,c^{k+1}S_3)\Rightarrow(a^{k+3}b^{k+3}Q_3,b^{k+2}S_2,c^{k+2}S_3)\Rightarrow(a^{k+3}b^{k+3}c^{k+2}S_3,b^{k+2}S_2, c^{k+2}S_3)\Rightarrow(a^{k+3}b^{k+3}c^{k+3},b^{k+3}S_2,c^{k+3}S_3)$. In this way it can be seen that both in returning and non returning mode of computation the generated language is $L(PC) = \{a^nb^nc^n \mid n>2\}$. This system is centralised.

The previous example showed that a regular PC grammar system of order 3 can generate a non context-free language. This is an interesting fact in view of the next result.

Theorem: The centralised and non centralised regular PC grammar systems of order at most two can generate only context-free languages.

Actually the difference between the returning and non returning mode can be used if there are more than one queries for the same component during a derivation. In these cases the synchronization, i.e., the derivation steps done by various components does matter. The next example shows this phenomenon.

Example (a non linear generated by a regular PC system)

Let $PC = (\{S_1,S_2\},\{Q_1,Q_2\},\{a,b\},(S_1,\{S_1\to aS_1,S_1\to aQ_2,S_2\to aS_1,S_2\to 1\}),\ (S_2,\{S_2\to bS_2\}))$. Then a derivation in returning mode: $(S_1,S_2)\Rightarrow^*(a^kS_1,b^kS_2)\Rightarrow(a^{k+1}Q_2,b^{k+1}S_2)\Rightarrow(a^{k+1}b^{k+1}S_2,S_2)\Rightarrow(a^{k+1}b^{k+1}aS_1, bS_2)\Rightarrow^*(a^{k+1}b^{k+1}a^mS_1,b^mS_2)\Rightarrow^*(a^{k+1}b^{k+1}a^{m+1}b^{m+1}\ldots a^{j+1}b^{j+1},bS_2)$. In this way, the generated language in returning mode is

$$L(PC)=\left\{a^{n_1}b^{n_1}\ldots a^{n_r}b^{n_r}\mid r\geq 1, n_i\geq 1, 1\leq i\leq r\right\}.$$

The example can be used in non returning mode as well and it generates another language:

$$L(PC)=\left\{a^{n_1}b^{m_1}\ldots a^{n_r}b^{m_r}\mid r\geq 1, n_i\geq 1, m_i=\sum_{i}^{k-1}n_k, 1\leq i\leq r\right\}$$

In the previous examples centralised systems were used. In the next example we present a non centralised PC system.

Example (a non centralised PC system)

Let $PC = (\{S_0,S_1,S_2\},\{Q_0,Q_1,Q_2\},\{a,b,c\},(S_0,\{S_0\to cS_0,S_0\to cQ_1,S_2\to l\}),(S_1,\{S_1\to aS_1,S_1\to aQ_2,S_2\to aS_1\}),(S_2,\{S_2\to bS_2\}))$ be a PC system used in returning mode. The last two components of the system are similar to the components of the previous example. The new master component writes letters c and uses the symbol Q_1 in queries, but it can use the result (i.e., it can continue to terminate the derivation) only if S_2 appears in it. In this way a successful derivation is possible only if the master component introduces the query symbol at the same time as the other component. In this situation, first the query by Q_2 is answered with b's from the last component and then the string described in the previous example goes to the master. In this way the language generated by the system in returning mode is

$$L(PC)=\left\{c^m a^{n_1} b^{n_1} \dots a^{n_r} b^{n_r} \mid r\geq 1, n_i\geq 1, 1\leq i\leq r, m=\sum_{r}^{k-1} n_k\right\}$$

Now we state some theoretical results.

Theorem: The generating power of non centralised regular PC systems is larger than the generating power of centralised regular PC systems (with the same parameters).

The next theorem is about the infinite hierarchies of the generated language classes.

Theorem: The centralised regular PC systems of order n+1 can generate more languages than the centralised regular PC systems with n components ($n\geq 1$). Similarly, the centralised linear PC systems with n+1 components can generate more languages than the centralised linear PC systems with n components ($n\geq 1$).

In the next part we consider context-free PC systems.

Theorem: For every context-free PC system working in non returning mode, there can be constructed a context-free PC system that generates the same language in returning mode.

The next result gives a normal form for these systems.

Theorem: For every context-free PC system that works in returning mode there is an equivalent PC system of the same type having rewriting rules only of the forms $A\to BC$, $A\to B$, $A\to a$, $A\to Q$, $A\to\lambda$, where $A,B,C\in N$ nonterminals, $a\in T$ terminal and $Q\in K$ query symbol.

The next result can be proven about the generating power of PC systems.

Theorem: Every recursively enumerable language can be generated by context-free PC system having at most 5 nonterminal symbols.

For non context-free PC systems we state the following important result.

Theorem: Every recursively enumerable language can be generated by a context-sensitive PC system of order at most 3. The context-sensitive PC systems with at most two components generate exatly the class CS of context-sensitive languages.

2.4.2.4.2.2.4.2.1. PC systems communicating by commands

The PC systems described so far used communication by queries. There are PC systems that use communication by command. The next definition describes such models.

Definition: A PC grammar system of order n ($n\geq 1$) communicating by command is a tuple $CCPC = (N,T,(S_1,H_1,R_1),\ldots,(S_n,H_n,R_n))$, where N and T, as before the finite sets of nonterminals and terminals, the triplets (S_i,H_i,R_i) are the components of the system ($1\leq i\leq n$). The symbols $S_i\in N$ are the axioms (startsymbols), the finite sets H_i contains rewriting rules over the alphabet $(N\cup T)$, and the sets $R_i\subseteq(N\cup T)^*$ are regular languages, R_i is the filter (language) for the i-th component.

Let $(p_1,\ldots,p_n)$ and $(q_1,\ldots,q_n)$ be two n-tuples, where $p_i,q_i\in(N\cup \mathrm{T})^*$ for every i ($1\leq i\leq n$). They are configurations of $CCPC$. Then $(q_1,\ldots,q_n)$ is directly derived from $(p_1,\ldots,p_n)$ in returning or non returning mode: $(p_1,\ldots,p_n)\Rightarrow(q_1,\ldots,q_n)$ if one of the following cases is fulfilled:

1. For every i ($1\leq i\leq n$), $p_i\Rightarrow^*q_i$ by rewriting rules of H_i and there is no $r_i\neq q_i$ such that $q_i\Rightarrow^*r_i$ (componentwise derivation step).

2. The communication step goes as follows: Let

$$d(p_i,j)=\begin{cases}\lambda, & \text{if } p_i\notin R_j \text{ or } i=j,\\ p_i, & \text{if } p_i\in R_j \text{ and } i\neq j,\end{cases}$$

for every $1\leq i,j\leq n$. In this way in the communication step every component i send its actual sentential form p_i to the communication channel. If p_i is not in the filter language R_j of the component j, then the component j does not receive anything from the component i (this is denoted by λ), otherwise it receives the word p_i. There is no self communication allowed, a component does not receive its own message. Let us see how the new configuration is built up from the messages. All the message that is sent by a component i: $d(i) = d(x_{i,1})d(x_{i,2})\ldots d(x_{i,n})$, the concatenation of the sent and received (by other components) messages. The received message of component i is the concatenation of all the received messages (i.e., messages that can be understood by the component: the messages of the others passing through the filter of component i). In this way, let $D(i) = d(x_{1,i})d(x_{2,i})\ldots d(x_{n,i})$, the concatenation of the messages received from the other components. Then let q_i (for every i) be defined as follows:

- let $q_i = D(i)$ if $D(i)\neq\lambda$, i.e., there were incoming (nonempty) message;

- let $q_i = p_i$ if $D(\mathrm{i}) = \lambda$ and $d(i) = \lambda$, i.e., the component was not involved to the communication step, it was not received any (nonempty) messages and noone was able to receive its message; finally

- let $q_i = S_i$ if $D(i) = \lambda$, but $d(\mathrm{i})\neq l$, i.e., the message of this component was used by other component but this component has not received any incoming (nonempty) messages that is usable.

The derivation in these systems starts by a (componentwise) derivation step starting from the configuration of start symbols. Since there is no further steps that can be made in this way (every component has done a maximal, not continuable derivation, in the same way as t-mode is used at CD systems) a communication step must turn. By definition after a communication step again a componentwise derivation step follows. (In case when only terminals were in a sentential form the given component keeps it without change during step.)

The language generated by the system $CCPC$ contains all the terminal words that can be obtained by the first component in the system by alternating use of componentwise derivation and communicating steps: $L(CCPC) = \{w\in T^* \mid (S_1,\ldots S_n)\Rightarrow^*(q_1,\ldots,q_n),\ q_1 = \mathrm{w}$, where the last step of the derivation is a componentwise derivation step$\}$.

As it can be seen from the definition, in a componentwise derivation step each component derives a sentential form such that the derivation cannot be continued by the given set of rewriting rules, in this way the number of derivation steps are not so synchronous as it was at PC systems communicating by queries. The communication in PC systems communicating by commands goes through filters. The messages sent by component i, i.e., $d(i)$ is computed only to know if any other component was able to use this message (sent to every other component). The componentwise derivation steps and the communication steps follow each other turn by turn.

Depending on the type of the grammars used in the system, we have various systems: if all components are regular, linear, context-free, etc., then the PC system is called regular, linear, context-free etc. PC grammar system comunicating by command.

Let us see the next example to see how a PC system communicating by command works.

Example (a non context-free language generated by regular PC system communicating by command)

Let $CCPC = (\{S_1,S_2,S_3,A,B,X\},\{a,b,c\},(S_1,\{S_1\to aS_1,S_1\to bS_1,S_1\to X\},\{a,b\}^*\mathrm{c}), (S_2,\{S_2\to A,X\to c\},\{a,b\}^*\mathrm{X}),(S_3,\{S_3\to B,X\to c\}\},\{a,b\}^*X))$. Then the derivation starts with the componentwise derivation: $(S_1,S_2,S_3)\Rightarrow(wX,A,B)$ for an arbitrary word $w\in\{a,b\}^*$. Then, since it is contained in the filter language of both the second and third components, but nor A, nor B is element of any filter language, the communication step leads to the configuration (S_1,wX,wX). Observe that all words in the configurations contained nonterminals so far. By the next derivation step $(w'X,wc,wc)$ is obtained, where $w'\in\{a,b\}^*$; then the communicating step leads to the configuration $(wcwc,w'X,w'X)$. By the next derivation step we obtain $(wcwc,w'c,w'c)$ and therefore the word $wcwc$ is derived. In this way it can easily be proven that the generated language is: $L(CCPC) = \{wcwc \mid w\in\{a,b\}^*\}$. This is a non context-free language and it is generated by a regular PC system of order 3 communicating by command.

About the generated language classes we state the following important results. It is interesting that the regular filter languages and the concatenation used to create the new sentential forms from the messages, the already derived parts, can increase the generating power. It is more interesting if we consider only regular components.

Theorem: The context-free PC systems of order 1 (i.e., with only 1 component) communicating by command generates exactly the class CF of context-free languages.

The context-free PC systems of order 2 (i.e., with 2 components) communicating by command generates exactly the class CS of context-sensitive languages.

The regular PC systems of order at most 2 (i.e., with only 1 or 2 components) communicating by command generates exactly the class REG of regular languages.

The regular PC systems of order 3 (i.e., with 3 components) communicating by command generates exactly the class CS of context-sensitive languages.

The PC systems communicating by command relates to the communication process that everybody sends his/her actual state (message) at the same (or nearly at the same) time to a communication channel and everybody has a filter to receive only the messages that are for him/her from the channel.

By closing this section we would like to mention that there are CD and PC systems not only for grammars but for other formal models of computation, e.g., for automata. We give some references in the literature. Also in some grammar or automata systems the active component is chosen by the help of some additional mechanism, e.g., a stack, a queue or a graph even in deterministic manner.

There are PC-systems of L-systems, ECO-grammar systems, colonies and other network systems that are closely related to the presented systems. One of the main questions is if the combined system has a larger generating power than the power of the component systems. Another important question is, how effectively these mixed systems can be used and what the descriptional complexity of the obtained systems is. In the literature there are various examples, e.g., modeling how the agents and the environment act to each other in a complex system.

2.4.3. Questions and exercises

1. What are the properties of parallel systems that can be modeled by CD and PC systems?

2. What are the main differences between the CD grammar systems and the PC grammar systems communicating by queries?

3. What is the complexity of the word (membership) problem of context-free CD grammar systems using mode t of computation?

4. How we can measure that a CD grammar system is fair? What is the difference between the weak and strong fairness?

5. What are the external and internal hybrid CD-systems? What is the main difference between these models?

6. What is the difference between the PC systems communicating by queries and PC systems communicating by commands?

7. What does the property "centralised" mean at PC grammar systems?

8. What is the difference between returning and non returning mode of computation at PC systems communicating by queries? How does this difference appear in blackboard interpretation of the model?

9. What is the filter at PC systems communicating by command and how is it used?

10. Give context-free CD system and PC grammar system that generates the language L = $\{ww| w\in \{a,b\}^*\}$ in various modes.

11. Give a context-free CD system and a context-free PC system communicating by queries that generate the non context-free language $\{a^nb^mc^nd^m \mid n,m>0\}$.

12. Is the CD system constructed for the previous example fair with some values of z?

13. Give a fair CD system that generates the language $\{a^nb^nc^n\}$ in some computation mode in fair way.

14. Give a fair CD system that generates the language $\{a^nb^mc^k \mid$ where the pairwise difference of n,m and k is at most $3\}$.

15. Give a hybrid CD system that generates a language in a "simpler way" than we can generate it without a hybrid system.

16. Give a regular PC system of order 2 communicating by queries such that it generates the language $\{a^nb^n \mid n>0\}$.

17. Give a centralised regular PC system that communicates by queries and generates the language $\{a^nb^nc^nd^n \mid n>4\}$.

18. Generate the langauge $\{a^nwa^nw \mid w\in \{b,c\}^*\}$ by

- centralised context-free PC system communicating by queries in returning mode,

- non centralised context-free PC system communicating by queries in returning mode,

- centralised context-free PC system communicating by queries in non returning mode,

- non centralised context-free PC system communicating by queries in non returning mode.

19. Generate the language $\{a^nwa^mw \mid w\in \{b,c\}^*, m>n\}$ by an appropriate PC grammar system.

20. Give a PC system communicating by command that generates the language $\{wcwcwc \mid w\in \{a,b\}^*\}$

2.4.4. Literature

H. Bordihn, B. Reichel: *On descriptions of context-free languages by CD grammar systems,* Journal of Automata, Languages and Combinatorics 7 (2002), 447-454.

H. Bordihn, M. Holzer: *On the number of active symbols in L and CD grammar systems,* Journal of Automata, Languages and Combinatorics 6 (2001), 411-426.

E. Csuhaj-Varjú, J. Dassow, J. Kelemen, Gh Paun: *Grammar systems - A grammatical approach to distribution and cooperation,* Gordon and Breach, Yverdon, 1994.

E. Csuhaj-Varjú, A. Salomaa: *Networks of Language Processors: Parallel Communicating Systems.* Current Trends in Theoretical Computer Science, Entering the 21th Century, World Scientific (2001) 791-810.

E. Csuhaj-Varjú: *Grammar Systems,* in: **C. Martín-Vide, V. Mitrana, Gh. Paun (eds.):** *Formal Languages and Applications, chapter 14, pp. 275-310., Springer, 2004.*

E. Csuhaj-Varjú, Gy. Vaszil: *On the computational completeness of context-free parallel communicating grammar systems,* Theoretical Computer Science 215 (1999), 349-358.

E. Csuhaj-Varjú, Gy. Vaszil: *On context-free parallel communicating grammar systems: synchronization, communication, and normal forms,* Theoretical Computer Science 255 (2001), 511-538.

E. Csuhaj-Varjú, Gy. Vaszil: *Parallel communicating grammar systems with bounded resources,* Theoretical Computer Science 276 (2002), 205-219.

J. Dassow, Gh. Paun, G. Rozenberg: *Grammar systems,* Chapter 4, in: **G. Rozenberg, A. Salomaa (eds.):** Handbook of Formal Languages, volume 2, pp. 155-213., Springer, 1997.

J. Dassow, V. Mitrana: *Fairness in grammar systems,* Acta Cybernetica 12 (1996), 331-345.

H. Fernau, M. Holzer, R. Freund: *Hybrid modes in cooperating distributed grammar systems: internal versus external hybridization,* Theoretical Computer Science 259 (2001), 405-426.

J. Kelemen: *Miracles, Colonies and Emergence,* in: **C. Martín-Vide, V. Mitrana, Gh. Paun (eds.):** Formal Languages and Applications, chapter 16, pp. 323-333., Springer, 2004.

A. Kelemenová: *Eco-Grammar Systems,* in: **C. Martín-Vide, V. Mitrana, Gh. Paun (eds.):** Formal Languages and Applications, chapter 15, pp. 311-322., Springer, 2004.

V. Mitrana: *Hybrid cooperating disitributed grammar systems,* Computers and Artificial Intelligence 12 (1993), 83-88.

B. Nagy: *On CD-Systems of Stateless Deterministic R(2)-Automata,* Journal of Automata, Languages and Combinatorics 16 (2011), 195–213.

B. Nagy, F. Otto: *CD-Systems of Stateless Deterministic R(1)-Automata Governed by an External Pushdown Store,* RAIRO - Theoretical Informatics and Applications, RAIRO-ITA 45 (2011), 413–448.

B. Nagy, F. Otto: *On CD-systems of stateless deterministic R-automata with window size one,* Journal of Computer and System Sciences 78 (2012), 780-806.

B. Nagy, F. Otto: *Deterministic Pushdown-CD-Systems of Stateless Deterministic R(1)-Automata,* Acta Informatica 50 (2013), 229-255.

Gh. Paun: *On the generative capacity of hybrid CD grammar systems,* J. Inform. Processing and Cybernetics EIK 30 (1994), 231-244.

Gy. Vaszil: *On simulating non-returning PC grammar systems with returning systems,* Theoretical Computer Science 209 (1998), 319-329.

3. fejezet - Parallel automata models

3.1. Recall: the traditional (sequential) finite state automata

In automata theory, traditionally, an automaton has only one reading head on a tape (only one read-write head at Turing machines). The automaton processes the input string by this head. In the next sections we investigate automata having more than one heads, and therefore they are able to process the input string in a parallel manner, and other parallel type of automata.

First of all we recall the definition of the traditional 1-head finite state automata and fix our notations.

Definition: Let $FA = (Q,T,q_0,d,F)$ be a quintuple. It is a finite (state) automaton, where Q is the finite, nonempty set of states, T is the input- (or tape-) alphabet, q_0 is the initial state, d is the transition relation, $F \subseteq Q$ is the set of final (or accepting) states. The form of transition relation d determines the type of the automaton, it can be

- non-deterministic with allowed empty word transitions: $d:Q\times(T\cup\{\lambda\})\to 2^Q$,

- non-deterministic without empty word transitions: $d:Q\times T\to 2^Q$,

- (partial) deterministic: $d:Q\times T\to Q$ (can be a partial function),

- completely defined deterministic: $d:Q\times T\to Q$ (completely defined function).

The configuration of a finite automata is an ordered pair (u,q), where $u\in T^*$ is the remaining part of the input word and $q\in Q$ is the actual state of the automaton. The initial configuration is (w,q_0), where w is the input word. A configuration (u,p) can be directly derived from the configuration (bu,q) (formally: $(bu,q)\Rightarrow(u,p)$) if $p\in d(q,b)$ (or in deterministic case $p = d(q,b)$). The reflexive and transitive closure of the direct derivation gives the notion of derivation (denoted by $\Rightarrow^*$). The sequence of direct derivations from the initial configuration is also called runs of the automaton (for a given input). The automaton accepts the word w if $(w,q_0)\Rightarrow^*(\lambda,p)$ for a state $p\in F$. The set of accepted words form the accepted (or recognised) language.

It is well known that the class of finite automata (any of the four versions defined above) recognises exactly the class of regular languages.

The theory of finite automata is a topic of several text books, we recommend, for instance, the book *Horváth Géza, Nagy Benedek: Formal Languages and Automata Theory, Typotex, 2014.*

3.2. Multihead automata

In this section we present some extensions of traditional finite automata having several reading heads. Let us start with the 2-head automata:

3.2.3.2.1. Watson-Crick finite automata

The Watson-Crick automata was originally introduced as models of automata working on DNA strands as input. The DNA strand, actually, can be considered as double string over a four-letter alphabet, where the two strings (the upper and the lower strands) uniquely determine each other. In this section we show a simple model: since the two strands uniquely determine each other, the work of the Watson-Crick automata are equivalent to the work of the 2-head automata on normal strings. Therefore in this book we show this simplified, but equivalent model.

Definition: An ordered quintuple $WK = (Q,T,q_0,d,F)$ is a (finite) WK-automaton, where Q is the finite, nonempty set of states, T is the input- (or tape-) alphabet, q_0 is the initial state, $d:Q\times(T\cup\{\lambda\})\times(T\cup\{\lambda\})\to 2^Q$ is the transition relation, and finally, $F \subseteq Q$ is the set of final- (or accepting-) states.

The configuration of a *WK*-automaton is an ordered triplet (u,v,q), where $u,v \in T^*$ are the parts of the input that is not yet read by the first and second head, respectively; $q \in Q$ is the actual state of the automaton. The initial configuration is (w,w,q_0), where w is the input word. The configuration (u,v,p) can be directly derived from the configuration (bu,cv,q) (formally: $(bu,cv,q) \Rightarrow (u,v,p)$) if $p \in d(q,b,c)$. The reflexive and transitive closure of the direct derivation gives the relation derivation (denoted by $\Rightarrow^*$). The automaton accepts a word w if $(w,w,q_0) \Rightarrow^* (\lambda,\lambda,p)$ for a state $p \in F$. The set of accepted words form the language accepted (or recognised) by the automaton.

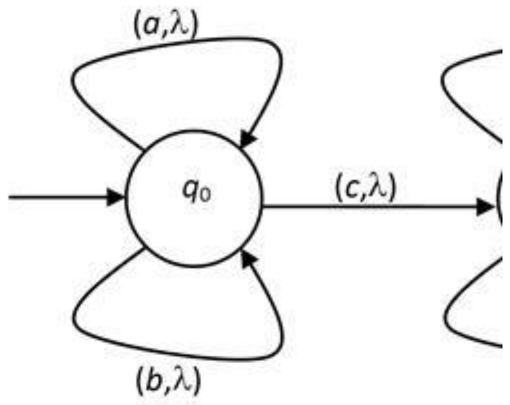
The WK automata are real extensions of the traditional finite automata: all regular languages can be accepted by WK automata. If the two heads step together (reading the same input letter) at each transition, then we simulate exactly the work of a traditional finite state automaton. We can graphically represent the traditional automata by their graph. Similarly, WK automata can also be represented by their graphs, with the following difference: at WK automata the edges of the graph are labelled by ordered pairs of letters: the letter (or the empty word) read by the first and the second head at the given transition, respectively.

The next examples show that the accepting power of WK automata is much larger than the accepting power of the 1-head finite automata.

Example (the marked copy language)

Let $WK = (\{q_0,q,p\},\{a,b,c\},q_0,d,\{q\})$, where d is defined as follows: $\{q_0\} = d(q_0,a,\lambda)$, $\{q_0\} = d(q_0,b,\lambda)$, $\{p\} = d(q_0,c,\lambda)$, $\{p\} = d(p,a,a)$, $\{p\} = d(p,b,b)$, $\{q\} = d(p,\lambda,c)$, $\{q\} = d(q,\lambda,a)$, $\{q\} = d(q,\lambda,b)$ and there is no other transition defined (i.e., $\{\} = d(q_0,\lambda,a)$, etc.). Then *WK* works as follows. The first head goes till it reads a *c* and the automaton changes its initial state to state *p* at this step. In this state both heads can read the same letter continuing the process. Then, without stepping with the first head, the second head can read a *c* and the automaton reaches state *q*. In this (final) state the first head cannot move, only the second head can read *a*'s and *b*'s to finish the input. Thus the accepted language is the non context-free language $L(WK) = \{wcw \mid w \in \{a,b\}^*\}$. Figure 3.1 graphically shows the automaton WK.

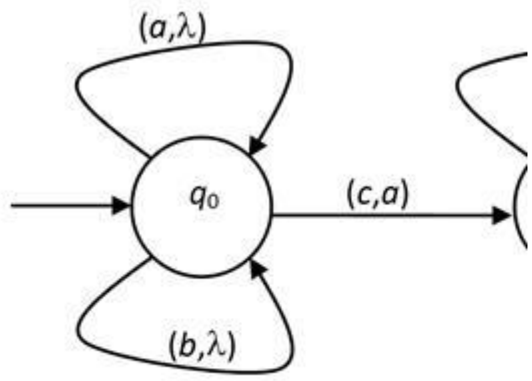
3.1. ábra - The WK-automaton accepting the language $\{wcw \mid w \in \{a,b\}^*\}$.



Example (the language of crossed dependencies)

Let $WK = (\{q_0,q,r,p\},\{a,b,c,d\},q_0,d,\{q\})$, where d is defined as follows: $\{q_0\} = d(q_0,a,\lambda)$, $\{q_0\} = d(q_0,b,\lambda)$, $\{p\} = d(q_0,c,a)$, $\{p\} = d(p,c,a)$, $\{r\} = d(p,d,b)$, $\{r\} = d(r,d,b)$, $\{q\} = d(r,\lambda,c)$, $\{q\} = d(q,\lambda,c)$, $\{q\} = d(q,\lambda,d)$ and there is no other transition. Then *WK* works as follows. The first head reads *a*'s and *b*'s while it reaches a *c*, and the automaton remains in its initial state during this process. It changes its state to *p* when the first head reads the *c* and the second head reads an *a* at the same time. In this state the two heads can read *c*'s and *a*'s, respectively. Then, the automaton changes its state to *r* when the first head reads a *d* and the second head reads a *b* at the same transition. In this state the two heads read *d*'s and *b*'s, respectively in a parallel manner. Finally, without stepping by the first head the second head can read a *c* and the automaton reaches its final state *q*. In the last part of the computation the first head cannot move, the second head can read *c*'s and *d*'s to finish the process. In this way, the accepted language is the non context-free language $L(WK) = \{a^n b^m c^n d^m \mid n,m>0\}$. Figure 3.2 shows the graph of this automaton.

3.2. ábra - The WK-automaton accepting the language $\{a_n b^m c^n b^m \mid n,m > 0\}$.



In the next part we investigate another type of Watson-Crick automata.

3.2.3.2.1.3.2.1.1. 5'→3' WK automata

In this subsection we define and analyse special Watson-Crick automata in which the heads are moving to opposite directions starting from the two extremes of the input. The DNA strands have directions due to the chemical bonds between the neighbour nucleotides. One of the ends of a strand is called the 5' end, the other is called the 3' end. The two-stranded DNA consists of two strands that have opposite biochemical directions: the 5' end of a strand is paired to the 3' end of the other strand and vice versa. According to this fact, if we consider the heads of a WK-automaton as enzymes, then they must move to the same biochemical direction along the DNA strands and thus physically, mathematically and computationally the heads must move to opposite directions. Therefore in 5'→3' WK automata the heads move in opposite directions, this is the main difference between the WK automata we already described and the 5'→3' WK automata. However this new setup allows that the work (the run on the given input) of the automata can be finished by the meeting of the heads, since at that time the whole input is processed: each position of the input was processed by either head. Let us analyse this 2-head model. Its definition differs from the definition of the previous WK-automata by the interpretation of the transition relation and by the configurations.

Definition: Let $WK' = (Q,T,q_0,d,F)$ be an ordered quintuple. It is a 5'→3' WK-automaton having the same components as the traditional WK-automata. The configurations are ordered pairs (u,q) where $u\in T^*$ is the part of the input that is not processed yet and $q\in Q$ is the actual state. The initial configuration is (w,q_0), where w is the input word (the whole input to be processed). The configurations are changing according to the transition relation during the computation of the automaton: $(aub,q)\Rightarrow(u,p)$ if $p\in d(q,a,b)$, where $u\in T^*$, $a,b\in T\cup\{\lambda\}$. In the usual way, WK' accepts an input word w if $(w,q_0)\Rightarrow^*(\lambda,\mathrm{p})$ for a state $p\in F$. The set of the words accepted by the automaton is the language $L(WK')$ accepted by the automaton.

WK' is deterministic if for every possible configuration there is at most one directly derivable configuration.

As we can see in a run of a 5'→3' WK-automaton each input letter is being read at most once, and so, the process can be really parallel. The next example shows a typical usage of these automata.

Example (the language of palindromes)

Let $WK' = (\{q,p\},\{a,b\},q,d,\{q,p\})$, where $\{q\} = d(q,a,a)$, $\{q\} = d(q,b,b)$, $\{p\} = d(q,a,\lambda)$ and $\{p\} = d(q,b,\lambda)$. There are no other transitions (i.e., the empty set is the set of possible next states). Then WK' can read the same letter by both of the heads similarly remaining in state q. In this way it can accept all the even length words that are formed by two parts such that the second part is the reverse of the first part. If the length of the input is odd, then *WK*' can accept it by reading this last (i.e., middle) letter by the first head and changing the state to p. Since there is no defined transition on this state the run stucks if this letter was not the last one and therefore the automaton does not accepts in this run. In this way the automaton accepts the language of palindromes (i.e., the words that are the same reading them from the end to the beginning) in a non-deterministic way.

Now we show that this language can be accepted by a deterministic automaton.

Example (the language of palindromes in a deterministic manner)

Let *WK*' = $(\{q,p,r\},\{a,b\},q,d,\{q,p,r\})$ be a 5'→3' WK-automaton, where $\{p\} = d(q,a,\lambda)$, $\{r\} = d(q,b,\lambda)$, $\{q\} = d(p,\lambda,a)$, $\{q\} = d(r,\lambda,b)$. Then *WK*' is a deterministic 5'→3' WK-automaton, and it accepts exactly the palindromes.

The next example shows how the efficiency of non-determinism can be used.

Example (the complement of the marked copy language)

Let us construct a non-deterministic 5'→3' WK-automaton that accepts the complement of the language $\{wcw|w\in\{a,b\}^*\}$. Let the following non-deterministic runs be defined in the automaton. Let the automaton accept every input word containing not exactly one *c*. (It is easy to do by reading the whole input and changing the state when a *c* is being read, and changing it again when another *c* is found. With such runs every word can be accepted that does not contain any *c*, or contain more than one *c*'s.) Another non-deterministic runs are accepting all the words of the form *ucv* with *u* and *v* having different lengths. (This could be done by reading one-one letter by each of the heads at the same time. If one of the two heads reaching the letter *c* before the other, then the automaton accepts the input.) Then we need to accept inputs of the form *ucv* where the length of *u* and *v* are the same, but $u\neq v$. These words can be written of the form $xeycx'e'y'$ where the lengths of *x* and *x*' are the same, and the lengths of *y* and *y*' are the same, moreover $e\neq e'$, $e,e'\in T$. Then let the accepting run of the automaton for these cases be the following. Let the first head read the subword *x*, then the automaton stores the letter *e* in its finite state memory. Then both of the heads read letters one-by-one: the first head reads the subword *y*, while the second head reads the subword *y*' (their lengths are the same, therefore the heads are finishing them at the same time). Then the first head is reading letter *c* and at the same time the second head reads *e'*. In this way the automaton checks if the stored *e* and the actually read *e*' differ. If they differ, then the automaton accepts the input after reading the subword *x'*. In opposite case this run is not an accepting run.

Theorem: The class of languages accepted by 5'→3' WK-automata coincides with the language class LIN of linear languages of the Chomsky-hierarchy.

Proof: The proof of the theorem is constructive. Let us consider the first direction: let a linear grammar $G = (N,T,S,H)$ be given in normal form. Then we construct the 5'→3' WK-automaton that accepts the language generated by *G*. Let *WK*' = $(N\cup\{q'\},T,S,d,\{q'\})$, where $q'\in N$, and *d* is defined in the following way based on the set of rules *H*:

$A\in d(B,a,b)$, if $B\to aAb\in H$, where $A,B\in N$, $a,b\in T\cup\{\lambda\}$,

$q'\in d(A,a,\lambda)$, if $A\to a\in H$, where $A\in N$, $a\in T\cup\{\lambda\}$.

It can be seen by induction on the number of steps that there is an accepting run for every terminating generation and vice versa. For the proof of the other direction, let *WK*' = (Q,T,q_0,d,F) be given. We are constructing a linear grammar *G* that generates the language that is accepted by *WK'*. Without loss of generality we may assume that the sets *Q* and *T* are disjoint (if this condition is not fulfilled, then it can be reached by renaming the elements of *Q*). Then let $G = (Q,T,q_0,H)$ be a linear grammar with the following rules:

let $p\to aqb\in H$ if $q\in d(p,a,b)$, $(p,q\in Q,\ a,b\in T\cup\{\lambda\})$;

let $p\to ab\in H$ if $q\in d(p,a,b)$, and $q\in F$.

It can be seen that the terminating derivations in *G* and the accepting runs of *WK*' coincide. The theorem is proved. √

The next result shows that the deterministic 5'→3' WK-automata have less accepting power than the non-deterministic versions have.

Theorem: The language class accepted by deterministic 5'→3' WK-automata is strictly included in the class LIN of linear languages.

The main essence of these 2-head automata is not only that they could process the input twice faster than the traditional automata due to the two heads, but they are accepting a wider class of languages, by accepting not only the class REG of regular, but the class LIN of linear languages.

For Watson-Crick automata one can allow the heads to read strings during a transition (but only finitely many transitions are defined for an automaton). It is common to consider some limited variations of WK-automata. These variations are, e.g.,

- if only one of the heads can make a move in a transition,

- if all states are final states (all-final),

- if there is only one state (stateless).

By biological motivations, in the all-final versions the automaton accepts every input that can be fully processed. The stateless models are also biologically motivated, since we can imagine enzymes reading the DNA without states. In the literature we give some references about the accepting powers of these models. One can also consider 5'→3' WK-automata in which both heads read the full input (moving the heads in opposite directions).

3.2.3.2.2. m-head automata

The 2-head automata can be extended by having more reading heads than two. The work of the heads at 5'→3' WK-automata naturally complemented each other (the input string has two extremes, therefore this model was natural). At traditional WK-automata the cooperation of the two heads is not so natural, but this model can easily be extended. The m-head finite automata are defined as follows.

Definition: The ordered tuple $MFA = (Q,T,k,q_0,<,>,d,F)$ is a k-head finite automaton, where Q is the finite, nonempty set of states, T is the input- (or tape-) alphabet, k is the number of heads, q_0 is the initial state, $<,> \notin T$ are the endmarkers marking the beginning end the end of the input, respectively, d is the transition function and $F \subseteq Q$ is the set of final (or accepting) states. By the form of the transition function d we distinguish various types of these automata:

- non-deterministic one-way k-head automata:

$d : Q\left(T\{<,>\}\right)^k 2^{Q\times\{0,1\}^k}$,

- deterministic one-way k-head automata:

$d : Q\left(T\{<,>\}\right)^k Q\times\{0,1\}^k$,

- non-deterministic two-way k-head automata:

$d : Q\left(T\{<,>\}\right)^k 2^{Q\times\{-1,0,1\}^k}$,

and

- deterministic two-way k-head automata:

$d : Q\left(T\{<,>\}\right)^k Q\times\{-1,0,1\}^k$.

The transition function does not allow to step further to any of the markers by any of the heads and therefore every head is located above a character of the part <*w*> of the tape at any time of the computation.

The configurations of these automata are ordered (k+2)-tuples of the form $(w,q,p_1,\dots,p_k)$, where the values $p_1,\dots,p_k \in \{0,\dots,|w|+1\}$ give the actual positions of the first, …, k-th heads, respectively, on the $w \in T^*$ input word, and $q \in Q$ is the actual state of the automaton. The initial configuration is $(w,q_0,1,\dots,1)$. On the tape content <*w*> 0 means the startmarker <, while the value $|w|+1$ means the endmarker >, each other value means a letter of the input word *w*.

The configuration $(w,q',r_1,\dots,r_k)$ can be directly derived from the configuration $(w,q,p_1,\dots,p_k)$ (formally: $(w,q,p_1,\dots,p_k) \Rightarrow (w,q',r_1,\dots,r_k)$) if $(q',d_1,\dots,d_k) \in d(q,a_1,\dots,a_k)$ and $r_i = p_i+d_i$ and a_i is the input letter at the position of the i-th head for each $1 \le i \le k$ (in the deterministic case $(q',d_1,\dots,d_k) = d(q,a_1,\dots,a_k)$). The reflexive and transitive closure of the direct derivation is the derivation relation (denoted by $\Rightarrow^*$). The automaton accepts the word w (in a run) if $(w,q_0,1,\dots,1) \Rightarrow^* (w,q,p_1,\dots,p_k)$ for a state $q \in F$. The set of accepted words form the accepted (or recognised) language.

As we can see from the definition, the input word is between two markers on the tape, but in case of one-way automata the initial marker < is useless. We underline that in multihead automata each of the heads reads the input letter under it even the head does not move in that transition. In this sense these automata differ from the traditional finite automata, because traditionally if the head reads a letter, then it must move through on that letter. However the traditional automata can easily be redefined without changing the class of accepted languages to work in this new way. Note that in the model used here the heads do not sense each other, the automaton usually has no information on the respective positions of the heads and it does not sense whether two or more heads are at the same position.

It is obvious by the definition that the one-way automata form a special case of the two-way automata. The deterministic versions are also special cases of the non-deterministic automata, both in the cases one-way and two-way automata, respectively.

As we have already seen at WK-automata, with two heads some non context-free languages are accepted. Now let us see what we can do with even more heads. Let us start with an example.

Example (3-head one-way deterministic finite automata)

Let $\{a^ib^jc^k | i,j,k \ge 1,\ i = j \text{ or } j = k \text{ or } k = i\}$. Then let the three heads be located at the beginning of the blocks a^i, b^j and c^k, respectively. This can be done in the following way. Let the second head move through on *a*'s till it sees a *b*. Let the third head move through on *a*'s and then through on *b*'s till it sees a *c*. Then all the three heads will read and move through on a letter: *a*, *b* and *c*, respectively, at the same time. Iterating the process, if the first head sees the first *b* at the same time as the second head sees the first *c*, then the automaton accepts (independently of the fact that the third head still sees *c* or it has already reached the endmarker >). The automaton also accepts if the first head sees the first *b* at the same time as the third head reaches the endmarker >. The automaton is reaching the accepting state if the second head sees the first *c* at the same time the third head reaches the endmarker > (independently of the fact if the first head still sees *a*'s or already reached the first *b*).

It can be proven that there is no one-way deterministic 2-head automaton that accepts the language of the previous example. In connection to the hierarchy of the classes of accepted languages depending on the number of heads we present an old and well-known result (Yao-Rivest, 1978) that says: k+1 heads are better than k.

Theorem: The class of languages accepted by one-way deterministic k+1-head automata is strictly includes the class of languages accepted by one-way deterministic k-head automata. Furthermore the class of languages accepted by one-way non-deterministic k+1-head automata is strictly includes the class of languages accepted by one-way non-deterministic k-head automata (for every $k \ge 1$).

Proof: There are languages that can be accepted by deterministic k+1-head automata, but they cannot be accepted by less heads. Let $L_n = \{w_1cw_2\dots cw_ncw_nc\dots w_2cw_1 | w_i \in \{a,b\}^*, 1 \le i \le n\}$. Then it can be seen that the language L_n can be accepted by one-way deterministic k-head automata (for some values of k) if and only if

$n \leq \binom{k}{2}$

Based on this fact the language

$L_{\binom{k+1}{2}}$.

can be accepted by a deterministic k+1-head finite automaton, but it cannot be accepted by any deterministic or non-deterministic k-head finite automata. √

Example (non-deterministic 3-head automaton)

Let $L = \{w_1cw_2\dots cw_ncv_nc\dots v_2cv_1 | n\geq 1,\ w_i,v_i \in \{a,b\}^*, 1\leq i\leq n$ and there is a value j $(1\leq j\leq n)$, such that $w_j\neq v_j\}$. Then this language can be accepted by a non-deterministic 3-head automaton that works as follows. First, let the first and the second head be positioned to the beginning of a non-deterministically chosen subword w_j. We call blocks the maximal lengths subwords of the form $\{a,b\}^*$ that can be found at the beginning of the word (between the marker < and the first c), in the end of the word (after the last c), and between two occurrences of c, i.e., the subwords w_j, v_j (for any applicable value of j). Then the first head is used to "count" how many blocks are from the current position till the endmarker > (and also to check if the number of blocks are even in the input word), while the third head also moves the same number of blocks from the beginning of the input. In this way the second and the third head are positioned to the beginning of blocks w_j and v_j, respectively. Then the two head moves together and the automaton checks if $w_j\neq v_j$. If it is fulfilled and only in this case the automaton changes its state to an accepting state. In this way for every accepted word there is a non-deterministic run of the automaton that accepts it, and other words are not accepted.

The language of the previous example, as we can see, is accepted by a non-deterministic 3-head automaton. On the other side, it can be proven that there is no deterministic multihead automaton (with any, but fixed number of heads) that accepts the same language. The next theorem (based on more complex languages than the one used in the previous example) states that the non-deterministic versions of multihead automata are more effective, i.e., the class of the accepted languages by them is wider.

Theorem: There is a language that can be accepted by one-way non-deterministic 2-head finite automata, but there is no such a value k for that there would be a one-way deterministic k-head finite automaton accepting the same language.

The language class accepted by one-way deterministic k-head (k>1) finite automata is not closed under the set theoretical operations union, intersection and complement. The language of palindromes cannot be accepted by any one-way non-deterministic k-head finite automata (with any value of k), on the other hand the complement of this language is accepted by a one-way non-deterministic two-head finite automaton. By these facts the following results can be proven about the hierarchy of the accepted language classes.

Theorem: Let $k\geq 2$ be fixed. Then the language class accepted by one-way non-deterministic k-head finite automata strictly contains the language class accepted by one-way deterministic k-head finite automata.

Consider now the two-way automata. The first result about these automata is that at the traditional 1-head case this property does not increase their accepting power.

Theorem: The language class accepted by 1-head two-way automata is exactly the class REG of regular languages, both in case of deterministic and non-deterministic automata.

Example (2-head two-way deterministic finite automaton)

Let $L = \{a^ib^jc^k | i,j,k\geq 1,\ i = j$ or $j = k$ or $k = i\}$. As we have already seen, to accept this language at least 3 heads were necessary in case of one-way deterministic automata. Now we show that 2 heads are enough in case of two-way automata. Let the second head move through on a's till it sees a b, then both heads step parallely by reading a's and b's, respectively. If the first head sees the first b at the same time as the second head sees the first c, then the second head checks if only c's are after the first c, and if so, then it accepts. In other cases, i.e., if the first head sees a *b* earlier than the second head sees the first *c*, or the second head sees a *c* before the first head sees the first *b*, then both heads are positioned to the beginning of the next block, i.e., the first head to the first *b* and the second head to the first *c*. Then they move stepwise by reading *b*'s and *c*'s, respectively, till the second head reaches the endmarker > (reading only *c*'s). If the first head reaches the first *c* in the same step, then the automaton accepts. In other case, i.e., the first head reads the first *c* in an earlier step, or it still reads *b*'s, then the first head goes back till it reads the first a in this direction (it is actually the last letter of the block a^i), and the second head is positioned to the last *c* (by stepping one in backward direction from the endmarker). Then both heads are moving backward in a parallel manner checking if the number of *a*'s and the number of *c*'s match (i.e., the first head reaches the initial marker < at the same time as the second head sees a *b* again). If so, then the automaton accepts (by entering the accepting state). If it is not fulfilled, or there were no defined transition (e.g., the input word is not of the form $a^*b^*c^*$), then the automaton stops without accepting.

In the following part some hierarchy results are shown regarding the two-way automata.

Theorem: The class of languages that is accepted by two-way deterministic k+1-head finite automata strictly includes the language class accepted by two-way deterministic k-head finite automata (for every value of $k \geq 1$).

In a similar way, the next theorem is stated for the non-deterministic case.

Theorem: The class of languages accepted by two-way non-deterministic k+1-head finite automata strictly includes the language class accepted by two-way non-deterministic k-head finite automata (for every value of $k \geq 1$).

By comparing the one-way and two-way automata we obtain the following results.

Theorem: Let $k \geq 2$ be fixed. Then, the language class accepted by two-way non-deterministic k-head finite automata strictly includes the language class that is accepted by one-way non-deterministic k-head finite automata. Similarly, the class of languages accepted by two-head deterministic k-head finite automata strictly includes the language class accepted by one-way deterministic k-head finite automata.

The following undecidability result shows that very sophisticated computations can be done by two-way finite multihead automata.

Theorem: It is algorithmically undecidable if the accepted language of an arbitrary two-way deterministic 2-head automata with 1 state is empty.

Another special case is considered in the next theorem.

Theorem: Over a 1-letter alphabet the multihead automata (one-way or two-way, deterministic or non-deterministic) accept only regular languages, thus any of the mentioned variations accept exactly the class REG of regular languages (over the unary alphabet).

For the sake of completeness, now we mention sensing versions of multihead automata. In these versions the automaton senses if two or more of its head are at the same position (reading the same letter of the input). In these cases, even there is no way to rewrite the input tape, but some heads can be used as markers and the automaton can remember to some (finite number of) positions. The next example shows how this property can be used in computations.

Example (a non-regular language over the unary alphabet is accepted by a sensing automaton)

Let *MFA* be a two-way deterministic 3-head finite automaton that works as follows.

step 1. Let step one with the first head, if it reaches the endmarker >, then the automaton accepts; otherwise the computation is continued in the following way.

step 2. Let the second head step two forward.

step 3. Then let the first and the third heads step parallely as many steps are needed to reach the second heads position with the third head.

step 4. Then the third head steps back to the beginning of the input (to the first letter).

Let the computation go in a cyclic way from step 1 till the automaton accepts or the automaton gets stuck during the computation (a head cannot step beyond the endmarker >).

In the computation the automaton, as it can easily been see, the positions of the second head follow the sequence of odd numbers, while by the help of the third head, the first head summing up these odd numbers.

Thus the accepted language is $L(MFA) = \{w \mid |w| = n^2, n \geq 1\}$.

By the previous example, sensing automata are capable of accept non-regular languages even over a unary alphabet.

The multihead automata are partially parallel systems, since the "degree of parallelism" is limited by the number of heads.

There are variations of the multihead automata in which only the first head can read the input, the other heads are blind. The blind heads recognise only the markers at the two extremes of the input. In these automata the transition function depends only on the input letter read by first head. However, the accepting power of these automata is still increasing by the number of the heads. There is a usual way to restrict the power of the (one-way/two-way) (deterministic/non-deterministic) multihead automata by having only one state of the automaton (stateless automaton). For these automata there are also interesting hierarchy results. Pushdown automata can also be extended to have several heads on the input tape. There is an infinite hierarchy of classes of languages accepted by these pushdown automata by the number of heads.

The theoretical importance of the two-way automata is essential, since at the case of multihead automata, some complexity classes can be naturally defined and some important problems concerning complexity, e.g., P-NP problem, can also be defined as simple problems on these automata.

Finally we note that the one-way 2-head automata are equivalent to the traditional WK-automata; the one-way 1-head automata are equivalent to the classical finite automata. In this way the classes of multihead finite automata can be seen extensions of the classical finite automata.

3.2.3. Questions and exercises

1. What is the main difference between the basic version of WK-automata and 5'→3' WK-automata? Give some examples that can be accepted by the first model, but it cannot be accepted by the second one.

2. How does the parallelism appear in the work of multihead automata?

3. What is the difference between one-way and two-way automata? Which class includes the other class?

4. What is the role of the markers in the two extremes of the input?

5. Give a WK automaton that accepts the language $\{a^n b^n c^n | n>0\}$. Draw also the graph of this automaton.

6. Describe the 5'→3' WK-automaton formally that was mentioned in the example accepting the complement of the marked copy language.

7. Give the WK-automaton that accepts the complement of the language of the palindromes.

8. Give a language that is accepted by some 5'→3' WK-automata, but there is no deterministic 5'→3' WK-automaton that accepts it.

9. Give a one-way deterministic 2-head finite automaton that accepts the language $\{a^i b^j c^k | i,j,k \geq 1,\ i = j \text{ or } j = k\}$.

10. Give a one-way deterministic 2-head finite automaton that accepts the language $\{abaaba^3b\ldots a^n b | n \geq 1\}$.

11. Give the one-way 3-head non-deterministic automaton that accepts $L = \{w_1cw_2\ldots cw_ncv_nc\ldots v_2cv_1 | n\geq 1, w_i,v_i\in\{a,b\}^*, 1\leq i\leq n$ and there is a value j $(1\leq j\leq n)$ such that $w_j\neq v_j\}$ in a formal way. Draw the automaton.

12. Give a multihead automaton that accepts exactly those words that have length of the Fibonacci numbers.

3.2.4. Literature

M. Chrobak: *Hierarchies of One-Way Multihead Automata Languages*, Theoretical Computer Science 48 (1986), 153-181.

Ö. Egecioglu, L. Hegedűs, B. Nagy: *Hierarchies of Stateless Multicounter 5′ → 3′ Watson-Crick Automata Languages,* Fundamenta Informaticae 110 (2011), 111-123.

R. Freund, Gh. Paun, G. Rozenberg, A. Salomaa: *Watson-Crick finite automata*, Proceedings of the Third Annual DIMACS Symposium on DNA Based Computers, Philadelphia, 1994, pp. 535-546.

L. Hegedűs, B. Nagy, Ö. Egecioglu: *Stateless Multicounter 5′ → 3′ Watson-Crick Automata: The Deterministic Case,* Natural Computing 11 (2012), 361-368.

J. Hromkovic: *One-way multihead deterministic finite automata,* Acta Informatica 19 (1983) 377-384.

O. Ibarra, S.M. Kim, L.E. Rosier: *Some Characterizations of Multihead Finite Automata*, Information and Control 67 (1985), 114-125.

O. Ibarra, J. Karhumaki, A. Okhotin: *On stateless multihead automata: hierarchies and the emptiness problem*, Theoretical Computer Science 411 (2012), 581-593.

M. Kutylowski: *Multihead One-Way Finite Automata,* Theoretical Computer Science 85 (1991), 135-153.

B. Monien: *Transformational methods and their application to complexity problems,* Acta Informatica 6 (1976) 95-108; Corrigenda: Acta lnformatica 8 (1977), 383-384.

K. Morita: *Two-Way Reversible Multi-Head Finite Automata,* Fundamenta Informaticae 110 (2011), 241-254.

B. Nagy: *On 5′→3′ sensing Watson-Crick finite automata,* DNA 13, Lecture Notes in Computer Science - LNCS 4848 (2008), 256-262.

B. Nagy: *On a hierarchy of 5′ → 3′ sensing Watson-Crick finite automata languages,* Journal of Logic and Computation (Oxford University Press) 23 (2013), 855-872.

B. Nagy: *DNS számítógépek és formális modelljeik,* Gyires Béla Tananyag Tárház, Typotex, 2014. [DNA computers and their formal models, in Hungarian]

Gh. Paun, G. Rozenberg, A. Salomaa: *DNA Computing: New Computing Paradigms,* Springer-Verlag, Berlin Heidelberg, 1998.

A. L. Rosenberg: *On multi-head finite automata,* IBM Journal of Research and Development, 10 (1966), 388-394.

A. C. Yao, R. L. Rivest: *k + 1 heads are better than k,* Journal of the ACM, 25:2 (1978), 337-340.

3.3. P automata

P automata are developed on the shoulder of a new computing paradigm, the membrane computing (that also called P systems) in 2002 by E. Csuhaj-Varjú and Gy. Vaszil. This theoretical model unifies the properties of automata with the massive parallelism of membrane computing. This concept is a biologically motivated concept, the membrane computing is motivated from the biochemical processes of the living cell. The model entirely differs from the previous models (maybe it is more related to the PC grammar systems), but we think that it is worth to mention here, in connection to parallel computing.

The membrane computing is closely connected to multiset computing, since instead of strings, multisets of objects are used for computing in this model. The next, widely used definition provides a connection between strings (words) and multisets.

Definition: Let the order of the letter of the alphabet T be fixed: $\{a_1,\dots,a_n\}$. Let $|w|_a$ denote the number of occurrences of $a \in T$ in w. Then, the mapping $T^* \rightarrow \mathbb{N}^n$

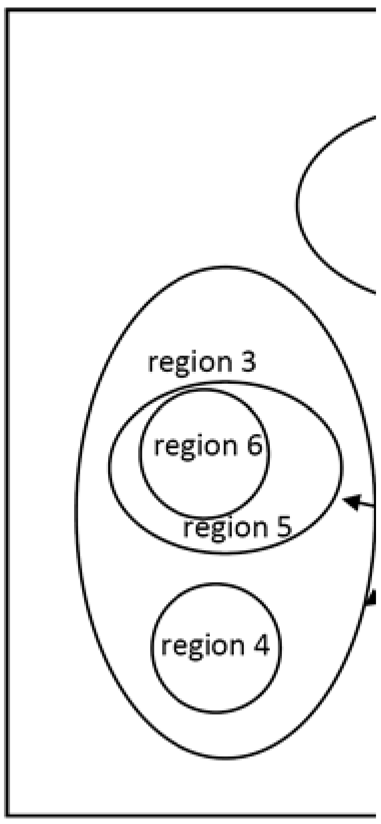
that results the vector

$(|w|_{a_1}, |w|_{a_2}, \dots, |w|_{a_n})$

for every word $w \in T^*$, is called Parikh mapping. The Parikh-set of a language contains exactly the Parikh-vectors of the words of the language.

In membrane computing the space is divided to regions (bordered by so-called membranes) in a hierarchical structure. The outer membrane is also called skin membrane. In every region there is a multiset of object symbols (letters) at any instant of time in the computation.

3.3. ábra - Tree structure of a membrane system.



The membrane structure that can be seen in Figure 3.3 can be written in the following way (by bracket expression): $(_1(_2)_2(_3(_4)_4(_5(_6)_6)_5)_3)_1$.

The P automaton is a membrane computing model in which the object symbols can move from a region to one of its neighbour regions driven by the communication rules. The input is defined by words that describe the objects entering the system from the environment. Some inputs are usually accepted by a P automaton and some inputs are not accepted. The acceptance is going by various accepting configurations. In membrane computing the multisets are usually represented by strings, in this way any words can be used that have the Parikh image of the represented multiset.

Definition: A P automaton containing n membranes is a construct $PA = (V,\mu,P_1,\ldots,P_n,c_0,F)$ where V is a finite alphabet (set of objects), μ is the tree-form membrane structure containing n membranes, c_0 is the initial configuration giving the number of various objects initially for every region, F is a computable set of accepting configurations (usually it is given by n elements such that the i-th element gives the possible multisets that can be the content of the i-th membrane in an accepting configuration), $P_1,\ldots,P_n$ are the finite sets of (communication or antiport) rules (for the region i the set of rules that can be used in this region is P_i) of the form (x out; y in) for some x,y multisets over the alphabet V. An antiport rule (x out; y in) describes a possible communication between the given region and its parent region (in the tree structure): while the object symbols of y enter to the region from its surrounding parent region, the object symbols of x leave the region through its bordering membrane and travel to its parent region (sometimes some rules are allowed that describe one-way communication, in this case x or y does not contain any object symbols, these rules are also called symport rules).

The configurations of the P automaton PA are n-tuples of multisets of objects. We say that the configuration c' can be directly derived from the configuration c (denoted by $c \Rightarrow c'$) if for every region the following is fulfilled: a maximal number of rules are applied, i.e., in configuration c there is not left such a multiset of objects (including only objects for which there is not any rule chosen to apply) such that one more rule application can be done together with the given ones in the system). This type of work of the system is called maximal parallel mode. In such a step the multiset v of objects that are entering the system through the outer membrane is the input processed in this (computation) step. (Usually any strings having Parikh-vector representing the multiset v can be considered when string representations are used.) These objects are coming from the environment that, by our assumption, contains enough number of objects of any kinds that needed for any computations. If the automaton reaches a configuration which is an element of the set of accepting configurations F, then the input that is processed until this time (i.e., until this computation step), i.e., the concatenation(s) of the strings that represent the input processed in the computation steps, is/are accepted by the given run of the system. The set of all input for which there is an accepting run of the system PA form the accepted language $L(PA)$ of the system.

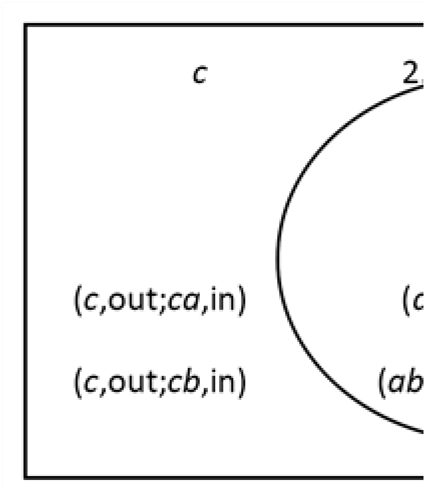
Note that by definition, the words that build up from various word representations of the same Parikh-vectors are equivalent to each other (allowing the same runs of the P automaton).

At P automata the input is not predefined. It is not given at the beginning of the computation, but it builds up during the computation by the rules of the system. The objects that represent the input are entering the system from the environment. In this way the P automata model is somewhere between the accepting and generating devices.

Example (P automaton for a non-regular language)

Let $PA = (\{a,b,c\},(_1(_2)_2)_1,\{(c,\text{out};ca,\text{in}),(c,\text{out};cb,\text{in})\},\{(ab,\text{in}),(abc,\text{in})\}, (\{c\},\{\}),(\{\},\{a,b,c\}^*))$. The system can be seen in Figure 3.4. In the initial configuration any of the rules of region 1 can be applied once to reach the second configuration. In this way a symbol a or a symbol b is also entering to region 1 with the symbol c. However there is only 1 c in the system, therefore only 1 rule can be applied for this region. If both symbols a and b are present in region 1, then rules of region 2 became active. Applying the rule (ab,in) objects are entering in region 2, but always a symbol a together with a symbol b. Applying the second rule, (abc,in) the symbol c also enters into region 2. At this point, if all the symbols a and b are entered region 2, and only in this case, the system reaches an accepting configuration. There is no applicable rule in the system, and so, the computation also halts in configurations when symbol c has entered into region 2. In accepting configurations the number of symbol a's and b's in region 2 are the same, there is only 1 c there, and region 1 is empty. Since the order of the processing the input, i.e., entering ac/ca or bc/cb can be done arbitrarily, and the important condition is only that their number match, the accepted language: $L(PA) = \{w \in \{ac,ca,bc,cb\}^* | w$ contains the same number of a's and b's and it is nonempty$\}$.

3.4. ábra - Example for P automaton.



P automata can also be used in sequential mode of computation, in these computations for every region exactly one rule must be applied in computation steps (except the regions for which there is no applicable rule). In the previous example the sequential mode leads to the same accepted language. In the next example we use more parallelism.

Example (P automaton with more parallelism)

Let *PA* = ({*a*,*b*,*c*},$(_1(_2)_2)_1$,{(*c*,out;*ca*,in),(*c*,out;*cb*,in)},{(*ab*,in),(*cc*,in)}, ({*cc*},{}),({},{*a*,*b*,*c*}*)). The structure of this system is very similar to the previous one. The main difference is that the initial configuration consists of two *c*'s. It allows to use two rules in parallel for region 1. In this way two new object symbols are entering into the system (with the two instances of *c*). In this system, the *a*'s and *b*'s are entering into section 2 together (one-one or two-two in a computation step). By using the rule (*cc*,in) of region 2, both *c*'s are entering into region 2. (This step leads to a halting configuration.) At this stage, if all *a*'s and *b*'s are entered to region 2 from region 1, and only in this case, the system reaches an accepting configuration. The accepted language is: *L*(*PA*) = {*w*∈{*acac*,*caca*,*aacc*,*ccaa*,*caac*,*acca*,*bcbc*,*cbcb*,*bbcc*,*ccbb*,*cbbc*, *bccb*,*accb*,*bcca*,*abcc*,*bacc*,*ccab*,*ccba*,*cabc*,*cbac*,*cacb*,*cbca*,*acbc*,*bcac*}*|*w* contains the same number of *a*'s and *b*'s}.

The previous examples showed how non-regular context-free languages can easily been accepted by P automata. Now we show an example for a non-context-free language using the power of maximal parallelism.

Example (P automaton for non-context-free language)

Let *PA* = ({*a*},$(_1)_1$,{(*a*,out;*aa*,in)},({*a*}),({*a*}*)). This is a very simple system containing only 1 membrane and using the unary alphabet. Deterministically two *a*'s entering into the system in the first step. In every other step the number of *a*'s that enter into the system is double of the number of a's of the previous step. All configurations are accepting configurations, therefore all possible input (that can be processed) are accepted. In this way, the accepted language is *L*(*PA*) = {a^n|n = 2^m-2 for all *m*≥1}.

P automata have several variations. Usually, it is allowed to use a mapping from the multiset of objects entering into the system in a step to the set of words of an alphabet. In this way, allowing words that are not having Parikh vector of the processed multiset, the accepting power of the systems can be increased. In some cases promoter and inhibitor multisets are used for the rules. If a rule is associated with an inhibitor multiset, then the rule can be applied only if the region does not contain the inhibitor multiset. If a rule is associated with a promoter multiset, then the rule can be applied if the region contains the promoter multiset (every element at least as many times as it can be found in the promoter multiset). Another possibility to increase the accepting power is the usage of active membranes, that allows to change the membrane structure during the computation. Both the context-sensitive and recursively enumerable language classes can be nicely characterized by P automata. For more details we recommend the literature that can be found after the section.

3.3.1. Questions and exercises

1. How does the parallelism appear at the work of P automata?

2. What are the main differences of the P automata and the multihead automata?

3. Give a P automaton that accepts the $\{w \mid w \in \{a,b\}^*, |w|_a = |w|_b$ and in every prefix of w there is at least as many a's than b's$\}$ Dyck language. (It is the language of correct bracket-expressions.)

4. Give a *P*-automaton that accepts a language that can be described by the powers of 3.

3.3.2. Literature

E. Csuhaj-Varjú, M. Oswald, Gy. Vaszil: *P automata,* in: **Gh. Paun, G. Rozenberg, A. Salomaa (eds.):** *Handbook of Membrane Computing, chapter 6, Oxford University Press, 2010, pp. 144-167.*

E. Csuhaj-Varjú, Gy. Vaszil: *P Automata or Purely Communicating Accepting P Systems,* International Workshop on Membrane Computing (WMC-CdeA), LNCS 2597 (2003), 219-233.

E. Csuhaj-Varjú, Gy. Vaszil: *P automata,* Scholarpedia 5 (2010), 9344.

B. Nagy: *Új számítási paradigmák,* Gyires Béla Tananyag Tárház, Typotex, 2013. [New computing paradigms, in Hungarian]

Gh. Paun: *Membrane Computing: An Introduction,* Springer, Berlin, 2002.

3.4. Cellular automata

In this section we present an old model of automata that is essentially parallel and it has some similarity with the array processors.

The introduction of cellular automata dates back before the World War II due to the pioneer work of John von Neumann and E. F. Codd. J. von Neumann originally used the name self-reproducing automata for the model, but later on the name cellular automata has been widespread. The words cell and cellular show the biological motivation. The basic problem that J. von Neumann solved by this model was the following: what formal organization is needed for an automata-model that can reproduce itself.

A cellular automaton can be seen as a parallel computing system, and also as a dynamic system. Therefore they can be applied in various fields in various ways: they can describe algorithms, recognise patterns and languages and also they can be applied in complexity analysis. On the other side, cellular automata can be seen as natural models of physical, chemical, biological and/or economical systems. In these models the aim is not only to give a description of the system by cellular automata, but also to predict some phenomena of the system.

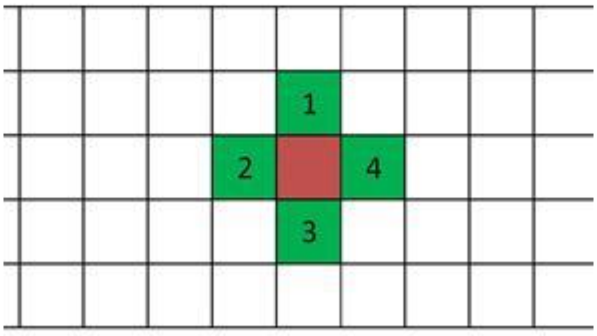
These abstract machines (or automata) can be imagined as systems formed by connected uniform, simple "cells": the architecture or universe is an infinite, regular discrete network; and in its every node there is a finite state machine (automaton) that is similar to the others. Usually regular grids are used and the dimension of the architecture defines the elements of the system: In one and two dimensions the pixels are usually squares since the square grid is used most frequently, while in three dimensions voxels are used. These elements of the universe together with the automata assigned to them, e.g. the squares of the infinite square grid and the automata "on them", are called cells. The whole machine, the cellular automata work in a discrete time: every cell communicates with its neighbours (the cells that are connected to it). This local, deterministic, (geometrically) uniform and synchronous (happening at the same time) communication determines the evolution of the machine, i.e., the state of the cells of the system at the next time instant.

Definition: A k-dimensional cellular automaton is an ordered quadruple $CA = (Z,Q,N,d)$, where Z is the (a) k-dimensional digital space, Q is a finite set of states, N is an ordered set, called neighbourhood (its elements are vectors, that give the respective positions of the neighbour points of the given point in the digital grid), and $d{:}Q^{n+1} \rightarrow Q$ is the local transition function.

By the previous definition the state of a cell depends (only) on its own and its neighbours previous states. We note here that in some cases only the states of the neighbours are used to determine the new state and it does not depend on its own previous state. Our general description allows these cases, moreover we allow such functions that the new state do not depend on the previous states at all.

Traditionally the cellular automata were investigated and analysed in two-dimensional space, especially on the square grid. Most of the results we know are about these automata. The basic von Neumann- and Moore-neighbourhood can be seen in Figure 3.5. These widely used neighbourhoods use the closest cells as neighbours.

3.5. ábra - Von Neumann- and Moore-neighbourhood (green and blue colours, respectively), the numbers show a possible order of the neighbourhood set.



These neighbourhoods are used in one- and more-dimensional spaces too. Generally, in dimension k, the points are addressed by vectors with integer coordinates, formally: the cells $x(x_1,x_2,\ldots,x_k)$ and $y(y_1,y_2,\ldots,y_k)$ are Moore-neighbours if $|y_i\text{-}x_i|\leq 1$ for every $1\leq i\leq k$. The cells represented by x and y are von Neumann-neighbours if

$$\sum_{k}^{i-1} |y_i - x_i| = 1,$$

i.e., their vectors differ in exactly one coordinate and the difference in this coordinate is ±1. It is easy to see that in the one-dimensional space the two definitions lead to the same neighbourhood: every cell has two neighbours, the left and the right neighbours. In two dimensions, as we could see on Figure 3.5, the von-Neumann neighbourhood contains the 4 closest neighbours, while the Moore-neighbourhood contains 4 more diagonal neighbours and has together 8 neighbour cells. In three dimensions they refer for 6 and 26 neighbours, respectively. Generally, in the k-dimensional (hypercubic-)grid $2k$ and 3^k-1 neighbour cells are defined in this way, respectively. By the definition of the cellular automaton, we give the neighbour cells in a fixed order (e.g., the ones showed in Figure 3.5). In the general case the neighbourhood can be characterized by the Euclidean distance r of the farthest neighbour and the number $|N|$ of the neighbour cells.

The work of the automaton, i.e., its dynamic can be described using the next concepts.

Definition: Let a k-dimensional cellular automaton, $CA = (Z,Q,N,d)$ be given. The configuration of the system at a time instant t, i.e., the global state of CA can be described by the function $c_t{:}Z\to Q$, that gives the actual state for every cell. The evolution of the cellular automaton is a sequence of the configurations $c_0,c_1,\ldots,c_t,c_{t+1},\ldots$ such that the initial configuration c_0 (it is given at time instant t = 0), and every other configuration can be given from the previous configuration by the direct derivation relation: $c_{i-1}\Rightarrow c_i$ if and only if the state of every cell x in c_i is given by the transition function d based on the states of x and its neighbours at the time istant i-1 (for all i>0). The relation ⇒ is called a computation step of CA. Based on this we define the global function D on configurations, that gives the next configuration: $D(c_i) = c_{i+1}$ (for every $i\geq 0$). The function D is usualy called the next-state-mapping.

During a computation special configurations may occur. Now we give some examples. So far we used infinite space and universe. The next definition is about finiteness in the infinite universe.

Definition: Let the cellular automaton $CA = (Z,Q,N,d)$ be given. Let $p\in Q$ be a resting state, i.e., $d(p,\ldots,p) = p$ (if a cell and all of its neighbours are in state p, then the cell remains in the same state in the next time instant also). Let the actual configuration c of CA be given. If the number of cells that have states different from p are finite in configuration c, then we say that c is a finite configuration of CA.

The finiteness plays an important role in computations with cellular automata, in pattern- and language-recognition, since in these cases the data is finite in the configurations.

The next definition may also mean that the system has a finite amount of data.

Definition: Let *CA* = (*Z*,*Q*,*N*,*d*) be a *k*-dimensional cellular automaton and let *c* be one of its configurations. Then, if there is a *k*-dimensional vector *y* such that for every cell (let it be addressed by vector *x*) the state of the cell (addressed by *x*) is the same as the state of the cell addressed by the vector *x*+*y* in configuration *c*, then we say that c is a periodic configuration (in space) of *CA*.

It is obvious that except the special case when the configuration *c* contains only the resting state *p*, the periodic configurations (in space) are not finite. However the data they contain are finite.

Definition: Let the cellular automaton *CA* = (*Z*,*Q*,*N*,*d*) and its configuration c_t be given. Then, if there is a value s such that $c_t = c_{t+s}$, i.e., after time s the configuration c_t is repeated, then we say that the work of *CA* for the configuration c_t is periodic (in time).

Obviously, if $c_t = c_{t+s}$, then so $c_t = c_{t+ms}$ for every natural number *m*. It can be proven that every periodic configuration (in space) implies that the work of the automaton is periodic (in time) on that configuration. However the converse does not necessarily hold: there are cellular automata whose work on some configurations are periodic (in time), but those configurations are not periodic (in space).

Definition: A configuration *c* of a cellular automaton *CA* = (*Z*,*Q*,*N*,*d*) is called Garden of Eden if there is not any configuration *c*' such that *D*(*c*') = *c*, i.e., the configuration c cannot be obtained during the work of the automaton *CA*, it can only be a starting configuration.

Von Neumann described a 29-state two-dimensional cellular automaton that is proved to be universal, i.e., from a computational point of view it is equivalent to the Turing machine. The system also has the ability to reproduce itself. This model used the infinite square grid with identical finite state automata and von Neumann-neighbourhood. The cells of the system have a resting state, and the system has a finite configuration at every time instant. The problems that were solved by J. von Neumann are as follows:

1. A universal Turing machine was embedded into the system.

2. An automaton A was embedded into the system that can built and start an arbitrary constructable automaton *B* (given the description of *B* in an embedded way) such that *B* is working independently of *A*.

3. To provide an automaton *A* that satisfies the previous point and it can build up itself.

Later on Codd gave an 8-state universal automaton that also fulfils von Neumann's aims and so, satisfies the condition of self-reproduction. Then Banks defined a 4-state universal cellular automaton.

The computational universality can also be obtained by one-dimensional cellular automata (using an enlarged neighbourhood):

Theorem: Let *TM* be a Turing machine with n states and let m be the size (i.e., the cardinality) of the tape alphabet. Then there is a one-dimensional cellular automaton with max{*n*,*m*}+1 states that can simulate *TM* using neighbourhood with 6 elements.

The next definition gives a subclass of the class of cellular automata.

Definition: A cellular automaton is totalistic if the transition function *d* does not depend on the order of the states given in the argument, i.e., the order of elements in the neighbourhood including the cell itself does not matter.

If the cells can have two states and the states are marked by numbers, then in a totalistic system the transition depends only on the sum (or equivalently on the average value) of the states of the appropriate cells. (We say that a cell has a value *x* meaning that the cell's current state is *x*.) In the literature totalistic cellular automata can also be used in more general: if only the sum (or the average) of the values of the neighbourhood is important to define the next state not only at two-state systems.

Definition: The cellular automaton *CA* is reversible, if for every possible configuration *c* there is exactly one configuration *c*' such that $c' \Rightarrow c$, i.e., *c* can be directly derived from exactly one configuration.

For cellular automata in two- or higher-dimensional spaces, the problem to decide if they are reversible is usually computationally (algorithmically) not decidable.

3.4.3.4.1. One-dimensional two-state cellular automata - Wolfram's results

Consider the one-dimensional case with $r = 1$, i.e., using the closest neighbourhood. Then the next state of a cell depends on its own state and the states of its two neighbours: three consecutive cells' states determine it. In binary case, each of these cells can have at most two values, therefore there are $2^{2 \cdot 2 \cdot 2} = 2^8 = 256$ possibilities, we can define 256 different systems (regarding their rules). Usually these systems are denoted by their binary code, or its decimal equivalent as follows.

Definition: Let $CA = (Z,\{0,1\},N,d)$ be a one-dimensional cellular automata, where *Z* is the set of integers, $N = \{+1,-1\}$ contains the positions of neighbour cells, *d* is the transition function. The transition function can be defined in the following way: consider the elements of $\{0,1\}^3$ (they are the three letter long words over the alphabet $\{0,1\}^*$) and order them alphabetically (using the relation $0<1$) in decreasing way. To determine *CA* we need to add the new state (0 or 1) for these eight cases. Since the order of these eight cases is fixed, a word over the alphabet $\{0,1\}^*$ with length eight gives the transition function. This is an eight-bit binary word and it is called the Wolfram-number of *CA*. Usually (beside or instead of the binary representation) the decimal value is used (it is between 0 and 255).

Example (Wolfram-number of a one-dimensional automaton)

Let the number 122 be given. Its binary form (in eight bits) is 01111010. As a Wolfram-number its meaning is as follows. The cell with its two neighbours can have the following value-triplets (written with three-letter words, in lexicographic order):

111,	110,	101,	100,	011,	010,	001,	000.	The transition function *d* assigns the values
0,	1,	1,	1,	1,	0,	1,	0	for these triplets.

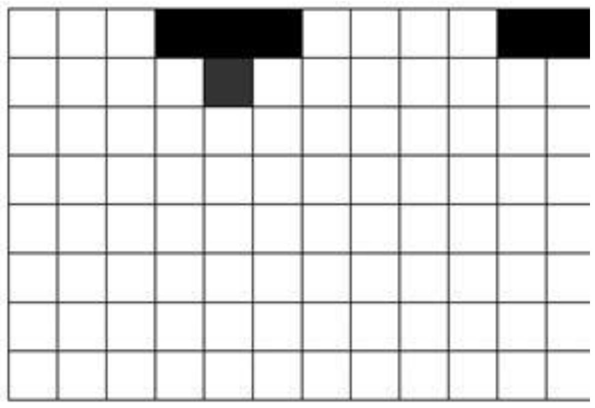
Thus, $CA = (Z,\{0,1\},\{+1,-1\},d)$ is a one-dimensional cellular automaton such that the next state of a cell will be 0 if its and both its neighbours' values are 1 in the previous time instant; the next state of a cell will be 1 if its and its left neighbour had value 1 and its right neighbour had value 0; 1 is the next state if the neighbours had value 1 while the cell itself had value 0; the value will be 1 if the left neighbour cell had value 1 and both the cell itself and its right neighbour had values 0; etc.

The dynamics of the one-dimensional cellular automata is usually depicted by a space-time diagram: the first row of these figures show configuration c_0. Every other row shows the configuration of the next time instant as the previous row. In these diagrams the states are marked by various colours. Naturally, only a finite interval (both in space and time) can be graphically shown, but these diagrams still give a good background to start to analyse such systems.

Let us analyse the one-dimensional two-state cellular automata using the smallest neighbourhood. These systems are classified by Wolfram (with an experimental approach) to the next four classes based on their dynamic properties starting from a random configuration.

Class 1. The first class includes those systems that lead to a constant (also called steady) configuration, i.e., configuration in which all the cells are in the same state, from any initial configurations. Figure 3.6 shows an example for a space-time diagram of one of these systems: this one has Wolfram-number 160. It can be seen that the constant 0 (all cells are going to have value 0) is obtained soon after the initial configuration. (In this figure the time is also shown by lightening the colours of the states step by step.) The automata with Wolfram-number 0 and 255 are also belonging to this class: in these cases all the pixels (cells) are changed to 0 (white) or 1 (black or grey) in the first derivation step.

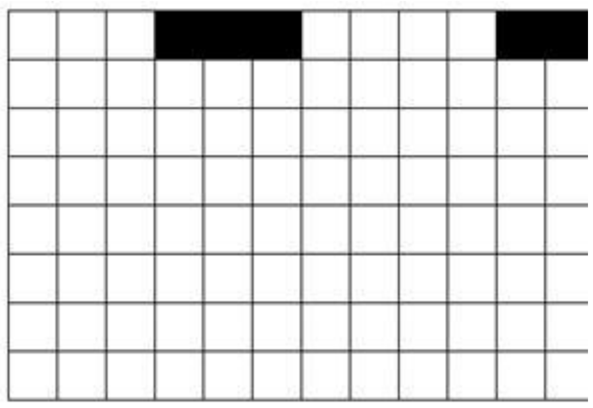
3.6. ábra - An example for the evolution of cellular automaton with Wolfram-number 160 from a given initial configuration.



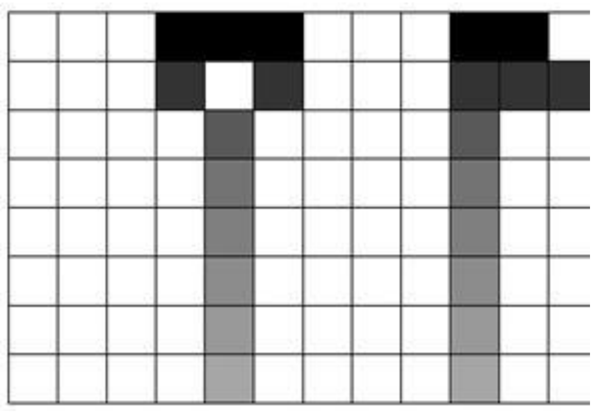
Class 2. This class includes those models that develops to configurations that are periodic (in time) with a short period (maybe 1) from any initial configurations. Figure 3.7 shows a part of a computation of the automaton with Wolfram-number 4. In this model the value of a cell will be 1 if and only if its value was 1 and its neighbours had values 0 (i.e., at 010). The system obtains a configuration with period 1, i.e., constant in time, these configurations are also called stable. Figure 3.8 presents an example for the work of automaton with Wolfram-number 108. As one can observe the work of this automata is periodic (in time).

It is a very important property of the automata in this class that a given pattern survives, maybe in the same place, maybe in a shifted way (travelling), as at automaton with Wolfram number 2. The maximal speed of the movement is one cell in a derivation step. In two third of these automata the pattern survives in a fixed size, while in the half of the remaining cases the pattern grows up to the infinity in a nice regular, periodic way (e.g., automaton with Wolfram-number 50). About one seventh of the automata have a more complex dynamical property, as we detail in the next classes.

3.7. ábra - Automaton with Wolfram-number 4 obtains a stable configuration.

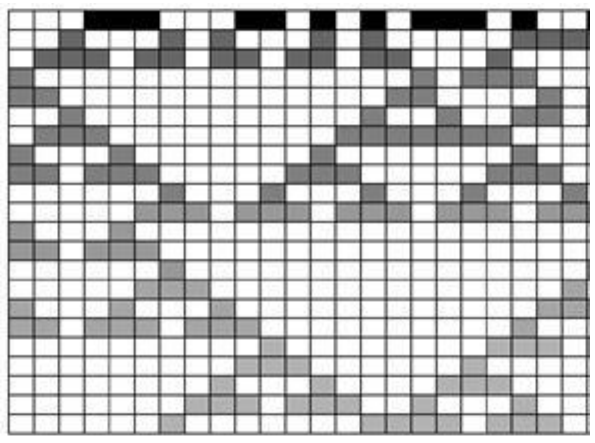


3.8. ábra - Automaton with Wolfram-number 108 leads to configurations that are periodic (in time).



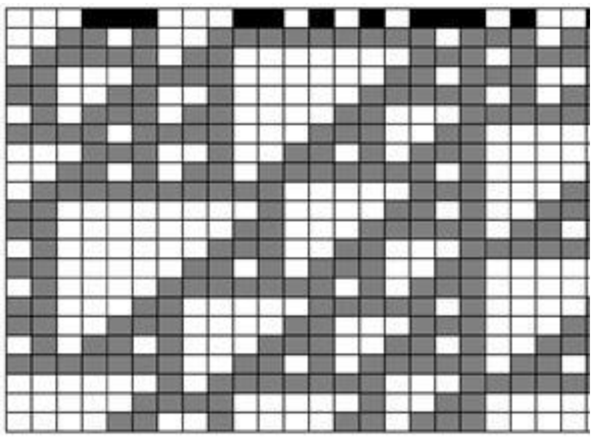
Class 3. This class contains the automata that are obtaining "random-like" patterns. There are ten transition functions that describe these automata with chaotic dynamical properties. The patterns generated by them are fractal like, opposite to the regularity that the previous classes have. There are three different patterns of these automata. Figure 3.9 shows automaton with Wolfram-number 22. Here a cell is going to have value 1, if and only if there was exactly one 1's among the values of its neighbours (including itself). This automaton starting from a configuration including only one cell with value 1 generates (in the space-time diagram) a version of the fractal called Sierpinski-triangle.

3.9. ábra - The evolution of the automaton with Wolfram-number 22 starting from a random-like configuration.



Class 4. This class includes models that behave in a "complex way", e.g., the cellular automata with Wolfram-number 110. The other three automata that are in this class are equivalent to this one (interchanging the roles of left-right and/or the black/white, i.e., the 0-1). Figure 3.10 shows a small part of the evolution of automaton with Wolfram-number 110. This model mixes the property of stability with some chaotic properties. Starting with a configuration having only one cell in state 1, we can observe periodic patterns and also a lane where the evolution seems to be chaotic. These two parts has a complicated interaction at their border. This model is proven to be Turing-universal (and thus, it is considered as the simplest Turing-universal computing model).

3.10. ábra - An example for the evolution of automaton with Wolfram-number 110.



It is very interesting that such a simple algorithm can behave in a surprisingly complex manner.

It is usual to do experiments starting from a configuration that has only one cell with value 1 and all other cells have value 0. In this case the automata with even Wolfram-number (i.e., those ones for which the value 0 is assigned for the triplet 000 at the next time instant) can be seen that generates the binary form of integers: configuration ci refers for the i-th element of the sequence.

Example (generating a sequence of numbers by cellular automaton)

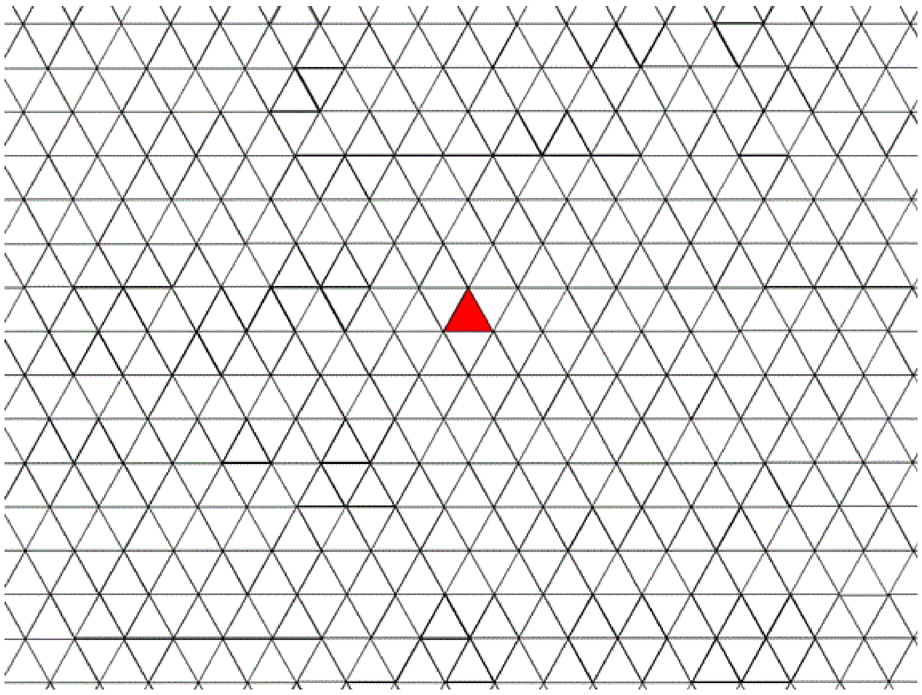
Let us consider the automaton with Wolfram-number 220. It is 11011100 in binary representation. At the beginning (at time instant 0) the configuration is the code of 1. The transition function assigns 1 to the triplets 111, 110, 100, 011 and 010. Based on these facts, it is easy to show that after the first derivation step the code 11 (it is the binary representation of 3) is obtained; after the second derivation step 111 (it is the code of 7), then 1111 (it is the code of 15) etc. can be obtained. The *i*-th element of the sequence generated by this automaton is exactly the code of 2^{i+1}-1.

So far we have analysed the simplest two-state (or binary) cellular automata (in one dimension with the smallest neighbourhood). We note here that in the case of 3-state automata (when a third colour is also used), even the smallest neighbourhood is used in one dimension, there is a very huge amount of possible automata: their number is the same as the number of strings over an alphabet with three letters of length 3^3 = 27. Wolfram analysed only the totalistic systems (using the more general definition of totalistic systems) among them and it is turned out that basically they have the same types of classes as the previously detailed two-state systems.

3.4.3.4.2. Pattern growth

As we have already seen interesting patterns can be obtained by the help of cellular automata. In this subsection we present pattern growth by two-dimensional cellular automata. These systems usually have a simple initial configuration (often there is exactly one cell that is not in resting state) and the rules of the system do not allow to a cell that is not in resting state to change its state (back) to resting state. In this way the pattern can change in growing way without decreasing.

Animation 3.1 shows how a pattern grows on the triangular grid using the closest neighbourhood (the three edge-joined triangle pixels). Red colour shows the new, active cells, yellow colour the next state and the old cells of the pattern are green. (The cells in resting state have white colour.) A cell that is in resting state becomes active (red) if it has exactly one red and two white neighbours.



Animation 3.1: A pattern is growing on the triangular grid.

In the next subsection we use cellular automata with two states and it is allowed that the state of a cell change from 0 to 1 and vice versa, depending on the states of the neighbours. In this way the pattern may grow and may be erased, therefore it is not a pattern growth…

3.4.3.4.3. Conway's game of life

Conway's game of life is a very popular type of the cellular automata. Conway designed the system according to three aims:

1. it should not be decided trivially if the pattern is growing unlimitedly from a finite initial configuration;

2. let there be some configurations which grow unlimitedly;

3. the initial patterns and their cells play (mimic) the rules of the life: cells birth, keep alive and die; let there be initial configurations that will die (finally the configuration having only cells in resting state is obtained), and initial configurations leading to a configuration that has a stable or periodic (in time) or maybe a travelling (moving) pattern.

The system is two-dimensional, binary, using the Moore-neighbourhood on the square grid. The system is direction independent, but it is not strictly totalistic because the number of neighbours having value 1 and the state of the cell itself determines the next state of the cell. These type of systems is called outer totalistic (the next state of a cell depends only on the average of its neighbours and the inner state of the cell).

Definition: Let $LG = (Z,\{0,1\},N,d)$ be a cellular automaton, where

- Z is the square grid, i.e., two-dimensional vectors with integer coordinates;

- N is the Moore-neighbourhood (in any order);

- d is defined in the following way:

+ if the cell has value 1, then it keeps value 1 if it has 2 or 3 neighbours with value 1,

+ if the cell has value 0, then it changes its value to 1 if it has exactly 3 neighbours with value 1,

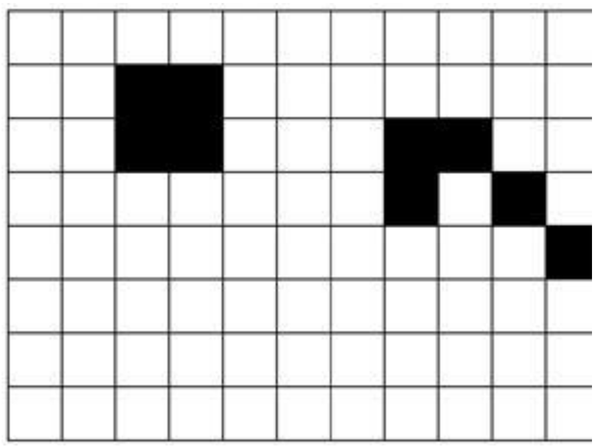
+ in every other case the value of the cell will be 0.

Then *LG* is called the game of life.

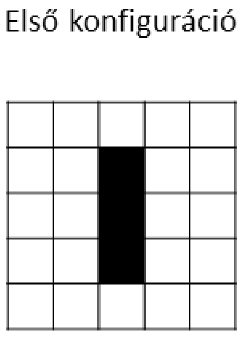
The cells having value 1 are called living cells, the cells having value 0 are called dead cells. Then the transition function can be interpreted in the following way: If a living cell has less than 2 living neighbours, then it will die (it has not enough connections, it is separated). If a living cell has 2 or 3 living neighbours, then it has an ideal environment and it suffices to keep it alive for the next generation. If a living cell has more than 3 living neighbours, then it will die, the region is overcrowded, it cannot support its life. A dead cell becomes alive if it has exactly 3 living neighbours, this is the way how the life spreads (birth, reproduction).

In this way the game of life can be viewed as a game with 0 players: after designing the initial configuration we could observe the evolution of the system, the spread of the life and the death etc. depending only on the initial configuration. It is easy to see that a one pixel size pattern will die in 1 derivation step obtaining the constant configuration having only dead cells. There are known such configurations (having a relatively small number of living cells) that can evolve for some hundreds or even thousands derivation steps before dying (arriving to the constant configuration). Figure 3.11 shows a stable configuration (they are also called still lifes): the next and all the next configurations are the same as the one in the figure. The living cells are black (value 1), the dead cells are white (value 0). (The figure contains three components that are stable by themselves: e.g., the left side shows the "block", the right side show the "beehive".)

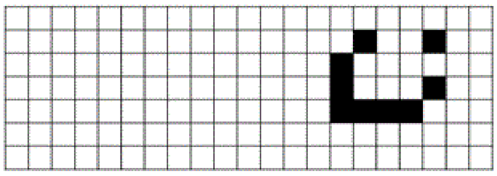
3.11. ábra - Still lifes: stable pattern(s) in the game of life.



The next animations show some typical, representative patterns that show the wide variability of the evolution of the patterns in game of life. In animation 3.2 an oscillating sign, the blinker can be seen. The numbers of living neighbours are also shown for helping to trace the evolution. There are patterns that are repeating with a longer period.



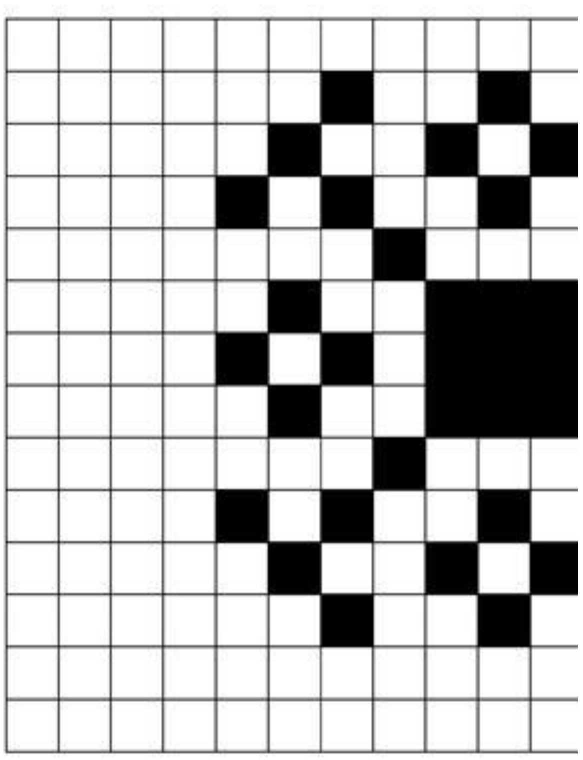
Animation 3.2: The oscillating blinker pattern (periodlength: 2).

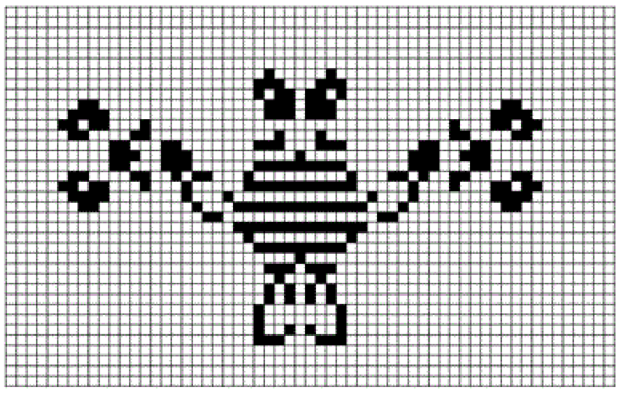


Animation 3.3: A "spaceship", a moving pattern.

The pattern in animation 3.3 is moving, while the pattern is repeated (in 4 derivation steps) it is moving by 1 cell to the right. Figure 3.12 shows the pattern "gun" that generates moving patterns. By the help of this pattern it is proven that there are such configurations in the game of life that grow forever and never die. Animation 3.4 presents a pattern that draws horizontal lines. Some patterns work in a similar manner filling the space during their infinite growth. For the help of the reader we used colour codes in this animation: the cells that have just died in the current step are marked by grey, while green colour shows the newly born cells; black cells are those who were alive in the previous and still are alive in the actual configuration.

3.12. ábra - The pattern "gun" that produces "gliders".



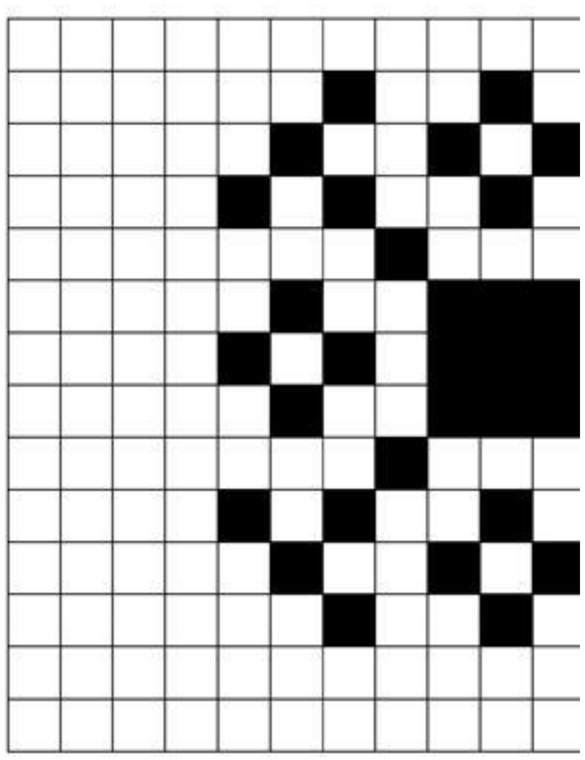


Animation 3.4: A pattern that "draws horizontal lines".

Thus, there are configurations (patterns) that die, are stable, are oscillating (periodic in time), are moving (travelling) and are growing unlimitedly. Moreover there are patterns that can "eat" gliders. The collision of gliders moving in various directions also results interesting patterns. These patterns gave the basis to use the game of life for simulation of a Turing machine. In this way, it is proven that the game of life is also a Turing universal model of computation.

There are patterns that are called "orphans": they contain finitely many living cell and they have the property that there are no such configuration from which they can be obtained by a derivation step. Therefore these patterns cannot be obtained in the game of life (they could only be initial configurations). In this way, it is obvious that the orphans are also Gardens of Eden. Moreover all Gardens of Eden contains an orphan pattern. In figure 3.13 an example is shown for an orphan (and so, for a Garden of Eden).

3.13. ábra - A Garden of Eden.



There are plenty of interesting facts about the Game of Life that can be found on the Internet. There are several thematic pages and sites, there is a dictionary of the names of various patterns, online simulations etc. The examples shown here form a very small portion of the Universe of game of life.

Finally we note that there are some variations of the game of life besides the classic Conway's version. This original version can be denoted by B3S23 that is for birth exactly 3 neighbours are needed, while for survive 2 or 3 neighbours are needed). In other versions the rules and sometimes the grid is also changed.

3.4.3.4.4. Other variations

In practice the cellular automata cannot work in an infinite space. Usually it is restricted for a finite part of the grid. In some cases the space is closed (e.g., the leftmost cells are right neighbours of the rightmost ones) and, in this way, the space is still homogeneous. Another possible solution is to have a fixed border from which the cells in the border get a fixed signal (e.g. the left neighbour of a leftmost cell has resting state in every time instant). In some applications the cells in the border are special and they "know about it" by having a special signal.

The synchronization problem of the cellular automata is known as the synchronization problem of firing squad. This problem is more that 50 years old. The task is the following: there is a finite, one-dimensional cellular automaton with the smallest neighbourhood. All cells work in the same way except the two in the border. In the beginning all cells (the soldiers) are in resting state but the leftmost cell (the general). The general is in state "fire if you are ready". This state gives the order to fire for the whole squad, but everybody can communicate only by his two neighbours. The aim is that everybody will fire at the same time. Therefore the cellular automaton should evolve to a configuration in which all the cells have became the state "fire" at the same derivation step (and no one before this step). The set of states and the transition function must not depend on the number of cells of the system. The challenge is to find cellular automaton that fires as soon as possible and use number states at least as possible. The standard solution is the following: a sign starts from the general and when it reaches the rightmost cell it is going back and meeting with a sign starting at the same time, but travelling with a less speed (one cell in three derivation steps) from left to right. In this way the middle soldier can be found. Then the middle soldier starts a signal that meets with a more slower sign starting from the general (at the beginning) and in the other side to a slower signal started from the rightmost soldier: in this way the soldiers at one fourth and three fourth positions are found… These squad firing synchronization problems have several variations; there are very active research going on this field.

Cellular automata can also be used to accept formal languages. For instance, the initial configuration of a one-dimensional cellular automaton consists of finitely many cells in non resting state that encodes the input word (similarly as at a Turing machine the finite input is written on an infinite tape filled by spaces before and after the input to the infinity). It is clear that the set of states of the automaton must include the input alphabet. The cell containing the first letter of the input can be specially marked, and the acceptance of the word can be defined by the fact if the marked cell has the value "accepted" after a finitely many derivation step.

Because the lack of space we shortly mention that there are cellular automata in higher-dimensional spaces and usually more than two states are used in a system (as we mentioned, for instance, at the part about the universality).

Cellular automata are widely used in simulation, and the research field "artificial life" is closely related to this field.

Finally we note that there are several related mechanisms that are able to generate/accept picture languages, i.e., two and higher-dimensional languages. Some examples are the collage- and array grammars and models connected to image processing. Some of these models are parallel.

3.4.5. Questions and exercises

1. What is a cellular automaton? How is it related to the previous automaton models? What are the differences between these models?

2. What does the self-reproduction mean at cellular automata?

3. What can be computed by cellular automata?

4. Describe the notion of the Wolfram-number! Which automata can be described by them?

5. What are the Wolfram's classes?

6. Define Conway's game of life! Why is this model interesting?

7. Write a program that simulates an arbitrary cellular automaton given by a Wolfram-number on the screen by space-time diagram

- starting from a configuration having one cell in state 1;

- starting from a random configuration;

- starting from a configuration given by the user!

8. Give a Wolfram-number and an initial configuration such that the given automaton do not changes this configuration (i.e., the next configuration is the same)!

9. Give a Wolfram-number and an initial configuration such that the given automaton works in a periodic way (in time) with a larger period than 2!

10. Check figure 3.11: why these patterns are stable?

11. Find more stable pattern in the game of life!

12. Give a configuration in the game of life such that it evolves periodically (in time) with a period larger than 2!

13. Check that the configuration in figure 3.13 is a Garden of Eden (write a program)!

14. Give a cellular automaton using a non-traditional grid!

3.4.6. Literature

S. Bandini, G. Mauri, R. Serra: *Cellular automata: From a theoretical parallel computational model to its application to complex systems,* Parallel Computing 27 (2001), 539-553.

M. Delorme, J. Mazoyer: *Cellular Automata: A Parallel Model,* Springer, 1998, 2010.

Drommerné Takács V. (eds.): *Sejtautomaták,* Gondolat Kiadó, 1978. [Cellular automata, in Hungarian]

M. Gardner: *The Game of Life, Parts I-III.,* in: Wheels, Life, and other Mathematical Amusements. New York: W. H. Freeman, 1983.

K. Imai, T. Hori, K. Morita: *Self-Reproduction in Three-Dimensional Reversible Cellular Space,* Artificial Life 8 (2002), 155-174.

M. Kutrib: *Non-deterministic cellular automata and languages,* Int. J. General Systems (IJGS) 41 (2012), 555-568.

K. Morita: *Simple Universal One-Dimensional Reversible Cellular Automata,* J. Cellular Automata 2 (2007), 159-166.

K. Morita: *Computation in reversible cellular automata,* Int. J. General Systems 41 (2012), 569-581.

B. Nagy: *Generalized triangular grids in digital geometry,* Acta Mathematica Academiae Paedagogicae Nyíregyháziensis 20 (2004), 63-78.

B. Nagy: *Characterization of digital circles in triangular grid,* Pattern Recognition Letters 25/11 (2004), 1231-1242.

B. Nagy, R. Strand: *A connection between Zn and generalized triangular grids,* ISVC 2008, 4th International Symposium on Visual Computing, Lecture Notes in Computer Science - LNCS 5359 (2008), 1157-1166.

B. Nagy, R. Strand: *Non-traditional grids embedded in Zn,* International Journal of Shape Modeling - IJSM (World Scientific) 14/2 (2008), 209-228.

R. Strand, B. Nagy, G. Borgefors: *Digital Distance Functions on Three-Dimensional Grids,* Theoretical Computer Science - TCS 412 (2011), 1350-1363.

G. Rozenberg, T. Bäck, J. N. Kok (eds.): *Handbook of Natural Computing,* Section I: Cellular Automata (vol. I, pp. 1-331), Springer, 2012.

H. Umeo: *Synchronizing square arrays in optimum-time,* Int. J. General Systems (IJGS) 41 (2012), 617-631.

S. Wolfram: *A New Kind of Science.* Champaign, IL: Wolfram Media, 2002.

4. fejezet - Traces and trace languages

4.1. Commutations and traces

The concepts of parallelism and concurrency are fundamental concepts of computer science, information theory and (parallel) programming. The specification and validation of concurrent programs are very important, since the everyday used programs that work on distributed systems and, in this way, their correctness and safety are based on these theories. In 1977, analysing basic networks, Mazurkiewicz introduced the concept of partial commutations. Two independent parallel events commute, i.e., their executing order can be arbitrary in a sequential simulation. By using the concept of commutations, the work of the concurrent systems can be described by traces. The main advantages of this description are the following: the fundamentality and essentiality of the concurrent behaviour can be kept; moreover this theoretical description is close to classical methods that describe the work of the sequential programs.

4.1.4.1.1. Commutations

In the literature, the semi-commutation is known as a mapping that allows permuting two consecutive letters of a word if their order was in the given one. The partial commutation allows permuting two consecutive letters independently of their original order. In this way, the semi-commutation can be seen as a generalisation of the partial commutation. Now we give formal definitions for these concepts.

Let T be a finite alphabet. The length of word w is denoted by $|w|$. Furthermore, let $|w|_a$ denote the number of occurrences of the letter $a\in T$ in w, and let $alp(w) = \{a\in T|\ |w|_a\neq 0\}$ be the set of the letters of w.

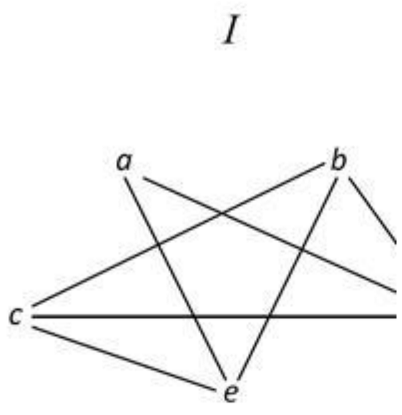
Definition: Let T be a finite alphabet, and let D be reflexive and symmetric binary relation on the alphabet T, i.e., $(a,a)\in D$ for every $a\in T$, further, if $(a,b)\in D$ for some letters $a,b\in T$, then also $(b,a)\in D$. Then D is a dependency relation on T. The independency relation (or also called commutation) induced by D is $I = (T\times T)\setminus D$. Obviously, I is irreflexive and symmetric.

Intuitively one can imagine that if a and b works on independent resources (or use independent data for their computations), then $(a,b)\in I$. Therefore the order of their execution is arbitrary: $ab = ba$ (they are equivalent). They can be executed in any order in sequential manner, or in parallel, simultaneous way. Opposite to this fact, the complement of the commutation, the dependency, consists of pairs that cannot be executed in a parallel manner. Both the dependency and the commutations can be drawn by graphs. The vertices of these graphs are the letters and those are connected by an edge that is in the given relation. We note here that the loop-edges of the dependency relation that is fulfilled for every vertex a are usually not drawn on the figures (the edge (a,a) is missing to make the figures simpler).

Example (commutation)

Let $T = \{a,b,c,d,e\}$ and $D = \{(a,a),(a,b),(a,c),(b,a),(b,b),(a,c),(c,c),(d,d), (d,e),(e,d),(e,e)\}$ be a dependency relation on T. Then the induced commutation is $I = \{(a,d),(a,e),(b,c),(b,d),(b,e),(c,b),(c,d),(c,e),(d,a), (d,b),(d,c),(e,a),(e,b),(e,c)\}$. Figure 4.1 shows both the graph of the commutation and the graph of the dependency relation.

4.1. ábra - Graph of commutation and graph of dependency.



Definition: Let *T* be an alphabet and let *I* be an independency relation on *T*. Let *w*≡*w*' if *w* = *uabv*, *w*' = *ubav* and *a*,*b*∈ *T*, *u*,*v*∈ *T**, (*a*,*b*)∈ *I*. The equivalence relation ≡ induced by *I* on *T** is called partial commutation. For each word *w*∈ *T** the set of words that are equivalent to *w* form the commutation class [*w*] of *w*. These classes are called traces. The definition can be easily extended to languages: [*L*] = {[*w*]| *w*∈ *L*}, then [*L*] is a trace language that consists of traces. Further, it is usual to use the language *L*' = {*w*'| there is a word *w*∈ *L* such that *w*'∈ [*w*]}. This language is the linearization of the trace language [*L*], or shortly (if it is not confusing) it is also called trace language.

Two words are in the same class, i.e., they are equivalent, if there is a finite sequence of words that connect them and two consecutive words of this sequence differ by the permutation of two consecutive and independent letters.

Example (the equivalence-class of a word)

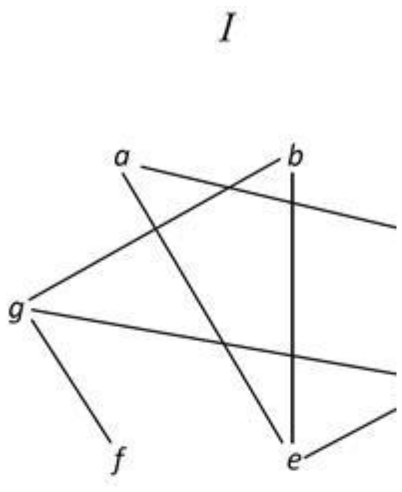
Let *T* = {*a*,*b*,*c*}, *w* = *abbcbaacb* and *I* = {(*b*,*c*),(*c*,*b*)} be a commutation. Then [*w*] = {*abbcbaacb*,*abbcbaabc*,*abbbcaacb*,*abbbcaabc*,*abcbbaacb*, *abcbbaabc*,*acbbbaacb*,*acbbbaabc*}.

Definition: Let *T* be an alphabet and *I* be a commutation. The equivalence class of the empty word is called empty trace. The concatenation of two traces is defined as [*w*][*w*'] = [*ww*']. A trace *t* (or a word *w*) is connected if *alp*(*t*) (resp. *alp*(*w*)) defines a connected subgraph in the dependency graph.

Example (connected and non-connected traces)

Let the independency relation shown in figure 4.2 (left) be given on the alphabet {*a*,*b*,*c*,*d*,*e*,*f*,*g*}. Then the trace *t* = [*abbed*] is connected (*alp*(*t*) = {*a*,*b*,*d*,*e*}), but the trace *t*' = [*acafegfea*] is not connected, since *alp*(*t*) = {*a*,*c*,*e*,*f*,*g*} (see also the right-hand-side of figure 4.2).

4.2. ábra - A non-connected trace



The definition can be extended to trace languages in the usual way. In this way, besides the usual set-theoretic operations, the concatenation and the Kleene-star (iteration) can also be used for these languages.

By dropping the symmetry condition in the definition of partial commutation, (i.e., by allowing not necessarily symmetric relations) the semi-commutation is obtained.

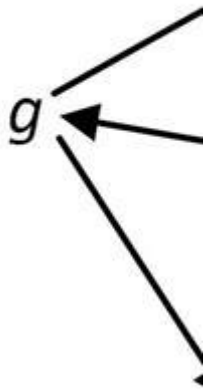
Definition: An irreflexive and not necessarily symmetric binary relation Z on a finite alphabet T is a semi-commutation. The semi-commutation system is a rewriting system (T,H) assigned to the semi-commutation Z such that the rewriting rules are: $H = \{ab \rightarrow ba | (a,b) \in Z\}$.

In a semi-commutation two consecutive letters can be permuted only if they were in the order fixed by the semi-commutation. A semi-commutation relation can be presented by digraphs (directed graphs). The relation induced by a semi-commutation may not be symmetric, and thus it is usually not an equivalence relation.

Example (semi-commutation)

Let $Z = \{(a,e),(b,e),(c,a),(c,d),(d,c),(d,e),(d,g),(e,a),(e,d),(g,b),(g,f)\}$ be a semi-commutation (see also figure 4.3). Then in the rewriting system defined by Z the following "derivations" are valid: $abbedgbffgbcd \Rightarrow abebdgbffgbcd \Rightarrow aebbdgbffgbcd \Rightarrow aebbgdbffgbcd$;
$abbedgbffgbcd \Rightarrow abbdegbffgbcd \Rightarrow abbdegbffgbdc$.

4.3. ábra - Representation of a semi-commutation.



Definition: The words w and *w*' are called letter-equivalent if $|w|_a = |w'|_a$ for every $a \in T$. Similarly, two languages *L* and *L*' are letter-equivalent if for every word *w* of *L* there is a word $w' \in L'$ that is letter-equivalent to *w*; and similarly, for every word *w*' of *L*' there is a word $w \in L$ that is letter-equivalent to *w'*.

Let *L* be a formal language, then the language *L*' = {*w*'| there is a word $w \in L$ such that *w* and *w*' are letter-equivalent} is called the commutative closure of *L*. A language is commutative if it is the commutative closure of itself.

In the special case, when the independency relation *I* contains all pairs of letters (except only the pairs of the form (*a*,*a*), since they cannot be included there by definition), then we obtain the commutative closure of the languages. In this case a word w is equivalent to every word that contains the same number of letters of each as *w*, i.e., the equivalence relation induced by *I* is exactly the letter-equivalence relation. In this way, by a fixed ordering of the letters of the alphabet (e.g., $T = \{a_1,a_2,\dots,a_n\}$), one can use the Parikh-vectors

$$(|w|_{a_1}, |w|_{a_2}, \dots, |w|_{a_n})$$

assigned to (the equivalent classes of) words instead of the original language over the alphabet *T*. In this Parikh-map (or Parikh-image) of the language the vectors refer for multisets (the order of the letters in a word does not matter, only their numbers are of interest), and thus these languages are multiset-languages. About the Parikh-sets of the languages (that are defined at the section about *P* automata) we present the most important definitions and results.

A set of vectors is linear if it can be written in the form

$$\{\vec{v}_0 + \sum_{m}^{i-1} x_i\vec{v}_i \mid x_i \in \mathbb{N}\ \textit{for every}\ 0 \le i \le m\}$$

for some fixed vectors

$$\vec{v}_i \in \mathbb{N}^n$$

($0 \le i \le m$). A set of vectors is semi-linear if it is a finite union of linear sets. The following results are known about the traditional language classes of the Chomsky hierarchy.

Theorem: Every context-free language is semi-linear (that is the widely known Parikh theorem). There is a letter-equivalent regular language for every context-free language. Over the one-letter alphabet every context-free language is regular.

By the previous theorem, the commutative closure of the regular languages is the same set as the commutative closure of the context-free languages. Moreover exactly these languages are the commutative semi-linear languages. There are not semi-linear context-sensitive languages, for instance the language of words with square lengths.

We introduce some notions for the traces based on the number of letters in them: let $t = [w]$. Then the numbers of the letters of the trace is denoted by $|t|$. Further, let $|t|_a$ denote the number (occurrences) of the letter $a\in T$ in t, and let $alp(t) = \{a\in T| \ |t|_a\neq 0\}$ be the set of the letters of t.

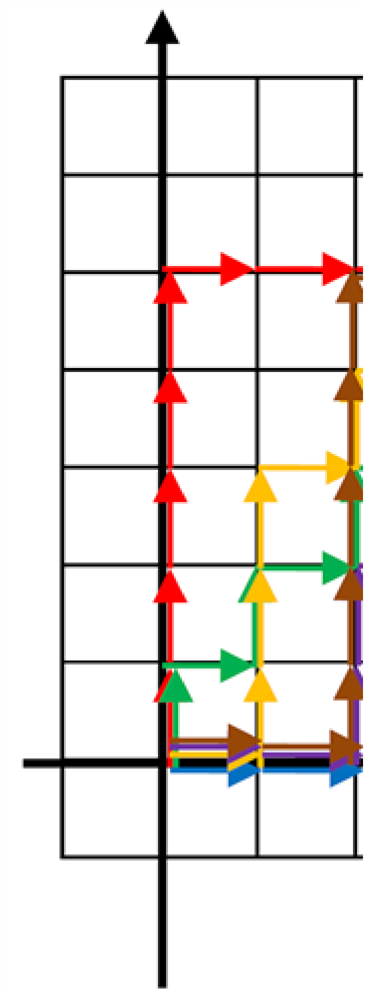
Definition: Let the commutation I be given over the alphabet T. Two traces t and r are independent if $alp(t)\times alp(r)\subseteq I$. Similarly, two words w and w' are called independent if $alp(w)\times alp(w')\subseteq I$.

It can be seen that t and r are independent if and only if $tr = rt$ and $alp(t)\cap alp(r) = \{\}$. For the empty word $alp(\lambda)= \{\}$, and thus, the empty word and the empty trace is independent of every word, and trace, respectively.

Example (paths in the square grid)

Let a square grid be given, there a shortest path (using only the edges of the grid) between the points (0,0) and (5,3) contains 5 steps to upward and 3 steps to the right. They can be given by words *uuuuu* and *rrr*. These two words and the traces based on them are independent. Moreover, by defining these words as the concatenation of their letters (or the traces based on them), the subwords (and the traces based on them) of *uuuuu* are independent of the subwords (and traces based on them) of *rrr*, i.e., the order of these steps is arbitrary. Figure 4.4 shows some of the shortest paths, i.e., some elements of the trace [*uuuuurrr*]. In fact, the multiset with Parikh vector (5,3) over the ordered alphabet {*u*,*r*} represents this trace.

4.4. ábra - In the two-dimensional square grid the shortest paths form a trace, since the order of the steps is arbitrary, all words with vector (5,3) give a shortest path.



By the definition of trace languages one can see that a language and a dependency- or an independency relation is needed: and thus the traces given by the commutations of the words of the language form the trace language. In the next subsection we analyse the trace languages connected to regular languages.

4.1.4.1.2. Trace languages

In this subsection we concentrate on the recognizable and rational trace languages.

The commutation on the words of a language can be understood as a kind of operation, and thus, a trace language can be seen as the result of the closure under this operation applied for the language.

Definition: Let the equivalence relations on T^* be given based on the dependency relation D, and let their class be denoted by $[M]$. Let $h:T^*\to[M]$ be the mapping that assign $[w]$ to each word $w\in T^*$.

The trace language $[L]\subseteq[M]$ is said to be recognizable if $h^{-1}([L])$ is a regular language over T^*.

In fact the language $[L]$ defined in this way does not contain words, but instead of them, it contains their commutation classes. The linearization of the trace language $[L]$ is a language over T^* and it is defined as $L' = \{w\in T^*|[w]\in L\}$.

The set of linearizations of recognizable trace languages over T^* defined by using the dependency relation D is denoted by REC(D).

The recognizable trace languages can be characterized by automata as follows.

Definition: Let $[M]$ be the set of equivalence classes induced by the dependency relation D among the words over the alphabet T. Then, the ordered quintuple $TA = (Q,[M],q_0,d,F)$ is called a trace automaton, where Q is the nonempty, finite set of states; q_0 is the initial state; $F \subseteq Q$ is the set of final (or accepting) states; and $d:Q\times[M]\rightarrow Q$ is the transition function with the following properties:

$d(q,[\lambda]) = q$ for every state $q \in Q$,

$d(q,uv) = d(d(q,u),v)$ for every state $q \in Q$ and $u,v \in [M]$.

Then, the set (trace language) $[L] \subseteq [M]$ is accepted by the automaton TA with the set (of accepting states) F if $[L] = \{u \in [M] | d(q_0,u) \in F\}$.

Example (trace automaton)

Consider the trace language that is accepted by the automaton $TA = (\{q_0,p,r\},[M],q_0,d,\{p\})$, where $[M]$ is defined by the dependency relation $D = \{(a,a),(a,c),(b,b),(b,c),(c,a),(c,b),(c,c)\}$ over the alphabet $\{a,b,c\}$, i.e., only the letters a and b are independent of each other; and the transition function d is defined as follows. Let $d(q_0,[a]) = q_0$, $d(q_0,[b]) = r$, $d(q_0,[c]) = p$, $d(r,[a]) = d(r,[b]) = d(r,[c]) = r$, $d(p,[a]) = d(p,[b]) = p$, $d(p,[c]) = r$. Then, the recognized trace language is $[a^*][c][a^*b^*]$, and its linearization is the regular language $a^*c(a+b)^*$.

The class of these automata characterize the class of recognizable trace languages:

Theorem: Let $[M]$ be the equivalence classes induced by the dependency relation D among the words over the alphabet T.

A trace language $[L] \subseteq [M]$ is accepted by some trace automaton if and only if $[L]$ is a recognizable trace language, i.e., $L' \in \mathrm{REC}(D)$, where L' is the linearization of $[L]$.

The recognizable trace languages have a nice connection to the recognisability by finite automata, and thus, they can easily be related to the regular languages, as we have already seen by our example. Opposite to this fact, the definition of rational trace languages shows an analogy to the structure of regular expressions (and thus by the description of regular languages by regular expressions), as we can see in the following iterative definition.

Definition: Let $[M]$ be the set of classes induced by the dependency relation D among the words over the alphabet T.

- Let $[L] \subseteq [M]$ be a finite trace language, that is $[L]$ consists of finitely many classes of commutations, and thus, its linearization is a finite language). Then $[L]$ is a rational trace language. (It is the base of the iterative definition.)

- If $[L]$ and $[L']$ are rational trace languages, then $[L] \cup [L']$, $[L][L']$ and $[L]^*$ are also rational trace languages. (This is the induction step with the regular operations.)

Further, every rational trace language can be given by some finite trace languages and finitely many usage of the regular operations.

On the set T^* the set of the linearizations of the rational trace languages defined by the dependency relation D is denoted by $\mathrm{RAT}(D)$.

The next theorem holds for rational trace languages.

Theorem: Let the set of equivalence classes induced by the dependency relation D on T^* be denoted by $[M]$. Let $h:T^*\rightarrow[M]$ be the mapping that assigns $[w]$ for every word $w \in T^*$.

Then L is a (linearization of a) rational trace language, if and only if there is a regular language L' such that $L = h^{-1}(h(L')) = \{w' | w' \in [w]$ for some word $w \in L' \}$.

In this book, in the next part, we concentrate on linearizations of trace languages; the trace languages are represented by their linearizations.

In a special case, when the independency relation I is empty, then the recognizable trace languages based on L and the rational trace language based on L are the same, more precisely we have the following result.

Theorem: Let T be a fixed alphabet and let the commutation I be empty (i.e., the dependency relation D be maximal, containing every pair of letters). Then REC(D) = RAT(D) = REG.

If the independency relation I is not empty, then is fulfilled with the dependency relation D induced by the commutation I.

The rational trace languages can be described by finite automata with translucent symbols. This description is given by Benedek Nagy and Friedrich Otto. First we define the (non-deterministic) finite automata with translucent letters.

Definition: The ordered tuple $TFA = (Q,T,Q_0,\$,t,d,F)$ is a (non-deterministic) finite state automaton with translucent letters, where Q is a finite, nonempty set of states;, T is the (tape)alphabet; $Q_0 \subseteq Q$ is the set of initial states; $\$ \notin T$ is the endmarker (of the tape); $t:Q \rightarrow 2^T$ is the translucency mapping; $d:Q \times T \rightarrow 2^Q$ is the transition mapping (function); $F \subseteq Q$ is the set of final (or accepting) states.

For every state $q \in Q$ the letters of $t(q)$ are translucent, that is, the automaton in this state cannot see these letters. The work of the automaton for an input word w_0 is as follows: first we choose non-deterministically an initial state $q_0 \in Q_0$. Thus, the initial configuration is $(q_0,w_0\$)$. Let us assume now that the current configuration is $(q,w\$)$. Let $w = a_1a_2\ldots a_n$, where $n \geq 1$ and $a_1,a_2,\ldots,a_n \in T$. Then we look for the first (leftmost) letter that is not translucent in the current state, that is let $w = uav$ such that $a \notin t(q)$, $u \in t(q)^*$. Then we choose a state q' from $d(q,a)$ in a non-deterministic manner, and thus, the next configuration will be $(q',uv\$)$. If $d(q,a) = \{\}$, then the automaton stops (halts) without accepting the input. If $w \in t(q)^*$, then the automaton sees the symbol \$ and stops. If the automaton sees the symbol \$ and the actual state is an accepting state, then the automaton accepts the input with this run (computation). The set of accepted words results the accepted language.

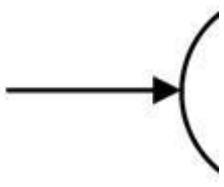
One can see in the definition that transitions without reading an input letter are not allowed in these models (there are no transitions by the empty word). However the automaton may have several initial states. It is evident that this model is a generalisation of the usual non-deterministic finite state automata. Considering the empty translucency mapping t in every state q: $t(q) = \{\}$, the automaton works in a similar way as the traditional non-deterministic finite automata (if we may choose the state where we start to read the input word by the help of transitions with the empty word from the initial state).

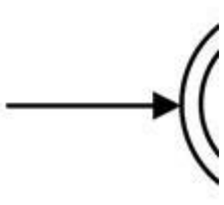
The next example shows an automaton with translucent letters.

Example (the language of the words built up by the same number of a's, b's and c's)

Let $TFA = (\{q_0,q_1,q_2,q_3\},\{a,b,c\},\{q_3,q_0\},\$,t,d,\{q_3\})$, where the translucency mapping t has the following values: $t(q_0) = \{b,c\}$, $t(q_1) = \{a,c\}$, $t(q_2) = \{a,b\}$, $t(q_3) = \{\}$; further the transition function d is defined as follows: $d(q_0,a) = \{q_1\}$, $d(q_1,b) = \{q_2\}$, $d(q_2,c) = \{q_3,q_0\}$. The automaton accepts the empty word in state q_3. For other choice, starting at the initial state q_0 the first occurrence of a's is erased from the tape (since all b's and c's are translucent), and the automaton reaches the state q_1. In this state the first occurrence of the latter b is erased (a's and c's are translucent) and the automaton reaches the state q_2. Then the letters a and b are translucent, and thus the first c is erased and the automaton enters into the state q_0 to continue the process or into the accepting state q_3 to accept if the whole input is processed (every of its letter has been erased). In this way the automaton accepts the language over $T = \{a,b,c\}$ that contains every word such that the number of the letters a, b, and c are the same. Figure 4.5 shows the graph of the automaton.

4.5. ábra - The graph of the finite automaton with translucent letters that accepts the language $\{w \in \{a,b,c\}^* \mid |w|_a = |w|_b = |w|_c\}$ (the translucent letters are marked by red colour).





As we can see from the previous example, the accepting power of the finite automata with translucent letters is much greater than the accepting power of traditional finite automata. Some non-context-free languages can also be accepted by finite automata with translucent letters.

The next example shows a deterministic finite automaton with translucent letters (that is, there is only one initial state and the transition function gives sets with at most one element).

Example (the Dyck language: correct bracket-expressions)

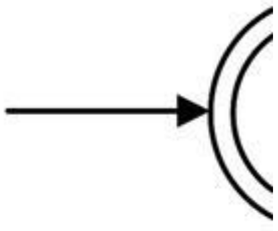
Let *TFA* = $(\{q_0,q_1\},\{0,1\},\{q_0\},\$,t,d,\{q_0\})$, where the values of the translucency mapping *t* are: $t(q_0) = \{\}$, $t(q_1) = \{0\}$; further the transition function d is defined as $d(q_0,0) = \{q_1\}$, $d(q_1,1) = \{q_0\}$. The work of the automaton is as follows: at the state q_0 the automaton accepts the empty word, since there are no translucent letters in this state. If the tape is not empty (yet), then at the state q_0 the letter 0 is erased provided it is the first letter of the tape and the automaton reaches the state q_1. In this state the first occurrence of the latter 1 is erased (the 0's are translucent) and the automaton goes back to the state q_0. In this way, it can be seen that every accepted word (and only those) fulfils the next two conditions:

- the number of 0's (opening) and 1's (closing) are the same,

- for every prefix of the word it is true that the number of 0's in this prefix is at least as many as the number of 1's in this prefix.

Thus, the accepted language is exactly the correct bracket-expressions. The graph of the automaton is presented in Figure 4.6.

4.6. ábra - The graph of the automaton accepting the correct bracket-expressions (Dyck language).



Theorem: For every finite automaton with translucent letters *TFA* there is an equivalent (that is, accepting the same language) finite automaton with translucent letters in a normal form $TFA' = (Q,T,Q_0,\$,t,d,F)$ such that the next conditions hold:

- for every state $q\in Q$ there is at most one letter $a\in T$ such that $d(q,a)\neq\{\}$.

- for every state $q\in F$ the mapping $t(q) = \{\}$ and $d(q,a) = \{\}$ for every letter $a\in T$.

We can observe that in an automaton in normal form at most one letter can be read in every state, moreover at the accepting states there are no translucent letters and no letters that the automaton can read. Since these automata are non-deterministic at every transition we may guess what the next letter can be that the automaton will read in the next transition and we can choose such a state that allows to read this letter. Based on this idea, the set of states of the automata in normal form can be defined as the Cartesian product of the states of the original automaton and the input alphabet, and consequently the transition function will contain all the ordered pairs having q as the first element instead of the state q itself. We could non-deterministically guess whether the input is fully processed, and choose to enter into an accepting state.

The next theorem is about the closure properties of the language class accepted by non-deterministic finite automata with translucent letters.

Theorem: The language class accepted by non-deterministic finite automata with translucent letters is closed under the regular operations: union, concatenation and iterations (Kleene-star and Kleene-plus).

Now we are continuing with the characterization of the rational trace languages by the help of non-deterministic finite automata with translucent letters.

Theorem: Let L be the linearization of a rational trace language. Then one can construct a non-deterministic finite automaton with translucent letters such that it accepts L, i.e., $L(TFA) = L$.

Proof: The construction goes in the following way. Let L be a rational trace language. Then there is a dependency relation D such that it is assigned to the mapping h that assigns the equivalence class defined by D to every word; moreover there is a regular language L' such that $L = h^{-1}(h(L'))$. Then let $FA = (Q,T,q_0,d,F)$ be a deterministic finite automaton that accepts L'. First, let us extend FA to the finite automaton with translucent letters $TFA = (Q,T,\{q_0\},\$,t,d,F)$ such that $t(q) = \{\}$ for every state. Then let $TFA' = (Q',T,Q_0,\$,t',d',F')$ be in normal form and equivalent to TFA. Let I be the commutation induced by D. Let us extend TFA' with the translucency mapping t'' to obtain $TFA'' = (Q',T,Q_0,\$,t'',d',F')$ as follows: for every state q and for every letter a let $a\in t''(q)$ if and only if $d(q,a')\neq\{\}$ for the letter $a'\in T$ and $(a,a')\in I$. Then it can be checked that $L(TFA'') = L$. √

Theorem: Let $TFA = (Q,T,Q_0,\$,t,d,F)$ be given in normal form. If the following conditions are fulfilled, then TFA accepts a rational trace language, i.e., $L(TFA)\in \mathrm{RAT}(D)$ for some dependency relation D.

- For every pair of states $p,q\in Q$ if $d(q,a)\neq\{\}$ and $d(p,a)\neq\{\}$ for the same letter $a\in T$, then $t(q) = t(p)$. Further,

- let R be the binary relation that can be defined in the following way: $(a,b)\in R$ if and only if there is a state $q\in Q$ such that $d(q,a)\neq\{\}$, and $b\in t(q)$. Then the relation R is symmetric.

In this way the relation R is a commutation, and it induces the dependency relation D.

Moreover, by the previous theorem that is proved in a constructive way, there is a *TFA* with the above properties for every rational trace language L over the alphabet T such that it accepts L.

The class of languages accepted by finite automata with translucent letters is closed under the regular (language) operations; and finite automata with translucent letters can effectively be constructed for every rational trace language, for their union, concatenation and Kleene-closure. In this way an effective calculus is given for the rational trace languages.

Regarding our examples, the language containing the words that are built up by the same number of a's, b's and c's is a rational trace language, since it can be obtained by using the maximal independency relation ($I = \{(a,b),(a,c),(b,a),(b,c),(c,a),(c,b)\}$) over the three-letter alphabet $T = \{a,b,c\}$. This language is given, e.g., by the commutation closure for the regular language $(abc)^*$, moreover it is exactly the commutative closure of the regular language $(abc)^*$. Our other example, the Dyck language is not a rational trace language, in this case the relation is not a commutation, but a semi-commutation: the rewriting only allowed in direction $10\rightarrow01$, and thus, the relation that defines this language form the regular language $(01)^*$ is not symmetric.

By closing this chapter we note here that the recognizable trace languages are accepted by asynchronous cellular automata, and this characterization is given by Zielonka. We have no space here to detail this construction. Finally, we note that similarly to the case of rational trace languages context-free trace languages can be defined; their analysis can be given in a similar manner as we did here using a concept similar to the pushdown automata with translucent letters. In the literature we give some papers on this topic, too.

2. Questions and exercises

1. What is the difference between the commutation and the semi-commutation?

2. What are the Parikh-vector of a word and the Parikh-set of a language? How these concepts relates to the commutations?

3. Compare the recognizable and the rational trace languages. What are their common properties and what are the main differences between them?

4. Let $T = \{a,b,c,d,e\}$ and $D = \{(a,a),(a,b),(a,c),(b,a),(b,b),(b,c),(c,a),(c,b), (c,c),(d,d),(d,e),(e,d),(e,e)\}$ be a dependency relation on T.

(a) Give the independency relation induced by D.

(b) Draw the graph representation of both the given dependency relation and the induced commutation.

5. Let the independency relation $I = \{(a,b),(a,c),(b,a),(b,d),(c,a),(c,e),(d,b),(e,c)\}$ be given over the alphabet $T = \{a,b,c,d,e\}$. Draw the graph of this commutation and also the dependency relation induced by.

6. Let $T = \{a,b,c,d\}$, $w = dcbaccd$ and the commutation $I = \{(b,c),(c,b),(c,d),(d,c)\}$ be given. Describe the set of words $[w]$. Are w and the word $w' = acaca$ independent?

7. Give a recognizable trace language, and a trace automaton that recognize it. Give a regular language that is the linearization of the given trace language.

8. Let $T = \{a,b,c,d\}$ and $I = \{(b,c),(c,b),(b,d),(d,b)\}$ over T. Give the finite automaton with translucent letters that accepts the (linearization of) the rational trace language based on $(abca)^*(bc)^*(a+bd)^*$ and I.

9. Let $T = \{a,b,c\}$ and let I be the maximal independency relation over T. Give the finite automaton with translucent letters that accepts the rational trace language based on the regular language $(aa)^*(bc)^*b^*$ using I, i.e., the language that contain exactly the words that contain even number of a's and the number of b's are not less than the number of c's.

3. Literature

O. Burkart, D. Caucal, F. Moller, B. Steffen: *Verification on infinite structures,* in: **J. A. Bergstra, A. Ponse, S. A. Smolka (eds.):** *Handbook of Process Algebra, Elsevier, 2001. pp. 545-623.*

M. Clerbout, M. Latteux, Y. Roos: *Semi-Commutations,* in: **V. Diekert, G. Rozenberg (eds.):** *The book of traces, World Scientific 1995. pp. 487-552.*

V. Diekert, Y. Métivier: *Partial Commutation and Traces,* Chapter 8, in: **G. Rozenberg, A. Salomaa (eds.):** *Handbook of formal languages, vol. 3, Springer, pp. 457-533. (1997)*

V. Diekert, G. Rozenberg (eds.): *The book of traces,* World Scientific 1995.

J. Kortelainen: *Remarks about Commutative Context-Free Languages,* Journal of Computer and System Sciences 56 (1998), 125-129.

A. Mateescu, G. Rozenberg, A. Salomaa: *Shuffle on trajectories: syntactic constraints,* Theoretical Computer Science 197 (1998), 1-56.

A. W. Mazurkiewicz: *Trace Theory,* Advances in Petri Nets 1986, LNCS 255 (1987), 279-324.

F. Moller: *A Taxonomy of Infinite State Processes,* Electronic Notes in Theoretical Computer Science 18 (1998), 20 pages.

B. Nagy: *Languages generated by context-free grammars extended by type AB?BA rules,* Journal of Automata, Languages and Combinatorics 14 (2009), 175-186.

B. Nagy: *Permutation languages in formal linguistics,* IWANN 2009, Lecture Notes in Computer Science 5517 (2009), 504-511.

B. Nagy: *Linguistic power of permutation languages by regular help,* in: Linguistics, Biology and Computer Science: Interplays, Cambridge Scholars (2011), 135-152.

B. Nagy, F. Otto: *CD-Systems of Stateless Deterministic R(1)-Automata Accept all Rational Trace Languages,* LATA 2010, Lecture Notes in Computer Science - LNCS 6031 (2010), 463-474.

B. Nagy, F. Otto: *An automata-theoretical characterization of context-free trace languages,* SOFSEM 2011, Lecture Notes In Computer Science - LNCS 6543 (2011), 406–417.

B. Nagy, F. Otto: *Finite-State Acceptors with Translucent Letters,* ICAART 2011 - 3rd International Conference on Agents and Artificial Intelligence, BILC 2011 - 1st International Workshop on AI Methods for Interdisciplinary Research in Language and Biology, 3-13.

B. Nagy, F. Otto: *CD-Systems of Stateless Deterministic R(1)-Automata Governed by an External Pushdown Store,* RAIRO - Theoretical Informatics and Applications, RAIRO-ITA 45 (2011), 413–448.

B. Nagy, F. Otto, M. Vollweiler: *Pushdown Automata with Translucent Pushdown Symbols,* 3rd International Workshop Non-Classical Models of Automata and Applications (NCMA 2011), Milan, Italy, Österreichischen Computer Gesellschaft, book@ocg.at, 193-208.

B. Nagy, F. Otto: *On CD-systems of stateless deterministic R-automata with window size one,* Journal of Computer and System Sciences 78 (2012), 780-806.

R. Schott, J.-C. Spehner: *Two optimal parallel algorithms on the commutation class of a word,* Theoretical Computer Science 324 (2004), 107-131.

W. Zielonka: *Safe executions of recognizable trace languages by asynchronous automata*, Logic at Botik '89, Symposium on Logical Foundations of Computer Science, LNCS 363 (1989), 278-289.

5. fejezet - Petri nets

5.1. Introduction

The theory of Petri nets is started by the PhD dissertation of Carl Adam Petri who worked at the University of Bonn and the dissertation was submitted to the University of Darmstadt in 1962 on the topic of mathematical description of concurrent systems. The big advantage of these systems that they give a nice graphical representation, and thus, they help and helped generations of engineers and other scientists to understand and learn the analysis of parallel and concurrent systems. The model shows both the static and dynamic properties of these systems. In the past years several researchers worked on this particular field and proved and showed the wide applicability of these systems. Nowadays besides/instead of the Neumann type architecture computers parallel architectures have been appeared and used, such as multicore processors, multiprocessor systems, networks, GRIDs, Internet, cloud computing. By this reason the parallel algorithms and the methods modelling them became essential. In this way the Petri nets and their various versions are in foreground both in education and research. In the next sections of this book we start form the simplest, elementary nets to more complex Petri nets and show various models and give some important facts about them.

The Petri nets are widely applied in the following fields: performance evaluation of systems, analysis of communication protocols, modelling and analysis of distributed software systems, distributed database systems, concurrent and parallel programs, dataflow computations, flexible manufacturing systems, multiprocessor systems, asynchronous systems, formal languages, logical programs, networks, digital filters, neural nets, decision supporting/making systems, economic modelling, etc.

For the (mathematical and computational) analysis of Petri nets there are various methods, e.g., algebraic methods, the state equation, that is connected to integer valued linear programming; the place-invariant and transition-invariant methods; the structural analysis, the state-space analysis, the reachability graph analysis; and various reduction techniques. To check the validity of a model, temporal logics can also be used, similarly to concurrent and parallel programming. In this book, because of lack of space, only the basic concepts and some important results are provided.

5.2. Binary Petri nets

In this section we show the simplest Petri nets. First we define these, so-called elementary or binary Petri nets in a formal way.

Definition: A Petri net is a directed bipartite graph ($G = (N,E)$) with the following properties. The nodes (vertices) of the graph are the places (P, denoted by circles and/or ellipses in figures) and the transitions (T, denoted by rectangles in figures), formally $P \cup T = N$ (where $P \cap T = \{\}$). The edges of the (di)graph $E \subseteq P \times T \cup T \times P$, i.e., they may go from places to transitions and from transitions to places. It is usual to define the neighbourhood of a node n by the nodes that are directly connected to the node n, formally:

$\bullet t = \{p \mid (p,t) \in E\}$, i.e., the preconditions of transition t are the input places of t, i.e., the places from where there is an edge to t;

$t\bullet = \{p \mid (t,p) \in E\}$, i.e., the postconditions of the transition t are the output places of the transition t;

$\bullet p = \{t \mid (t,p) \in E\}$, i.e., the input transitions of p are the transitions from which there are edges to p;

$p\bullet = \{t \mid (p,t) \in E\}$, i.e., the output transitions of p are the transitions having in-edge from p.

A transition t is called a source transition if $\bullet t = \{\}$; and transition t is called a sink transition if $t\bullet$; $= \{\}$.

The semantics of a binary Petri net is the following:

The configuration (global state or marking) of a binary Petri net is defined as a function of the form $P \rightarrow \{0,1\}$. Thus the configuration of an elementary Petri net can be described as a binary vector of dimension $|P|$: the elements of the vector are assigned to the places of the system in a bijective way. The value 1 in the vector means that the given place has a token, while 0 means that the given place has not any token. Usually the initial configuration is given by the Petri net, i.e., it is given which places having tokens and which have not, initially. Then the dynamics of the system can be given along the discrete time scale as follows: we say that transition *t* is enabled (in the given configuration of the system), if each of the places of $\cdot t$ having a token and none of the places of $t\cdot$ having any tokens. There can be several enabled transitions at the same time, and it can happen that none of the transitions are enabled in a given configuration. Then the development (the change of the global state) of the system is given as follows: let us choose (in a non-deterministic manner) an enabled transition *t*. Let the tokens be deleted from its input places and let a token appear in each out place of *t*. This step is called the firing (switching) of transition *t*. The new token distribution (the configuration) determines the enabled transitions, etc. The steps that the system makes one after the other define the so-called firing sequences.

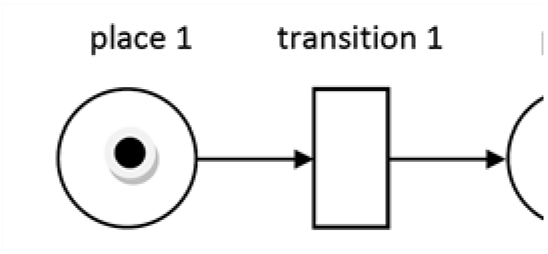
We note here that the source transitions are always enabled.

Typical examples for using Petri nets are the modelling of parallel/distributed/concurrent systems. We continue with some simple examples.

Example (binary Petri net with sequential run)

Let 3 places and 2 transitions in a Petri net graph be as it is shown in Figure 5.1. Then, in configuration (1,0,0) only the first transition is enabled, the second is not. After firing the first transition the configuration of the system will be (0,1,0). Now the first transition is not enabled, but the second one is enabled. In this way the two transitions are firing in a sequential manner. After the firing of the second transition the system enters to the (global) state (0,0,1) and there is no more enabled transition.

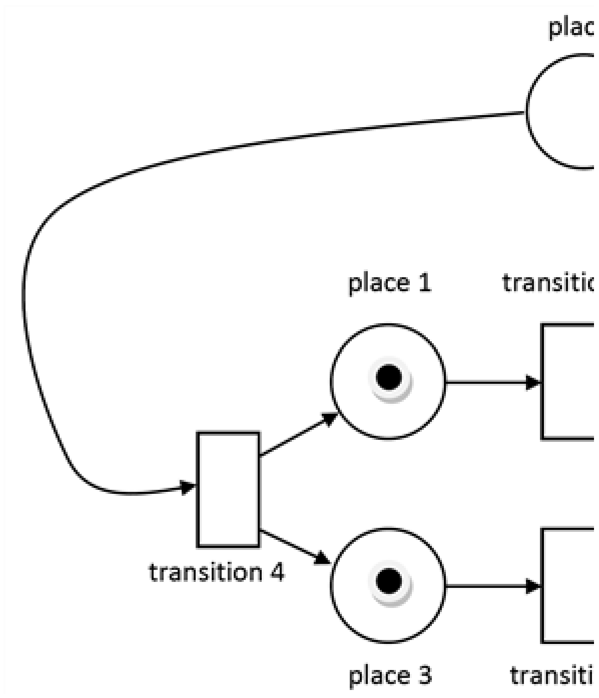
5.1. ábra - Petri net with sequential run with initial marking.



Example (binary Petri net with simultaneous run)

Let the graph and the initial marking of a Petri net be given as it is in Figure 5.2. Transitions 1 and 2 are enabled; they can be fired in any order (or in a parallel way). Until these two transition are fired, no other transition is enabled, and thus they are also synchronized. After both of these transitions have been fired, the marking will be (0,1,0,1,0) enabling transition 3. By firing transition 3 the only token of the system will be at place 5 by enabling only transition 4. After its firing the initial configuration (1,0,1,0,0) is obtained. This system models a deterministic parallel run: the parallel branches are activated by transition 4 and their run is finished by transition 3.

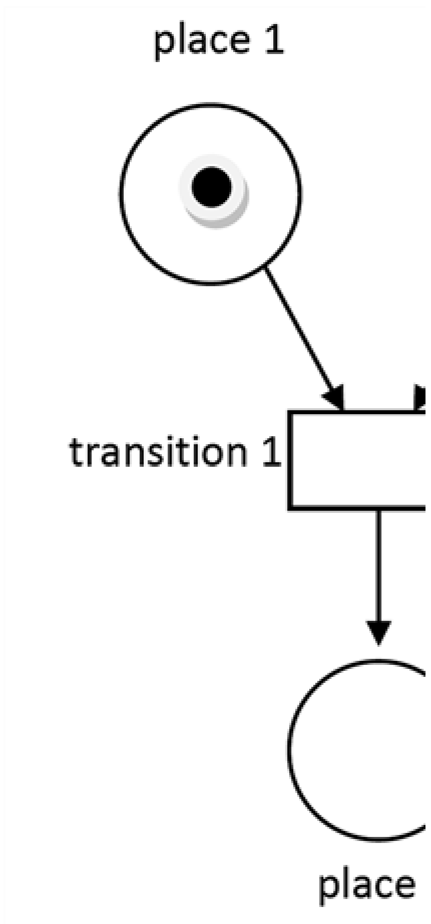
5.2. ábra - A parallel run in Petri net with initial marking.



Example (modelling conflict situation with binary Petri net)

Consider the Petri net that is given in Figure 5.3 (both its graph and its initial marking). Then both the transitions 1 and 2 are enabled. Whatever is chosen to fire, none of them will be enabled after the firing. These two transitions are in a conflict situation. It is not possible to fire both of them, only one (any) of them. This system models a non-deterministic choice.

5.3. ábra - Conflict situation, modelling a non-deterministic choice.



Typical interpretations of Petri nets are, for example, to see the input places, the transitions and the output places as data/signal, computing steps, output data/signals; or preconditions, events and postconditions; or needed resources, task/job and released resources; or conditions, clause in logic and conclusion(s), respectively.

There are some properties of the Petri nets that depend on the initial configuration, they are called behavioural properties. However there are properties that are independent of the initial marking, they depend only on the graph of the net, these properties are called structural properties.

Definitions: Let a Petri net and its initial marking be given. The system is contact-free if it has the following property. If every input place of transition t has a token, then t is enabled, i.e., in all these cases the condition that the output places of t are empty is automatically fulfilled.

Example (a contact-free binary Petri net and its work)

Let the system have 2 places and 3 transitions: transition A and B have input place "place 1" and output place "place 2"; transition C has input place "place 2" and output place "place 1". Let the initial marking be (1,0), i.e, the only token at place 1. According to this both transitions A and B are enabled. Any of them may fire resulting the configuration having a token only at place 2, and thus none of these two transitions are enabled after this firing. At the same time transition C become enabled, and after its firing the initial marking is obtained. In this example transitions A and B are concurrent to each other.

The next result holds for contact-free systems.

Theorem: Let a Petri net and its initial marking be given. Then a contact-free system can be constructed such that the possible states of this system have a bijective mapping to the states of the original system (and also the possible firing of these nets have a bijective mapping to each other). We say that the two systems are configuration-equivalent.

In connection to Petri nets the following questions and problems are arisen and we can solve/answer them (we can analyse the following properties of the modelled systems by the help of Petri nets):

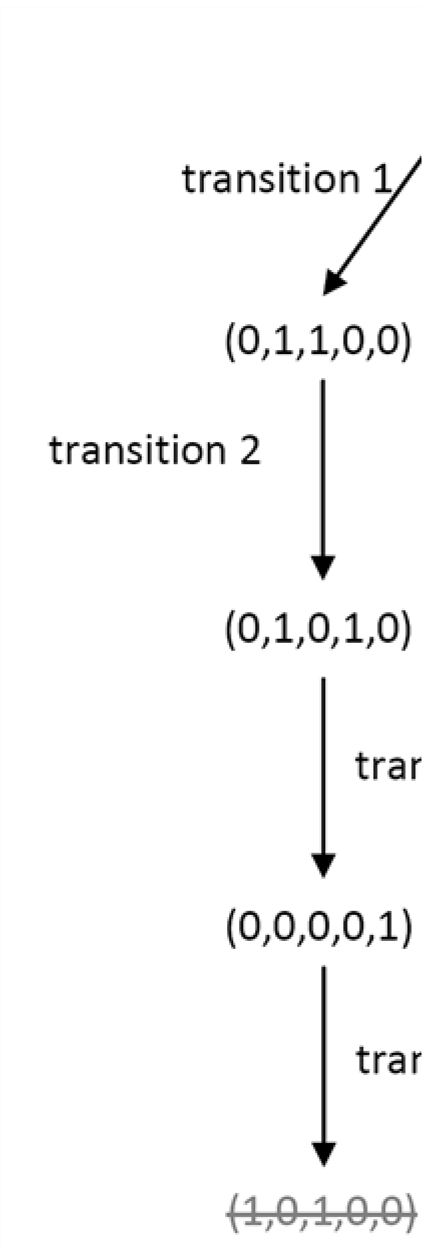
Definition: The reachability problem: can a given marking be obtained from the initial marking by any firing sequence?

The covering problem: can a marking be obtained from the initial marking by any firing sequence such that all given places have tokens (all other places are not important in this problem, some of them may have or may not have any tokens).

These problems can be solved using the concept reachability tree. If the initial marking of a Petri net is given, then we can construct the tree (graph) such that its nodes are the possible markings of the net, and its edges show how the system develops from a marking to another one by switching an enabled transition (thus transitions are assigned to the edges). The root of the tree is the initial marking. Since there are finitely many enabled transitions in every configuration of the system, a node can have only finitely many children node. It is enough to put only those markings into the tree that were not there already. Since, at the case of binary nets, the number of possible markings of as net is finite, the reachability tree is obtained after a finitely many steps and the solution of the above problems can be read out from this tree.

As an example, let us see the reachability tree of the Petri net shown in Figure 5.2. Figure 5.4 shows the tree (we have presented the states that are already in the tree in a crossed way and with grey colour).

5.4. ábra - Example for reachability graph.



Definition: The equivalence problem: there are two Petri nets given (with the same number of places, in an appropriately fixed order) with their initial markings, are the sets of reachable markings (vectors) are the same?

The containment problem: there are two Petri nets given (with the same number of places) with their initial configurations, does the set of possible states (vectors) of the second system contain the set of possible states of the first system?

By the help of the reachability tree all the possible markings of a system can be described and thus, by doing it for both systems, these problems can be solved.

The liveness problem, that is closely connected to, e.g., the deadlock-freeness property of operating systems, is the following:

Definition: There are various degrees of the liveness properties. Let a Petri net and its initial marking be given:

the transition *t* is

- (L0) dead if it is not element of any firing sequence.

- (L1) potentially firable if there is a firing sequence that contains it.

- (L2) arbitrarily often firable if for every positive value of *k*, there is a firing sequence such that *t* is enabled at least k times.

- (L3) infinitely often firing if there is an (infinite) firing sequence that contains *t* infinitely many times.

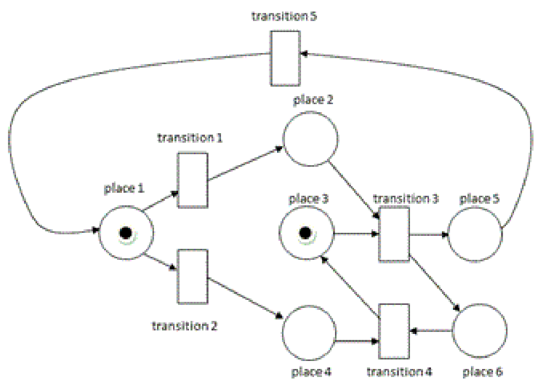
- (L4) always firable (or simply live) if for every reachable marking there is a firing sequence such that in the end of this sequence *t* is enabled.

A Petri net is always firable/infinitely often firing/arbitrarily often firable/potentially firable if it holds for every of its transitions, respectively.

It can be seen that among the liveness properties the last one, the always firable, fulfils the strongest condition. Further, the following hierarchy of these properties can easily be checked: every always firable transition/Petri net is also infinitely often firing. Every infinitely often firing transition/Petri net is also arbitrarily often firable. Further, every arbitrarily often firable transition/Petri net is also potentially firable transition/Petri net.

Example (potentially firable Petri net)

Let the Petri net be given with the initial marking as it is shown in Animation 5.1. Then there can be two (maximal long) firing sequence and both of them are shown in the animation. Both of them are of finite length. The first sequence contains all the transitions: 13524. Then, we can say that every transition is potentially firable, and thus the Petri net itself so. Since we have only finitely many (finite) firing sequences, there is no arbitrarily often firable transition. Thus the Petri net itself is not potentially firable; and conditions (L3) and (L4) are also not fulfilled (by any transitions and by the net itself).



Animation 5.1: The dynamics of a non arbitrarily often firable, but potentially firable Petri net.

Definition: The reversibility problem: a Petri net is reversible if for every reachable marking (that can be reached from the initial marking) there is a firing sequence such that it leads to the initial marking, and thus the net is able to work in a cyclic way.

To solve the reversibility problem one can use a similar method to the method to compute the reachability tree. Since the number of possible markings of the system is finite, this question is solvable.

Our previous example is not reversible as we could observe from the animation: it cannot work in a cyclic way. The Petri net shown in Figure 5.2 is reversible. It can also be seen by continuing the algorithm that results the reachability tree: as it is shown by grey colour in Figure 5.4, the initial marking can be reached wherever we are in the tree.

By the analysis of the persistence property, one may check whether the parallel branches has some influence on each other.

Definition: A Petri net is persistent if for any two enabled transitions, the firing of any of them does not disbale the other one.

In a persistent Petri net, if a transition t is enabled, then it remains enabled until it fires.

By checking the fairness property one can decide whether the parallel events of the system can fire without firing the others. A system is fair if every transition will fire before or after.

Definition: Two transitions are in the bounded fair relation if one can fire only in a bounded number without firing the other. A Petri net is bounded fair if any two of its transitions are in the bounded fair relation.

There is another fairness concept, the so-called global/unconditional fairness.

Definition: A firing sequence is globally (or unconditionally) fair if it is finite, or every transition occurs infinitely many times in it. A Petri net is globally fair if its every firing sequence is globally fair.

The two types of fairness concepts are not independent of each other. In case of binary nets they are equivalent:

Theorem: An elementary Petri net is globally fair if and only if it is bounded fair.

By the structure of the graphs of the Petri nets we can define the following properties.

Definition: A Petri net is cycle-free if its graph is cycle-free.

A Petri net is conflict-free if for every of its places *p*, either the number of its output transitions is at most one, or each output-transition *t* of the place *p* has *p* as output place (i.e., *p*-*t*-*p* is a directed cycle in the graph). In these Petri nets every place having more than one out edges is the output place of every of its output transition.

A Petri net is communication-free if its every transition has exactly one input place. These Petri nets are also called BPP nets.

A Petri net is conservative if the number of its tokens is constant (it is if and only if every transition needs the same number of tokens from its input places as it produces at its output places).

The Petri nets having one or more properties of the previous definition form special subclasses, and thus some of the problems can be solved more easily for them. For example, the cycle-free Petri nets are not reversible, moreover they can have at most potentially firable transitions (depending on the initial marking), transition that is arbitrarily often firable cannot be in these nets.

In fact, the binary Petri nets are closely connected to finite automata. The main difference is that in these Petri nets the marking may contain several "states" at the same time (in this way these nets are more related to non-deterministic finite automata). However the state-space (the possible markings) is finite for these Petri nets and therefore their analysis is relatively simple, the problems about them are usually solvable.

A subclass of the binary Petri nets is analogous to the finite automata regarding their work also.

Definition: The finite-state machines are Petri nets with the following properties: every transition has exactly one input place and exactly one output place, and the initial marking has exactly one token.

By the definition one can see that the finite-state machines are conservative BPP nets. In these very simple Petri nets the parallel phenomena (such as the synchronization) cannot be modelled, but, for instance, a finite state vending machine can be modelled: the system is in exactly one "state" at every time, i.e., there is exactly one token and it is in a well-defined place.

5.3. Place-transition nets

In a binary Petri net the number of the possible markings is finite, thus the expressing power of these systems is similar to the expressing power of finite automata (the regular languages/processes can be described by their help).

It is a widely used extended model that allows to have not only one, but any number (non-negative integer) of tokens in a place at the same time. In these nets it is also allowed to have a multiple connection between a place and the transition, i.e., weighted arcs can be used as follows. The weight w of an edge has the meaning that the number of tokens that move on the given edge at the same time is w: if the edge is from a place to a transition, then this number of tokens needed to fire the transition and if the transition fires the number of tokens will be decreased by w in this place; if this edge is from a transition to a place, then at the firing of the transition the number of tokens will increase by w on this output place. A transition is enabled if there is enough number of tokens at its each input place. It could also happen in these nets that some firing sequence increases the number of tokens in the net unlimitedly.

Definition: Let the Petri net ($G = (P\cup T,E)$) be given and let its edges have positive integer weights.

The work (semantics) of a place-transition Petri net is as follows:

The configuration of a place-transition net is defined as a function of the form $P\to\mathbb{N}$, accordingly a state of these Petri nets can be denoted by a $|P|$-dimensional vector with non-negative integer elements: the elements of the vector can be mapped to the places of the system in a bijective way. A value in a vector means the number of tokens of the assigned place. Usually an initial marking (initial vector with the initial number of tokens of the places) is assigned to the net. Then the work (dynamics) of the system is given by the discrete time line in the following way: the transition t is enabled (in a given state of the system) if every place of $\bullet t$ has at least as many tokens as the weight of the edge from this place to the transition t. The change of the state of the system can be given as follows: let us choose (non-deterministically) an enabled transition t. Let us erase as many tokens from every input place of t as the weight of the edge from this place to the transition t and let us put as many tokens to every output place of t as the weight of the edge from t to that place. The firing sequences can be then defined analogously to the case of binary Petri nets.

As one may observe from the definition, by allowing more than one tokens on the places the condition that the output places of the firing transition are empty is released. Later on we will see that in some cases it could be important to use such type of conditions.

In the following we will use weights that are different from 1, i.e., if we do not put a specific weight for an edge of the net, then it is used as an edge with a unit weight. Figure 5.5 shows a schematic place-transition net model how to assembly a bike.

5.5. ábra - A (simplified) model of bike assembly.

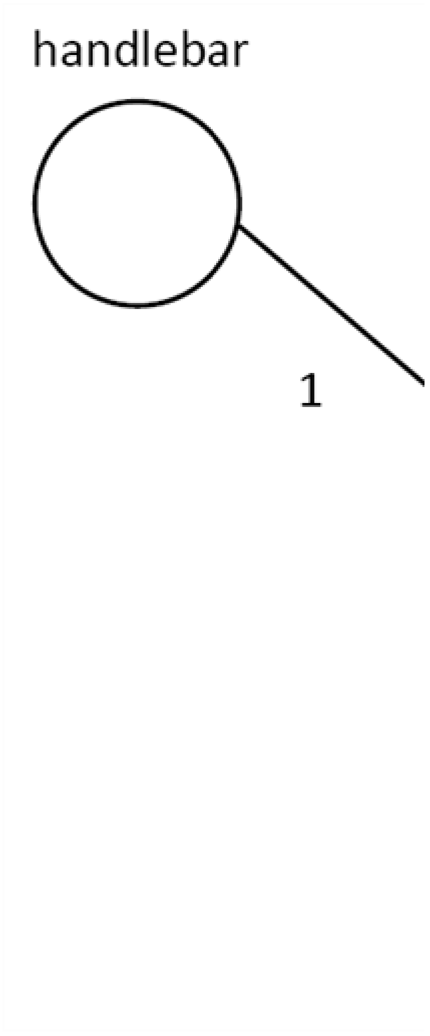
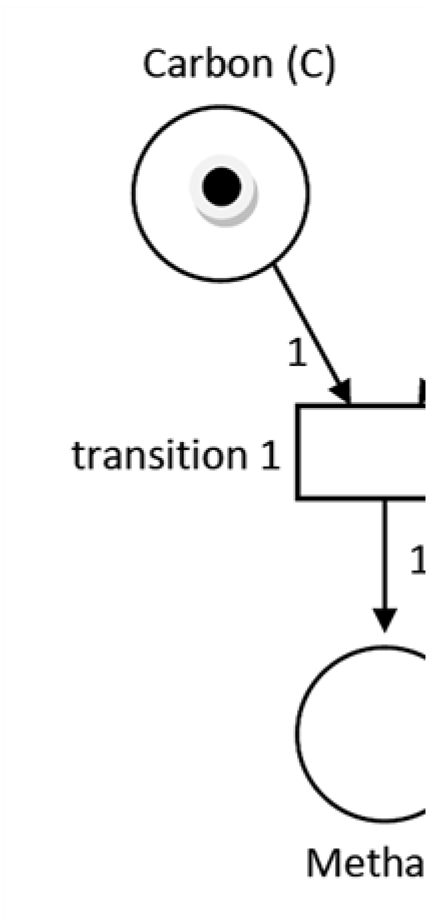


Figure 5.6 shows a place-transition net that describes chemical reactions: assembly of Water and Methane molecule from their elements.

5.6. ábra - Modelling chemical reactions.



At place-transition nets the same questions and problems are arisen as at binary nets. The reachability, the equivalence and the containment problems are defined in a similar way as there, such as the liveness properties. The covering problem can be redefined in the following way to place-transition nets:

Definition: Let a place-transition net and its initial marking and an additional marking be given. The covering problem is to decide whether a marking can be obtained from the initial marking such that it contains at least as many tokens in each place than the given additional marking.

Furthermore there is an important problem regarding the place-transition nets that was not a problem in binary nets:

Definition: The problem of boundedness is to decide if the number of reachable marking from the initial marking is finite or infinite.

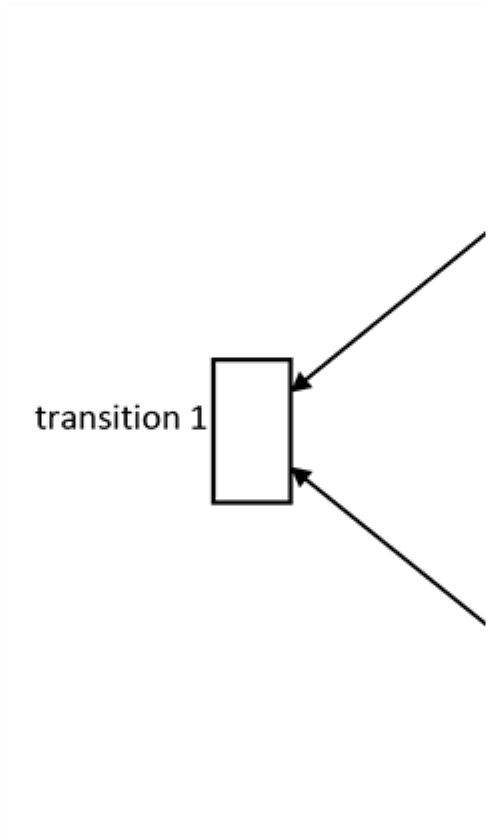
The cycle-free Petri nets are bounded. In case of cycle-free nets the reachability problem and other problems can easily be solved (not only in binary case, but in case of place-transition nets also).

Now we give an example for various degrees of the liveness property in a place-transition net.

Example (various liveness properties in a place-transition net)

Let the Petri net be given as it is shown in Figure 5.7 with its initial marking. (There are no weights shown, therefore each edge has a weight 1.) The initial marking is (1,0,0) and by using only transition 3 the marking (1,*n*,0) can be obtained for an arbitrary non-negative integer n. Here, transition 3 is infinitely often firing (L3) since it can fire infinitely many times. If in a time instant transition 2 fires, the vector (0,*n*,1) describes the state of the system. Thus, transition 2 is potentially firable (L1). In this marking only transition 4 is enabled and it can fire at most n times obtaining the marking (0,0,1). In this state of the system there is no enabled transition. In this way transition 4 is arbitrarily often firable (L2). To check all the possibilities, it is clear that transition 1 is dead (L0), because it cannot be enabled starting from the given initial marking. One can also check that no stronger liveness property is fulfilled by any of the transitions of the system than we have already described.

5.7. ábra - Various liveness properties in a place-transition net.



The place-transition net given in the previous example is not bounded, since in place 2 the number of tokens can increase unlimitedly.

In some cases the boundedness is required or we can restrict the work of the system for a bounded state space by the properties of the modelled problem.

Definition: Let a place-transition net be given with its initial marking. The net is called safe if it cannot happen that a place has more than one tokens.

The definition above seems to related to binary Petri nets, and so they are related by the fact that binary vectors can be used to describe markings. However there are essential differences between these types of systems: At binary nets a transition is enabled only if there is no token in its any output-places. This condition does not hold for place-transition nets, and thus, in safe nets by definition. In safe place-transition nets the structure and the initial marking is given in such a way that it cannot occur that more than one tokens have the same place, even if this condition is not forbidden by the definition of the place-transition nets:

Theorem: The safe place-transition nets are exactly the contact-free binary nets.

The concept of safeness can be generalised by allowing more than one, but only limited number of tokens at the same place in the system.

Definition: Let a place-transition net be given with its initial marking. The net is *k*-bounded if there is no reachable marking containing more than *k* tokens in any place. The place-transition systems that are *k*-bounded for some positive integer *k* are called bounded nets.

It is clear that the boundedness problem is closely connected to the bounded nets: the number of reachable marking is finite in exactly those nets that are bounded.

Now we show an algorithm that decides if a net is bounded with the given initial marking. The construction of the covering graph is a variation of the construction of the reachability tree such that it can be used in non-bounded place-transition Petri nets.

We will denote by the symbol ω that in a place there can be any number (it can be called unlimited or infinite) of tokens. According to this ω is greater than every natural number *n*, and furthermore $\omega+n = \omega$, $\omega-n = \omega$ and $\omega\geq\omega$.

The algorithm itself is a graph-searching algorithm from artificial intelligence.

Let us start from the initial marking, let it be the first (initial) node of the graph and let us mark that it is not yet analysed.

Until the graph has not analysed node let us do the following.

Let us choose one such node and look for the marking *M* assigned to this node. Let us consider every transition that are enabled at this marking.

If the resulted marking *M'* is already included in our graph, then we simply add a directed edge from the analysed node to the node having the obtained marking and let the label of this edge be the analysed transition.

If the obtained marking is not in our graph, then let us check the following: are there any node in the path from the initial node to the analysed node such that its marking *M*'' satisfies the following: the newly obtained marking *M*' has at least as many tokens in each place as the *M*'' has (i.e., *M*' covers *M*'') and *M*' and *M*'' are different. Then let us change every element of *M*' that is larger than the respective element of *M*'' to the symbol ω. Since *M*' has at least as many tokens in every place as *M''*, obviously every firing sequence that is allowed in *M*'' is allowed in the marking *M'*, and thus the number of tokens in the respective places can be increased unlimitedly.

Let the new node with the newly obtained (and possibly modified) marking *M*' be added to the graph as a new, not yet analysed node if it was not in the graph; and let the label of the edge from the analysed node (with marking *M*) to the new node be the transition that is used to obtain *M'*.

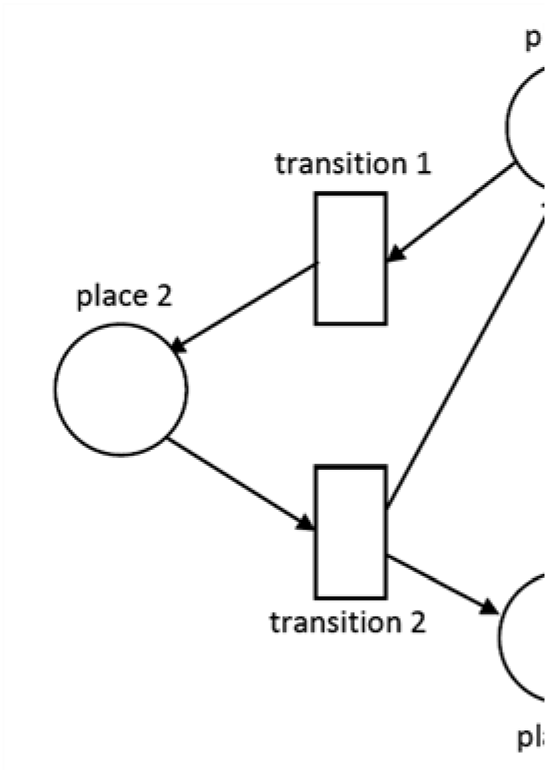
Finally, if every possible transition is analysed to the analysed node, let this node be already analysed and let us continue the algorithm by choosing another not yet analysed node (if there is any).

The algorithm stops after finitely many steps and provides the covering graph. If the symbol ω appears in the graph, then the net is not bounded with the given initial marking; if ω does not occur in any of the vectors assigned to the nodes of the graph, then the analysed net is bounded. In bounded case the largest value that occurs in the vectors of the nodes also gives the value *k* such that the net is *k*-bounded (for this value and all the largest values). If the largest value that occurs in the vectors assigned to the nodes is one, then the net is safe. Furthermore, exactly those transitions are dead that do not occur in the graph as labels of any edge.

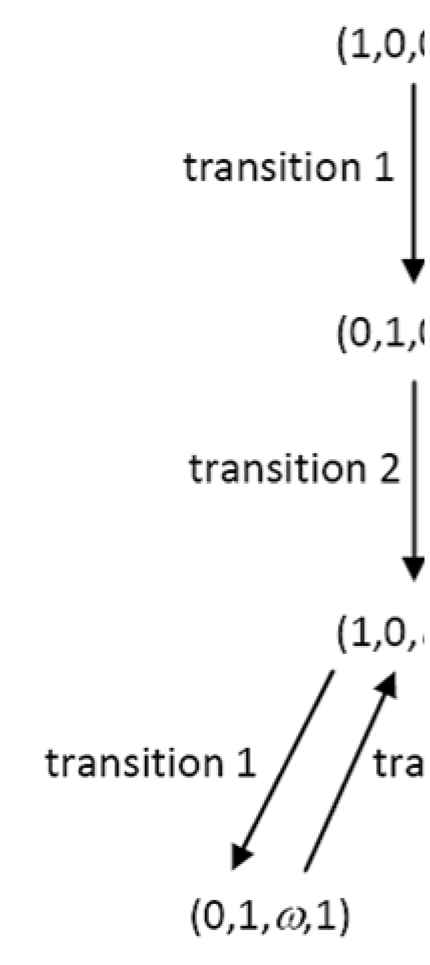
Example (a net with infinitely many reachable markings)

Let the place-transition net be given as it is shown in Figure 5.8 with the shown initial marking (every edge has a unit weight). Then let us construct the covering graph. The initial marking is (1,0,0,1), this is the initial node of the graph. This is the only not analysed node, therefore we choose it; and there is only one enabled transition (transition 1) in this marking. After firing this transition we obtain the marking (0,1,0,1) and store it in the graph as a new, not yet analysed node. At this state of the system only transition 2 is enabled, after its firing the marking (1,0,1,1) is obtained. This is not in our graph yet, but it covers the initial marking: at place 3 the number of tokens is increased, thus the modified marking (1,0,ω,1) is added to the graph. This is the only not yet analysed node, and transitions 1 and 3 are enabled in this marking. By firing transition 1 the marking (0,1,ω,1) is reached and it is added to the graph. By firing transition 3, the marking (0,0,ω,1) is obtained and it is also added to the graph. Now there are two not yet analysed nodes. Let us choose the one with marking (0,1,ω,1). Only transition 2 is enabled obtaining the marking (1,0,ω,1), but this is already in the graph, thus we add only an edge to the graph showing that we can reach that node from the currently analysed node. The only one not yet analysed node has marking (0,0,ω,1); there is no enabled transition here, no new node is obtained. Figure 5.9 shows the obtained covering graph. Based on this graph we can say that this net is bounded.

5.8. ábra - A place-transition Petri net with its initial marking.



5.9. ábra - Covering graph for the net given in Figure 5.8.



Based on the covering graph the covering problem can also be easily answered.

Definition: A place-transition net is structurally bounded if it is bounded with every possible initial marking.

It can easily be proven that the conservative Petri nets are structurally bounded.

In other way, without having and proving this restriction by the structure, we may explicitly give this condition into the definition introducing a new class of Petri nets. In Petri nets, based on real information systems, the number of tokens can be limited in each place: we may explicitly give the maximal number of tokens that are allowed to occur at the same time in each place. This condition is trivially fulfilled by binary Petri nets.

Definition: Let a place-transition net be given with capacity bounds on some places. Let, for instance, at place p the given bound be *k*. In the net only such firing sequences are allowed that obtain markings with the condition that at place *p* there are not more tokens than *k*. These nets are called capacity nets.

If there is a capacity bound (a positive integer) for each place of the net, then we say that the whole net is capacity bounded (or the net is of finite capacity).

Those nets that have only a limited number of tokens during their work (because, for instance, they are bounded or capacity bounded) can be analysed by the reachability tree shown for the binary nets, since there are only finitely many reachable markings, and therefore the reachability tree can be constructed and it can be used to answer the arisen questions/problems.

Every bounded fair place -transition net is globally fair. Its converse holds for every bounded place-transition net, that is, every globally fair bounded place-transition net is bounded fair.

The equivalence and the containment problems can be solved for those nets in which there are only finitely many reachable markings. In the general case, these questions are more difficult. Usually there are infinitely many reachable markings and therefore the following result can be obtained.

Theorem: The equivalence and the containment problems are not algorithmically solvable for place-transition nets in the general case.

The proof of the thorem is very hard, it can be done by using Hilbert's 10th problem.

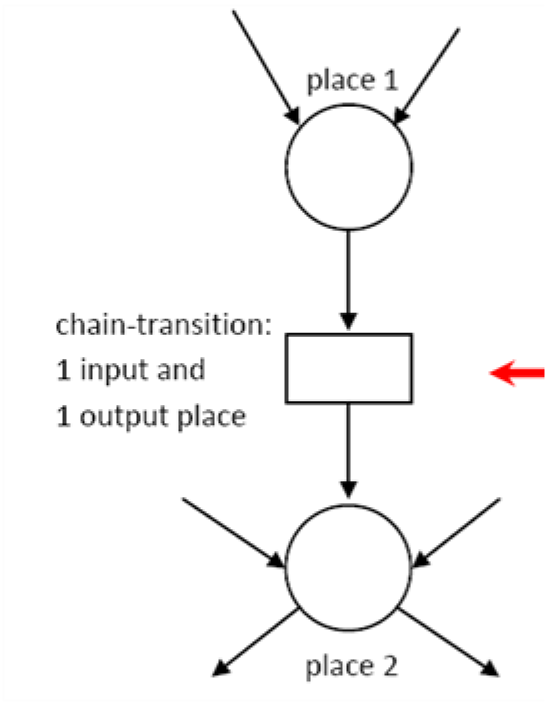
At Petri nets with some special restrictions positive results can be proven for these problems.

5.3.5.3.1. Reduction techniques

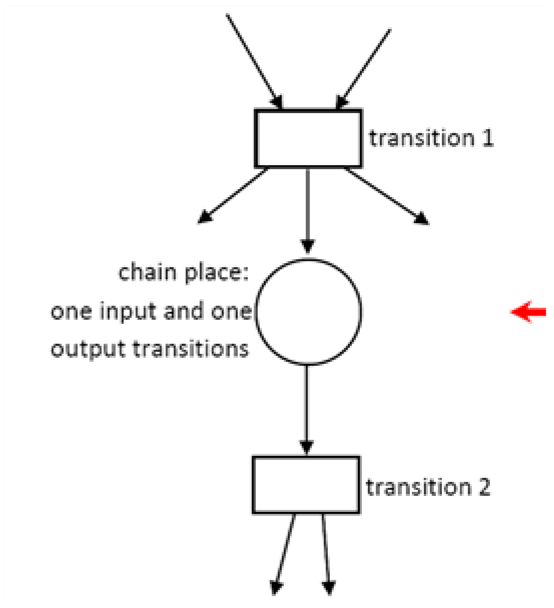
For the analysis of place-transition nets such reduction steps are used several times that change the net without modifying the liveness, the boundedness and the safety properties. By the help of these steps the net can be simplified and these questions can be answered by analysing a simpler net. Now we show some reduction steps. In Figures 5.10-15 the left side shows a (part of the) net that can be reduced to the net shown in the right side (the above mentioned properties are the same for the nets shown in both sides).

In Figure 5.10 we show how places can be united if they were connected by a transition having exactly one input and one output place. The reduction goes by erasing this, so-called chain transition; it is important that this transition is the only output transition of place 1 in the left hand side. (This reduction step may also be called fusion of series places.) In a similar manner, as it is shown in Figure 5.11 we can unite such transitions that are connected by a place having exactly one input and exactly one output transition. In this case, the so-called chain place can be erased. In this case it is important that transition 2 has only one input place, the chain place, in the original net (left hand side). This step can be viewed as a fusion of series transitions.

5.10. ábra - Eliminating chain-transition.

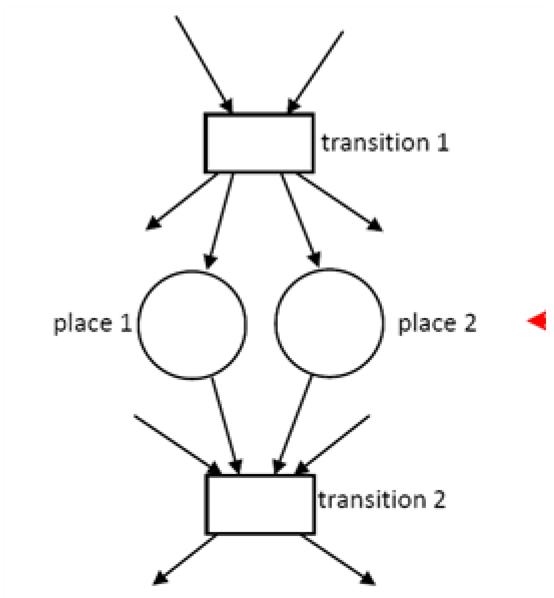


5.11. ábra - Eliminating chain place.

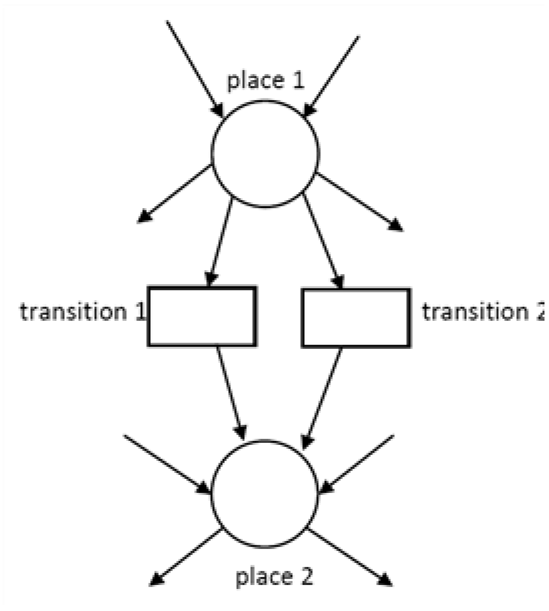


Figures 5.12 shows a reduction step in which there are two places such that their input transition is the same transition and their output transition is the same transition (they both have exactly one input and exactly one output transition). These, so-called double places can be united as it is shown on the right side of Figure 5.12. Figure 5.13 shows an analogous reduction step in which there are two transitions such that their exactly one input place is the same and their exactly one output place is the same. Figure 5.13 on the right side shows the united transition instead of the double transition. These steps can be seen as fusion of parallel places/transitions.

5.12. ábra - Reduction of double places.

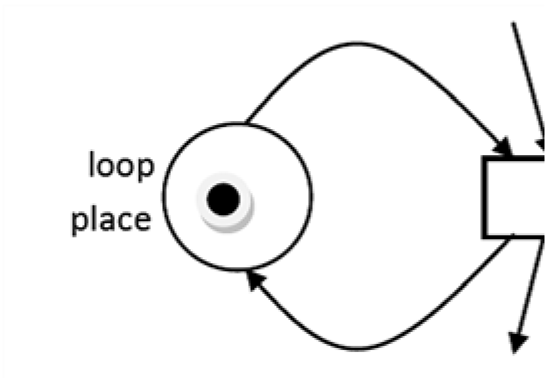


5.13. ábra - Reduction of double transitions.

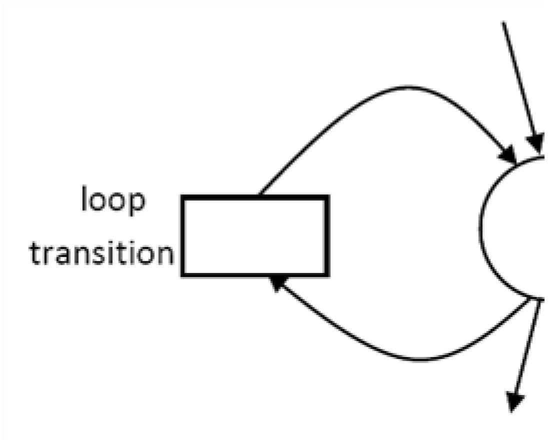


Figures 5.14 and 5.15 shows how we can eliminate a (self-)loop place (with the condition that it has a token) and eliminate a (self-)loop transition. A (self-)loop place has exactly one input and exactly one output transition, moreover they are coincide. Analogously, a (self-)loop transition has exactly one input and exactly one output place and they are the same.

5.14. ábra - Eliminating loop place.



5.15. ábra - Eliminating loop transition.



The place-transition nets are closely connected to vector addition systems. They are equivalent systems in a mathematical way. Based on this fact, some formalism used in vector-addition systems can be used to prove some properties of the place-transition nets. Here, because the lack of space, we do not detail these algebraic and related methods.

5.3.5.3.2. Special place-transition nets

So far we have already mentioned some special Petri nets, such as models with finite state space. In this subsection we recall some classes of nets with special properties that are widely used in practice.

5.3.5.3.2.5.3.2.1. Marked graph model

The finite-state machines were already mentioned at the binary Petri nets. The nets called marked graphs are analogous to them in a certain sense.

Definition: A place-transition net is a marked graph (net) if every of its edges has a unit weight and every place has exactly one input and exactly one output transition.

There cannot be a conflict situation in marked graphs. Their name comes from an alternative graphical representation: they can be viewed as directed graphs where the edges represent the places, and so the tokens are assigned to the edges, while the nodes are the transitions. In this representation the firing of a transition is implemented by decreasing the number of tokens by one on its every input edges and increasing the number of tokens by one in each of its output edges.

In marked graphs the directed cycles play important role regarding the problems we are interested. A given directed cycle has exactly one input and exactly one output edge at each of its nodes. Based on this fact we can say that the number of tokens in each directed cycle of the net is constant. This token-invariant property helps to obtain a result about (L4) liveness of these nets.

Theorem: A marked graph with a given initial marking has the (L4) liveness property if and only if every directed cycle of the graph has at least one token in the initial marking.

In the next subsection we show a more general case.

5.3.5.3.2.5.3.2.2. Free-choice nets

The previously shown marked graphs are very special Petri nets. The free-choice nets are the generalisations of the finite-state machines and the marked graphs, but still special models. Formally we defined them as follows.

Definition: A place-transition net is a free-choice net, if each edge has a unit weight and for each edge from a place to a transition fulfils at least one of the following conditions:

- this is the only edge from this place,

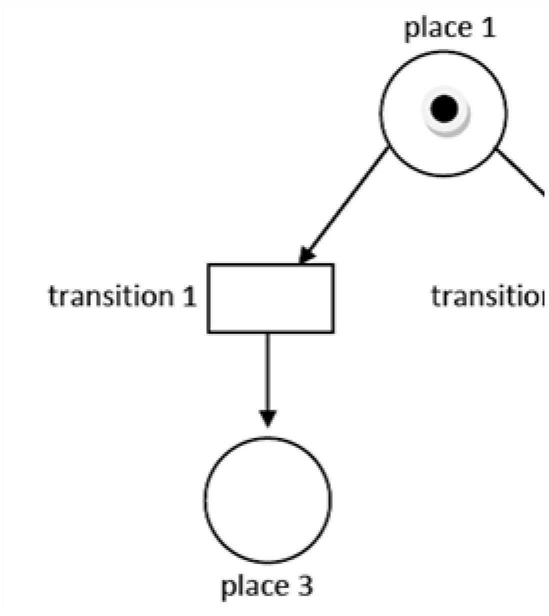
- this is the only edge to that transition.

The condition written in the definition can equivalently be formed as follows: if the sets of output transitions of two distinct places are not disjoint (i.e., there are two input edges to a transition), then each of these two places has exactly one output transition (there is only one outedge from each of these places) and thus this output transition is the same for these places. There is further generalised concept of nets, the extended free-choice nets.

Definition: A place-transition net with unit weights is an extended free-choice net if the following condition holds. If any two transitions have common input place(s), then the set of input places are the same for these two transitions.

The concept of (extended) free-choice nets do not allow the so-called (symmetric) confusion that can be seen in Figure 5.16: in this situation transition 1 and 3 are concurrent (and they can be firing simultaneously), while both of them in a conflict situation with transition 2.

5.16. ábra - (symmetric) confusion.



In the analysis of free-choice nets the concepts siphon and trap play essential roles.

Definition: In the Petri-net ($P \cup T, E$) with unit weight the subset P' of places is called a siphon (or deadlock) if the set of its input transitions is a subset of the set of its output transitions.

In this net the subset P'' of the places is a trap if the set of its output transitions is a subset of the set of its input transitions.

It is clear that for a siphon, if there is no token in any of its places, then after any possible firing sequence the siphon remains without any tokens. Analogously, for a trap the next statement holds: if any of its places has a token, then after any possible firing sequence the trap still has at least one token. It can easily be seen that any union of siphons is also a siphon. A siphon/trap is a basic siphon/trap if it cannot be obtained as the union of other siphons/traps. Every siphon/trap can be obtained as a union of basic siphons/traps. A siphon/trap is called minimal siphon/trap if it does not contain other siphon/trap (as a subset). The minimal siphons/traps are basic siphons/traps, but, in contrary, there are basic siphons/traps that are not minimal.

Theorem: An (extended) free-choice net is (L4) live if and only if every of its siphons contain a trap that has a token.

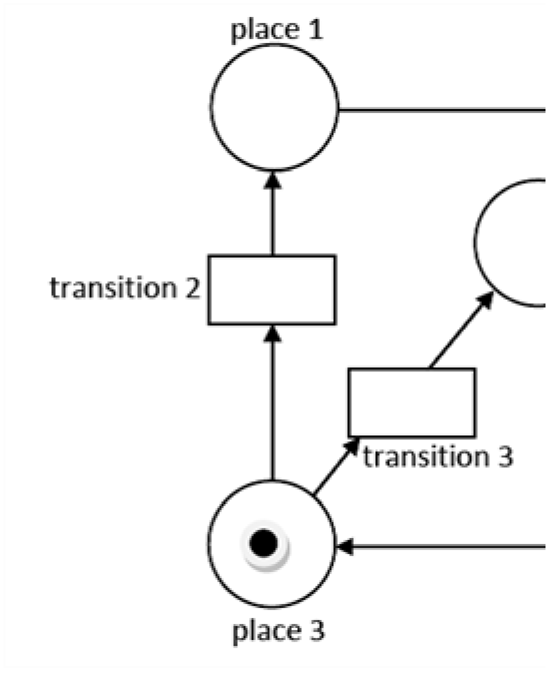
Let us see an example.

Example (determining siphons and traps)

Let the free-choice net be given as it is shown in Figure 5.17. Then, it can be checked that the subset {2,3,4} of places is a siphon, because the input transitions of these places are transitions 2, 3 and 4, and the output transitions of these places consists of all the four transitions of the net. Consider the subset {2,4} of places. The set of its input transitions contains transitions 1, 3 and 4. The set of output places of this set of places contain the transitions 1 and 4. Therefore the set {2,4} of places is a trap. Similarly it can be shown that the set {1,3,4} of places is a siphon. The set {1,2,3,4} is a siphon and a trap at the same time. The set {1,2,4} of places is also a trap. Further, the sets {2,3,4} and {1,3,4} are minimal siphons, and thus, basic siphons. All the sets {2,4}, {1,2,3,4} and {1,2,4} are basic traps, but only {2,4} is a minimal trap.

In this example the siphon {1,3,4} does not contain any trap, and thus, this net is not (L4) live for any initial marking.

5.17. ábra - A free-choice net.



5.4. Further Petri net models

The Petri nets we dealt with so far have less expressive power than the Turing machines have. Therefore there are systems that are given in a mathematically adequate description such that they cannot be modelled by the previous Petri nets (only semi-linear systems can be modelled by the previous nets: only systems that can be described by sets that are defined by Presburger arithmetic). To solve this problem several more complex models of Petri nets are created, e.g., coloured Petri nets, Petri nets with inhibitor arcs, timed Petri nets, priority Petri nets. The lack of the previous (infinite state) systems, in fact, that they cannot provide zero-tests, i.e., a transition cannot be enabled by the emptiness of a place. If this condition can be solved in a net, then it is easy to model a two-counter machine (that is a Turing equivalent universal computing machine). Unfortunately by extending the expressive power in this way, we (obviously) obtain such systems that the most of the important problems/questions about the analysis of these nets become algorithmically undecidable in general. In the next part we briefly recall the extended systems we have already mentioned.

5.4.5.4.1. Coloured Petri nets

In all the previously described Petri nets the tokens were identical (such as the electrons in physics). It looks obvious to generalise the nets by allowing various types of tokens. In graphical representations usually colours are used to mark the differences among tokens, the name of these nets comes from this fact. In coloured Petri nets the tokens may contain data (e.g., colour), and these data may change during firing a transition. Sentinels can be used to give further conditions for the enable transitions using also the data of the actual tokens.

The expressive power of the coloured Petri nets is the same as the expressive power of the Turing machines.

There are systems such that the tokens use unique identifiers, and thus the number of "colours" can be unlimited in these systems.

5.4.5.4.2. Inhibitor arcs in Petri nets

As we already mentioned, at binary nets the emptiness of the output places is explicitly needed for a transition to be enabled. This condition was relaxed at place- transition nets. However, in practice, in several modelling problems, it is a real requirement that an event cannot be happen in case of some conditions are fulfilled. The inhibitor arcs give special tools for checking some conditions in these nets.

The inhibitor arcs are special edges that give enabling condition from the given (input) place if there are not enough tokens on that place. In this way a transition that has also inhibitor arcs is enabled if the weight of every of its input (normal) edges is at most as many the number of tokens at the input place from where this edge is coming and for every inhibitor arc coming to the transition the number of tokens of the given input place is less than the weight of the inhibitor arc.

In most cases it is enough to restrict the usage of inhibitor arcs to arcs with unit weight. By using inhibitor arcs it is easy to give priorities to some transitions over others.

By the help of inhibitor arcs the zero-tests can be done directly, and thus, these nets are Turing universal.

5.4.5.4.3. Timed Petri nets

At timed Petri nets time is given to the system in an explicit way. The firing of the transitions in these models does not go in a time instant, but takes time. In several applications these models can be used, e.g., in timing problems. The expressive power of timed Petri net is also the same as the expressive power of the Turing machines.

5.4.5.4.4. Priority Petri nets

The non-deterministic work of the Petri nets can be restricted by using priority relation among the transitions. In a priority Petri net a transition with lower priority cannot fire until there is an enabled higher priority transition. (The firing order among the enabled transitions with the same priority is still non-deterministic.) The priority relation can determine the order of the firing transitions in a possible firing sequence, but it may also make impossible some firing sequences.

The priority Petri nets are Turing equivalent.

5.4.5.4.5. Other variations of Petri nets

There are several other extensions of the traditional Petri nets known. In this subsection we recall some of them without the sake of completeness.

At the Turing equivalent models shown in this section previously, the most of the important problems/questions are algorithmically not solvable: they can be traced back to the halting problem of Turing machines. Therefore the demand is arisen to develop Petri nets having less expressive power than the Turing equivalent models, but it is larger than the expressive power of the traditional place -transition nets. These systems, for example, are the reset nets and the transfer nets.

5.4.5.4.5.5.4.5.1. Reset nets

The reset nets have special reset edges. When a transition is firing with an out reset edge, then the number of tokens at the place where this reset edge goes became zero.

5.4.5.4.5.5.4.5.2. Transfer nets

In the transfer nets there are special transfer edges: these are going from a place through a transition to a place. When a transition on a transfer edge is firing, first all tokens are removed that give enability to the transition, then all the remaining tokens will be moved from the input places to the output places by the transfer edges, and finally, as usual, the output tokens will be given to the output place by the out edges of the transition.

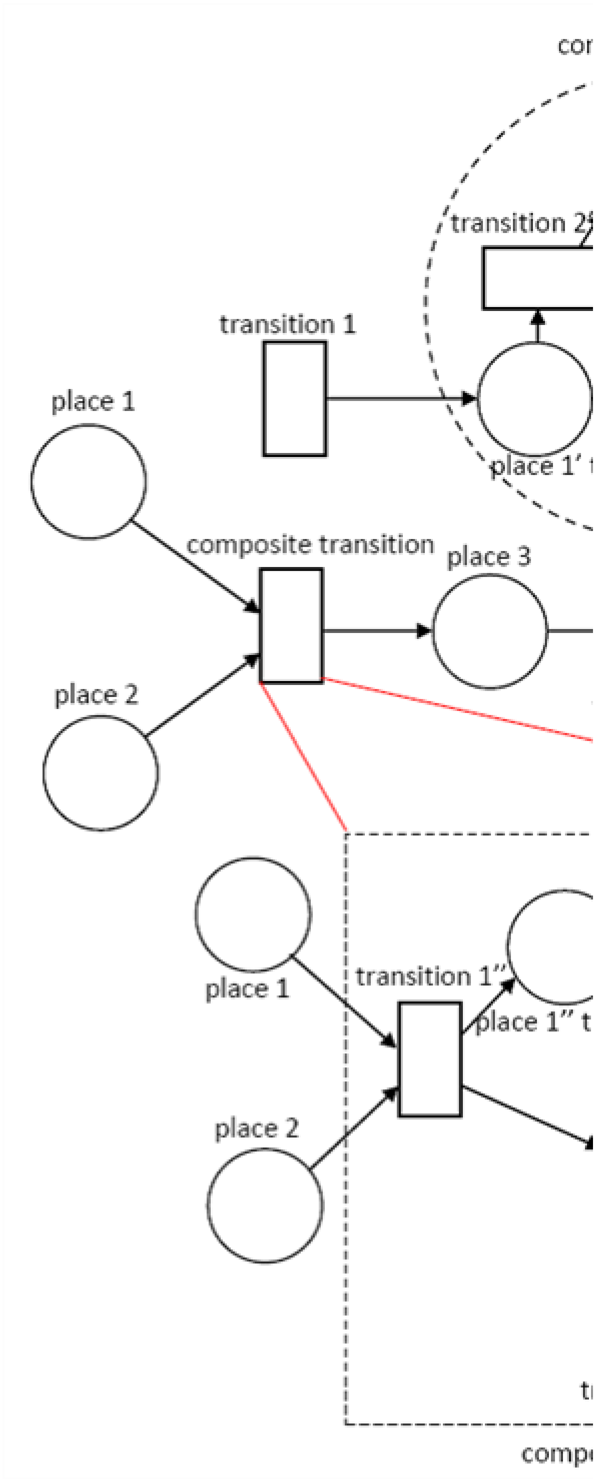
5.4.5.4.5.5.4.5.3. Dual nets

The role of places and transitions can be used in a unified global way in dual Petri nets. These systems allow to check the static and dynamical properties in the same manner. The system can be seen as it develops from given states (marking) to other ones by firing transitions. Alternatively, it can be seen as follows: by firing of given transitions it develops through on some markings (where tokens are on places) to the "firing (state)" of some other transitions.

5.4.5.4.5.5.4.5.4. Hierarchy nets

The hierarchy Petri nets are built up in a hierarchic way, e.g., some transitions can mean subnets. An example is shown in Figure 5.18. This technique can be very useful in practical modelling. As it can be seen in our example both places and transitions can be built up by subnets.

5.18. ábra - A hierarchic model having subnets for a place and a transition.



5.4.5.4.5.5.4.5.5. Stochastic, continuous, hybrid, fuzzy and OO Petri nets

In the stochastic Petri nets, similarly to the timed Petri nets the firing needs time, but in these models these time intervals are not deterministically given, but they can be given by probabilities. These nets can be closely related to Markovian chains.

There are continuous and hybrid Petri nets that are widely used in control theory. The mixture of fuzzy theory and the theory of Petri nets leads to the field of fuzzy Petri nets (allowing, e.g., fuzzy tokens).

For the object oriented programming philosophy the modelling can be done by OO Petri nets. In these systems every class includes an object net that describes the inner activities of the objects, e.g., how they react receiving messages.

5.4.5.5. Petri net languages

Formal languages can also be defined by Petri nets, and also the work of a Petri net can be described formally by a formal language. Considering the set of transitions as an alphabet, a firing sequence is a word over this alphabet. Other, more general possibility is to say explicitly which letter is assigned to a given transition.

All possible firing sequence from an initial marking defines a formal language. The Petri net languages are closely connected to commutations, and obviously to traces and trace languages.

It is usual to give a goal marking that can be reached in the system, and then, only those firing sequences are counted as words of the language that obtain the given goal marking (the final/accepting state of the system) from the initial marking. The language class defined in this way is closed under the operations union, intersection and concatenation.

In a similar way a formal language can be defined in the following way: let the reached final marking cover the given goal marking.

Another possible way to define formal languages by the firing sequences that cannot be continued (they reach such markings that do not have any enabled transition).

The language class defined by place-transition Petri nets contain all regular languages and it contains only context-sensitive languages and it is incomparable with the class of context-free languages with respect to the set theoretical inclusion. By allowing to have more than one token in a place, it is very easy to implement some places as counters, and by their help it is easy to define non context-free languages. However there are some context-free languages that cannot be defined by these systems.

We note here that the BPP nets are closely related to the context-free grammars and by the basic parallel processes. Without the distinction of the tokens in simulation of a grammar the concept of commutative grammars is obtained. With special coloured Petri nets the grammars of the Chomsky hierarchy can also be modelled.

5. Questions and exercises

1. What is the main difference between Petri nets and finite automata?

2. What can be modelled by Petri nets?

3. What is the main difference between the binary Petri nets and the place-transition nets?

4. Is the binary Petri net in Figure 5.1 cycle-free? What can we say about its liveness properties?

5. Analyse the liveness properties of the net given in Figure 5.2. What about its fairness? Is it cycle-free? Is it persistent? Is it conflict-free? Is it communication-free? Is it conservative?

6. Give an example for a finite-state machine. When a finite-state machine is (L4) live?

7. Is an (L4) live finite-state machine is reversible? Is it cycle-free?

8. Is the place-transition net shown in Figure 5.7 safe?

9. Is the place-transition net shown in Figure 5.8 safe?

10. Construct the reachability tree of the potentially firable Petri net shown in Figure 5.9.

11. Is the net shown in Figure 5.6: modelling chemical processes bounded? Give the reachability tree of this net. What about its liveness properties?

12. Is the place-transition net shown in Figure 5.7 cycle-free? Is it conservative? Is it bounded?

13. Give an extended free-choice net that is not free-choice net. Construct its covering graph.

14. Give an example for an (L4) live free-choice net. Determine the siphons and traps. Which are the minimal siphons and traps? Which are the basic siphons and traps?

15. Draw the whole net of the hierarchic model presented in Figure 5.18. Use the reduction rules to simplify the net.

6. Literature

G. Balbo, J. Desel, K. Jensen, W. Reisig, G. Rozenberg, M. Silva: *Petri Nets 2000*, 21st International Conference on Application and Theory of Petri Nets, Aarhus, Denmark, June 26-30, 2000, Introductory Tutorial - Petri Nets

R. David, H. Alla: *Discrete, continuous, and hybrid Petri Nets*, Springer, 2005.

E.P. Dawis, J.F. Dawis, Wei-Pin Koo: *Architecture of Computer-based Systems using Dualistic Petri Nets*, Systems, Man, and Cybernetics, 2001 IEEE International Conference on, Volume 3, 2001, pp. 1554-1558.

C. Dufourd, P. Jancar, P. Schnoebelen: *Boundedness of reset P/T nets*, International Colloquium on Automata, Languages and Programming (ICALP), LNCS 1644, pp. 301-310, Springer-Verlag, 1999.

J. Esparza: *Petri nets, commutative context-free grammars and basic parallel processes*, Fundamenta Informaticae 30 (1997), 24-41.

M. Hack: *The recursive equivalence of the reachability problem and the liveness problem for Petri nets and vector addition systems*, FOCS, pp. 156-164, 1974.

R. Howell, L. Rosier , H. Yen: *Normal and sinkless Petri nets*, Journal of Computer and System Sciences 46, 1993, 1-26.

P. Jancar: *Non-primitive recursive complexity and undecidability for Petri net equivalences*, Theoretical Computer Science 256 (2001), 23-30.

K. Jensen: *Coloured Petri Nets and the Invariant-Method*, Theoretical Computer Science 14 (1981), 317-336.

J. Kleijn, M. Koutny: *Formal Languages and Concurrent Behaviours*, in: **G. Bel-Enguix, M. D. Jiménez-López, C. Martín-Vide (eds.):** *New Developments in Formal Languages and Applications, Springer, 2008, pp. 125-182.*

L. Landweber, E. Robertson: *Properties of conflict-free and persistent Petri nets*, JACM 25:3 (1978), 352-364.

C.G. Looney: *Fuzzy Petri Nets and Applications*, in: Fuzzy Reasoning in Information, Decision and Control Systems (International Series on Microprocessor-Based and Intelligent Systems Engineering, volume 11, Part 5, 1994), 511-527.

E. Mayr: *An algorithm for the general Petri net reachability problem*, SIAM J. Comput. 13 (1984), 441-460.

T. Miyamoto, S. Kumagai: *A Survey of Object-Oriented Petri Nets and Analysis Methods,* IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Volume E88-A Issue 11, 2005, 2964-2971.

M.K. Molloy: *Performance analysis using stochastic Petri nets,* IEEE Transaction on Computers, vol. C-31 (9), 1982, pp. 913-917.

T. Murata: *Petri nets: Properties, analysis and applications,* Proceedings of the IEEE 77:4 (1989), 541-580.

B. Nagy: *Derivation "trees" and parallelism in Chomsky –Type Grammars,* Triangle 8 (Languages. Mathematical Approaches) (2012), 101-120.

B. Nagy: *Graphs of Grammars – Derivations as Parallel Processes,* in: Computational Intelligence in Engineering, Studies in Computational Intelligence - SCI 313, Springer (2010), pp. 1-13.

A. Pataricza (eds.): *Formális módszerek az informatikában,* Typotex Kiadó, Budapest, 2004 [chapter 2: Petri hálók, pp. 31-111, Petri nets, in: Formal methods in computer science, in Hungarian]

J. L. Peterson: *Petri Nets,* Computing Surveys 9 (1977), 223-252.

J. L. Peterson: *Petri Net Theory and the Modeling of Systems.* PRENTICE-HALL, INC., ENGLEWOOD CLIFFS, NJ 07632 , 1981, 290 pp

C. A. Petri: *Kommunikation mit Automaten,* PhD Dissertation, Rheinisch-Westfalisches Institut fűr Intrumentelle Mathematik an der Universitat Bonn, Bonn, 1962.

Petri Nets home page: http://www.informatik.uni-hamburg.de/TGI/PetriNets/

W. Reisig: *Petri Nets: An Introduction,* EATCS monographs on Theoretical Computer Science, Springer-Verlag New York, Inc., New York, NY, 1985.

W. Reisig, G. Rozenberg (eds.): *Lectures on Petri nets I: Basic Models (Advanced Petri nets),* LNCS 1491, Springer 1998.

W. Reisig, G. Rozenberg (eds.): *Lectures on Petri nets II: Applications (Advanced Petri nets),* LNCS 1492, Springer 1998.

Hsu-Chun Yen: *Introduction to Petri Net Theory,* in: **Z. Ésik, C. Martín Vide, V. Mitrana, (eds.):** *Recent Advances in Formal Languages and Applications, Springer, 2006. pp. 343-373.*

6. fejezet - Parallel programs

6.1. Elementary parallel algorithms

In the following, we will observe a few fundamental problems and their solver algorithms, in particular the possibility of parallelization.

The first problem is searching.

Problem (Searching):

In: *n*, *A*[*n*], *x* (*x* is called the key)

Out: an index *i* with *A*[*j*] ≠ *x* if such index exists; 0 otherwise.

This problem has a number of different solutions. The simplest one is to read the whole sequence of numbers until we find the key. In the worst case we need as many comparison operations as the number of members of the set.

```
Algorithm 1.
1. $i \leftarrow n$
2. while ($i \geq 0$) and (($x \neq A[i]$)) do
3.     $i \leftarrow i - 1$
```

Statement:

Let *t*(*n*) be the time complexity of Algorithm 1. Then *t*(*n*) ∈ O(*n*).

To get meaning to the statement, we have to give exactly what we mean by time complexity in this case. There are several definitions and actually, for any of them - which are expressive enough - the statement would be true. The most general, when we count every executed command, makes the calculation more difficult in most of the cases. This is why it is practical to choose the most important commands: those which use the most expensive resources or those which are useful to estimate the number of all the other executed commands. In the case of the present algorithm (and those which are related to the problem) the time complexity will be understood as the number of comparisons. However, the complexity of the comparison itself remains hidden.

The proof of the statement is easy, it can easily be proved with a basic experience in programming, so we leave it to the reader.

A more interesting solution of the problem is when we suppose that the incoming data is in an ordered structure. In this case, theoretically it is sufficient to carry out log(*n*) comparisons. The following example is the so called binary search algorithm.

In: *n*, *A*[*n*], *x*, where *A*[*j*] ≤ [*j* + 1] for all 1 ≤ *j* ≤ *n*.

Out: an index *i* with *A*[*i*] = *x* if such index exists; 0 otherwise.

```
Algorithm 2.
 1. $i \leftarrow 1$
 2. $j \leftarrow n$
 3. while $i < j$ do
 4.
```

$$k \leftarrow \frac{i+j}{2}$$

```
 5.    if A[k] = x then
 6.        j ← i
 7.    else if A[k] < x then
 8.        i ← max{i + 1, k}
 9.    else
10.        j ← k
11.    endif endif
12. endwhile
13. if A[i] ≠ x then
14.    i ← 0
15. endif
```

Theorem:

The time complexity $t(n)$ of Algorithm 2 is $\Theta(\log(n))$.

Similarly to the previous algorithm, we define the time complexity by the number of comparisons.

Proof:

There are comparisons only in the 5th and 7th line of the algorithm, thus we make two comparisons during the execution of the loop in lines 3-12.

The execution of the loop continues until $i < j$. Since j and i are integer, this means that $1 \leq j - 1$. (*)

Assume that during the lth execution of the loop body $i = i_l$ and $j = j_l$, $\forall\ 0 < l$ is fullfilled, then we find that

$$j_{l+1} - i_{l+1} < \frac{1}{2}\left(j_l - i_l\right).$$

Since $j_1 - i_1 = n$, then after the lth iteration

$$j_{l+1} - i_{l+1} < \frac{1}{2^l} \cdot n$$

holds.

By (*)

$$1 \leq j_{l+1} - i_{l+1} < \frac{1}{2^l} \cdot n$$

holds.

Rearranging the inequality and taking the logarithm of both sides we get the following relation:

$2^l < n$

$1 < \log(n)$

This exactly means that the loop body can be executed at a maximum of $\log(n)$ times. All together, we do not make more comparisons than $2log(n)$, so the time complexity is $O(log(n))$. √

During the observation of the algorithm, we face the problem that we have executed a sequence of operations that did not completely read the incoming data. On the other hand, we have to notice that the whole input is immediately available for the algorithm. These two observations seem to be problematic for the first view if we try to construct the algorithm with the traditional Turing machine model. The difficulty is that the speed of the transmission of the information can become arbitrary big. (Transmission of the information: the algorithm gets the input and transmits the result to somewhere.) However, it does not have such a big importance as basically we do not solve the problems with Turing machines but with some machine with particular architecture. The solution: we do not provide the input (output) in the traditional way, it is simply available. (e.g., it is the result of the previous computation in the memory.) In most cases it models quite well the problem to solve, but we have to notice that the abilities of the algorithm in practice (~ program) are limited because of the bounds on the resources. (Basically a computer is a finite automaton.) Accepting these kinds of limiting factors, it does not make any sense to observe the time complexity problems only from theoretical point of view, since we can give a finite number of answers to a finite number of inputs in just one step. Obviously, we can only get practical results if we bring in more limiting factors during the observations.

Returning to the original problem, we can give a parallel solving algorithm, as well. This compares the element to be found with all the elements of the set at the same time and gives positive answer only if any of the comparing modules finds a match.

```
Algorithm 3.
1. for.p $j$ ← 1 … $n$ do.p    // parallel for command
2.    if $A[j]$ = $x$ then
3.        $F[j]$ ← $j$
4.     else
5.        $F[j]$ ← 0
6.     endif
7. endfor.p
8. $i$ ← $max(F[j], j = 1 … n)$
```

Structurally this algorithm is similar to the first one, the simplest sequential algorithm but we do everything parallel whatever we can. As we have seen in the chapter on Super Turing machines, theoretically line 8 can be executed in the time of one comparison. This implies the following:

Statement: Let $T(n)$ be the total time complexity and t(n) be the absolute time complexity of Algorithm 3. Then $T(n) \in O(n)$ and $t(n) \in O(1)$.

If we define the time complexity by the number of comparisons (as before), then the proof of the statement is easy. It gives additional refinement to the interpretation that in general the command in line 9 - depending on the model (or architecture) - cannot be necessarily executed during constant time.

There is the question though, whether Algorithm 3 is an algorithm in the traditional sense, since basically we need different architectures for each task. The problem can only be solved in the theoretically determined time, if there is a suitable amount of processors available, with the necessary architecture. This practical consideration is the reason of the introduction of the limited complexity concepts, as the size of the host system generally cannot be changed. The FPGA architectures, which are more and more common at the computations, give new possibilities. With them we can change not only the software but the hardware as well to fit to the problem.

The second basic problem is sorting. We define it in the following way:

Problem (Sorting): Let A be a set with a sorting relation $\leq$ and let $T[1 \dots n]$ be an array such, that its entries are from the set A.

Let's sort the entries of array T such, that $T[i] \leq T[i + 1]$ be true for all $i \in \{1 \dots n - 1\}$. During the sorting the entries cannot disappear and new ones cannot be created.

In: n, $A[n]$, x

Out: $B[n]$, where $\forall i \in \{1, \dots, n - 1\}$: $B[i] \leq B[i + 1]$, and $\exists\, \pi$ permutation, such that $\forall\ i \in \{1, \dots, n\}$: $A[i] = B[\pi(i)]$.

In the case of the general sorting algorithms we are in a special situation. The sorting belongs to a class of problems where we can prove exact lower bound for the time complexity.

Theorem: Let R be a general sorting algorithm with time complexity $t(n)$. Then $n \cdot \log(n) \in O(t(n))$.

This means that a general sorting algorithm cannot be faster than $c \cdot n \cdot \log(n)$.

Several known algorithms reach this theoretical bound, so except for a constant factor, it gives an exact lower bound to the time complexity.

One of these algorithms is the algorithm of merge sort.

6.1.6.1.1. Merge sort

This method is based on the principle of divide and conquer

```
ÖFR(n, A)
1.
```

$$A_2 \leftarrow A\left[\frac{n}{2}+1 \ldots n\right]$$

$$B_1 \leftarrow ÖFR\left(\frac{n}{2}, A_1\right)$$

```
2.
```

$$A_2 \leftarrow A\left[\frac{n}{2}+1 \ldots n\right]$$

$$B_2 \leftarrow ÖFR\left(\frac{n}{2}, A_2\right)$$

```
3.
```

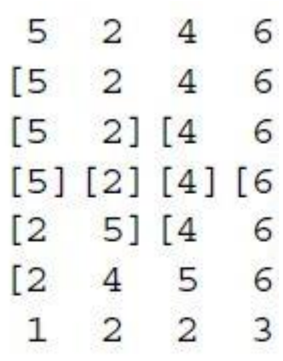
$$C \leftarrow ÖF\left(\frac{n}{2}, B_1, B_2\right)$$

The algorithm of merging:

```
ÖF(n,A,B)              // A and B sorted
 1. i ← 1
    j ← 1
    k ← 1
 2. while (i ≤ n) or (j ≤ n) do
 3.    if (j > n) then
 4.        C[k] ← A[i]
           i ← i + 1
 5.    else if (i > n) then
 6.        C[k] ← A[j]
           j ← j + 1
 7.    else if (A[i] < B[j]) then
 8.        C[k] ←A[i]
           i ← i + 1
 9.    else
10.        C[k] ← A[j]
           j ← j + 1
11.    endif; endif; endif
12.    k ← k + 1
13. endwhile
```

We assume that the two input sets have the same amount of elements, but it works similarly in case of different cardinality.

Example: Let’s sort the array A = [5,2,4,6,1,3,2,6].



```
 5   2   4   6
[5   2   4   6
[5   2] [4   6
[5] [2] [4] [6
[2   5] [4   6
[2   4   5   6
 1   2   2   3
```

By analyzing the algorithm of merging we can see that it is a typical sequential one, and by parallelizing we cannot make it a lot faster. (A minimal acceleration is possible, if we start the comparison of the next elements with pipelining while executing the data movement.)

Real growth can only be achieved by the reconsideration of merging.

6.1.6.1.2. Batcher's even-odd sort

This procedure makes more effective the tight cross section of the algorithm, the merging.

A big advantage is that its steps can be parallelized.

Let *A*[1 … *n*] and *B*[1 … *n*] be two sorted arrays with the same number of entries. For simplicity assume that $n = 2^k$ for some positive integer *k*.

The used algorithm is the following: We create two new arrays from each original array: the sequences of the even and odd indexed elements. The merging of these arrays provides sorted arrays with very good properties.

```
BÖF(n,A,B)          // A and B is sorted
1. if n > 1 then
2.
   $A_1 \leftarrow A\left[2i-1, i \leftarrow 1 \dots \frac{n}{2}\right]$
   $A_2 \leftarrow A\left[2i, i \leftarrow 1 \dots \frac{n}{2}\right]$
   $B_1 \leftarrow B\left[2i-1, i \leftarrow 1 \dots \frac{n}{2}\right]$
   $B_2 \leftarrow B\left[2i, i \leftarrow 1 \dots \frac{n}{2}\right]$
3.
   $C_1 \leftarrow BÖF\left(\frac{n}{2}, A_1, B_2\right)$
   $C_2 \leftarrow BÖF\left(\frac{n}{2}, A_2, B_1\right)$
4.    pfor i ← 1 … n do
5.       C[2i − 1] ← min{C_1[i], C_2[i]}
         C[2i] ← max{C_1[i], C_2[i]}
6.    endpfor
7.  else
8.    C[1] ← min{A[1], B[1]}
      C[2] ← max{A[1], B[2]}
```

The result of merging is in *C*. Although, it does not affect the time complexity related to the number of comparisons, it is worth to consider to reduce the data movement operations. The corresponding computer program can have significant speedup, if we apply an in place sort. Of course, in architectures, where data movement is a simple wiring (e.g. FPGA), it is not an issue.

Theorem:

The Batcher's merging is correct, i.e. a sorted array is created from the entries of the two original arrays.

Proof. For simplicity, we assume that every element of the array are different. The statement is true in the more general case, too, but the proof would be slightly more complicated. It is clear that the algorithm is swapping the elements, thus a value neither gets in nor can disappear. Thus the correctness of merging depends on the observation that the instructions executed in step 4 are moving the elements of the array to the correct place.

The entries $C[2i-1]$ and $C[2i]$ are on the correct place relative to each other, we only have to prove that there cannot be a bigger entry before them, and a smaller one after them.

Before the pair of $C[2i-1]$ and $C[2i]$ there can only be such kind of elements that were the elements of arrays C_1 and C_2 with the indexes $j < i$. Then it is clear that $C_1[j] < C_1[i]$ and $C_2[j] < C_2[i]$ $\forall$ $0 < j < i$, so we only have to show that $C_2[j] < C_1[i]$ and $C_1[j] < C_2[i]$ $\forall$ $< 0 < j < i$.

Since C_1 was created be the merging of the sorted arrays A_1 and B_2, thus *k* entries are from A_1 and $i-k-1$ from B_2 in C1[1, …, $i-1$]. More precisely $C_1[1, \ldots, i-1]$ is the merge of $A_1[1, \ldots, k]$ and $B_2[1, \ldots, i-k-1]$.

Assume that $C_1[i]$ was originally in *A*. Since A_1 contains the odd and A_2 contains the even entries, thus A_2 contains exactly *k* entries which are less than $C_1[i]$ and consequently

$$\frac{n}{2}-k-1$$

which are greater. Furthermore, since B_2 stends of the even entries of *B*, thus B_1 have $i-k-1$ entries which are less

and

$$\frac{n}{2}-(i-k)$$

entries which are greater than $C_1[i]$. However, the $(i-k)$th is not known. This yields that in a C_2 there exist $k+i-k-1 = i-1$ entries which are less and

$$\frac{n}{2}-k+\frac{n}{2}-(i-k)=n-i$$

entries which are greater than $C_1[i]$. Only $C_2[i]$ cannot be determined in advance how is it related to $C_1[i]$. The algorithm requires this comparison operation during step 5.

By similar arguments, we get the same result if $C_1[i]$ was originally not in *A* but in *B*.

√

6.1.6.1.3. Full comparison matrix sort

For the sorting we will create an $n \times n$ processor matrix.

The processors in the intersection will compare the elements corresponding to its row and column index. If in this comparison the entry corresponding to the row index is bigger, it sends 1 to the adder collecting data from the column, otherwise it sends 0. At the bottom of the column there will be the sum of these values. If we assume that the members of the array to sort are pairwise different, then the sum represents the index of position where the member, corresponding to the column, should be placed. The algorithm has time complexity $O(log(n))$, while it uses $O(n^2)$ processors.

6.2. Parallelization of sequential programs

In case of sequential algorithms it is common, that to get the final result we have to determine intermediate results that are not based on each other. If, for example, we are looking for the divisors of the positive integer *n*, one possible way is to check each number from 2 to *n* whether it divides *n*. Of course these trials can be done separately. Thus the algorithm can be divided automatically to independent threads. These threads do not need any communication, do not wait for each other and give different intermediate results of the problem. In this chapter we will see how to resolve an algorithm to steps that can be executed in a parallel, after a syntactic analysis.

In this chapter we will see how to decompose an algorithm into steps that can be executed parallelly, after its analysis due to its (syntactic and semantic) description.

During our observations we will be still using the command based (imperative) programming languages. To make it more understandable we will examine a language which is quite general, but as simple as it can be. In this language there are only 3 + 1 kind of commands:

```
0. skip
1. X ← E
2. if B then C1 else C2 endif
3. while B do C endwhile
```

0: does not do anything; it will have an important role at 2, when the conditional command is not complete.

1: the usual assignment command; after its execution the value of *X* will be equal to the value of the expression *E*. The expression *E* can be any well defined operation resulting an object from the domain of *X*. From practical point of view, of course, only computable operations are important.

2: conditional execution; if the logical expression *B* is true, C_1 will be executed, in any other cases C_2; if the command is incomplete, for example, it does not have an else branch, we will substitute skip to C_2. The properties of *B* are similar to the ones of *E* with difference, that the result is a Boolean value.

3: conditional loop; while the logical expression *B* is ture, we execute the loop body *C*.

We will examine the possibilities of the parallelizing of sequential programs on this very simple language. For more complex analysis, the programming language can be extended by the necessary instructions.

We assume that the variables have no type, i.e. they have a universal type.

6.2.6.2.1. Representation of algorithms with graphs

In the introduction of the chapter we saw a frequently used way to represent an algorithm, the so called pseudo code. With this we can describe the procedures in the structure used for the imperative languages. For those who are experienced programmers that is a tool expressive enough to describe the operations and their order. The internal relationships of the algorithms given by a pseudo code are only visible after a detailed analysis (in case if we can find them at all).

One of the most important aims of the representation by graph is that we try to give the above mentioned relationships during the description of the algorithm in a visually understandable way.

According to the first generation of the programming languages, in practice, the representation of algorithms by flow charts was very usual.

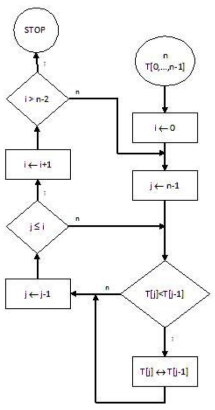
6.2.6.2.1.6.2.1.1. Flow charts

The flow chart is basically a directed graph, its vertices represent the instructions, its edges the sequentialness.

In case of simple algorithms the interdependence of the instruction execution can be represented very expressively with it, but, because of the spreading of the structural programming languages its usability has been reduced, as some kinds of flow chart structured are very difficult to be converted into program structures. (Like jumping from one loop to another, for example. This can make a well-structured program completely broken, the controlling will become untraceable.)

On Figure 6.1 we can see the algorithm of the bubble sort.

6.1. ábra - The flow chart of bubble sort.



Though the algorithm is well interpretable by the flow chart, it is not completely straightforward, how to write a well-structured goto free C code of it.

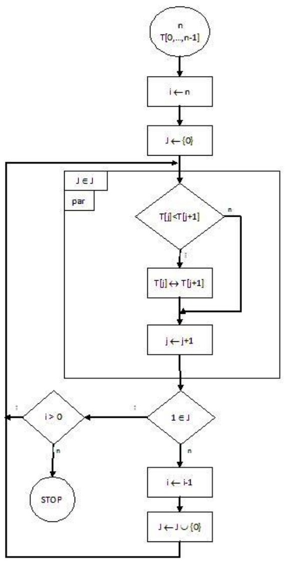
It is also difficult to say what kind of parallelization steps can be used.

6.2.6.2.1.6.2.1.2. Parallel flow charts

We can generalize the flow chart model in the direction of parallelizing. Basically, the directed edges are expressing the sequentialness of the instructions here too, but we have to introduce the parametrical edge as well (that is belonging to the parallel loop). With this we can express that we are executing the instructions in the quantity of the parametrical edge.

On the following flow chart we can see the parallelized algorithm of the bubble sort.

6.2. ábra - The flow chart of parallel bubble sort; J is a set type variable.



From here we have one more generalizing step to the data flow graph. In this model the interdependence of the instructions is completed with the representation of the possible data flows.

6.2.6.2.1.6.2.1.3. Data flow graphs

In a data flow graph, as before, the instructions are in the vertices, which are called actors.

The edges of the graphs are representing not only simple sequential execution, but the real data flow possibility. Basically we can consider them as data channels, thus the operations of the algorithm represented by the graph can be observed by graph theoretic tools. (The determination of the time complexity will be reduced to the searching of the longest path and the maximal flow capacity of the graph will get a meaning in case of the so called “pipeline” operation. In this case we can start the computation on the new input before getting the result of the precedent computation.)

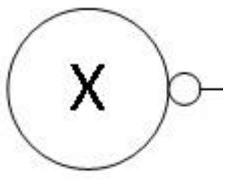
We call the data packets flowing in the data flow graph tokens.

An actor will become active (allowed), if there are available tokens completing the conditions given on the inputs.

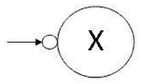
When a node of the graph become active, it starts working immediately, executes the dedicated operation and puts the result as a token on the output edge. On one output edge there can be several tokens, but their processing order has to be the same as their creation order. (So the edges can be considered a kind of FIFO data storage.)

We may notice that the data flow graphs are similar to the Petri nets, with one difference: the tokens here are particular data packets.

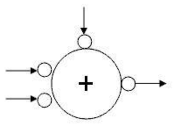
6.3. ábra - Input actor.



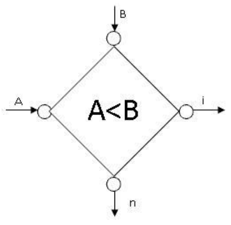
6.4. ábra - Output actor.



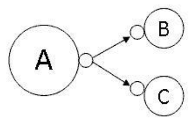
6.5. ábra - Adder actor.



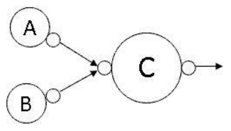
6.6. ábra - Condition generator.



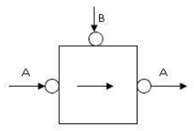
6.7. ábra - Port splitter; the output token of A appears at the input ports of B and C in the same time.



6.8. ábra - Collector actor; the output token of A or B appears on the input if it becomes empty.



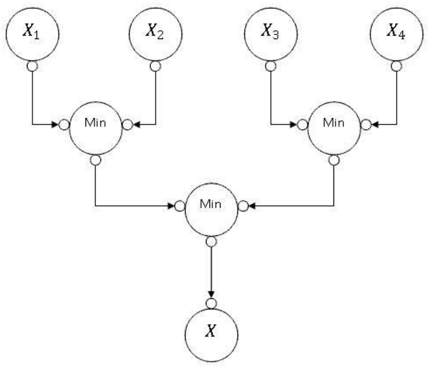
6.9. ábra - Gate actor; if a signal arrives from B, then A is transferred from the input port to the output port.



We can give different conditions to each component of the data flow graph, thus we get different models.

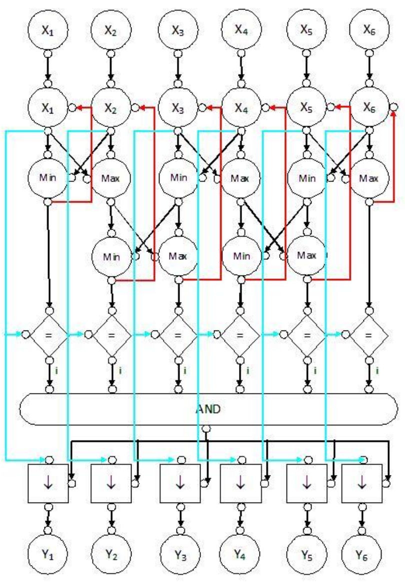
In the simplest case we do not let feedback in the graph (directed acyclic). For example choosing the minimum can be seen in Figure 6.10. :

6.10. ábra - Selecting the minimum; $X = min\{xX_1, xX_2, xX_3, xX_4\}$.



In more complex models we allow feedback, but we may require for example that an actor only starts to work if there is no token on the output edges. (So the previous output has already been processed by the actor.) For example: bubble sort.

6.11. ábra - The data flow graph of bubble sort.



In one of the most advanced data controlled model the tokens can pile up at an actor. We resolve the synchronization by numbers assigned to the tokens. We will give the same number to the related input data. If the tokens with proper numbers have arrived to an actor, it will be activated and computes its output tokens, giving them the same number as the input has. When tokens with other numbers arrive, it puts them aside till all of them that are needed for the activation are arrived. Thus the result of the computation will not necessary appear in the same order we gave at the input.

6.2.6.2.1.6.2.1.4. Data dependency graphs

The typical observation method of graph representation is that we switch the roles of the vertices and edges. In our case it means that in the new graph model the edges will represent the instructions and the nodes represent the data that the instructions are applied on. As each instruction can execute an operation on several data at the same time, it is practical to choose the graph as a hypergraph. Thus we can represent the dependency relations between the data. Because of the traceability, we represent the different occurrences of the data differently. There are several levels of the dependency of the data, we will present them later.

6.3. The method of program analysis

During the analysis of a program, we will tell about a sequentially represented algorithm (program), how its steps (instructions) depend from each other.

In practice, one can create programs in such a way, that we cannot decide easily if the instructions are depending from each other or not, i.e. whether we can change the order of their execution.

By considering only the theoretical points of view, the independent parts of the program can be executed in paralell, considering their variable values. The analysis of algorithms or programs that contain recursive parts can be more complicated than those that are built from iterative components only. The usage of complex data structures causes further difficulties and we can create quite complicated programs using pointers and direct memory access. We will not deal with the cases of procedure call though these are not that complicated if we keep in mind some simple rules.

From a simple observation we could say that the dependence of the instructions is not complicated for the loop free program parts, but it is for the sequences that contain loops. After all, it is logical to decompose the code into linear segments, as a first step.

After decomposition, we can start the detailed analysis of each sequence.

Example (decomposition of the bubble sort into linear sequences)

Sequence I

```
1. $i \leftarrow 0$
2. while $i < n - 2$ do
   Sequence II
   3. $j \leftarrow n - 1$
   4. while $j > i$ do
      Sequence III
      5. if $T[j] < T[j - 1]$ then
         6. $T[j] \rightleftarrows T[j - 1]$
      7. endif
      8. $j \leftarrow j - 1$
   9. endwhile
   Sequence IV
   10. $i \leftarrow i + 1$
11. endwhile
```

When the decomposition is done, we start the analysis of the sequences independently from each other.

The most important part of this process is the creation of a dependency graph.

The vertices of this graph are the different instructions and the elements which are in a direct dependence relation are connected with directed edges.

Example (dependency graph for solving second order equation)

In: $ax^2 + bx + c = 0$

```
1. $t \leftarrow ac$
2. $t \leftarrow 4t$
3. $s \leftarrow b^2$
4. $d \leftarrow s - t$
5. if $d < 0$ then
   6. $n \leftarrow 0$
7. else if $d = 0$ then
   8. $n \leftarrow 1$
   9. $x \leftarrow \frac{b}{2}$
10. else
   11. $n \leftarrow 2$
```

12.

$g \leftarrow \sqrt{d}$

13. $t \leftarrow b - g$

14.

$x_1 \leftarrow \frac{t}{2}$

15. $t \leftarrow b + g$

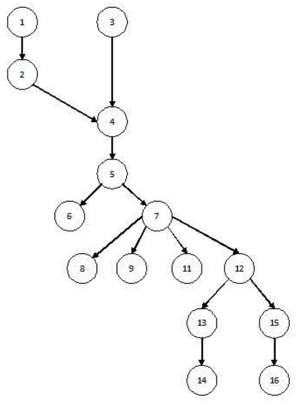
16.

$x_2 \leftarrow \frac{t}{2}$

17. endif

18. endif

6.12. ábra - The dependency graph of the solver of quadratic equations.



The type of the constructed dependency graph depends not only on the program, but on the analysis as well.

Example (several ways to create a dependency graph)

1. $x \leftarrow a$

2. $y \leftarrow b$

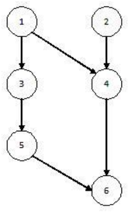
3. $t \leftarrow x$

4. $t \leftarrow x + y$

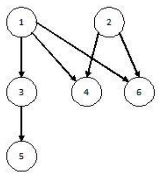
5. $x \leftarrow t$

6. $y \leftarrow xy$

6.13. ábra - The syntactical dependency graph of the program.



6.14. ábra - Expressing deeper dependency for the same code.



In the case of the observed simplified programs, the dependence of each instruction is only determined by the dependence of the variables. Since the observed segments are linear, the data flow is one way. The order of the execution of the instructions is determined only by the direction of the data flow.

The dependency relation of two commands is not as easy to be defined as it seems, since the dependency is changing if we choose another (deeper) analysing method. For first, we are dealing with a quite rough determination: we say that instruction C_2 is depending directly on instruction C_1, if at least one of the outputs of C_1 is the input of C_2 in the same form.

Example (fake dependency)

1. $x \leftarrow a$

2. $y \leftarrow b$

3. $x \leftarrow x + y$

4. $x \leftarrow x$ – y

Here 4 is depending on 3 only virtually, just as 3 and 4 were not actually executed.

We can decompose the instructions into levels based on their position in the graph. The levels can be defined recursively in the following way:

Definition: To the instructions of an acyclic program we can give the instruction levels in the following recursive way:

level 0: instructions that are not depending from anything.

level i ($0 < i$): if we have already defined the level i - 1, those instructions are belonging here that we have not defined their level yet and are only depending on instructions we have already defined.

Remark: If an instruction is belonging to level i, it has to depend from an instruction that is on the level i - 1. If it was not be so, due to the definition it would belong to level i - 1 (or even higher).

Remark: The above definition assigns a unique level value to every instruction.

Since a program contains a finite number of instructions, the graph will have a lowest level. We call this level an output level, while the level 0 is the input level.

The dependency of the instructions are related to the data and variables used by the instructions. We can simplify the observation of the dependency of the instructions to the observation of the dependency of the data. For this, we will need a new definition, namely the actual value of the data.

Definition: Let C be a program and $Var(C) = \{X_1, X_2, \ldots, X_n\}$ be the set of variables used by C. A state of the program is the set$(x_1, x_2, \ldots, x_n)$, where $X_i = x_i$. Among the values we allow the symbol #, which denotes udefined variables.

In case of programs containing a loop to decide the dependency of the data, we decompose the program into blocks and analyze whether a block uses its variables in a local or global way, namely the variables in the segments are getting the values from outside or they are storing an internal independent intermediate result. An index variable of a loop is typically a local variable. If in a block there is nothing but local variable, then the block can be executed independently from the previous part of the program.

We may give the set of in and output variables to a block of commands. Intuitively we will call output variables those that are getting a new value during the execution of the block and input variables that are taking part at the computation of the values of the output variables.

We distinguish two levels: when only formal (syntactic) and when real (semantic) dependency can be established between the in and output variables.

Examples:

Example (formal dependency)

1.

$T := X$

$X := Y$

$X := T$

Output variables: T, X

Input variables: X, Y, T

2.

$X := Y$

$Y := 2Y$

$Y := Y - x$

Output variables: X, Y

Input: X, Y

Example (fake dependency)

3.

$T := X$

$X := Y$

$X := T$

Output variables: T

Input variables: X

4.

$X := Y$

$Y := 2Y$

$Y := Y - X$

Output variables: X

Input variables: Y

Definition (formal in and output for acyclic programs):

Let C be a block of instructions. The set $SOut(C)$ of the output variables of C and the set $SIn(C)$ of the input variables can be given by the following inductive definition:

1. $SOut(skip) = \emptyset$

$SIn(skip) = \emptyset$

2. $SOut(X \leftarrow E) = \{X\}$

$SIn(X \leftarrow E) = Var(E)$

3. $SOut(C_1;C_2) = SOut(C_1) \cup SOut(C_2)$

$SIn(C_1;C_2) = \mathrm{SIn(C1)} \cup (SIn(C_2) \setminus SOut(C_1))$

4. $SOut(\text{if } B \text{ then } C_1 \text{ else } C_2) = SOut(\mathrm{C}_1) \cup SOut(C_2)$

$\mathrm{SIn}(\text{if } B \text{ then } C_1 \text{ else } C_2) = Var(B) \cup SIn(C_1) \cup SIn(C_2)$

To establish the real in and output variables of a block of program, we have to set up tighter connections between the variables. In some cases it may require the usage of quite strong tools and we may even find the problem impossible to solve.

6.3.6.3.1. Dependency of the variables

The dependence relation of the variables of a program can be established by a multiple leveled analysis. The simplest is when we are observing only if the variable has got a value before, due to another variable. Doing this recursively only that value dependency will be left that has the other variable's dependency as well. We call it syntactical dependency. The dependency established like this describes the connection a bit more precisely between the syntactic in and outputs.

We can establish a higher level dependency if we interpret the connection of the variables and observe not only if the other variable played a role before in some assignments. This kind of dependency is called semantic dependency.

There are several different types of the dependency of the variables:

0. X is not depending from Y even syntactically;

1. X is depending from Y syntactically, the dependency of the represented data can be resolved by a programming technical change;

2. X is depending from Y structurally and there is no chance to the resolution of the dependency.

Although the structural dependency is stronger than the syntactical, it is still not powerful enough to describe real dependency of the variables.

6.3.6.3.2. Syntactical dependency

Among the different dependencies, the establishment of the syntactical dependency is the easiest. We will not get however an exact picture of the real dependency by it, since it does not give the real relation between the variables. Thus in some cases it gives too many relations.

Let's see for example the following part of a program:

```
1.  X ← X - Y
2.  X ← X + Y
```

It is clear that this program does not change the value of the variables in any ways, so it does not change the dependency relation. But if we consider only the syntactical dependency, *X* is dependent on *Y*.

However this observation has a point, namely we may filter out several variables in the first step and this will simplify the establishment of higher level dependencies.

Definition: Denote by *Var*(*E*) the set of variables that are in expression *E*. We define it due to the complexity of *E* in the following way:

1. *Var*(*c*) = ∅ , where *c* is a constant;

2. *Var*(*X*) = {*X*}, where *X* is a variable;

3.

$$Var(f(E_1,\dots,E_n) = \bigcup_{i=1}^{n} Var(E_1),$$

if *f*(.) is a function.

Definition (real syntactic dependency of the variables of acyclic programs):

Let *SDep*(*X*,*C*) be a set of variables that the output variable *X* is depending on syntactically, during the execution of program *C*. We define *SDep*(*X*, *C*) in a recursive way:

1. *SDep*(*X*, *skip*) = {*X*};

2. *SDep*(*X*, (*Y* ← *E*)) = *Var*(*E*), if *Y* = *X* és {*X*}, if *Y* ≠ *X*;

3. *Sdep*(*X*, (*C*1; *C*2)) = $\bigcup_{Y\in SDep(X,\,C2)} SDep(Y, C_1)$;

4. *SDep*(*X*, (if *B* then C_1 else C_2)) = *SDep*(C_1) ∪ *SDep*(C_2).

The establishment of the real dependency of the variables is somewhat more complicated. For this, it is not enough to determine which input variables are the output variables depending on, but the way of its dependence as well. Due to the dependency of the variables we can establish the connection between the data. The data represented by independent variables can be considered independent; the data represented by dependent variables demand deeper analysis to establish their real dependence.

In the most of the cases it is not easy to establish the dependency relation of the variables, but there are some when it can be done.

Definition (real dependency of the variables of acyclic programs):

Let *RDep*(*X*, *C*) be the function that is realized by the program *C* due to the output variable *X*. we define *RDep*(*X*, *C*) in a recursive way:

1. $RDep(X, \text{skip}) = X$;

2. $RDep(X, (Y \leftarrow E)) = f(X_1, \dots, X_n)$, if $Y = X$, where E is realizing the function $f(X_1, \dots X_n)$

and X, ha $Y \neq X$;

3. $RDep(X, (C_1; C_2)) = f(g_1(X_1, \dots, X_n), \dots, g_k(X_1, \dots, X_n))$, where

$f(Y_1, \dots, Y_k) = RDep(X, C_2)$ and $g_i(X_1, \dots, X_n)$;

4. $RDep(X, (\text{if } B \textit{ then } C_1 \text{ else } C_2)) = f(X_1, \dots X_n)$, where

$f(X_1, \dots, X_n) = RDep(X, C_1)$, if $B(X_1, \dots, X_n) = I$ (or 1) and

$f(X_1, \dots, X_n) = RDep(X, C_2)$, if $B(X_1, \dots, X_n) = H$ (or 0).

Remark: If the values of the expressions are real numbers, the definition in the 4th point can be simply given by:

$f(X_1, \dots X_n) = B(X_1, \dots, X_n) \cdot RDep(X, C_1) + (1 - B(X_1, \dots, X_n)) \cdot RDep(X, C_2)$.

If we could establish the dependency of the variables, we can define the real in and output variables.

Definition (for cyclic programs containing real in and output variables):

Let C be a given block of instructions. The set $Rout(C)$ of the real output variables and the set $RIn(C)$ of the real input variables of C can be given by the following definition:

$ROut(C) = \{X | RDep(X, C) \neq X\}$.

$RIn(C) = \bigcup_{X \in Rout(C)} Dom(RDep(X, C))$.

There are several possibilities to practically describe the dependency of the variables. One of these methods is the appropriate modification of Dijkstra's weakest precondition calculus. It will be presented in some later parts of the chapter. With that we may establish what are the necessary computation and their order to give the final result, and the scope of the needed variables.

Of course there are way more effective parallelizing possibilities in the case of a particular algorithm. A nice example is the fast Fourier transformation.

During the computation of Fast Fourier Transformation (regardless the practical application and meaning of the operation) basically a multiplication $M \cdot a$ should be computed, where M is a matrix of size $n \times n$ and a is an n-dimensional vector. The exclusivity of the transformation depends on the structure of matrix M.

The matrix of transformation is similar to the following:

$$\begin{bmatrix} 1 & 1 \\ -1 & 1 \\ -1 & -1 \\ 1 & -1 \end{bmatrix}$$

Let $n = 2^s$ and $M = m_{j,k}$. Then M can be constructed by the recursive definition below:

```
for k ← 1, …, n - 1
    m0,k = m0,0
for l ← 1, …, s
   for j ← 2^(l-1),…, 2^l - 1
      for k ← 0, …, n - 1
```

$$m_{j,k} \leftarrow m_{j-2^{l-1},k} \cdot (-1)^{\left[\frac{k}{2^{l-1}}\right]+1}$$

In other words, the recursion is the following:

1. The rows with indices 2^l - 1 are given by the proper periodic alternation of 1 and -1;

2. The rest of the rows are the composition of them:

$$m_{2^l-1+k} = m_{2^l-1} x\, m_k$$

During the recursion - as one can observe by the relation - $M_j = M_{sj}$.

By this correspondence, instead of the original n^2 basic multiplication, it is enough to execute n, while the number of additive operations are $n \cdot \ s = n \cdot \ \log n$.

By the recursive definition of M we may find, that all basic multiplication can be done simultaneously. Thus, the processor complexity of the algorithm is n, while the time complexity is log$n$.

6.3.6.3.3. Banerjee’s dependency test

We have already observed the establishment of the variable dependency of acyclic programs. When our program contains loops, it is more complicated.

As we could see above, the establishment of the dependency relations of the instructions is more difficult in the case of loops. We can use Banerjee's dependency test to make it simpler. Basically, this is a simplified, very effective data dependency test that is used for automatic parallelizing of the loops of programs. Its operation is based on the fact that we try to detect the dependency caused by the indexed variables (for example arrays) through the observation of the indexes.

Example:
```
for $i$ from 0 to 100 do
   for $j$ from 0 to 50
                do
      for $k$ from 0 to 40
                do
         $X(a0 + i + 2j - k)$ = ···
         $X(b0 + 3i - j + 2k)$ =
                     ...
          enddo
    enddo
enddo
```

Principally, we establish relations by equations between the variables between the limits of the indexes (here, for example, $0 \le i \le 100$, $0 \le j \le 50$, $0 \le k \le 40$). We will get the dependency by the results of the equations. Since the variables in the equations can only be integers, the solution of the equations is quite simple, it can be done by the methods to diofantic equations.

The sequential program can be transformed into a program that can be executed effectively on a parallel architecture computer with such kind of compiler that restructures the code.

The task of the compiler is to reveal the hidden parallelities and to make them opened.

As a first step, we determine the dependencies arise during the execution of the sequential program.

The parallel version of the program has to fulfill the same dependency relations to have the same results as the original program.

If in the sequential program operation B is depending on operation A, then A has to be executed first in the parallel program as well.

To establish the dependency of the two operations, we have to know if they are using the same memory space, and if yes, in which order.

6.3.6.3.4. Modified Dijkstra's weakest precondition calculus

Dijkstra's weakest precondition calculus is a well-known and well-studied method of the proof of program correctness. There are several automatized systems to execute the operations described in it, so it is worth to expand it to the observation of parallelizing.

Definition:

1 $wp(skip, Q) = Q$

2. $wp(X \leftarrow E, Q) = Q(X \leftarrow E)$

3. $wp((C_1; C_2), Q) = wp(C_1, wp(C_2, Q))$

4. $wp((if\ B\ then\ C_1\ else\ C_2), Q) = (B \wedge wp(C_1, Q)) \vee (\neg B \wedge wp(C_2, Q))$

5. $wp((while\ B\ do\ C), Q) = \exists k\ (k \geq 0) \wedge p_k$, where P_0 & $\neg B \wedge Q$δ: Q \ H and $P_{k+1} = B \wedge wp(C, P_k)$ (informally: $\exists k\ (k \geq 0) \wedge P_k = P_0 \vee P_1 \vee P_2 \vee \ldots)$

This way we can express in a block which output variables are depending from which input variables, and after that we can describe the structure of the interdependence between the blocks.

Consider the following simple part of a program:

1. $X \leftarrow 3$

2. $Y \leftarrow 2 \cdot X$

3. if $Y < 4$ then

$X \leftarrow 2$

In this block of instructions the variables X and Y will get output values, however it has no input - at least not that kind which takes part in the creation of the output.

We can tell that the values of X and Y can be computed in the same time, independently from each other:

They will get the following values: $X = 3$ and $Y = 6$

Output variables: X_r, Y_r

By the 3rd line the dependency: $D_x = ((Y < 4) \wedge (X_r = X)) \vee ((Y \geq 4) \wedge (X_r = 2))$

$D_y = (Y_r = Y)$

By the 2nd line the dependency: $D_x = ((2 \cdot X < 4) \wedge (X_r = X)) \vee ((2 \cdot X \geq 4) \wedge (X_r = 2))$

$D_y = (Y_r = 2 \cdot X)$

By the 1st line the dependency: $D_x = ((2 \cdot 3 < 4) \wedge (X_r = 3)) \vee ((2 \cdot 3 < 4) \wedge (X_r = 2))$

$D_y = (Y_r = 2 \cdot 3)$

We can see that it is easy to tell the dependency relations if the program has only 1-4 type instructions. The difficulty starts at using loops.

Let's observe these three simple loops:

I.)

```
1. X ← 0
2. while X < n do
```

```
3.    $T[X] \leftarrow 2 \cdot T[X]$
4. endwhile
```

II.)

```
1. $X \leftarrow 0$
2. $S \leftarrow 0$
3. while $X < n$  do
4.    $S \leftarrow S + T[X]$
5. endwhile
```

III.)

```
1. $X \leftarrow 0$
2. $Y_1 \leftarrow 0$
3. $Y_2 \leftarrow 1$
4. while $X < n$ do
5.   $T \leftarrow Y_1$
6.   $Y_1 \leftarrow Y_2$
7.   $Y_2 \leftarrow T + Y_2$
8. endwhile
```

In the first case it is clear that the instructions of the loop body can be executed in the same time, the order of execution of the operations $T[X]$ ⇅ $2 \cdot \; T[X]$ is irrelevant, neither of them uses the result of the other.

In the second case the problem is not that simple. The order of execution of the operations can be arbitrary, but the variable S is kind of a collector variable, it has an accumulator rule.

At last, in the third example the reversal of the order of operations is not interpretable, as we clearly have to use the result of the previous computation in the next iteration step. Here, the parallelization can be done only after higher level mathematical considerations and analysis, basically with a unique solution to each problem.

Of course there can be a structure of instructions where we hid the dependency of the variables, so basically it cannot be found out. We may assume though that during the creation of the program code it was not the point to make it difficult to understand, but to cooperate somehow to reach parallelization.

We may say that in case of each loop we can reach a higher level of parallelization with a unique arrangement of the instructions.

The next level of the parallelization of the blocks of instructions is the block in the example III, line 5-7. Here the relation between the values of the variables is more complex, despite that the method is simpler. However Dijkstra's analysis shows that the output values are clearly depending on the input values:

$$D_{y_1} = \left(\{Y_1\}_r = Y_2\right)$$

$$D_{y_2} = \left(\{Y_2\}_r = Y_1 + Y_2\right)$$

We can see that the parallelizing is possible, but we need a kind of synchronizing, and an appropriate delay before the storage of the results.

6.3.6.3.5. The construction of the program structure graph

If we have determined the dependency of the variables, we can start to determine the dependency of the instructions.

Using the direct dependency of the variables we can set up directly a simple dependency graph.

Decompose the program into maximal linear sequences. (A sequence is linear if it does not contain a loop. It will be maximal if we cannot expand it to any direction such that it stays linear.)

Inside a linear segment, we link instruction i with instruction j (in a directional way, $i \to j$), if $j > i$, one of the variables (let's say X) in instruction i gets a value, the value of X will be used in instruction j and during the instructions $i + 1 \ldots j - 1$, X does not get a new value. Thus we can describe the syntactical dependency inside a sequence. (We cannot observe the semantic dependency this way.)

If the dependency relation of the instructions are built segment by segment, we can tell the in and output variables of each one. Input variables are those that are taking part in some assignment or in some condition evaluation in the sequence, such that they did not get a value during the sequence. Output variables are those that get a new value during the sequence.

This analysis can have different variants. Let's see for example the following part of a program:

```
C =
1. if X < 10 then
2.       Y ← 2Z
3. else
4.       X ← 2Y
5. endif
```

If we take this due to the previous definition, the sets of the in and output variables:

$RIn(C) = \{X, Y, Z\}$

$ROut(C) = \{X, Y\}$

We can see that Y and Z are not inputs to instruction 2 and 4, so it would have no sense to delay the execution because of an occasional waiting for a seeming variable.

But we can rewrite the program to an equivalent code:

```
C' =
1. B ← ( X < 10 )
2.   ( B ) Y ← 2Z
3.   ( ¬ B ) X ← 2Y
```

The advantage of this transcription will appear when we create the dependency graph of each instruction. Then we do not have to treat two instructions together when we establish the connection, but we can use the simple instructions.

We have to remark though that we have only done syntactical analysis, we cannot simplify the following type of program:

```
1. if X < 10 then
2.     Y ← Z
3. else
4.     Y ← Z
5. endif
```

The set of syntactical in and output variables can be defined with the following method:

Let the instructions be numbered inside a given sequence from 1 to *n*.

```
1. SIn ← Ø
2. SOut ← Ø
3. for i ← 1 … n  do
4.     SIn ← SIn ∪ (SIn(C_i)\SOut)
5.     SOut ← SOut ∪ SOut (C_i)
6. enddo
```

4. Questions and exercises

1. Write an algorithm to sum *n* numbers in *log*(*n*) time!

2. Write an algorithm to compute a^n in *n* time!

3. Give the above algorithms by flow chart and data flow graph representation!

5. Literature

J. W. Bakker: *Mathematical theory of program correctness;* Prentice-Hall International, 1980

T. H. Cormen, C. E. Leiserson, R. L. Rivest: *Algoritmusok (Introduction to Algorithms)* Műszaki Könyvkiadó, 1999 (The MIT Press, 1990)

D. E. Knuth: *A számítógép-programozás művészete (The Art of Computer Programming)*; 1-3. kötet: Alapvető algoritmusok (Fundamental Algorithms), Szeminumerikus algoritmusok , (Seminumerical Algorithms), Keresés és rendezés (Sorting and Searching), Műszaki Könyvkiadó, 1994 (Addison-Wesley Professional)

L. Rónyai, G. Ivanyos, R. Szabó: *Algoritmusok;* Typotex, 1998 [Algorithms, in Hungarian]

J. A. Sharp: Data Flow Computing: *Theory and Practice;* Intellect Books, 1992

7. fejezet - The forms of parallelism

7.1. The two basic forms of the parallelism

In the last part of this book, instead of a summary, we investigate the two basic forms of the parallelism as two extremes. In most cases at parallel computations and parallel algorithms they appear in mixed forms. The backtracking search strategy is very similar to the way as a human try to find the way out from a labyrinth: it is a sequential algorithm, similarly as a one-tape Turing machine works sequentially. Opposite to this, if there are several agents, then one can solve a problem in a parallel manner. The well-known graph-search algorithms have several active nodes of the search tree at the same time, and this gives a possibility to parallelise the search. The genetic algorithms also use several candidate solutions at the same time, but in this case the random factor plays also an important role.

The two basic forms of the parallelism in computations are the "and-parallelism" and the "or-parallelism". We start with the latter concept.

7.1.7.1.1. The "or-parallelism"

The "or-parallelism" is similar to the path searching model called "Chinese army". The task is to carry a message to the other side of a mountain. A massive parallel solution is the following: each solider has got the message and they start at the same time. At every branching of the path the group reaching this branching is divided to as many subgroups as many path follow the one they arrived to this branching, and thus some soldiers follow the path in every direction. Since it is the Chinese army, there is enough number of soldiers to fulfil this condition at every branching. In this way, if there is a path to the other side of the mountain, then some soldiers find it and fulfil the task.

At this type of the parallelism, in many cases, the same algorithm is running on various data and any of these data (and so these runs) can give a/the solution. In this sense this type of the parallelism is closely connected to the non-deterministic algorithms/guessing. It would be enough to compute only one path of the possible paths/one branch of the or-branches, the others not giving anything to the solution. But without knowing which path/branch will give the solution, we may try/compute every at the same time if we have enough parallel resources. This type of parallelism characterizes also the data-parallel computations. Using sequential computer with a good luck (having a good idea, oracle, or heuristics) the solution can be provided in the same time as with or-parallel computation. This factor of "good-luck" can be eliminated by deterministic, but maximal (massively) parallel algorithm.

Our super Turing-machine works in a similar manner in the example to find/copy the longest tape-content by working on all data (tape) at the same time.

7.1.7.1.2. The "and-parallelism"

Opposite to the above detailed or-parallelism, the "and-parallelism" is based on the "divide and conquer" principle. The problem is divided to subproblems that can be solved in a parallel way and their solution together gives the solution of the original problem. This concept can be imagined as the so-called problem-reduction way of problem solving in artificial intelligence with the condition that the subproblems are independent of each other. Among the models shown in this book, for instance, the L-systems can be considered to use this type of parallelism, since to obtain the derived word all the (massively parallel applied) rewriting steps are needed.

This type of parallelism is used in the so-called "high-performance computing". This type of parallelism also effects the time to obtain the solution that can be reduced very effectively by using the parallel resources in a clever way.

The problem is usually that it is not easy at all to divide the problem to subproblems in an appropriate way. There is no automatic method to do this to use the parallel resources in a clever way.

2. Questions and exercises

1. Present the two basic form of the parallelism.

2. Characterize the models that are presented in this book. Which type of parallelism can be seen in various models? Which model has (mostly) one type of parallelism? In which model can both type of parallelism observed?

3. Literature

R. Loos, B. Nagy: *On the Concepts of Parallelism in Biomolecular Computing*, Triangle: Language, Literature, Computation 6 (2011), 109-118.

B. Nagy: *On the Notion of Parallelism in Artificial and Computational Intelligence*, Proceedings of the 7th International Symposium of Hungarian Researchers on Computational Intelligence, Budapest, Hungary, (2006), 533-541.