Applied Automata Theory

Prof. Dr. Wolfgang Thomas
RWTH Aachen

Course Notes compiled by
Thierry Cachat
Kostas Papadimitropoulos
Markus Schlütter
Stefan Wöhrle

November 2, 2005

Note

These notes are based on the courses "Applied Automata Theory" and "Angewandte Automatentheorie" given by Prof. W. Thomas at RWTH Aachen University in 2002 and 2003.

In 2005 they have been revised and corrected but still they may contain mistakes of any kind. Please report any bugs you find, comments, and proposals on what could be improved to
`skript@i7.informatik.rwth-aachen.de`

Contents

Chapter 0

Introduction

Automata are systems consisting of states, some of them designated as initial or final, and (usually) labeled transitions. In classical automata theory recognizable languages or sets of state sequences have been investigated. Non-classical theory deals with several aspects, e.g. infinite sequences and non terminating behavior, with bisimulation, or with different kinds of input objects, e.g. trees as input structures.

Automata (more generally: transition systems) are the main modeling tools in Computer Science. Whenever we want to design a program, a protocol or an information system, or even just describe what it is intended to do, its characteristics, and its behavior, we use these special kinds of diagrams. Consequently, there is a variety of applications of automata in Computer Science, for example:

- as sequential algorithms (pattern matching, see Figure 1)

  They implement exactly those tasks that we normally connect to automata, namely they accept letters as an input, and on termination they reach a state, which describes whether or not the text that has been read possesses a certain property.

- as models of the execution structure of algorithms (flow diagrams, see Figure 2.)

  Before actually implementing an algorithm in some programming language, one usually describes it by using some 2-dimensional model, that represents the steps that a run of this algorithm may take and the phases that it may go through.

- as a formalism to system description (see Figure 3)

- as a formalism to system specification (see Figure 4)
  Before developing a system, we want to express its potential capabilities.

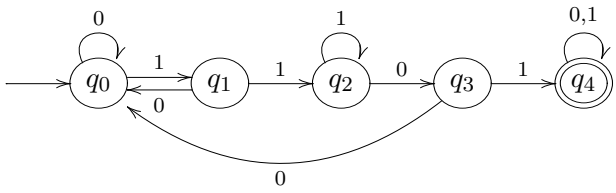


Figure 1: Deterministic automaton searching for 1101

In this course we are almost always concerned with finite automata, which is the area of automata theory where most questions are decidable. As we know, even Turing Machines can be seen as automata, but in this case, all main questions are undecidable. On the contrary, in the area of finite automata, practically all interesting questions that one may ask can be answered by an algorithm; and this is what makes them so special and useful in Computer Science.

Considered as mathematical objects, automata are simply labeled, directed graphs. However, this view on automata is not very interesting, since it can only be used to answer rather elementary questions like connectivity or reachability. What makes automata theory more complicated, is the fact that these special kinds of graphs have a certain behavior, they express a special meaning. This leads us to the *semantics* of finite automata. Some typical kinds of semantics are:

- the language being recognized (consisting of finite or infinite words)
- the word function being computed (automata with output)
- a (bi-)simulation class
- a partial order of actions (in Petri Nets)
- a set of expressions (in tree automata)

By considering these semantics, we can for example define an equivalence between two completely different automata in case they recognize exactly the same language. Moreover one can ask, which one of these two automata is the "best" in terms of space complexity (e.g. number of states) and whether there exists a better automaton. Furthermore, we can ask whether there is a way even to compute the best possible automaton that can recognize this language. Some other questions and problems that arise are listed below.

- Compare automata with respect to their expressive power in defining languages, connect this with alternative formalisms like regular expressions or logic.

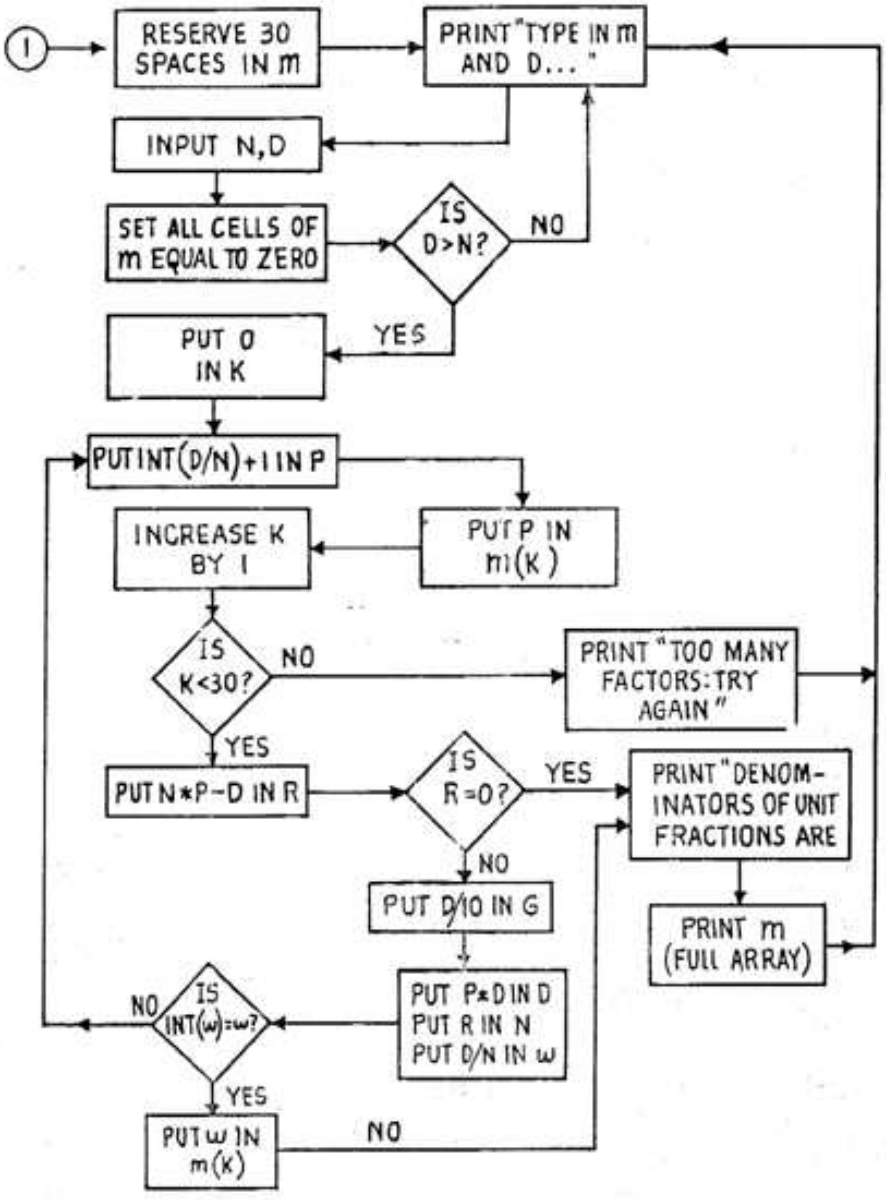


Figure 2: Flow diagram

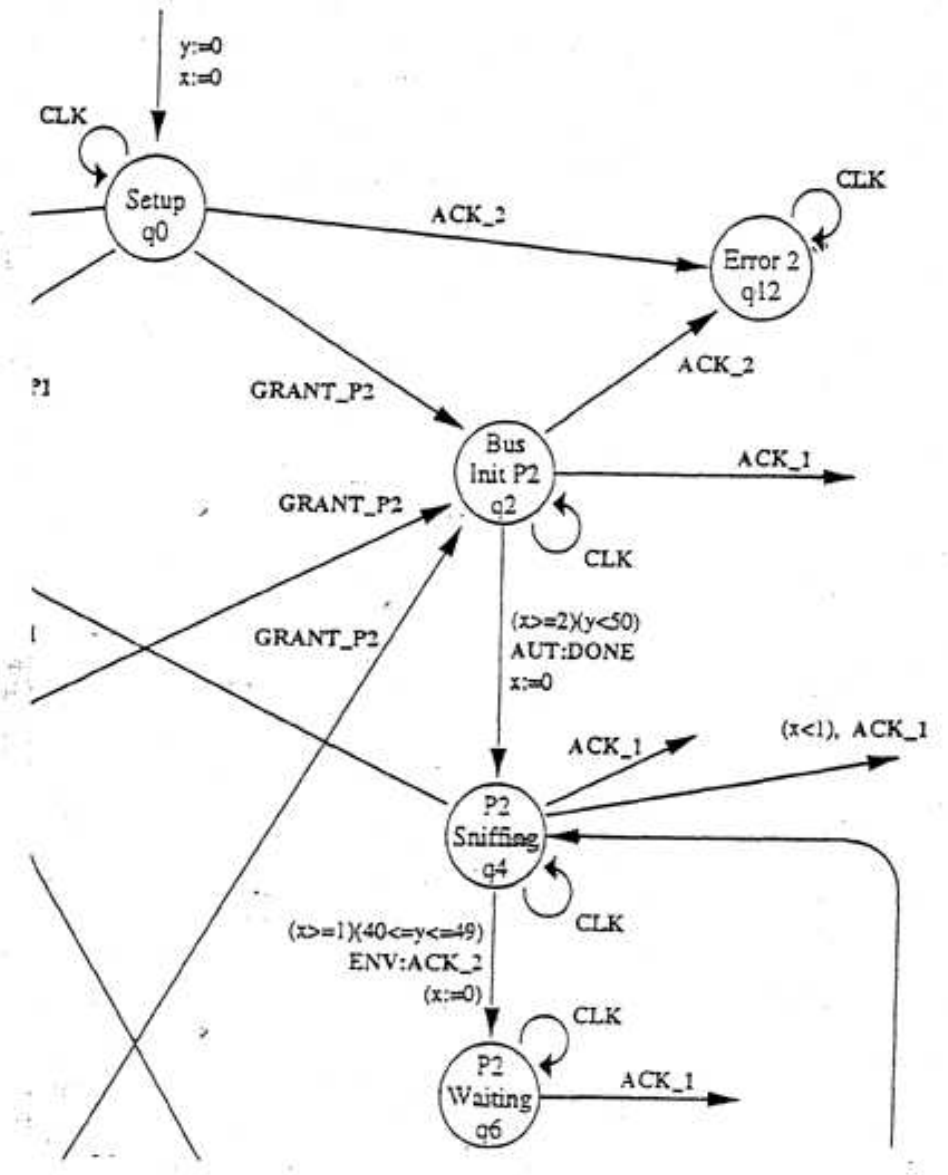


Figure 3: Transition diagram

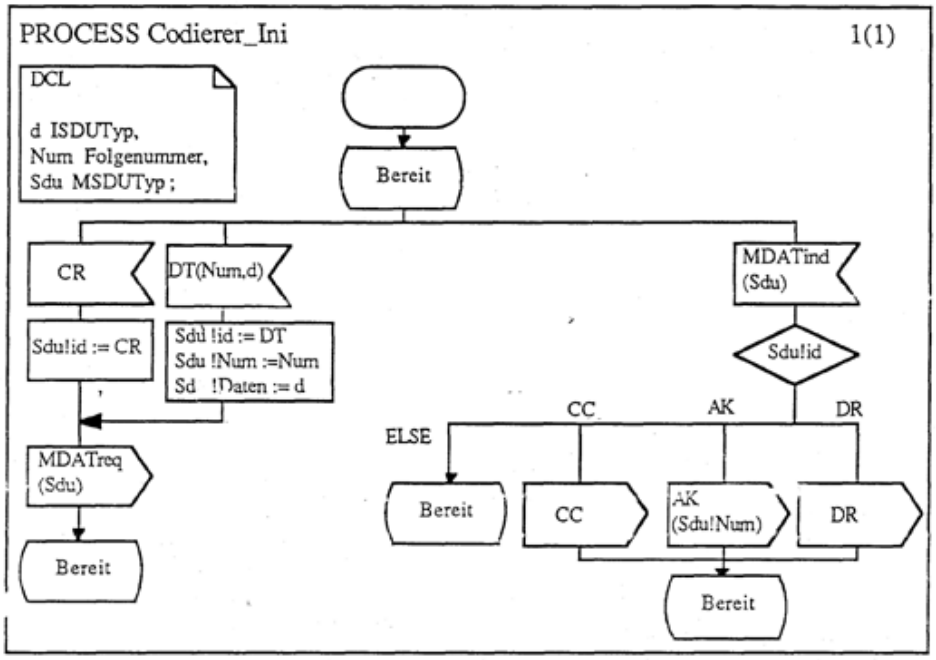


Figure 4: SDL-diagram

- Describe and evaluate algorithms for transformation and composition of automata and evaluate their complexity.
- Is it possible to find the simplest structure or expression to recognize a given language? In the case of automata, simplicity is measured on the number of states (minimization). In the case of logic formulas it may be measured e.g. on its length.
- Which questions about automata are decidable and, for those that are, what complexity do the corresponding algorithms have (e.g. concerning equivalence)?

Consequently, automata theory has to intersect with other related areas of Computer Science like:

- computability and complexity theory
- logic
- circuit theory
- algorithmic graph theory

The course and therefore these course notes are structured as follows: Chapter 1 deals with the connection of automata and logic and with the paradigm of model checking. In Chapter 2 equivalence issues including minimization and bisimulation are discussed. Chapter 3 focuses on tree automata and their applications. In Chapter 4 pushdown systems as a first model with an infinite set of configurations are investigated. The next two chapters deal with automata models which allow communication and concurrency between different processes, namely communicating finite state machines in Chapter 5, including the graphical formalism of message sequence charts, and Petri nets in Chapter 6.

0.1 Notation

An *alphabet* is a finite set. Its elements are called *symbols*. We usually denote an alphabet by Σ, and its elements by $a, b, c, \ldots$. A *word* over an alphabet Σ is a finite sequence of letters $a_1 a_2 \ldots a_n$ with $a_i \in \Sigma$ for $1 \leq i \leq n$. n is called the *length* of $a_1 a_2 \ldots a_n$. Words are usually named $u, v, w, u_1, u_2, \ldots$. By ϵ we denote the word of length 0, i.e. the *empty word*. Σ^* (Σ^+) is the set of all (nonempty) words over Σ. The *concatenation* of words $u = a_1 \ldots a_n$ and $v = b_1 \ldots b_m$ is the word $uv := a_1 \ldots a_n b_1 \ldots b_m$. Finally, we use capital calligraphic letters $\mathcal{A}, \mathcal{B}, \ldots$ to name automata.

A set $L \subseteq \Sigma^*$ is called a *language* over Σ. Let $L, L_1, L_2 \subseteq \Sigma^*$ be languages. We define the *concatenation* of L_1 and L_2 to be the set $L_1 \cdot L_2 := \{uv \mid u \in L_1 \text{ and } v \in L_2\}$, also denoted by $L_1 L_2$. We define L^0 to be the language $\{\epsilon\}$, and $L^i := LL^{i-1}$. The *Kleene closure* of L is the language $L^* := \bigcup_{i \geq 0} L^i$, and the *positive Kleene closure* L^+ is the Kleene closure of L without adding the empty word, i.e. $L^+ := \bigcup_{i \geq 1} L^i$.

Regular expressions over an alphabet Σ are recursively defined: $\emptyset$, ϵ, and a for $a \in \Sigma$ are regular expressions denoting the languages $\emptyset, \{\epsilon\}$, and $\{a\}$ respectively. Let r, s be regular expressions denoting the languages R and S respectively. Then also $(r + s)$, $(r \cdot s)$, and (r^*) are regular expressions denoting the languages $R \cup S$, RS, and R^*, respectively. We usually omit the parentheses assuming that * has higher precedence than $\cdot$ and $+$, and that $\cdot$ has higher precedence than $+$.We also denote the regular expression $(r + s)$ by $r \cup s$, and $r \cdot s$ by rs.

0.2 Nondeterministic Finite Automata

Definition 0.1 A *nondeterministic finite automaton* (NFA) is a tuple $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ where

- Q is a finite set of states,
- Σ is a finite alphabet,

- $q_0 \in Q$ is an initial state,
- $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation, and
- $F \subseteq Q$ is a set of final states.

A *run* of $\mathcal{A}$ from state p via a word $w = a_1 \dots a_n \in \Sigma^*$ is a sequence $\varrho = \varrho(1) \dots \varrho(n+1)$ with $\varrho(1) = p$ and $(\varrho(i), a_i, \varrho(i+1)) \in \Delta$, for $1 \leq i \leq n$. We write $\mathcal{A} : p \xrightarrow{w} q$ if there exists a run of $\mathcal{A}$ from p via w to q.

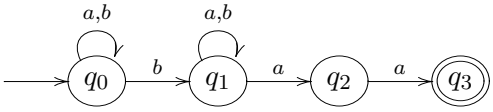
We say that $\mathcal{A}$ accepts w if $\mathcal{A} : q_0 \xrightarrow{w} q$ for some $q \in F$.

The language *recognized* by an NFA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ is

$$L(\mathcal{A}) = \{w \in \Sigma^* \mid \mathcal{A} : q_0 \xrightarrow{w} q \text{ for some } q \in F\}.$$

Two NFAs $\mathcal{A}$ and $\mathcal{B}$ are *equivalent* if $L(\mathcal{A}) = L(\mathcal{B})$. An NFA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ is called *complete* if for every $p \in Q$ and every $a \in \Sigma$ there is a $q \in Q$ such that $(p, a, q) \in \Delta$.

Example 0.2 Let $\mathcal{A}_0$ be the NFA presented by the following transition graph.



The language accepted by $\mathcal{A}_0$ is

$$L(\mathcal{A}_0) = \{w \in \{a, b\}^* \mid w \text{ contains a } b \text{ and endswith } aa\}.$$

The word *abbaaa* is accepted by $\mathcal{A}_0$, because there exists a run, e.g. 1122234, yielded by this word that starts in the initial state and leads the automaton to a final one. We notice the nondeterminism in state q_0, where on the occurrence of b the automaton can choose between staying in the same state or proceeding to state q_1, as well as in state q_1, where an analogous choice has to be made when reading an a. ⊠

0.3 Deterministic Finite Automata

Definition 0.3 (DFA) A *deterministic finite automaton* (DFA) is a tuple $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $q_0 \in Q$ is an initial state,
- $\delta : Q \times \Sigma \to Q$ is a transition function, and

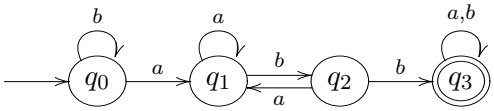
- $F \subseteq Q$ is a set of final states.

We can extend the transition function δ to words by defining

$$\begin{array}{rccl} \delta^* : & Q \times \Sigma^* & \to & Q \\ & (p, w) & \mapsto & q \text{ if } \mathcal{A} : p \xrightarrow{w} q \end{array}$$

This can also be defined inductively by $\delta^*(p, \epsilon) = p$ and $\delta^*(p, wa) = \delta(\delta^*(p, w), a)$. The language accepted by a DFA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ is $L(\mathcal{A}) := \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$. To simplify notation we often write δ instead of δ^*. Note that by definition every DFA is complete. As every automaton with a partially defined transition function can easily be transformed in to DFA by adding an additional state, we often also consider automata with partial transition function as DFAs.

Example 0.4 Consider the following DFA $\mathcal{A}_1$.



It is easy to evaluate the extended transition function δ^* by following the corresponding path in the automaton, e.g. $\delta^*(q_0, aaa) = q_1$ or $\delta^*(q_0, aabb) = q_3$. The language accepted by $\mathcal{A}_1$ is $L(\mathcal{A}_1) = \{w \in \{a, b\}^* \mid w \text{ has an infix } abb\} = \Sigma^* abb \Sigma^*$. $\mathcal{A}_1$ is indeed complete, because from every state, there is a transition for every letter of the alphabet. ⊠

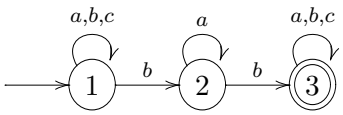
Chapter 1

Automata and Logical Specifications

1.1 MSO-Logic over words

This chapter is concerned with the relation between finite automata and logical formulas. In the first section we focus on the so called *Monadic Second-Order (MSO) Logic*, by explaining its characteristics and expressiveness. Before going into the details of the definition of this logic, we illustrate the motivation that leads us to study it as a helping tool in the area of automata.

Example 1.1 We consider the following automaton over $\Sigma = \{a, b, c\}$:



It is rather easy to see that this nondeterministic automaton accepts the words that have the following property:

"There are two occurrences of b between which only a's occur."

This can also be expressed in terms of logic by requiring the existence of two positions in the word (say x and y) labeled with b, such that x is before y, and every position that comes after x and before y (if there is such a position at all) is labeled by a. A logical formula φ that expresses this kind of property is:

$$\exists x \exists y (x < y \wedge P_b(x) \wedge P_b(y) \wedge \forall z((x < z \wedge z < y) \rightarrow P_a(z)))$$

A word w is accepted by the above automaton iff w satisfies φ. ⊠

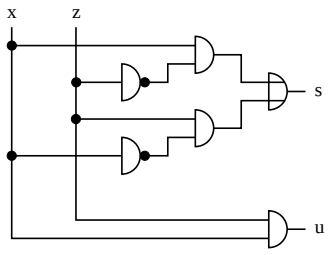


Figure 1.1: A half-adder.

Our final goal is to find a general equivalence between automata and logical formulas. This means that the language that we choose to represent logical formulas has to be of exactly the same expressive power as finite automata. Furthermore, we want to find a way to construct for a given automaton an equivalent logical formula and vice versa. This may have been simple in the above example because we did not really translate the automaton directly to the logical formula. What we actually did, was, using our common sense, comprehend and capture in mind the language that is defined by the automaton, then intuitively express the properties of this language in such colloquial English that it can be easily coded into a logical formula, and finally do the respective translation.

Unfortunately, these transformations cannot (yet) be performed by a machine. A machine needs explicitly defined instructions to translate one structure to another. And in this case it is not an easy task. On the one hand logical formulas can be build up by explicit parts; these parts are basic logical formulas themselves, we can partially order them in increasing complexity and we can combine them to result in a new logical formula that somehow possesses the power of all its components. In other words, logical formulas can be defined inductively. On the other hand, we cannot decompose an automaton to any meaningful basic parts; at least not in the general case. Neither are there operators in its definition that always combine its components to construct the whole automaton. We can of course disassemble an automaton to its states, but each one of these generally does not have any meaning anymore. It is the complex way the states are connected over labeled transitions what makes the automaton work in the desired way.

We know a similar connection between two different structures from Circuit Theory, namely the equivalence between Boolean Logic and Circuits.

Example 1.2 Consider the well known circuit (half-adder) of Figure 1.1:

Its behavior can be described by the following Boolean formulas:

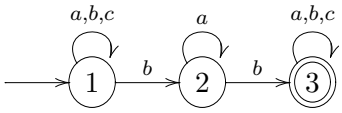
$$s = x \cdot \overline{z} + \overline{x} \cdot z, \qquad u = x \cdot z$$

⊠

There is a translation from circuits to Boolean formulas and vice versa. The main difference between circuits and automata is that the former are acyclic graphs. In the above example we can notice how data flows "from left to right". In the case of automata we may have loops, consisting of one or more states, and this is what makes them more complex to handle.

We also know some other roughly speaking "logical formulas" that describe the behavior of automata, namely regular expressions over Σ. These are built up by the symbols of Σ, $\emptyset$, ϵ with $+$ (for union), $\cdot$ (for concatenation) and * (for the iteration of concatenation).

Example 1.3 Consider the following automaton:



The language that it recognizes can be defined by the following regular expression (the concatenation symbol $\cdot$ is usually omitted):

$$(a+b+c)^* \cdot b \cdot a^* \cdot b \cdot (a+b+c)^* \text{ , short } \Sigma^* b a^* b \Sigma^* \ .$$

⊠

Theorem 1.4 (Kleene) *A language L is definable by a regular expression iff L is DFA or NFA recognizable.*

But if we already have such an equivalence between automata and regular expressions why do we need more? Why are regular expressions not enough and what is it that makes us need the logical formulas?

To realize the advantages of logical formulas, we consider Example 1.1 as well as two variations of it, namely:

(Example 1.1) "There are two occurrences of b, between which only a's occur"

Example 1.5 "There are no two occurrences of b, between which only a's occur" ⊠

Example 1.6 "Between any two occurrences of b only a's occur" ⊠

We notice that 1.5 is the complement of 1.1, whereas 1.6 turns the existential statement into a universal statement. These two relations between these examples can be easily expressed using logical formulas (e.g. in the case of 1.5):

$$\neg\exists x \exists y (x < y \wedge P_b(x) \wedge P_b(y) \wedge \forall z ((x < z \wedge z < y) \rightarrow P_a(z)))$$

On the other hand, by using regular expressions one can conveniently describe the occurrences of patterns but it takes a lot of effort to describe the non-occurrence of patterns. This holds even when defining a language by an automaton. The language of Example 1.5 is defined by the following regular expression:

$$(a + c)^*(\epsilon + b)(a^*c(a + c)^*b)^*(a + c)^*$$

So far we have used the following notation in our logical formulas:

- variables $x, y, z, \ldots$ representing positions of letters in a word (from 1 (= min) up the length n (= max) of a word)
- formulas $P_a(x)$ ("at position x there is an a"),
  $x < y$ ("x is before y")

Before extending the expressive power of our logical formulas, we look at some further examples:

Example 1.7 Consider the language consisting of words with the property:

"right before the last position there is an a"

Regular expression: $\Sigma^* \cdot a \cdot \Sigma$
Logical formula: $\exists x\ (P_a(x) \wedge x + 1 = \max)$

Instead of the operation "+1" we use the successor relation

$$S(x, y) \quad (\text{"}x \text{ has } y \text{ as a successor"}).$$

Now the formula looks like this:

$$\exists x\ (P_a(x) \wedge S(x, \max))$$

⊠

Example 1.8 The property of even length:
Even length, as a property of non-empty words, is regular:

$$(\Sigma\Sigma) \cdot (\Sigma\Sigma)^*$$

A corresponding logical formula would be:

$$\exists x\ (x + x = \max)$$

But introducing addition of positions leads us outside the class of regular languages. In other words, the +-operator provides us with too much

expressive power. By using it we can e.g. express the non-regular language $\{a^i b^i \mid i > 0\}$ by the formula

$$\exists x \ (x + x = \max \wedge \forall y(y \leq x \rightarrow P_a(y)) \wedge \forall z(x < z \rightarrow P_b(z)))$$

Is there a way to capture the property of even length without using the +-operator? To this purpose we consider a set of positions of a word and require that it:

- contains the first position,
- then always contains exactly every second position,
- does not contain the last position

We use X as a variable for sets of positions and we write $X(y)$ to express that "position y is in X". Now we are ready to express the property of even length in the following way:

$$\exists X \ (X(\min) \wedge \forall y \forall z \ (S(y,z) \rightarrow (X(y) \leftrightarrow \neg X(z))) \wedge \neg X(\max))$$

⊠

Definition 1.9 (Monadic Second Order Logic (MSO-Logic)) *MSO formulas* are built up from

- variables $x, y, z, \ldots$ denoting positions of letters,
- constants min and max
- variables $X, Y, Z, \ldots$ denoting sets of positions
- the atomic formulas (with explicit semantics)
  - $x = y$ (equality)
  - $S(x, y)$ "x has y as a successor"
  - $x < y$ "x is before y"
  - $P_a(x)$ "at position x there is an a"
  - $X(y)$ "$y \in X$"
  - (instead of x, y min, or max can also be used)
- the usual connectors $\neg, \wedge, \vee, \rightarrow$, and $\leftrightarrow$ and the quantifiers $\exists$, $\forall$

A non-empty word $w = b_1 \ldots b_m$ over the alphabet $\Sigma = \{a_1, \ldots, a_n\}$ defines the *word model*

$$\underline{w} = \big(\underbrace{\{1, \ldots, m\}}_{=: \mathrm{dom}(w)}, S^w, <^w, \min^w, \max^w, P^w_{a_1}, \ldots, P^w_{a_n}\big)$$

Where:

- $\mathrm{dom}(w)$ is the set of (all) positions $1, \ldots, |w|$
- S^w is the successor- and $<^w$ the smaller-than-relation on $\mathrm{dom}(w)$
- $\min^w = 1$ and $\max^w = |w|$
- $P_{a_i}^w := \{j \in \mathrm{dom}(w) \mid b_j = a_i\}$ for $i = 1, \ldots, n$

Remark 1.10 *We decide to use only non-empty words for models for two reasons. First, in mathematical logic it is generally convenient to use models that have at least one element. Otherwise, we have to bother with some additional considerations and special cases. Second, in case we accept the empty word ϵ to define a model, we are no longer able to make use of the constants* min *and* max*.*

MSO stands for "monadic second-order":
Second-order because it allows quantification not only over (first-order) position variables but also over (second-order) set variables.
Monadic because quantification is allowed at most over unary (monadic) relations, namely sets. For an alphabet Σ we get the $\mathrm{MSO}_\Sigma[S, <]$-formulas. In the special case that we use quantifiers only over first-order variables (ranging over positions) we get the $\mathrm{FO}_\Sigma[S, <]$-formulas. Usually, the lowercase Greek letters $\phi, \chi, \psi, \ldots$ are used for formulas. A variable x_i or X_i is *free* in a formula if it does not occur within the scope of a quantifier. The notation $\phi(x_1, \ldots, x_m, X_1, \ldots, X_n)$ for a formula indicates that at most the variables $x_1, \ldots, x_m, X_1, \ldots, X_n$ occur free in ϕ.

Assume $\phi(x_1, \ldots, x_m, X_1, \ldots, X_n)$ is given. To interpret the truth value of ϕ we need:

- a word model $\underline{w}$ (with predicates P_a^w for the symbols $a \in \Sigma$)
- positions $k_1, \ldots, k_m$ as interpretations of $x_1, \ldots, x_m$
- sets $K_1, \ldots, K_n$ of positions as interpretations of $X_1, \ldots, X_n$

Consequently our complete model is:

$$(\underline{w}, k_1, \ldots, k_m, K_1, \ldots, K_n)$$

Now,

$$(\underline{w}, k_1, \ldots, k_m, K_1, \ldots, K_n) \models \phi(x_1, \ldots, x_m, X_1, \ldots, X_n)$$

expresses the fact that ϕ holds in $\underline{w}$, if x_i is interpreted by k_i and X_i by K_i. In short notation we write:

$$\underline{w} \models \phi[k_1, \ldots, k_m, K_1, \ldots, K_n]$$

Example 1.11 Let $\Sigma = \{a, b\}$, $w = abbab$ and

1. $\phi_1(x_1, X_1) := P_a(x_1) \wedge \forall y\ (x_1 < y \wedge X_1(y) \rightarrow P_b(y))$
2. $\phi_2(x_1, x_2) := \exists z\ (x_1 < z \wedge z < x_2 \wedge P_b(z))$
3. $\phi_3 := \exists x \exists y\ (P_a(x) \wedge S(x, y) \wedge P_b(y))$

Then:

1. $(\underline{w}, 1, \{2, 3, 4\}) \not\models \phi_1(x_1, X_1)$
2. $(\underline{w}, 2, 4) \models \phi_2(x_1, x_2)$
3. $\underline{w} \models \phi_3$

⊠

The last example-formula does not have any free variables, and can therefore be interpreted given just a word as a model. Such formulas are called *s*entences. If we collect all words that satisfy such a sentence, and put them into a set, then this set is the language defined by the sentence. Formally, for an $\mathrm{MSO}_\Sigma[S, <]$-sentence ϕ, let $L(\phi) := \{w \in \Sigma^+ \mid \underline{w} \models \phi\}$ be the language defined by ϕ. Such a language is called $\mathrm{MSO}_\Sigma[S, <]$-definable (short: MSO-definable).

In case no set quantifiers are needed in ϕ, we call the language $\mathrm{FO}_\Sigma[S, <]$-definable or just FO-definable. FO stands for first-order, because now quantifiers range only over position variables.

Example 1.12 For $\Sigma = \{a, b, c\}$

- $\Sigma^* b a^* b \Sigma^*$ is FO-definable, namely by
  $\exists x \exists y\ (x < y \wedge P_b(x) \wedge P_b(y) \wedge \forall z\ ((x < z \wedge z < y) \rightarrow P_a(z)))$
- $\Sigma^* a \Sigma$ is FO-definable, namely by
  $\exists x\ (P_a(x) \wedge S(x, \max))$
- $(\Sigma\Sigma) \cdot (\Sigma\Sigma)^*$ is MSO-definable, namely by
  $\exists X\ (X(\min) \wedge \forall y \forall z\ (S(y, z) \rightarrow (X(y) \leftrightarrow \neg X(z))) \wedge \neg X(\max))$

Furthermore, we shall see in some future section that this formula cannot be expressed in FO-logic at all. ⊠

Remark 1.13 *In FO- and MSO-formulas we can eliminate min and max.*

Example 1.14 Instead of

$$\exists x\ (P_a(x) \wedge S(x, \max))$$

we write

$$\exists x\ (P_a(x) \wedge \exists y\ (S(x, y) \wedge \neg\exists z\ S(y, z)))$$

⊠

In general, substitute $\psi(\max)$, by:

$$\exists y(\psi(y) \wedge \neg\exists z\ y < z)\ .$$

Analogously for min.

Remark 1.15 *In MSO-formulas we can eliminate* $<$.
Proof: *Replace* $x < y$ *by*

$$\exists X\ \big[\neg X(x) \wedge \forall z\forall z'\ (X(z) \wedge S(z, z') \rightarrow X(z')) \wedge X(y)\big]$$

Example 1.16 Some more examples over $\Sigma = \{a, b, c\}$:

1. $\{w \in \Sigma^+ \mid$ in w every a is followed only by a's, until a b occurs$\}$ is definable by:

$$\forall x\ \big(P_a(x) \rightarrow \exists y\ \big(x < y \wedge P_b(y) \wedge \forall z\ (x < z \wedge z < y \rightarrow P_a(z))\big)\big)$$

2. a^+ is definable by $\forall x\ P_a(x)$.

3. $a(ba)^*$ is definable by

$$P_a(\min)\wedge\forall x\forall y\ \big(S(x, y) \rightarrow (P_a(x) \rightarrow P_b(y))\wedge(P_b(x) \rightarrow P_a(y))\big)\wedge P_a(\max)$$

⊠

1.2 The Equivalence Theorem

Now that we have defined the logic we are going to use, we shall present a way to translate any given automaton to a formula of this logic and vice versa. To this purpose we do the following preparation: A formula $\phi(X_1, \ldots, X_n)$ is interpreted on word models $(\underline{w}, K_1, \ldots, K_n)$ with $K_i \subseteq \mathrm{dom}(w)$. The set K_i of positions serves as an interpretation of X_i. To code such models by words that can be processed by an automaton we collect for every position $k \in \mathrm{dom}(w)$ the information if $k \in K_1$, $k \in K_2, \ldots, k \in K_n$ in the form of a bit vector. Formally, we represent a word $b_1 \ldots b_m$ with sets of positions $K_1, \ldots, K_n$ by a word over the alphabet $\Sigma \times \{0, 1\}^n$:

$$\begin{bmatrix} b_1 \\ (c_1)_1 \\ \vdots \\ (c_1)_n \end{bmatrix} \quad \begin{bmatrix} b_2 \\ (c_2)_1 \\ \vdots \\ (c_2)_n \end{bmatrix} \quad \cdots \quad \begin{bmatrix} b_m \\ (c_m)_1 \\ \vdots \\ (c_m)_n \end{bmatrix}$$

and we set $(c_k)_j = 1$ iff $k \in K_j$.

Example 1.17 Let $\Sigma = \{a, b\}$, $w = abbab$, $K_1 = \emptyset$, $K_2 = \{2, 4, 5\}$ and $K_3 = \{1, 2\}$.

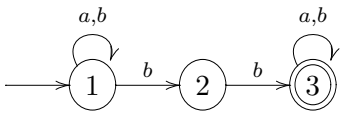
$$\begin{array}{c|ccccc} w & a & b & b & a & b \\ K_1 & 0 & 0 & 0 & 0 & 0 \\ K_2 & 0 & 1 & 0 & 1 & 1 \\ K_3 & 1 & 1 & 0 & 0 & 0 \end{array}$$

⊠

Theorem 1.18 (Büchi, Elgot, Trakhtenbrot 1960) *A language $L \subseteq \Sigma^+$ is regular if and only if it is MSO-definable. The transformation in both directions is effective.*

For the proof we present regular languages by NFAs. Let us give an example first.

Example 1.19 Given the following automaton $\mathcal{A}$:



we have to construct a sentence $\phi_{\mathcal{A}}$ with

$$\underline{w} \models \phi_{\mathcal{A}} \text{ iff } w \in L(\mathcal{A}) \ .$$

In other words, $\phi_{\mathcal{A}}$ has to express that there exists an accepting run of $\mathcal{A}$ on w. To this purpose, along with $\phi_{\mathcal{A}}$ we assume the existence of three sets X_1, X_2, X_3, such that:

$$X_i = \text{set of positions, at which } \mathcal{A} \text{ is in state } i.$$

Example 1.20 *abbab.* An accepting run for this word is 112333. We encode X_1, X_2, X_3 in the following way (without the last state):

w	:	a	b	b	a	b
X_1	:	1	1	0	0	0
X_2	:	0	0	1	0	0
X_3	:	0	0	0	1	1

⊠

Naturally the automaton can only be in one state at each point of time. Therefore there is just one 1 in every column of the run. How can we describe a successful run? That is, how do we set constraints to $X_1, \ldots, X_n$? The formula we are looking for is a conjunction over 4 basic properties:

$\phi_{\mathcal{A}} := \exists X_1 \exists X_2 \exists X_3 \ \big[$"$X_1, X_2, X_3$ form a Partition"

$\wedge$ $X_1(\min)$

$\wedge$ $\forall x \forall y \ (S(x,y) \rightarrow$ "at x, y one of the transitions is applied"

$\wedge$ "at max the last transition leads to a final state"$\big]$

Now let us explain each one of these properties (except the second, which is obvious) and give the respective MSO-formulas.

- "X_1, X_2, X_3 form a Partition": Partition is an expression for the above mentioned unambiguity of the automaton state. Since there is just one 1 in every X-bitvector, X_1, X_2, X_3 have to form a partition of $\mathrm{dom}(w)$. In terms of MSO-logic, we write:

  $\forall x \ (X_1(x) \vee X_2(x) \vee X_3(x))$

  $\wedge$ $\neg \exists x \ (X_1(x) \wedge X_2(x))$
  $\wedge$ $\neg \exists x \ (X_2(x) \wedge X_3(x))$
  $\wedge$ $\neg \exists x \ (X_1(x) \wedge X_3(x))$

- "at x, y one of the transitions is applied": In other words, we need a formula to represent the whole transition relation Δ:

  $(X_1(x) \wedge P_a(x) \wedge X_1(y)) \ \vee \ (X_1(x) \wedge P_b(x) \wedge X_1(y)) \ \vee$
  $(X_1(x) \wedge P_b(x) \wedge X_2(y)) \ \vee \ (X_2(x) \wedge P_b(x) \wedge X_3(y)) \ \vee$
  $(X_3(x) \wedge P_a(x) \wedge X_3(y)) \ \vee \ (X_3(x) \wedge P_b(x) \wedge X_3(y))$

- "at position max the last transition leads to a final state": Since we want the formula to be true if and only if the word is accepted by the automaton, we have to force that the run on w ends in a final state of $\mathcal{A}$:

  $(X_2(\max) \wedge P_b(\max)) \vee (X_3(\max) \wedge P_a(\max)) \vee (X_3(\max) \wedge P_b(\max))$

$\boxtimes$

Proof of Theorem 1.18: Let $\mathcal{A} = (\underbrace{\{1, \ldots, m\}}_{Q}, \Sigma, 1, \Delta, F)$ be an NFA. The formula expressing that there exists a successful run of $\mathcal{A}$ on a word $w \in \Sigma^+$ is then given by

$\phi_{\mathcal{A}} = \exists X_1 \ldots \exists X_m \ \big[\forall x \ \big(X_1(x) \vee \ldots \vee X_m(x)\big) \wedge \bigwedge_{i \neq j} \neg \exists x \ \big(X_i(x) \wedge X_j(x)\big)$

$\wedge\ X_1(\min)$

$\wedge\ \forall x \forall y\ (S(x,y) \rightarrow \bigvee_{(i,a,j)\in\Delta}\ (X_i(x) \wedge P_a(x) \wedge X_j(y)))$

$\wedge\ \bigvee_{(i,a,j)\in\Delta, j\in F}\ (X_i(\max) \wedge P_a(\max))\big]$

Then $\mathcal{A}$ accepts w iff $\underline{w} \models \phi_{\mathcal{A}}$.

Remark 1.21 *$\phi_{\mathcal{A}}$ does not contain any universal quantifiers over set variables but only existential ones. Such an MSO-formula, namely of the form*

$$\exists X_1 \ldots \exists X_m \psi(X_1, \ldots, X_m)$$

where ψ does not contain any set quantifiers at all, is called an existential MSO-formula (EMSO-formula).

Remark 1.22 *For m states, $\lceil \log_2(m) \rceil$ set quantifiers suffice.*

Example 1.23 In the case of 4 states: Instead of the sets $X_1, \ldots, X_4$ it suffices to use X_1, X_2 along with the convention that the states $1, \ldots, 4$ correspond to the column vectors $\binom{0}{0}, \binom{0}{1}, \binom{1}{0}, \binom{1}{1}$, respectively. ⊠

Suppose now that we are given an MSO-formula ϕ and want to construct an NFA $\mathcal{A}_\phi$ such that $L(\mathcal{A}_\phi) = L(\phi)$. The main idea is to proceed by induction on the construction of $\phi(x_1, \ldots, x_m, X_1, \ldots, X_n)$.

First we show how to eliminate the first-order variables x_i and get equivalent formulas containing only set variables. To this purpose we represent the element x by the set $\{x\}$ and then work with formulas of the form $\phi(X_1, \ldots, X_n)$. We call such formulas MSO_0-formulas. Atomic MSO_0-formulas are:

- $X \subseteq Y$ and $X \subseteq P_a$
- $\mathrm{Sing}(X)$ ("X is a singleton set")
- $S(X,Y)$ ("$X = \{x\}$, $Y = \{y\}$ and $S(x,y)$")
- $X < Y$ ("$X = \{x\}$, $Y = \{y\}$ and $x < y$")

As usual, formulas are built up using $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$, $\exists$, and $\forall$.

Lemma 1.24 (MSO_0-Lemma) *Every MSO-formula $\phi(X_1, \ldots, X_n)$ is (over word models) equivalent to an MSO_0-formula.*

Proof: We proceed by induction over the structure of the given MSO-formula. In some of the intermediate steps we have to handle formulas with free first-order (element) variables. As described above we transform these variables into singleton set variables. More formally, by induction over

the structure of an MSO-formula $\psi(y_1, \ldots, y_m, X_1, \ldots, X_n)$, we construct a corresponding MSO_0-formula $\psi^*(Y_1, \ldots, Y_m, X_1, \ldots, X_n)$ with

$$(\underline{w}, k_1, \ldots, k_m, K_1, \ldots, K_n) \models \psi \text{ iff } (\underline{w}, \{k_1\}, \ldots, \{k_m\}, K_1, \ldots, K_n) \models \psi^*$$

We omit the details. The idea is illustrated in the example below. □

Example 1.25 For

$$\phi(X_1) = \forall x \ \big(P_a(x) \rightarrow \exists y \ (S(x,y) \wedge X_1(y))\big)$$

construct

$$\phi^* = \forall X \ \big(\mathrm{Sing}(X) \wedge X \subseteq P_a \rightarrow \exists Y \ (\mathrm{Sing}(Y) \wedge S(X,Y) \wedge Y \subseteq X_1)\big)$$

⊠

Corollary 1.26 *To translate formulas $\phi(X_1, \ldots, X_n)$ to automata it suffices to consider* MSO_0*-formulas.*

A formula $\phi(X_1, \ldots, X_n)$ defines a language over the alphabet $\Sigma \times \{0,1\}^n$.

Example 1.27 Let $\Sigma = \{a, b\}$ and $\phi(X_1) = \forall x \ \big(P_a(x) \rightarrow \exists y \ (S(x,y) \wedge X_1(y))\big)$

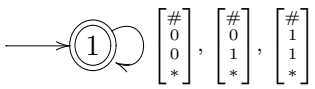
$\binom{b}{0}\binom{a}{0}\binom{a}{1}\binom{b}{1}$ satisfies ϕ

$\binom{b}{0}\binom{a}{0}\binom{a}{1}\binom{b}{0}$ does not satisfy ϕ.

$L(\phi)$ = the set of words over $\Sigma \times \{0,1\}$, such that after every a in the first component there is a 1 in the second component. ⊠

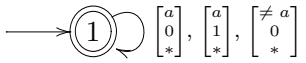
In the general case, for each $\phi(X_1, \ldots, X_n)$ we have to inductively construct a finite automaton over $\Sigma \times \{0,1\}^n$. For the base case we consider the atomic MSO_0-formulas.

- $X_1 \subseteq X_2$:
  Given a word $w \in (\Sigma \times \{1,0\}^n)^+$, an automaton has to check that whenever the first component is 1, the second component is 1 as well. This is performed by the following NFA:

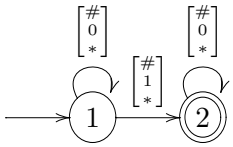


  that is, it accepts everything except $\begin{bmatrix}\#\\1\\0\\*\end{bmatrix}$, where # stands for arbitrary letters in Σ and $*$ for arbitrary bit vectors in $\{0,1\}^{n-2}$.

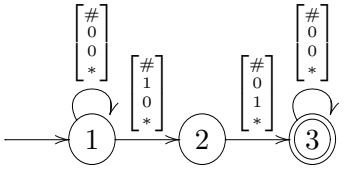
- $X_1 \subseteq P_a$:



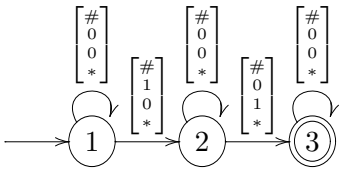
- $\mathrm{Sing}(X_1)$:



- $S(X_1, X_2)$:



- $X_1 < X_2$:



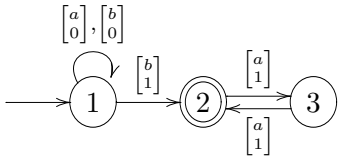
For the induction step, we suppose that for the formulas ϕ_1, ϕ_2 the corresponding NFAs $\mathcal{A}_1$, $\mathcal{A}_2$ are given and we only consider the connectors $\vee$, $\neg$ and the quantifier $\exists$. The other operators ($\wedge$, $\rightarrow$, $\leftrightarrow$ and $\forall$) can be expressed as a combination of the previous ones. Regarding the negation, $\neg\phi_1$ is equivalent to the complement automaton of $\mathcal{A}_1$. As we know, we can construct it using the subset construction on $\mathcal{A}_1$ and then, in the resulting DFA, declare all states $Q \setminus F$ as final ones. For the disjunction $\phi_1 \vee \phi_2$ we construct the automaton that recognizes $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$. As usually we do this by constructing the union automaton out of $\mathcal{A}_1$ and $\mathcal{A}_2$. To handle the existential quantifier we need some preparation.

Lemma 1.28 (Projection Lemma) *Let $f : \Sigma \rightarrow \Gamma$ be a projection of the alphabet Σ into the alphabet Γ extended to words by $f(b_1 \dots b_m) := f(b_1) \dots f(b_m)$. If $L \subseteq \Sigma^*$ is regular, then $f(L)$ is regular as well.*

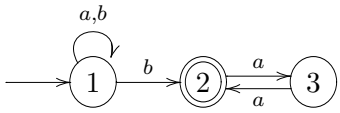
Proof: Given an NFA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ that recognizes L, we construct the NFA $\mathcal{B} = (Q, \Gamma, q_0, \Delta', F)$ with $\Delta' := \{(p, f(a), q) \mid (p, a, q) \in \Delta\}$. $\square$

Remark 1.29 *The NFA for the projection results by projecting the transition labels.*

Example 1.30 Consider the (incomplete) DFA over $\{a,b\} \times \{0,1\}$ that recognizes
$L = \left(\binom{a}{0} + \binom{b}{0}\right)^* \ \binom{b}{1} \ \left(\binom{a}{1}\binom{a}{1}\right)^*$:



The projection $f : \binom{a}{c} \mapsto a, \binom{b}{c} \mapsto b$ yields the NFA:



which recognizes $f(L)$. ⊠

Now that we have defined the necessary notions of the projection, we can deal with the existential quantifier.

Lemma 1.31 (Quantifiers Lemma) *Assume $L \subseteq (\Sigma \times \{0,1\}^n)^+$ is defined by $\phi(X_1, \ldots, X_n)$. Let $f : (\Sigma \times \{0,1\}^n) \to (\Sigma \times \{0,1\}^{n-1})$ be the canonical projection $(a, c_1, \ldots, c_n) \mapsto (a, c_1, \ldots, c_{n-1})$. Then the formula $\exists X_n \phi(X_1, \ldots, X_n)$ defines the language $f(L)$.*

Corollary 1.32 *If $\phi(X_1, \ldots, X_n)$ defines the regular language L, then $\exists X_n \phi(X_1, \ldots, X_n)$ defines a regular language as well, namely $f(L)$.*

Remark 1.33 *The NFA for $\exists X_n \varphi(X_1, \ldots, X_n)$ is constructed from the automaton for $\varphi(X_1, \ldots, X_n)$ by "omitting the last component" in the transition labels.*

Proof: (Quantifiers Lemma)
$w = b_0 \ldots b_{m-1} \in (\Sigma \times \{0,1\}^{n-1})^+$ satisfies $\exists X_n \phi(X_1, \ldots, X_n)$

iff there exists $K \subseteq \mathrm{dom}(w)$ such that $(\underline{w}, K) \models \phi(X_1, \ldots, X_n)$

iff there exist $c_0, \ldots, c_{m-1} \in \{0,1\}$, such that

$\begin{bmatrix} b_0 \\ c_0 \end{bmatrix} \ldots \begin{bmatrix} b_{m-1} \\ c_{m-1} \end{bmatrix}$ satisfies $\phi(X_1, \ldots, X_n)$

iff $b_0 \dots b_{m-1} = f(u)$ for some u with

$\underline{u} \models \phi(X_1, \dots, X_n)$, i.e. for some $u \in L$

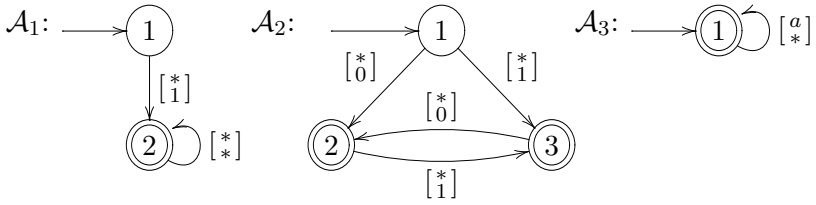
iff $b_0 \dots b_{m-1} \in f(L)$

□

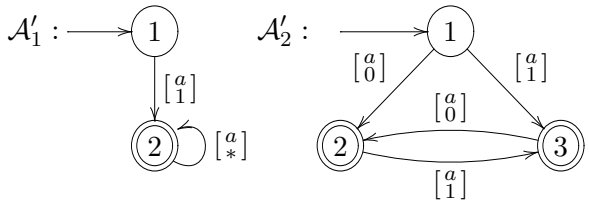
Example 1.34 For the following formula we construct an equivalent automaton.

$$\varphi = \exists X_1 \big(\underbrace{X_1(\min)}_{\psi_1(X_1)} \wedge \underbrace{\forall x \forall y (S(x,y) \rightarrow (X_1(x) \leftrightarrow \neg X_1(y)))}_{\psi_2(X_1)} \wedge \underbrace{\forall z \; P_a(z)}_{\psi_3(X_1)} \big)$$

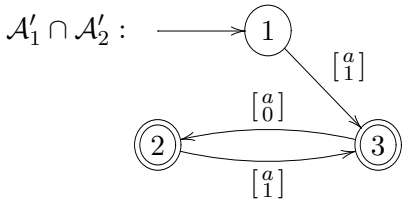
We suppose that we are directly given automata $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$ for the sub-formulas $\psi_1(X_1)$, $\psi_2(X_1)$, $\psi_3(X_1)$ respectively:



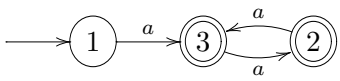
We consider the conjunctions directly (without eliminating them by using negation and disjunction). $\mathcal{A}_3$ merely requires that only a's occur in the first component. Since $\mathcal{A}_1$ and $\mathcal{A}_2$ make no restrictions about the first component, we can intersect $\mathcal{A}_3$ with both of them by always replacing the $*$ in the first component with an a:



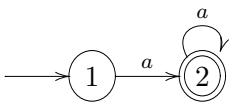
Since $\mathcal{A}'_1$ only requires that in the first position in the word model a 1 occurs in the second component, we form the intersection of $\mathcal{A}'_1$ and $\mathcal{A}'_2$ by removing the transition from $1 \rightarrow 2$. So for the formula $\psi_1(X_1) \wedge \psi_2(X_1) \wedge \psi_3(X_1)$ we get the following equivalent automaton:



The projection on Σ leads to the final automaton $\mathcal{A}$:



Obviously $L(\mathcal{A}) = a^+$, so a (better) equivalent automaton is:



⊠

The BET-Theorem allows the transformation of an MSO-formula into an equivalent automaton and vice versa. How effective are these transformations? Let us focus on the computational effort (in terms of time complexity) that is needed to perform the first one, assuming that we are given the automata for the atomic formulas. If each one of two NFAs $\mathcal{A}_1$, $\mathcal{A}_2$ has at most n states, then the NFA recognizing the

- union language has at most $2n + 1$ states (because we create a new initial state and depending on the first letter that is read, the new automaton branches to one of the two initial ones)
- intersection language has at most n^2 states
  (because we have to construct the product automaton)
- complement language has at most 2^n states (because we have to use the subset construction and turn the given NFAs to DFAs before switching final and non-final states)
- projection language has at most n states (because we use the same automata and omit one component)

Does this mean that the transformation requires exponential time with respect to the number of states of the automata? Unfortunately not. The fact that the projection may turn a DFA into an NFA and the fact that constructing the complement automaton requires to make an automaton first deterministic (subset construction) means that, whenever an alternation between negation and projection occurs (e.g. when a universal quantifier occurs), an additional exponential burden is added to our computational effort. So, in the general case, if k is the number of logical operators in the formula, then the number of states in the resulting automaton grows like $2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}\Big\}k$. The following theorem shows that we cannot hope a better bound.

Theorem 1.35 (Meyer, Stockmeyer 1971) *There is no translation of MSO-formulas φ to automata in which the number of states of the resulting automaton can be bounded by a function of the form*

$$2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}\Big\}k$$

with a constant k.

Empirical observation (MONA system of Basin, Klarlund):
If each DFA that is produced by the translation as an intermediate result is immediately minimized, then even in the case of long formulas the size of the automaton mostly remains moderate. But doesn't minimization add another computational burden? Not really, because we will see in the next chapter that for the case of a DFA, minimization can be done efficiently. Consequently, the examples for an hyper-exponential growth are not typical.

Example 1.36 We give to the MONA system the following formula:

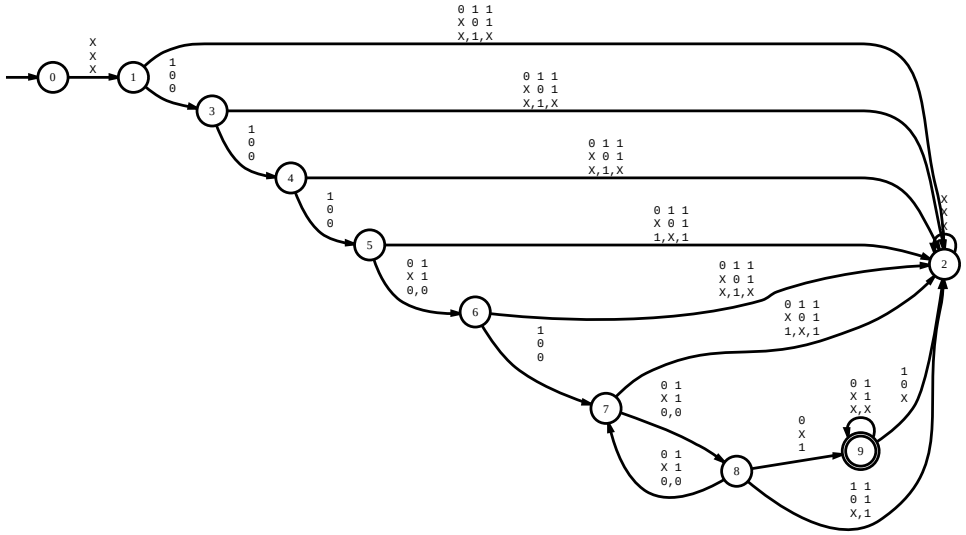
$$\varphi(X,Y,x) : \text{“}X \setminus Y = \{0,1,2,4\}'' \ \wedge \ \psi(x) \ \wedge \ \neg X(x)$$

with $\psi(x) := \exists Z(Z(0) \wedge Z(x) \wedge \forall y(0 < y \leq x \rightarrow$
$((Z(y) \rightarrow \neg Z(y-1)) \wedge (\neg Z(y) \rightarrow Z(y-1)))$

MONA input:

```
var2 X,Y;
X\Y = {0,4} union {1,2};
pred even(var1 x) = ex2 Z: 0 in Z & x in Z &
  (all1 y: (0 < y & y <= x) =>
  (y in Z => y-1 notin Z) &
  (y notin Z => y-1 in Z));
var1 x;
even(x) & x notin X;
```

The resulting automaton is (MONA output)



Now that we have connected automata to logic, we can make a comparison between them, as well as consider the applications of this connection.

- Automata: operational, formulas: descriptive
- Automata: implementation, formulas: specification
- Automata have local transitions, formulas have global operators (in particular quantifiers)
- The semantics of automata are defined over the arbitrary construction of state sequences, whereas the semantics of formulas are defined inductively (compositional)
- algorithmic questions on automata are often efficiently solvable, whereas the ones on formulas are hard to solve (non-emptiness vs. satisfiability)
- For automata the minimization problem is approachable, for formulas it remains still incomprehensible.

How does all this relate to the "logic" of regular expressions? These can be enriched by further operators that make them more convenient to use without increasing their expressive power. We add $\sim$ (for the complement) and $\cap$ (for the intersection).

Definition 1.37 Generalized regular expressions over Σ are built up by letters $a \in \Sigma$ and $\emptyset$, ϵ by using $+, \cdot, \cap$ (binary) and $^*, \sim$ (unary).

Example 1.38 The following property over words in $\Sigma = \{a, b, c\}$:

"there are no two occurrences of b, between which only a's occur, and at most three c's occur"

is defined by

$$\sim(\Sigma^* b a^* b \Sigma^*) \ \cap \ \sim(\Sigma^* c \Sigma^* c \Sigma^* c \Sigma^* c \Sigma^*)$$

$\boxtimes$

Remark 1.39 *A language is definable by a generalized regular expression if, and only if, it is regular.*

For the definition of word sets (languages) generalized regular expressions are an alternative to MSO-logic; they may be not so flexible, but in many cases they are much more convenient to use. Their translation into automata is of the same computational complexity as the translation of the formulas.

Remark 1.40 *The BET-Theorem has been extended to other types of models instead of finite words:*

1. *Automata on ω-words and their equivalence to MSO-logic (Büchi 1960)*
2. *Automata on finite trees and their equivalence to MSO-logic (Doner, Thatcher/Wright 1968)*
3. *Automata on infinite trees and their equivalence to MSO-logic (Rabin 1969)*

Application: Decidable theories. The initial motivation to reduce formulas to automata was "First-order arithmetics", i.e. the first-order theory of the structure $(\mathbb{N}, +, \cdot, 0, 1)$, which was proven to be undecidable (Gödel 1931). Based on this result Tarski formulated the following problem: What is the result if in its signature we keep only $+1$, but allow set quantifiers?

Corollary 1.41 (of the Büchi, Elgot, Trakhtenbrot Theorem) *: The second-order theory of the structure $(\mathbb{N}, +1)$ in which the second-order quantifiers range only over finite sets is decidable.*

This theory is usually referred to as the weak second-order theory of one successor (WS1S).

Theorem 1.42 *The weak second-order theory of one successor (WS1S) is decidable.*

The use of tree automata and automata on infinite trees lead to further results:

- The theory S1S refers to the structure $(\mathbb{N}, +1)$ with arbitrary set quantifiers.

- The theory WS2S concerns the structure of the infinite binary tree (with two successor functions), in which the set quantifiers range only over finite sets.
- The theory S2S refers to the infinite binary tree with arbitrary set quantifiers.

All these theories are decidable using the corresponding automata and similar techniques as in the proof of Theorem 1.18.

1.3 Consequences and Applications in Model Checking

First of all, the Equivalence Theorem has a purely logical consequence:

Theorem 1.43 *Every MSO-formula $\phi(X_1, \ldots, X_n)$ is over word models equivalent to an EMSO-formula, i.e. to a formula of the form*

$$\exists Y_1 \ldots \exists Y_m \psi(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$$

where ψ is an FO-formula.

Proof: Use the BET-Theorem in both directions: $\phi \rightsquigarrow \mathcal{A}_\phi \rightsquigarrow \exists Y_1 \ldots \exists Y_m\ \psi$.

So, given an MSO-formula $\phi(\overline{X})$, we first construct the equivalent NFA $\mathcal{A}_\phi$ over the alphabet $\Sigma \times \{0,1\}^n$ and then transform it into an equivalent EMSO-formula $\exists Y_1 \ldots \exists Y_m\ \psi$. □

Theorem 1.44 *Satisfiability and equivalence of MSO-formulas over word models are decidable problems.*

Proof: Apply the transformations into automata:

- φ is satisfiable iff $L(\varphi) \neq \emptyset$ iff $L(\mathcal{A}_\varphi) \neq \emptyset$
- $\varphi \equiv \psi$ iff $L(\varphi) = L(\psi)$ iff $L(\mathcal{A}_\varphi) = L(\mathcal{A}_\psi)$

The constraints on the right can be tested algorithmically. □

We look at the terminating behavior of finite state systems. The *model checking problem* is the following: Given a finite state system represented by an NFA $\mathcal{A}$ and a specification formulated as an MSO-formula ϕ, does every word accepted by $\mathcal{A}$ satisfy the specification ϕ? This can be succinctly restated as "$L(\mathcal{A}) \subseteq L(\phi)$?".

To solve the model checking problem we introduce error scenarios. A word w is an *error scenario* if $w \in L(\mathcal{A})$ but $\underline{w} \not\models \phi$, i.e. $w \in L(\mathcal{A}) \cap L(\mathcal{A}_{\neg\phi})$.

The solution of a model checking problem consists of two steps. First we construct from a formula ϕ an automaton $\mathcal{A}_{\neg\phi}$ using the BET-Theorem. Then we construct the product automaton $\mathcal{B}$ of $\mathcal{A}$ and $\mathcal{A}_{\neg\phi}$. $\mathcal{B}$ accepts precisely the error scenarios. Hence, we apply the emptiness test for $\mathcal{B}$. If $L(\mathcal{B})$ turns out to be non-empty, then not every word accepted by $\mathcal{A}$ satisfies the specification ϕ. In this case the emptiness test returns a counter-example to ϕ, which can be used for debugging the system.

So, given an automaton $\mathcal{A}$ and an MSO-formula ϕ, to solve the model checking problem we have to do the following:

1. Construct the automaton $\mathcal{A}_{\neg\phi}$
2. Proceed to the product automaton $\mathcal{A} \times \mathcal{A}_{\neg\phi}$ that recognizes $L(\mathcal{A}) \cap L(\mathcal{A}_{\neg\phi})$.
3. Apply the non-emptiness test on the product automaton.
4. In case of non-emptiness, return a word that is accepted by the product automaton (error scenario).

Example 1.45 MUX (Mutual exclusion) protocol modeled by a transition system over the state-space $\mathbb{B}^5$.

```
Proc0: loop
(00) a0: Non_Critical_Section_0;
(01) b0: wait unless Turn = 0;
(10) c0: Critical_Section_0;
(11) d0: Turn := 1;

Proc1: loop
(00) a1: Non_Critical_Section_1;
(01) b1: wait unless Turn = 1;
(10) c1: Critical_Section_1;
(11) d1: Turn := 0;
```

A state is a bit-vector $(b_1, b_2, b_3, b_4, b_5) \in \mathbb{B}^5$ (value of turn, line no. of process 0, line no. of process 1). On this protocol we can define the system automaton $\mathcal{A}_{\text{MutEx}}$ over the state space $\{0,1\}^5$, with initial state (00000), the alphabet $\mathbb{B}^5$ and the following transitions:

For process 0:

$$
\begin{array}{rcl}
b_1\ 0\ 0\ b_4\ b_5 & \rightarrow & b_1\ 0\ 1\ b_4\ b_5 \\
1\ 0\ 1\ b_4\ b_5 & \rightarrow & 1\ 0\ 1\ b_4\ b_5 \\
0\ 0\ 1\ b_4\ b_5 & \rightarrow & 0\ 1\ 0\ b_4\ b_5 \\
b_1\ 1\ 0\ b_4\ b_5 & \rightarrow & b_1\ 1\ 1\ b_4\ b_5 \\
b_1\ 1\ 1\ b_4\ b_5 & \rightarrow & 1\ 0\ 0\ b_4\ b_5
\end{array}
$$

For process 1: analogously

If we additionally label each transition with its target state, and declare all reachable states as final ones, the words being accepted by this automaton are exactly the system runs. Finally, for matters of system specification we can represent the 5 bits by predicates $X_1, \ldots, X_5$ and introduce abbreviated compact state properties that can be described by MSO-formulas:

$$\begin{array}{rcll} \text{ata0}(t) & := & \neg X_2(t) \wedge \neg X_3(t) & \text{(states of the form } (X00XX)\text{)} \\ \text{Turn}(t) = 0 & := & \neg X_1(t) & \text{(states of the } (0XXXX)\text{)} \end{array}$$

The first formula specifies all states where some part of the system is "at a0", which means that process 0 is in line 00. The second one specifies all those states where the value of the synchronization variable happens to be 0. For the general case where state properties $p_1, \ldots, p_k$ are considered, a framework is used that is called *Kripke structure*.

Definition 1.46 A *Kripke structure* over propositions $p_1, \ldots, p_k$ has the form $\mathcal{M} = (S, \rightarrow, s_0, \beta)$ with

- a finite set S of states
- a transition relation $\rightarrow \subseteq S \times S$ (alternatively $\subseteq S \times \Sigma \times S$)
- an initial state s_0
- a labeling function $\beta : S \rightarrow 2^{\{p_1,\ldots,p_n\}}$ assigning to each $s \in S$ the set of those p_i that are true in s

Usually, we write a value $\beta(s)$ as a bit vector $(b_1, \ldots, b_n)$ with $b_i = 1$ iff $p_i \in \beta(s)$.

Some very essential specifications of the MUX System would be the safety condition and the liveliness condition. For the first we have to specify that no state in which process 0 is in c0 and process 1 is in c1 will ever be reached:

$$\varphi := \forall x \neg\big(\text{atc0}(x) \wedge \text{atc1}(x)\big)$$

According to our model, we have to forbid all states of the form $(X1010)$:

$$\varphi(X_1, \ldots, X_5) = \forall x \neg\big(X_2(x) \wedge \neg X_3(x) \wedge X_4(x) \wedge \neg X_5(x)\big)$$

In the framework of model checking, this is an inclusion test, namely whether every (accepting) run of the finite system is also a (correct) model for the MSO-specification: $L(\mathcal{A}_{\text{MutEx}}) \subseteq L(\varphi(X_1, \ldots, X_5))$.

For an example specification of the second kind, we require that "whenever process 0 wants to enter the critical section (at b0), it will eventually indeed enter c0":

$$\forall x\big(\text{atb0}(x) \rightarrow \exists y(x < y \wedge \text{atc0}(y))\big)$$

The MSO formulation of this condition actually makes sense when it comes to infinite system runs, which goes beyond the scope of these course notes. ⊠

1.4 First-Order Definability

For the most cases of specification the expressive power of FO[$S, <$]-formulas suffices. A natural question to ask is what are the exact limits of this power, i.e. which regular languages can be expressed by FO-formulas. To this purpose we are going to introduce two new formalisms, namely *star-free (regular) expressions* and the *temporal logic LTL* ("linear time logic"), and compare them with FO-logic. Without giving a complete proof, we shall see that all three formalisms define exactly the same class of languages. Finally, we will see that this class is a *proper* subset of the class of regular languages, i.e. there is a special class of languages that is regular (which means MSO-definable) but not FO-definable.

1.4.1 Star-free Expressions

Definition 1.47 A *star-free* regular expression is a (generalized) regular expression built up from the atoms $\epsilon, \emptyset, a \in \Sigma$ using only the operations $\cdot$ for concatenation, $+$ for union, $\cap$ for intersection, and $\sim$ for complementation. The language $L(r)$ defined by such an expression r is also called star-free.

Example 1.48 Let $\Sigma = \{a, b, c\}$

1. Σ^* is star-free, defined by $\sim\emptyset$
2. a^+ is star-free, defined by $\sim\epsilon \cap \sim(\Sigma^*(b+c)\Sigma^*)$
3. $b(ab)^*$ is star-free, defined by the constraint:
   "never c, begin with b, never aa nor bb, end with b", i.e.:

$$\sim(\Sigma^* c \Sigma^*) \cap b\Sigma^* \cap \sim(\Sigma^*(aa+bb)\Sigma^*) \cap \Sigma^* b$$

⊠

Remark 1.49

a. *The use of complementation $\sim$ is essential.*
   Indeed, complementation is the operation that enables us to represent infinite languages because by complementing a finite language we get an infinite one. With union, concatenation, and intersection we can only produce finite languages starting from the finitely many symbols of the alphabet.

b. *The standard star-free expressions are produced out of $a \in \Sigma$, $\epsilon, \emptyset$ by using only $+$ and $\cdot$; these expressions define exactly the finite languages.*

Theorem 1.50 *Every star-free language $L \subseteq \Sigma^+$ is FO-definable.*

This theorem follows from the more technical lemma below.

Lemma 1.51 *For every star-free expression r there is an FO-formula $\varphi_r(x,y)$ that expresses that "the segment from x up to y is in $L(r)$", that is $w \in L(r)$ iff $\underline{w} \models \varphi_r[\min, \max]$ for all $w \in \Sigma^+$.*

Proof: By induction on the construction of the star-free expression.
Base cases:

- $r = a$: Set $\varphi_r(x,y) := P_a(x) \wedge x = y$
- $r = \emptyset$: Set $\varphi_r(x,y) := \exists z(x \leq z \wedge z \leq y \wedge \neg(z = z))$

Induction hypothesis: Let s, t be equivalent to $\varphi_s(x,y)$, $\varphi_t(x,y)$.
Induction step: For $r = s + t$, $r = s \cap t$ and $r = (\sim s) \cap \sim\epsilon$ set:
$\varphi_r(x,y) := \varphi_s(x,y) \vee \varphi_t(x,y)$, $\varphi_s(x,y) \wedge \varphi_t(x,y)$, and $\neg\varphi_s(x,y)$, respectively.
For $r = s \cdot t$ we require the existence of two successive positions z, z' that are between x and y. From x up to z, φ_s holds, and from z' up to y, φ_t holds:

$$\begin{aligned}\varphi_r(x,y) := \exists z \exists z'(&x \leq z \wedge z \leq y \wedge x \leq z' \wedge z' \leq y \wedge \\ &\varphi_s(x,z) \wedge S(z,z') \wedge \varphi_t(z',y))\end{aligned}$$

□

1.4.2 Temporal Logic LTL

The (propositional) temporal logic LTL (linear time logic) was first proposed in 1977 by A. Pnueli for purposes of system specification. Its main advantages are:

- variable-free compact syntax
- transformation into automata more efficiently than in the case of FO-formulas
- equivalence to FO$[S, <]$-logic

In this section we handle LTL only over finite words.

Definition 1.52 (LTL) Syntax: A *temporal formula* is built up from *propositional variables* $p_1, p_2, \ldots$ using the connectives $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$ and the *temporal operators* X (next), F (eventually), G (always), and U (until). We write $\phi(p_1, \ldots, p_n)$ for a temporal formula ϕ to indicate that at most the propositional variables $p_1, \ldots, p_n$ appear in ϕ.

Semantics: A temporal formula $\phi(p_1, \ldots, p_n)$ is evaluated in a model (w, j) where $1 \leq j \leq |w|$ is a position in the word w, and w is of the form

$$w = \begin{pmatrix} b_1(1) & & b_1(|w|) \\ \vdots & \cdots & \vdots \\ b_n(1) & & b_n(|w|) \end{pmatrix}$$

The value $b_i(k) \in \{0, 1\}$ codes the truth value of proposition p_i at position k in w.

We define the semantics of temporal formulas as follows:

$$\begin{array}{lllll} (w,j) & \models & p_i & \text{iff} & b_i(j) = 1 \\ & \models & F\phi & \text{iff} & \exists k\ (j \leq k \leq |w| \text{ and } (w,k) \models \phi) \\ & \models & G\phi & \text{iff} & \forall k\ (j \leq k \leq |w| \ \Rightarrow\ (w,k) \models \phi) \\ & \models & X\phi & \text{iff} & j < |w| \text{ and } (w, j+1) \models \phi \\ & \models & \phi U \psi & \text{iff} & \begin{cases} \exists k_1 (j \leq k_1 \leq |w| \text{ and } (w,k_1) \models \psi \\ \text{and } \forall k\ (j \leq k < k_1 \ \Rightarrow (w,k) \models \phi)) \end{cases} \end{array}$$

The Boolean connectives have the standard semantics.

Intuitively, $(w, j) \models \phi$ means that ϕ is satisfied at position j in the word w. We write $w \models \phi$ for $(w, 1) \models \phi$ and define the language $L(\phi)$ accepted by a temporal formula containing the propositional variables $p_1, \ldots, p_n$ to be $L(\phi) = \{w \in (\{0,1\}^n)^+ \mid w \models \phi\}$.

Example 1.53 The LTL-formula

$$\varphi := \mathrm{G}(p_1 \rightarrow \mathrm{X}(p_1 \mathrm{U} p_2))$$

expresses the following property:

"Whenever p_1 holds, then from the next moment on the following holds:
p_1 holds until eventually p_2 holds."

Consider the model:

$$w = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The formula holds on position 2, but not on position 1. Consequently, we write $(w, 2) \models \varphi$ but $w \not\models \varphi$. ⊠

Example 1.54

$$\mathrm{G}(p_1 \rightarrow \mathrm{X}(\neg p_2 \mathrm{U} p_3))$$

expresses in (w, j) with $w \in (\{0,1\}^3)^+$ the following property:

"From position j onwards, whenever the first component is 1, then from the next position onwards the second component is 0, until at some position the third component is 1".

⊠

Example 1.55 Let $n = 2$ and $w = \begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}$

- $w \models \mathrm{G}(p_1 \to \mathrm{F}p_2), \quad w \not\models \mathrm{G}(p_2 \to \mathrm{F}p_1)$
- $w \models \mathrm{X}(\neg p_2 \mathrm{U} p_1), \quad w \models \neg p_2 \mathrm{U} p_1$
- $w \not\models p_1 \mathrm{U}(p_1 \wedge p_2)$

⊠

Theorem 1.56 *Every LTL-definable language $L \subseteq \Sigma^+$ is FO-definable.*

Lemma 1.57 *For every LTL-formula φ one can construct an equivalent* FO$[S, <]$*-formula $\varphi^*(x)$ with*

$$(w, j) \models \varphi \quad \textit{iff} \quad (\underline{w}, j) \models \varphi^*(x)$$

Proof: Inductive definition of $\varphi^*(x)$:

- $(p_i)^*(x) \quad := X_i(x)$
- $(\neg\varphi)^*(x) \quad :=$ $\neg(\varphi^*(x))$, analogously for $\vee, \wedge, \to, \leftrightarrow$
- $(\mathrm{G}\varphi)^*(x) \quad := \forall y(x \leq y \to \varphi^*(y))$
- $(\mathrm{F}\varphi)^*(x) \quad := \exists y(x \leq y \wedge \varphi^*(y))$
- $(\mathrm{X}\varphi)^*(x) \quad := \exists y(S(x, y) \wedge \varphi^*(y))$
- $(\varphi \mathrm{U} \psi)^*(x) \quad := \exists y\big(x \leq y \wedge \psi^*(y) \wedge \forall z(x \leq z \wedge z < y \to \varphi^*(z))\big)$

□

Example 1.58

$$\begin{array}{rcl} \varphi & = & \mathrm{G}(p_1 \to \mathrm{X}(\neg p_2 \mathrm{U} p_3)) \\ \varphi^*(x) & = & \forall y(x \leq y \wedge X_1(y) \to \exists z(S(y, z) \wedge \exists s(z \leq s \wedge X_3(s) \wedge \\ & & \forall t(z \leq t \wedge t < s \ \to \ \neg X_2(t))))) \end{array}$$

⊠

So far we have shown that:

- If L is star-free, then L is FO-definable.
- If L is LTL-definable, then L is FO-definable.

The converses also hold but won't be proven in these course notes since this requires much bigger technical effort:

Theorem 1.59 (McNaughton 1969, Kamp 1968) *The conditions "FO-definable", "star-free", and "LTL-definable" for a language L are equivalent.*

1.5 Between FO- and MSO-definability

The question that arises now is whether there is an MSO-definable language that is not FO-definable. Using the above equivalences, this questions means that we are looking for regular languages that are not star-free. To this purpose we introduce the "counting"-property, by which regular languages can be distinguished from star-free ones. In particular, we show that:

- There is at least one regular language that is "counting".
- Every star-free language is "non-counting".

Definition 1.60 Call $L \subseteq \Sigma^+$ *non-counting* if

$$\exists n_0 \; \forall n \geq n_0 \; \forall u, v, w \in \Sigma^* : uv^n w \in L \;\Leftrightarrow\; uv^{n+1} w \in L.$$

This means for $n \geq n_0$ either all $uv^n w$ are in L, or none is.

Remark 1.61 *The above condition is a strong pumping-property; namely if for a sufficiently large n the segment v^n occurs in a word of L, then we remain in L if we use higher powers of v. Furthermore, note that the position of v^n is arbitrary. The standard Pumping Lemma guarantees only the* existence *of a position to pump.*

Definition 1.62 L is *not* non-counting (short: L is *counting*) iff

$$\forall n_0 \; \exists n \geq n_0 \; \exists u, v, w \in \Sigma^* : (uv^n w \in L \Leftrightarrow uv^{n+1} w \notin L).$$

Example 1.63

- $L_1 = (aa)^+$ is counting: Given n_0 take some even number $n \geq n_0$ and set $u, w := \epsilon$, $v := a$. Then $a^n \in L_1$, but $a^{n+1} \notin L_1$.
- $L_2 = b(a^*bb)^*$ is counting: Given n_0 take some even number $n \geq n_0$ and set $u := b$, $v := b$, $w := \epsilon$. Then $bb^n \in L_2$, but $bb^{n+1} \notin L_2$.
- $L_3 = b(a^+bb)^*$ is non-counting because it is FO-definable by:

$$P_b(\min) \wedge P_b(\max) \wedge \forall u(S(\min, u) \rightarrow P_a(u))$$
$$\wedge \forall x \forall y(\ S(x,y) \wedge P_a(x) \wedge P_b(y) \rightarrow$$
$$\exists z(S(y,z) \wedge P_b(z) \wedge \forall u(S(z,u) \rightarrow P_a(u)))) \ .$$

Theorems 1.65 and 1.59 imply that FO-definable languages are non-counting.

⊠

Remark 1.64 *By "counting" we mean modulo-counting successive occurrences of a particular pattern (the word v in the definition). $L = a(ba)^*$ is also sort of a counting language since it contains only words of odd length, but it is not modulo-counting in the above sense.*

Theorem 1.65 *Every star-free language is non-counting.*

Proof: By induction on the structure of star-free expressions.

Base cases: For $a \in \Sigma$, $\emptyset$, ϵ take $n_0 = 2$ respectively. Then, for each of these languages L, for each $n \geq n_0$, and for all $u, v, w \in \Sigma^*$ the following holds:

$$uv^n w \in L \ \Leftrightarrow \ uv^{n+1} w \in L$$

Induction step "$\sim$": If for a suitable n_0 and for every $n \geq n_0$

$$uv^n w \in L \ \Leftrightarrow \ uv^{n+1} w \in L$$

holds, then obviously this also holds for the complement of L.

Induction step "$\cap$": By induction hypothesis, for L_1, L_2 there are n_1, n_2, such that for all $n \geq n_i$

$$uv^n w \in L_i \ \Leftrightarrow \ uv^{n+1} w \in L_i \quad (i = 1, 2)$$

Set $n_0 = \max(n_1, n_2)$. Then for $n \geq n_0$ the following holds:

$$uv^n w \in L_1 \cap L_2 \ \Leftrightarrow \ uv^{n+1} w \in L_1 \cap L_2$$

Induction step "$\cdot$": Take the induction hypothesis as before and set $n_0 = 2 \cdot \max(n_1, n_2) + 1$. For $n \geq n_0$ consider $uv^n w \in L_1 \cdot L_2$. Consider the decomposition of $uv^n w$ in two segments, $uv^n w = z_1 z_2$ with $z_1 \in L_1, z_2 \in L_2$ as illustrated in Figure 1.2. By the choice of n_0 one of the following cases applies.

Case 1: z_1 contains at least n_1 v-segments. Then by induction hypothesis z_1' (with one v-segment more than z_1) is in L_1.
Case 2: z_2 contains at least n_2 v-segments. Then by induction hypothesis z_2' (with one v-segment more than z_2) is in L_2.
In both cases we get $uv^{n+1} w \in L_1 L_2$. The converse (from $uv^{n+1} w \in L_1 L_2$ to $uv^n w \in L_1 L_2$) can be proven analogously. □

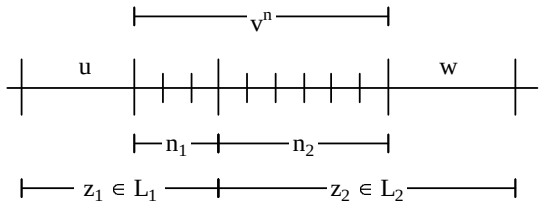


Figure 1.2: Illustration

Now that we distinguished these classes of languages on the level of logic, the natural thing to ask is whether there is a way to do the same on the level of automata. That is, is there a property of automata that characterizes the star-free expressions? This question is answered positively in Section 2.4.

In the following we introduce a finer complexity scale for (standard) regular expressions that refers to the number of occurrences of the star operator, and relate this measure to a property of automata.

Definition 1.66 The star-height of a regular expression is defined inductively as follows:

- $\mathrm{sh}(a) = \mathrm{sh}(\emptyset) = \mathrm{sh}(\epsilon) = 0$
- $\mathrm{sh}(r + s) = \mathrm{sh}(r \cdot s) = \max(\mathrm{sh}(r), \mathrm{sh}(s))$
- $\mathrm{sh}(r^*) = \mathrm{sh}(r) + 1$

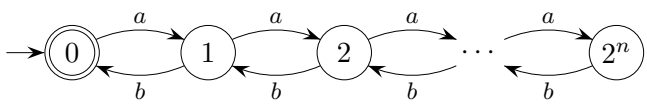
For a regular Language L, the star-height of L is

$\mathrm{sh}(L) = \min\{n \mid \text{there is a regular expression } r \text{ with } L = L(r) \text{ and } \mathrm{sh}(r) = n\}$

Remark 1.67 *L is finite iff* $\mathrm{sh}(L) = 0$.

Theorem 1.68 (Eggan 1963) *The star-height hierarchy is strict (over alphabets with at least two letters): For each $n \geq 0$ there exists a language L_n with* $\mathrm{sh}(L_n) = n$.

For L_n we can take the family of languages, recognized by the following NFA (see exercises):

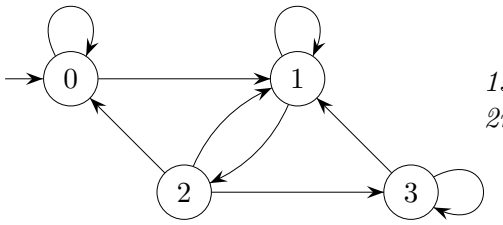


The property of NFAs that corresponds to the star-height of expressions is the "loop complexity" (also "feedback number").

Definition 1.69 The loop complexity $\mathrm{lc}(\mathcal{A})$ of an NFA $\mathcal{A}$ is the minimum number of steps of the following kind required to eliminate all loops of $\mathcal{A}$:

"remove one state in each strongly connected component of the current transition graph"

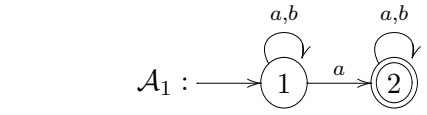
Example 1.70 Verify that the following automaton has loop complexity 2:



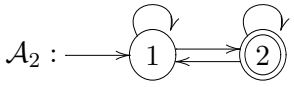
1st step: Remove 1
2nd step: Remove 0 and 3

⊠

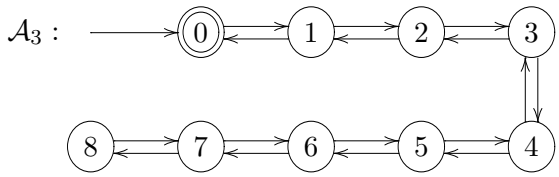
Example 1.71 $\mathrm{lc}(\mathcal{A}_1) = 1$, $\mathrm{lc}(\mathcal{A}_2) = 2$, and $\mathrm{lc}(\mathcal{A}_3) = 3$.



1st step: Remove 1 and 2.



1st step: Remove 1 (or 2)
2nd step: Remove 2 (or 1)



1st step: Remove 4
2nd step: Remove 2 and 6
3rd step: Remove 0 and 7

⊠

Analogously, we can define the loop complexity of a regular language.

Definition 1.72 The loop complexity of a regular language is

$$\mathrm{lc}(L) = \min\{n \mid \text{there exists an NFA } \mathcal{A} \text{ with } L = L(\mathcal{A}) \text{ and } \mathrm{lc}(\mathcal{A}) = n\}$$

Theorem 1.73 (Eggan 1963) *For every regular language L, $\mathrm{sh}(L) = \mathrm{lc}(L)$.*

The proof is a refinement of the one for Kleene's theorem (Theorem 1.4). For further details, please refer to J.R. Büchi: Finite Automata, Their Algebras and Grammars, Sect. 4.6.

Theorem 1.74 (Hashiguchi 1988) *Given a regular language L (say defined by an NFA) the star-height can be calculated algorithmically.*

For the case of generalized regular expressions the *generalized star-height* gsh is defined analogously to the one of the standard star-height.

Remark 1.75

- $\mathrm{gsh}(L) \leq \mathrm{sh}(L)$ *(i.e. we may be able to reduce the star-height by substituting some stars by the extended operators of the generalized regular expressions) but in general we cannot reduce it to 0 because*
- $\mathrm{gsh}(L) = 0$ *iff L is star-free (by definition).*
- *There exist regular languages L with* $\mathrm{gsh}(L) \geq 1$ *(compare Example 1.63 on page 35).*

Open problem (since 1971): Are there regular languages L with $\mathrm{gsh}(L) \geq 2$?

We suspect that in case there is a solution, it goes to the direction of nesting modulo-counting since this is the identifying property of non-star-free expressions. We define the languages:

$E = (00)^*1$ ("even block")
$O = (00)^*01$ ("odd block")

The suggestion of using the language specifying an even number of even blocks $(((O^*E)(O^*E))^*O^*)$ failed after a few years.

Theorem 1.76 (Thomas., Theor. Comput. Sci. 13 (1981))

$$\mathrm{gsh}(((O^*E)(O^*E))^*E^*) = 1$$

Another proposed language L as candidate for a language with $\mathrm{gsh}(L) = 2$ was $(O^+E^+O^+E^+)^*$. Also for this language it was later proven that $\mathrm{gsh}(L) = 1$. For details, please refer to M. Robson, More languages of generalized star height 1, Theor. Comput. Sci. 106 (1992), 327-335.

1.6 Exercises

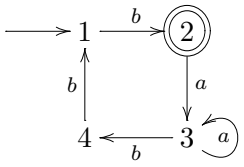
Exercise 1.1 Give FO- or MSO-formulas for the languages that are defined by the following regular expressions or descriptions in colloquial language:

(a) a^+b^*,

(b) aab^*aa,

(c) there are at least three occurrences of b, and before the first b there are at most two occurrences of a.

Exercise 1.2 Consider the following NFA $\mathcal{A}$:



In the lecture it was shown how to construct an equivalent formula $\varphi_{\mathcal{A}}$ with four set variables $X_1, \cdots, X_4$. According to the hint of the lecture, give a formula with only two set variables. (The states $1, 2, 3, 4$ should correspond to the vectors $\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}$). Is $L(\mathcal{A})$ FO-definable?

Exercise 1.3 (a) For the MSO-formula

$$\exists x \forall y (x < y \rightarrow P_a(y))$$

give an equivalent MSO_0-formula.

(b) Let $\Sigma = \{a, b\}$. The following MSO_0-formula defines a language over $\Sigma \times \{0,1\}^3$. Give (by direct construction) an equivalent NFA.

$$(X_1 \subseteq X_2) \wedge (X_1 < X_3) \wedge (X_3 \subseteq P_a) \ .$$

Exercise 1.4 * Show that every regular language $L \subseteq \Sigma^+$ is definable by an EMSO-sentence with only one single set quantifier.

Exercise 1.5 Let $\Sigma = \{a, b\}$ and

$$\begin{aligned} \varphi(X_1, y) &:= \big(\neg P_a(y) \rightarrow X_1(y)\big), \\ \psi(X_1) &:= \quad \forall y \ \varphi(X_1, y) \qquad \text{and} \\ \chi &:= \quad \exists X_1 \ \psi(X_1). \end{aligned}$$

(a) Write φ as an MSO_0-formula $\varphi(X_1, Y)$.

(b) Construct an NFA over $\Sigma \times \{0,1\}^2$ recognizing $L(\varphi(X_1, Y))$.

(c) Using (b) construct an NFA over $\Sigma \times \{0,1\}$ recognizing $L(\psi(X_1))$.

(d) Using (c) construct an NFA over Σ recognizing $L(\chi)$.

Exercise 1.6 Consider the following two decidability problems:

(a) Given an MSO-sentence φ, does φ hold for all nonempty word models?

(b) Given an MSO-sentence φ, does φ hold for all nonempty word models of even length?

Describe algorithms for solving these problems.

Exercise 1.7 Consider the word model

$$w := \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

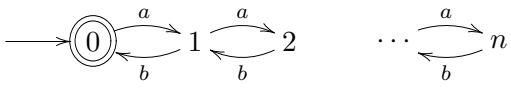
and the formulas

1. $\varphi = p_2 U (p_1 \wedge \neg p_2)$ and
2. $\psi = G(p_1 \to X p_2)$

(a) For each $j = 1, \cdots, 6$ answer whether the formulas hold in (w, j) or not.

(b) Give equivalent FO-formulas and star-free expressions.

Exercise 1.8 * For $n \geq 1$ let $\mathcal{A}_n$ be the following NFA:



(a) Show that $L(\mathcal{A}_n)$ is star-free.

(b) Show that for $L(\mathcal{A}_{2^n})$ there exists a conventional regular expression (i.e. only using $+, \cdot, ^*$) of star-height n.
(The star-height of an expression is the maximal number of nested stars. Example: $(a(aa)^* + (bb)^*)^*$ has star-height 2)

(c) Give an LTL-formula for $L(\mathcal{A}_n)$.

Hint: For (a),(b) it is useful, to consider next to $\mathcal{A}_n$ also the automata $\mathcal{A}_n^+$, $\mathcal{A}_n^-$; $\mathcal{A}_n^+$ has initial state 0, final state n, $\mathcal{A}_n^-$ has initial state n, final state 0.

Exercise 1.9 Give star-free expressions for the languages defined by the following LTL-formulas. The alphabet is $\Sigma = \left\{ \begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix} \right\}$. Write $\begin{bmatrix}1\\*\end{bmatrix}$ for $\begin{bmatrix}1\\0\end{bmatrix} + \begin{bmatrix}1\\1\end{bmatrix}$, etc.

(a) $XX(p_1 \wedge G\neg p_2)$

(b) $FG(p_1 \wedge \neg X p_2)$

(c) $F(p_1 \wedge X(\neg p_2 U p_1))$

Exercise 1.10 Decide whether the following languages are "counting" and verify your assumption.

(a) $b(abb)^*$

(b) $a(bbb)^* aa$

Chapter 2

Congruences and Minimization

In this chapter we are interested in reducing the size of automata while keeping their functionality. For finite automata (on words or on trees) the size usually refers to the number of states. The states of an automaton or a system represent the distinctions that are made in order to fulfill a certain task. The goal is to make this number of distinctions as small as possible. For this purpose we now take the graph theoretic view (as opposed to the logical view from the previous chapter) of automata. As we will see, there are elegant techniques that allow us to compare and improve automata on the system level, rather than on the specification level.

2.1 Homomorphisms, Quotients and Abstraction

There are cases in the analysis and design of a system, where we are interested only in a rough view of its structure and want to spare ourselves the local details. In this case, we actually represent by a model only the topology and interconnection between the subsystems and abstract away from their own functionality.

Example 2.1 The NFA $\mathcal{A}^2$ on the right-hand side of Figure 2.1 is a rough view on the NFA $\mathcal{A}^2$ on the left-hand side, where the subsystem of the states 1, 2, and 3 (counting a's modulo 3) is represented by state B. This merging of states corresponds to the following *homomorphism*:

$$0 \mapsto A, \quad 1 \mapsto B, \quad 2 \mapsto B, \quad 3 \mapsto B$$

$\boxtimes$

Definition 2.2 Let $\mathcal{A} = (A, R^A, f^A, c^A), \mathcal{B} = (B, R^B, f^B, c^B)$ be two structures, with

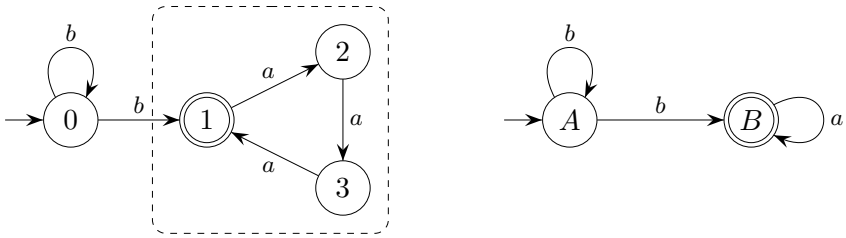


Figure 2.1: Merging states in an NFA

- universes A, B,
- binary relations $R^A \subseteq A \times A$, $R^B \subseteq B \times B$,
- binary functions $f^A : A \times A \to A$, $f^B : B \times B \to B$ and
- distinguished elements ("constants") $c^A \in A, c^B \in B$.

A *homomorphism* from $\mathcal{A}$ to $\mathcal{B}$ is a function $h : A \to B$ with

- for $x_1, x_2 \in A$: If $(x_1, x_2) \in R^A$, then $(h(x_1), h(x_2)) \in R^B$
- for $x_1, x_2 \in A$: $h(f^A(x_1, x_2)) = f^B(h(x_1), h(x_2))$
- $h(c^A) = c^B$

The definition is analogous for relations and functions of other arity.

Example 2.3 The "Parikh-mapping":
$\mathcal{A} = (\{a, b\}^*, \cdot, \epsilon)$, $\mathcal{B} = (\mathbb{N} \times \mathbb{N}, +, (0, 0))$
$h(w) = (|w|_a, |w|_b)$ (Numbers of a's and b's in w)
E.g.: $h(aabab) = (3, 2)$
The function h is a homomorphism because:

- for $w_1, w_2 \in \{a, b\}^*$: $h(w_1 \cdot w_2) = (|w_1 \cdot w_2|_a, |w_1 \cdot w_2|_b) = (|w_1|_a + |w_2|_a, |w_1|_b + |w_2|_b) = (|w_1|_a, |w_1|_b) + (|w_2|_a, |w_2|_b) = h(w_1) + h(w_2)$
- $h(\epsilon) = (|\epsilon|_a, |\epsilon|_b) = (0, 0)$

⊠

Example 2.4 Consider again the NFAs $\mathcal{A}^1$ and $\mathcal{A}^2$ from Figure 2.1:

$(\{0, 1, 2, 3\}, 0, R^1_a, R^1_b, \{1\})$ with
$R^1_a = \{(1, 2), (2, 3), (3, 1)\}$ and
$R^1_b = \{(0, 0), (0, 1)\}$

$(\{A, B\}, A, R^2_a, R^2_b, \{B\})$ with
$R^2_a = \{(B, B)\}$ and
$R^2_b = \{(A, A), (A, B)\}$

The domain of this homomorphism h is the state set of $\mathcal{A}^1$ and has the following discrete values: $h(1) = h(2) = h(3) = B, h(0) = A$. It is indeed a homomorphism, because every transition from R_a^1 is mapped to a transition in R_a^2. Analogously for R_b^1 and R_b^2. ⊠

Definition 2.5 Let $\mathcal{A}^1 = (Q^1, \Sigma, q_0^1, \Delta^1, F^1)$ and $\mathcal{A}^2 = (Q^2, \Sigma, q_0^2, \Delta^2, F^2)$ be NFAs. An *NFA-homomorphism* from $\mathcal{A}^1$ to $\mathcal{A}^2$ is a function $h : Q^1 \to Q^2$ with

- $h(q_0^1) = q_0^2$
- $(p, a, q) \in \Delta^1 \;\Rightarrow\; (h(p), a, h(q)) \in \Delta^2$
- $q \in F^1 \Rightarrow h(q) \in F^2$

Remark 2.6 *This is covered also by the general definition of homomorphism if Δ^1 is decomposed into binary relations R_a^1 for each $a \in \Sigma$ with:*

$$(p, q) \in R_a^1 \text{ iff } (p, a, q) \in \Delta^1$$

and analogously for Δ^2.

Remark 2.7 *Let h be an NFA-homomorphism from $\mathcal{A}^1$ to $\mathcal{A}^2$. Then the following holds: If $\mathcal{A}^1$ accepts the word w, then so does $\mathcal{A}^2$. Hence, $L(\mathcal{A}^1) \subseteq L(\mathcal{A}^2)$. In other words* abstraction by homomorphism produces an NFA that recognizes a super-language of the one defined by the initial NFA.

Example 2.8 Consider $\mathcal{A}^1$ and $\mathcal{A}^2$ from Example 2.4:

$$L(\mathcal{A}^1) = b^+(aaa)^*$$
$$L(\mathcal{A}^2) = b^+a^*$$

⊠

Remark 2.9 *An NFA-homomorphism h of $\mathcal{A}$ yields a matching (or equivalence) of the states of $\mathcal{A}$: p, q are h-equivalent iff $h(p) = h(q)$. In short terms: A homomorphism yields a state equivalence.*

The converse of Remark 2.9 also holds, namely: *A state equivalence yields a homomorphism.* To this purpose we consider a partitioning of the states of the NFA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ and we write $p \sim q$ if p, q are in the same class $q/_\sim = \{p \in Q \mid p \sim q\}$ (i.e. they are "equivalent").

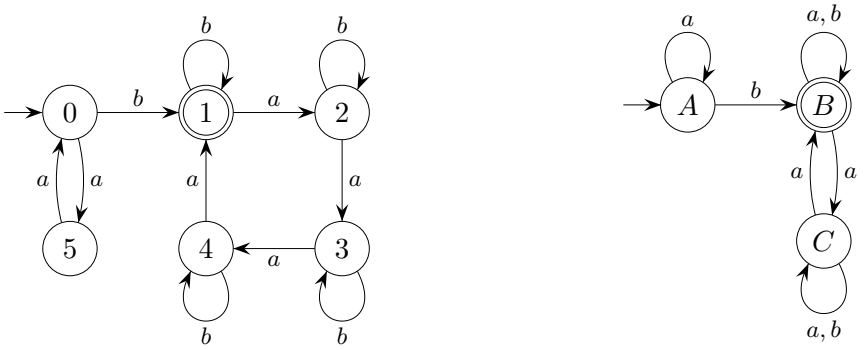
Definition 2.10 The *quotient automaton* $\mathcal{A}/_\sim$ of $\mathcal{A}$ is defined by $\mathcal{A}/_\sim = (Q/_\sim, \Sigma, q_0/_\sim, \Delta/_\sim, F/_\sim)$ with

- $Q/_\sim$ = set of $\sim$-classes $q/_\sim$ for $q \in Q$

- $(P, a, R) \in \Delta/_\sim$ iff there is $p \in P, r \in R$ with $(p, a, r) \in \Delta$, for $\sim$-classes P, R
- $P \in F/_\sim$ iff there exists $p \in P$ with $p \in F$

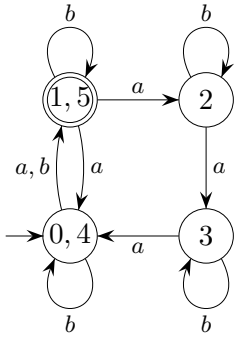
Then: $h : q \mapsto q/_\sim$ is a homomorphism from $\mathcal{A}$ to $\mathcal{A}/_\sim$.

Example 2.11 In the following automaton we merge the pairs of states $(0, 5)$, $(1, 2)$ and $(3, 4)$ and obtain the automaton on the right-hand side.



Note that we draw a b-transition from A to B, although there is such a transition only from 0 to 1 (but not from 5 to 1) in the initial automaton. This is because the definition requires only the *existence* of a member in each class that are linked with each other over such a transition. ⊠

Example 2.12 On the same automaton as before we can merge the pairs $(1, 5)$ and $(0, 4)$:

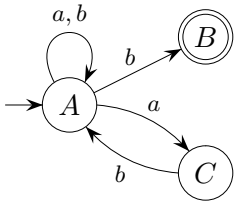


⊠

In both previous examples we observed that if $\mathcal{B}$ results from $\mathcal{A}$ by merging states, then $L(\mathcal{A}) \subseteq L(\mathcal{B})$. If we could find a way to merge the correct states, so that the resulting automaton recognizes exactly the same language as the initial one ($L(\mathcal{A}) = L(\mathcal{B})$), then we would have improved the initial automaton by reducing its number of states, without affecting its

functionality. Furthermore, the best possible merging of states would lead us to an equivalent automaton with the minimum number of states. This is what we call the minimization problem of NFAs; namely given an NFA $\mathcal{A}$, to find an equivalent NFA $\mathcal{B}$ that has the minimum number of states. But can we really achieve this by merging states? The following two examples give an answer to this question.

Example 2.13 Let $\mathcal{A}$ be an NFA and call $\mathcal{A}_q$ the same NFA as $\mathcal{A}$ but with a different initial state, namely q. To minimize $\mathcal{A}$ we suggest to merge two states p, q, if $L(\mathcal{A}_p) = L(\mathcal{A}_q)$. That is, if $\mathcal{A}$ accepts the same language both starting from p and from q, then we shall merge p and q.



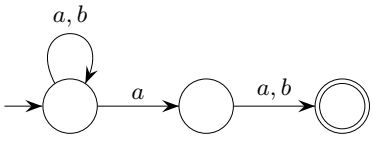
In the automaton above there is no pair that can be merged in way described before:

- A vs. C: From A, b is accepted, from C not,
- A vs. B: From B, ϵ is accepted, from A not,
- B vs. C: From B, ϵ is accepted, from C not.

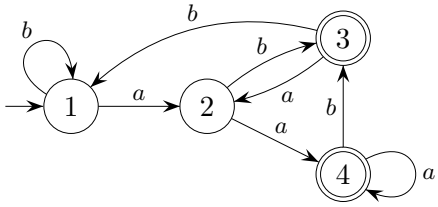
But still, it is obvious that the automaton can be minimized since states A and B suffice to accept the same language, namely $\Sigma^* b$. ⊠

Unfortunately, things are even worse since there are cases where even if we try all possible mergings (without following a specific rule as in the previous example), we will not find the minimal NFA.

Example 2.14 Consider the language $L = \Sigma^* a \Sigma$, over $\Sigma = \{a, b\}$. Obviously the minimal automaton $\mathcal{A}$ that recognizes L is the following:



Another (not minimal) automaton $\mathcal{B}$ recognizing L is the following:



We observe that there is no proper quotient automaton of $\mathcal{B}$ that recognizes L. This means that in the general case of the minimization of an NFA it does not suffice to merely merge the existing states. ⊠

2.2 Minimization and Equivalence of DFAs

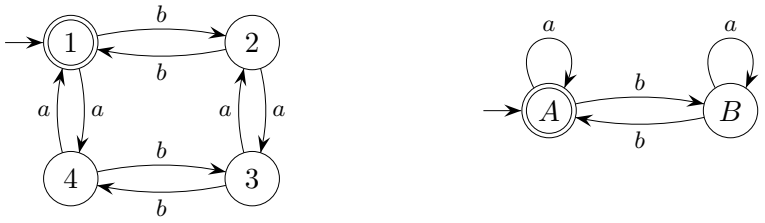
We first look at the minimization problem for DFAs. In this case we can successfully apply the techniques that we studied in the previous section and furthermore we have an efficient algorithmic solution. Before starting, we have to adopt a more abstract view on DFAs, namely consider them as *algebras*.

Definition 2.15 Let $\Sigma = \{a_1, \ldots, a_n\}$. A DFA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ can be represented as a structure $\underline{\mathcal{A}} = (Q, q_0, \delta_{a_1}, \ldots \delta_{a_n}, F)$ with $\delta_{a_i} : Q \to Q$, defined by $\delta_{a_i}(p) := \delta(p, a_i)$. This is an algebra with a distinguished subset F.

Every word $w = b_1 \ldots b_m$ induces a function $\delta_w : Q \to Q$, which is defined as a composition of $\delta_{b_1}, \ldots, \delta_{b_m}$: $\delta_w(p) = \delta_{b_m}(\delta_{b_{m-1}}(\ldots \delta_{b_1}(p) \ldots))$. $\mathcal{A}$ accepts w iff $\delta_w(q_0) \in F$.

We are going to apply the homomorphisms on the structures $\underline{\mathcal{A}}$.

Example 2.16



Looking at the automaton on the right-hand side we observe that with a we remain at the same state, whereas with b we switch from A to B and vice versa. So, δ_a^2 is the identity function and δ_b^2 is the permutation $A \mapsto B, B \mapsto A$.

To the automaton on the left-hand side we apply a homomorphism h defined by $h(1) = h(4) = A$, $h(2) = h(3) = B$. To ensure that it is indeed a

homomorphism we have to check for each $p \in \{1, 2, 3, 4\}$ the homomorphism condition:

$$h(\delta_a^1(p)) = \delta_a^2(h(p)), \quad h(\delta_b^1(p)) = \delta_b^2(h(p))$$

or in standard syntax:

$$h(\delta^1(p, x)) = \delta^2(h(p), x) \text{ for } x \in \Sigma \ .$$

⊠

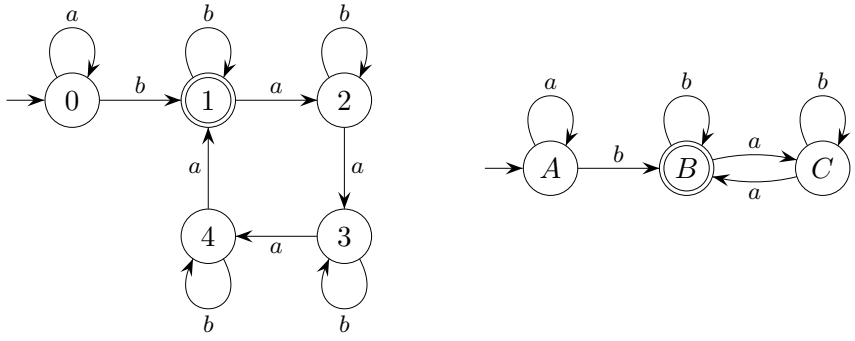
Next, we adapt the definition of NFA-homomorphism introduced in the previous section to the case of DFAs.

Definition 2.17 Let $\mathcal{A}^1 = (Q^1, \Sigma, q_0^1, \delta^1, F^1)$ and $\mathcal{A}^2 = (Q^2, \Sigma, q_0^2, \delta^2, F^2)$ be DFAs. A *DFA-homomorphism from* $\mathcal{A}^1$ *to* $\mathcal{A}^2$ is a homomorphism $h : Q^1 \to Q^2$ between the corresponding algebras. This means:

- $h(q_0^1) = q_0^2$
- $h(\delta^1(p, a)) = \delta^2(h(p), a)$
- $h(F^1) \subseteq F^2$

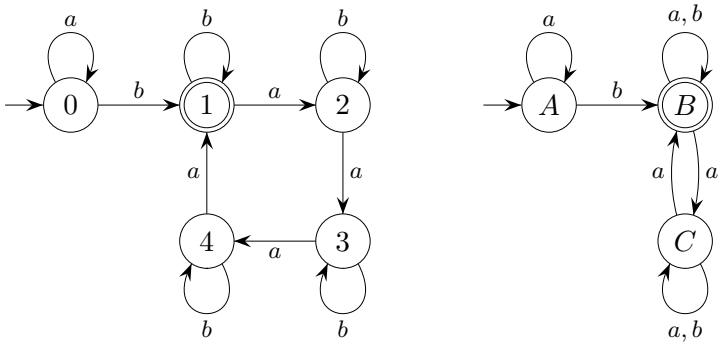
If h is a bijective function and $h(F^1) = F^2$, then h is called an *isomorphism.*

Example 2.18 From the left automaton we proceed to the right one by merging 1 with 3 and 2 with 4.



This corresponds to a homomorphism h with: $h(0) = A$, $h(1) = h(3) = B$, $h(2) = h(4) = C$. This homomorphism is also a DFA-homomorphism, because both the initial and the resulting automata are DFAs. ⊠

Example 2.19 On the same automaton as in the previous example we make a different merging of states, namely 1 with 2 and 3 with 4.



The mapping $h(0) = A$, $h(1) = h(2) = B$, $h(3) = h(4) = C$ yields an NFA-homomorphism, but not a DFA-homomorphism. We observe that if we want to preserve determinism in the resulting automaton, we cannot merge any pairs of states, even if this adheres to the conditions of the NFA-homomorphism. There must be some additional relation between the two states before they are merged. ⊠

Remark 2.20 *Every DFA-homomorphism induces an NFA-homomorphism; the converse does not hold. This means that if a DFA-homomorphism from $\mathcal{A}$ to $\mathcal{B}$ exists, then $L(\mathcal{A}) \subseteq L(\mathcal{B})$ holds (as a result of the induced NFA-homomorphism).*

Remark 2.21 *A partitioning of states with a corresponding equivalence $\sim$ defines a DFA-homomorphism iff $\sim$ is a* congruence with respect to the transition functions δ_a *for each $a \in \Sigma$ and preserves the distinction between final and non-final states. Formally:*

$$\begin{aligned} & p \sim q \quad \Rightarrow \quad \delta_a(p) \sim \delta_a(q) \\ \textit{and} \quad & p \sim q \quad \Rightarrow \quad (p \in F \Leftrightarrow q \in F) \end{aligned}$$

Definition 2.22 A DFA-homomorphism from $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ to $\mathcal{B}$ defines a quotient-DFA of $\mathcal{A}$ in the following way: We set $p \sim q$ iff $h(p) = h(q)$ and define $\mathcal{A}/_\sim = (Q/_\sim, \Sigma, q_0/_\sim, \delta/_\sim, F/_\sim)$ with

- $Q/_\sim$ = the set of $\sim$-classes $q/_\sim$ for $q \in Q$
- $\delta/_\sim(p/_\sim, a) = \delta(p, a)/_\sim$
- $p/_\sim \in F/_\sim$ iff $p \in F$

Explanation: The definition of $\delta/_\sim(p/_\sim, a)$ is independent of the representative p. This goes also for $F/_\sim$.

Definition 2.23 For a DFA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$, where w.l.o.g. we suppose that all states are reachable from q_0 we define $p \sim_\mathcal{A} q$, if $L(\mathcal{A}_p) = L(\mathcal{A}_q)$. Equivalently:

$$\delta_w(p) \in F \text{ iff } \delta_w(q) \in F \text{ for all } w \in \Sigma^* \ .$$

Remark 2.24 *a)* $\sim_{\mathcal{A}}$ *is a right congruence, i.e. an equivalence relation and*
if $p \sim_{\mathcal{A}} q$ *and* $a \in \Sigma$*, then* $\delta_a(p) \sim_{\mathcal{A}} \delta_a(q)$*.*

b) The DFA $\mathcal{A}/_{\sim_{\mathcal{A}}}$ *is called the* reduced DFA *of* $\mathcal{A}$*, short* $\mathcal{A}_{\text{red}}$ *and* $L(\mathcal{A}) = L(\mathcal{A}_{\text{red}})$*.*

Definition 2.25 (Nerode congruence) Let $L \subseteq \Sigma^*$.

$$u \sim_L v \text{ iff for all } w \in \Sigma^* \;\; uw \in L \Leftrightarrow vw \in L.$$

This means that for any future w, u and v need not be distinguished with respect to L. We often use the negation of this equivalence to prove that two words belong in different equivalence classes, namely $u \not\sim_L v$ iff there exists some w with $uw \in L$ and $vw \notin L$ (or $uw \notin L$ and $vw \in L$).

Example 2.26 Consider the language $L = a(bb)^*$. a and b are not equivalent: Take $w = \epsilon$, then $aw = a \in L$ but $bw = b \notin L$. $a, abb, abbbb, \ldots$ are for example equivalent. ϵ, a, b, ab are pairwise not equivalent. ⊠

Remark 2.27 $\sim_L$ *is a right congruence with respect to concatenation, which means that if* $u \sim_L v$*, then* $ua \sim_L va$ *and generally* $uw \sim_L vw$*.*

Proof: Show that $ua \sim_L va$, i.e. $uay \in L \Leftrightarrow vay \in L$ for all $y \in \Sigma^*$:

$$uay \in L \;\Leftrightarrow\; u(ay) \in L \;\Leftrightarrow \text{by assumption } v(ay) \in L \;\Leftrightarrow vay \in L$$

Notation: For the $\sim_L$-class of u we write $[u]_L$ or just $[u]$.

In a DFA $\mathcal{A}$ that recognizes L, we observe that if both u and v lead to the same state, then $u \sim_L v$. This holds because from $\delta(q_0, u) = \delta(q_0, v)$ we imply $\delta(q_0, uw) = \delta(q_0, vw)$ and consequently $uw \in L \Leftrightarrow vw \in L$. This means that there can be at most that many $\sim_L$-classes as there are states:

Remark 2.28 *If a DFA* $\mathcal{A}$ *recognizes the language* L*, then:*

$$\textit{Number of } \sim_L\textit{-classes} \leq \textit{number of } \mathcal{A}\textit{-states.}$$

It follows that if L is regular, then $\sim_L$ has a finite index*, i.e. there are finitely many equivalence classes.*

Definition 2.29 (Canonical DFA) Let L be a language with finite *index*$(\sim_L)$. The *canonical DFA* $\mathcal{A}_L = (Q_L, \Sigma, q_{0L}, \delta_L, F_L)$ for L is defined as follows:

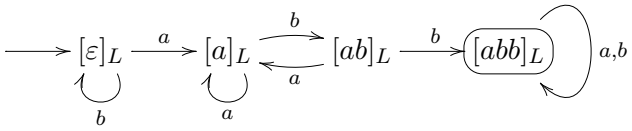
- $Q_L := \{\, [u]_L \mid u \in \Sigma^* \}$ the (finite) set of $\sim_L$-classes,
- $q_{0L} := [\varepsilon]$,

- $\delta_L([u], a) := [ua]$ and
- $F_L := \{[u] \mid u \in L\}$.

Example 2.30 Let $L = \{a, b\}^* abb\{a, b\}^*$. There are four different $\sim_L$-classes: $[\varepsilon]_L, [a]_L, [ab]_L, [abb]_L$. Since

$$[\varepsilon]_L = [b]_L \ , \ [aba]_L = [aa]_L = [a]_L \ , \ [abba]_L = [abbb]_L = [abb]_L$$

we obtain the following canonical automaton for L:



⊠

Proposition 2.31 *The canonical automaton $\mathcal{A}_L$ accepts L.*

Proof: $\mathcal{A}_L$ accepts $w \Leftrightarrow \delta(q_{0L}, w) \in F_L \Leftrightarrow [w]_L \in F_L \Leftrightarrow w \in L$ □

$\mathcal{A}_L$ has $index(\sim_L)$ many states. So $\mathcal{A}_L$ is a DFA accepting L with the minimum possible number of states. The following theorem shows that $\mathcal{A}_L$ is unique up to isomorphism.

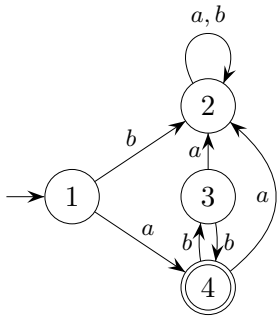
Theorem 2.32 (Main DFA-Minimization Theorem) *If the DFA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ recognizes L, then the DFA $\mathcal{A}_{\text{red}}$ is isomorphic to $\mathcal{A}_L$.*

Proof: We have to show that for $\mathcal{A}_{\text{red}} = (Q, \Sigma, q_0, \delta, F)$ there is a bijection $f : Q \longrightarrow Q_L$ with $f(q_0) = q_{0L}$, $f(F) = F_L$ and $f(\delta(q, a)) = \delta_L(f(q), a)$.

Definition of f: Pick for any state q of $\mathcal{A}_{\text{red}}$ a word u_q with $\delta(q_0, u_q) = q$ and let $f(q) := [u_q]_L$. Obviously f is an isomorphism between $\mathcal{A}_{\text{red}}$ and $\mathcal{A}_L$. □

Example 2.33 Consider the following automaton, which is obviously the minimal one that can recognize the language $L = a(bb)^*$. To map this automaton to a canonical one, we choose the following isomorphism:

$$1 \mapsto [\epsilon] \quad 2 \mapsto [bab] = [b] \quad 3 \mapsto [ab] \quad 4 \mapsto [abb] = [a]$$



⊠

Our aim is to compute from any DFA $\mathcal{A}$ a DFA $\mathcal{B}$ as in Theorem 2.32, and thus obtain a *minimal deterministic automaton*. Given a DFA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ we have to determine the equivalence classes of $\sim_{\mathcal{A}}$. But to fix two equivalent states, we have to examine *all* possible words that the automaton may read from those states, and examine whether they lead the automaton to a final state or not. That is why we prefer the converse: we identify all pairs of states that are *not* $\sim_{\mathcal{A}}$-equivalent. In other words, we identify those pairs p, q for which the following holds:

"there exists a w, such that one of the two states $\delta(p, w)$ and $\delta(q, w)$ is in F and the other one not"

To fix an appropriate w we proceed stepwise on its length. In the first step we take $w = \epsilon$ and separate all final states from the non-final ones: $(F, Q \setminus F)$. In the next step (because $p \sim_{\mathcal{A}} q \;\Rightarrow\; \delta(p, a) \sim_{\mathcal{A}} \delta(q, a)$) we declare two (not yet separated) states p and q as non-equivalent, if for some $a \in \Sigma$, $\delta(p, a)$ and $\delta(q, a)$ are non-equivalent as well. According to this concept we refine the classification $(F, Q \setminus F)$ as long as this is possible. Formally, the partition $Q \setminus F$ has to be split in two with respect to F if:

"there exist $p, q \in Q \setminus F$ and a letter a, with $\delta(p, a) \in F$ and $\delta(q, a) \notin F$; then the partition $Q \setminus F$ is split into $\{p \in Q \setminus F \mid \delta(p, a) \in F\}$ and $\{p \in Q \setminus F \mid \delta(p, a) \notin F\}$".

For every following step and a currently available partition $(B_1, \ldots, B_k)$ the task is generally described as follows:

B_i is being split in two with respect to B_j and a, if there exist p, q in B_i, such that $\delta(p, a) \in B_j$, but $\delta(q, a) \notin B_j$

This partitioning splits B_i into

$$(*)\ \{p \in B_i \mid \delta(p, a) \in B_j\} \text{ and } \{p \in B_i \mid \delta(p, a) \notin B_j\}$$

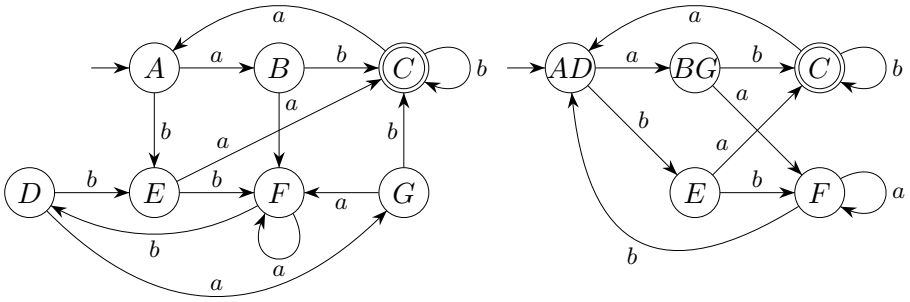


Figure 2.2: Partition refinement

Partition refinement algorithm:

1. Initialization: $B_1 = F,\ B_2 = Q \setminus F$

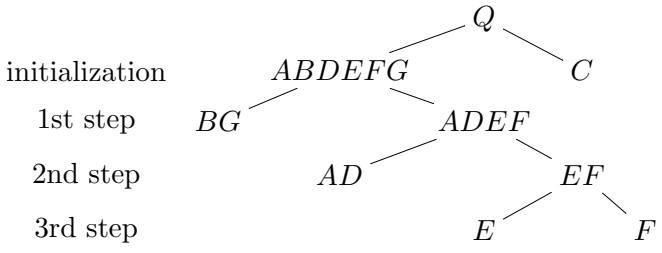
2. While there exist B_i, B_j, a with $(*)$, partition B_i with respect to B_j and a.

Example 2.34 We want to find equivalence classes in the automaton from the left-hand side of Figure 2.2. The initial partition is $(ABDEFG, C)$ and we now stepwise split these blocks according to $(*)$:

1st step: Splitting $ABDEFG$ w.r.t. C and b yields BG and $ADEF$.

2nd step: Splitting $ADEF$ w.r.t. BG and a yields AD and EF.

3rd step: Splitting EF w.r.t. C and a yields E and F.



The resulting minimized automaton is shown on the right-hand side of Figure 2.2. ⊠

Complexity of the algorithm: There can be at most $|Q|-1$ splits. Every loop requires at most to calculate $\delta(q,a)$ for $q \in Q$ and $a \in \Sigma$ (for checking condition $(*)$). Altogether this sums up to $O(|\Sigma| \cdot |Q|^2)$. We conclude that the minimization of a DFA can be done in second-order polynomial time with respect to the number of states. This can be improved by the algorithm of Hopcroft, which requires only $O(|Q| \log |Q|)$ time.

Now that we know an efficient algorithm to minimize DFAs, we can use it to construct an efficient equivalence test, according to the following corollary.

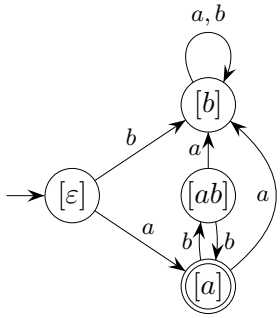
Corollary 2.35 (of the Main Minimization Theorem) *Two DFAs over Σ are equivalent, if and only if the corresponding reduced automata are isomorphic.*

This means that we can use the following algorithm to check, whether two DFAs are equivalent:

1. Given two DFAs $\mathcal{A}, \mathcal{B}$ construct their reduced versions $\mathcal{A}_{\text{red}}, \mathcal{B}_{\text{red}}$.
2. Starting from the initial state perform breadth-first search in $\mathcal{A}_{\text{red}}$, $\mathcal{B}_{\text{red}}$ and name each reached state q with the smallest word that leads to it, according to the canonical order.
3. Check whether the transition tables and the final states are identical.

Time complexity: $O(n \log n)$, like for the minimization.

Example 2.36 We consider the canonical automaton of Example 2.33 and apply the equivalence test algorithm. With breadth-first search we first visit



	a	b
$[\epsilon]$	$[a]$	$[b]$
$[a]$	$[b]$	$[ab]$
$[b]$	$[b]$	$[b]$
$[ab]$	$[b]$	$[a]$

the initial state $[\epsilon]$ and write it into the transition table. In the next loop we visit $[a]$ and from there $[b]$. We enter both under $[\epsilon]$. In the next loop we reach again $[b]$, but take no action, because we have already seen this state. In the last loop we visit $[ab]$ and write it down into the transition table. We complete the transition table for every state that we reached

and every symbol of the alphabet. If we follow the same procedure on the reduced version of any other automaton, we can find out whether it recognizes the same language as the one above, by simply comparing the resulting transition tables. ⊠

To end up the minimization of DFAs we consider a special case of languages, namely the ones with specific word length over the alphabet $\Sigma = \{0, 1\}$. It is obviously about finite languages $L \subseteq \{0, 1\}^n$, i.e. the sets of all bit sequences of length n.

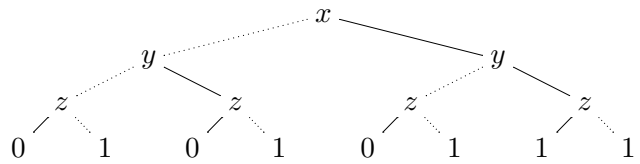
Remark 2.37 *A language $L \subseteq \{0, 1\}^n$ specifies a boolean function $f : \{0, 1\}^n \to \{0, 1\}$ according to the following mapping:*

$$f(b_1, \ldots, b_n) = 1 \text{ iff } b_1 \ldots b_n \in L$$

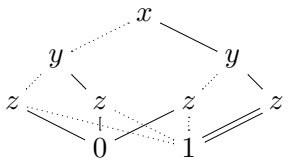
and vice versa.

Such languages can be defined by a special kind of DFA, namely one that has no cycles, but rather the form of a decision tree. The natural question to ask now, is whether we can apply some similar kind of partition refinement in this kind of a DFA to construct a "reduced diagram". We illustrate this idea by the following example.

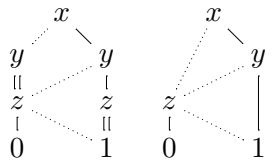
Example 2.38 Consider the formula $(x \wedge y) \vee \neg z$. We write the corresponding decision tree, where a continuous line denotes a transition with a 1 (as an interpretation of the starting node) and a dotted one, a transition with a 0. A double line means that the same transition is taken either with a 0 or with a 1.



Because the boolean function can take only two values, namely 0 or 1, we only need two leaves. By merging similar leaves, we get the following diagram, which is not a tree anymore:



We observe that the 3 leftmost z's have the same view on the part of the tree that is below them, i.e. they "see" a 1 through a transition with a 0, and a 0 through a transition with a 1. That is why we can merge these nodes and get the following diagram (left):



Finally, a node that leads to its (only) successor with both a 0 or a 1 can be eliminated, because it does not affect the final result. This leads us to the final minimized diagram above (right). ⊠

Formally the procedure can be described by the following reduction rules:

1. Merge two nodes p, q if they have isomorphic structures. (This corresponds to the relation $p \sim_{\mathcal{A}} q$ for a DFA $\mathcal{A}$)

2. In case there exist transitions $p \to_b q$ and $q \to_0 r$, $q \to_1 r$, strike out q and introduce the transition $p \to_b r$.

Note that the second rule makes use of the fact that the nodes are labeled by the names of the variables. This way, striking out a node does not lead to processing the wrong bits. If we apply these reduction rules on a decision tree, until no rules can be applied anymore, then the resulting diagram is called a *"Reduced Ordered Binary Decision Diagram"* (ROBDD).

Theorem 2.39 (Main Theorem on ROBDDs) *The ROBDD for a boolean function does not depend on the order, in which the reduction rules are applied: it is unambiguously defined up to isomorphism.*

Corollary 2.40 *The equivalence test for definitions of boolean functions can be reduced on the isomorphism test of ROBDDs.*

2.3 Equivalence and Reduction of NFAs

In this section we examine whether the problems that we efficiently solved in the last section on DFAs are likewise efficiently solvable in the case of NFAs. The main motivation towards this goal is the fact that minimal NFAs can be exponentially smaller than equivalent minimal DFAs with respect to the number of states.

Example 2.41 Consider the language L_n = the set of words over $\{a, b\}$ having an a on the n-th position before the end. A minimal NFA can recognize this language with $n + 1$ states by guessing when the n-th last position is being read and initiate a run of n transitions to the final state. On the contrary, a minimal DFA would always have to memorize the last n letters that it has processed by having built in a path for each possible combination. This needs obviously 2^n states. ⊠

By this example we realize that the minimization of NFAs can save up a lot of space when modeling a system. Unfortunately, this does not come without a cost. In particular, in this section we deal with the following issues:

- The PSPACE-hardness of the equivalence problem and the minimization problem for NFAs
- An efficient reduction of NFAs using the bisimulation equivalence between states
- The Universal NFA and the search for a minimal NFA

We address only the decision problems of minimization, equivalence and non-universality and show that even these are PSPACE-hard.
NFA Minimization Problem

- Given: NFAs $\mathcal{A}, \mathcal{B}$ over the same alphabet
- Question: Is $\mathcal{B}$ a minimal NFA equivalent to $\mathcal{A}$?

NFA Equivalence Problem

- Given: NFAs $\mathcal{A}, \mathcal{B}$ over the same alphabet
- Question: Are $\mathcal{A}$ and $\mathcal{B}$ equivalent?

NFA Non-Universality Problem

- Given: NFA $\mathcal{A}$ over the alphabet Σ.
- Question: Does $L(\mathcal{A}) \neq \Sigma^*$ hold?

At the first glance this last problem looks very similar to our fairly known (and by the way very efficiently solvable) non-emptiness problem. However, we will see that there is a big difference between them.

Before we start, let us refresh our memory with some issues out of Complexity Theory. We know that a decision problem is P, NP or PSPACE if it can be solved by a

- Turing Machine (TM) in polynomial time
- non-deterministic TM in polynomial time
- TM using at most polynomial space

respectively.

Remark 2.42 *We code a decision problem by a language L as follows: L contains the word codings of those instances, for which the expected answer is "yes".*

Furthermore, we also know that P $\subseteq$ NP $\subseteq$ PSPACE and we strongly suspect that the inclusions are proper.

Definition 2.43 A language L_0 is PSPACE-complete, if

1. L_0 belongs to PSPACE
2. Every language $L \in$ PSPACE, say over the alphabet Σ, is reducible in polynomial time to L_0 (short: $L \leq_p L_0$), i.e. there exists function F on Σ^*, such that it is computable in polynomial time and for all $w \in \Sigma^*$ $w \in L$ iff $F(w) \in L_0$ holds. In colloquial English, this means that a problem that has this property is at least as hard as any other problem in PSPACE.

L_0 is *PSPACE-hard*, if the second condition holds.

Example 2.44 QBF, namely the evaluation problem for quantified boolean formulas is PSPACE-complete. ⊠

Our aim is now to prove that all three decision problems mentioned above are PSPACE-hard, i.e. they are not efficiently solvable like in the case of DFAs. (It can be also shown that they belong to PSPACE, but we are not going to deal with that part of the proof.) For the case of the Non-Universality Problem of NFAs we have to prove that for every L in PSPACE "$L \leq_p$ NFA Non-Universality Problem" holds.

Lemma 2.45 *For an arbitrary language $L \subseteq \Sigma^*$ that belongs to PSPACE, there exists a transformation F that is computable in polynomial time, such that $F : \Sigma^* \rightarrow$ set of all NFAs over Σ' with $(*)$ $w \in L$ iff NFA $F(w)$ does not accept all words over Σ'.*

Proof: Let M be a Turing Machine that decides L using polynomial space. We define M by a tuple $M = (Q, \Sigma, \Gamma, q_0, \delta, F)$ with

- state set Q, initial state $q_0 \in Q$, final state set $F \subseteq Q$
- working alphabet Γ, input alphabet $\Sigma \subseteq \Gamma$ (with a distinguished blank symbol $\sqcup$ in $\Gamma \setminus \Sigma$)
- transition function $\delta : (Q \setminus F) \times \Gamma \rightarrow \Gamma \times \{N, L, R\} \times Q$

A transition $\delta(p, a) = (b, L/R/N, q)$ implies: "when being in state p and reading an a on the working cell, write there a b, move the head to the left / move the head to the right / don't move the head and go to state q". Now assume the TM M decides L and is space-bounded by some polynomial $p(n)$. The computational situation, in which M is, is described by its configuration, which is defined by the word that is written on the left of the

working cell, the current state and the word that is written on the right of the working cell. So, a configuration of M has the form uqv.

Let $w = a_1 \dots a_n$. An *accepting computation* M, w is a word over $\Gamma \cup Q \cup \{\#\}$ of the following form: $u_0 q_0 v_0 \# u_1 q_1 v_1 \# \ \dots \ \# u_s q_s v_s \#$, where

1. $u_0 = \sqcup \dots \sqcup$ and $v_0 = w \sqcup \dots \sqcup$, and u_0, v_0 have been chosen in such a way that throughout the whole computation M visits only the cells occupied by u_0, v_0, so $|u_i| + |v_i| = p(n)$

2. $u_{i+1} q_{i+1} v_{i+1}$ is a successor configuration of $u_i q_i v_i$

3. $u_s q_s v_s$ is a configuration with $q_s \in F$

The purpose of the reduction is to find a transformation F computable in polynomial time with $F : \Sigma^* \to$ set of all NFAs over the alphabet Σ' having the property:

$$(*) \quad w \in L \ \text{ iff } \ \text{NFA } F(w) \text{ does not accept all words in } (\Sigma')^*$$

We choose $\Sigma' = \Gamma \cup Q \cup \{\#\}$ and we define the NFA $F(w)$ in such a way that it accepts exactly the words that are *not an accepting* M, w*-computation.* Then $(*)$ holds! To achieve this, the NFA $F(w)$ has to check whether on the input u one of the constraints 1. to 3. is indeed violated. In other words, the NFA has to verify that:

1. The input word does not begin with $u_0 q_0 v_0 \#$,

2. OR there are two consecutive segments $u_i q_i v_i \# u_{i+1} q_{i+1} v_{i+1} \#$, such that $u_{i+1} q_{i+1} v_{i+1}$ is not a successor configuration of $u_i q_i v_i$

3. OR that the last configuration does not contain a final state.

The first condition can be tested with $p(n)+2$ states (because $|u_0 v_0| = p(n)$). The second condition can be tested by guessing 3 consecutive cells, whose update (after $p(n) + 2$ positions) is not correct. $O(p(n) + 2)$ states suffice for this test.

Explanation: Assume that the transition function of the Turing Machine implies $\delta(p, b) = (a, L, q)$ and consider the following two consecutive configurations (without the square brackets):

$$\dots \underbrace{[aab][apb] \dots \# \dots}_{p(n)+2} [aab][qaa]$$

The automaton may compare the three consecutive cells in the first square brackets with the ones that are $p(n) + 2$ positions away. In the case that none of these cells is currently (or in the previous step) occupied by the

head of the Turing Machine, one cell would suffice for the comparison. But in case one of the cells contains the state of the Turing Machine (like in the second square brackets) three cells are required. In the example above the NFA verifies that the Turing Machine changed from state p to q, moving the head to the left (that is why the q is now written on the left of the a that was previously on the left of the p) and printing an a on the b. In this example the cells adhere to the transition function. In the general case the Turing Machine has to guess such a pair of triples where this does not hold. It is now obvious that $O(p(n) + 2)$ states suffice for this test, where the constant represents the space that is required to save a triple of cells like it was mentioned above.

The last condition can be trivially tested by using a memory state to remember the information "no final state has been seen between the last two #". Consequently, from M and w we can construct the appropriate NFA $F(w)$ as a union-NFA from the three automata mentioned above with $p(n)+2$, const$\cdot(p(n)+2)$ and 5 states respectively. Altogether, this NFA has polynomially many states with respect to $n = |w|$, which also means that the transformation $w \mapsto F(w)$ is computable in polynomial time. Thus, we have proven the following theorem:

Theorem 2.46 *The NFA-Non-Universality Problem is PSPACE hard.*

Now that we have extensively dealt with the non-universality problem on NFAs, let us compare it with a much more easily solvable problem that looks (at least at the first glance) similar to it, namely the non-emptiness problem for NFAs. So, given an NFA $\mathcal{A}$ we compare the following questions:

1. $L(\mathcal{A}) = \emptyset$? or the negation $L(\mathcal{A}) \neq \emptyset$?
2. $L(\mathcal{A}) = \Sigma^*$? or the negation $L(\mathcal{A}) \neq \Sigma^*$?

The first question is about the existence of an appropriate w and the existence of some corresponding accepting run on it, which, as we know, is solvable in linear time (by performing breadth-first or depth-first search). On the contrary, the second problem is about the existence of an accepting run for each possible w, or for the existence of such a w that all possible runs on it are rejecting. Now we can see that there is a big difference in terms of logic between the two problems. The first one contains two consecutive existential quantification whereas the second is expressed by a universal quantification followed by an existential one (or vice versa in the case of the negation). The PSPACE hardness of the second problem relies exactly on this alternation of the existential and the universal quantifier. It is easy to see that the universality problem of an NFA $\mathcal{A}$ over Σ is a special case of the equivalence problem between $\mathcal{A}$ and the standard NFA $\mathcal{A}_0$ with just one state that is also a final one (obviously accepting Σ^*). Likewise,

it is a special case of the minimization problem "is $\mathcal{A}_0$ a minimal NFA for $\mathcal{A}$?". Hence, both the NFA-equivalence problem and the NFA-minimization problem have to be at least as hard as the NFA-universality problem.

Theorem 2.47 *NFA-equivalence problem and the NFA-minimization problem are PSPACE hard.*

Since we have proven that we cannot minimize NFAs in an efficient way, we try to find a way at least to reduce them, e.g. by merging two states that have been found to be equivalent. For this reason we consider the reduction of NFAs, by using the state equivalence: $p \sim_{\mathcal{A}} q$ ("whether starting from p or from q the same language is recognized"). For NFAs there are still two problems:

1. As it was shown before: The quotient NFA does not have to be the minimal NFA.
2. Fixing pairs of equivalent states is hard, because it requires an NFA equivalence test.

The question now is, whether there exists a state equivalence that is both efficiently computable and language preserving (i.e. such that the quotient NFA is equivalent to the original one). To define this equivalence we need to introduce a special kind of NFAs with the convention that the final states are exactly those states without outgoing transitions.

Definition 2.48 (Labeled Transition System) A *Labeled Transition System* (LTS) is a tuple $\mathcal{A} = (Q, \Sigma, \Delta)$, where Q is a set of states, Σ is an alphabet and Δ is a transition relation. By adding an initial state $q \in Q$, we get $(\mathcal{A}, q)$, a *pointed LTS*.

We consider a pointed LTS $(\mathcal{A}, q_0)$ as a usual NFA, where the set of final states $F_{\mathcal{A}}$ is implicitly defined as $\{q \in Q \mid \text{ there is no } (q, a, q') \in \Delta\}$.

Definition 2.49 (Bisimulation game) A *bisimulation game* $\mathrm{BG}(\mathcal{A}, p_0, \mathcal{B}, q_0)$ is played between players I (proposer) and II (responder) on labeled transition systems $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \Delta_{\mathcal{A}})$ and $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, \Delta_{\mathcal{B}})$.

A *configuration* is a pair $(p, q) \in Q_{\mathcal{A}} \times Q_{\mathcal{B}}$. The *initial configuration* is (p_0, q_0). A *round* of a game has two steps. From configuration (p, q)

- player I picks a transition $(p, a, p') \in \Delta_{\mathcal{A}}$ or $(q, a, q') \in \Delta_{\mathcal{B}}$ and
- player II responds with a transition $(q, a, q') \in \Delta_{\mathcal{B}}$ or $(p, a, p') \in \Delta_{\mathcal{A}}$ respectively.

The new configuration is now (p', q'). If player I cannot choose a transition he loses. If player II cannot respond correctly he loses. If a play goes on forever, player II wins.

A *strategy* is a mapping $\sigma : (Q_A \times Q_B)^+ \to (Q_A \cup Q_B)$ for player I and similarly $\sigma : (Q_A \times Q_B)^+(Q_A \cup Q_B) \to (Q_A \cup Q_B)$ for player II. A *winning strategy* for player II ensures that for every configuration (p, q) that is reached while II plays according to this strategy, player I will not win the $\mathrm{BG}(\mathcal{A}, p, \mathcal{B}, q)$.

$(\mathcal{A}, p_0)$ and $(\mathcal{B}, q_0)$ are called *bisimilar*, $(\mathcal{A}, p_0) \sim (\mathcal{B}, q_0)$, if player II has a winning strategy in $\mathrm{BG}(\mathcal{A}, p_0, \mathcal{B}, q_0)$. In the special case that $(\mathcal{A}, p) \sim (\mathcal{A}, q)$ we call p and q $\sim$-equivalent ($p \sim q$).

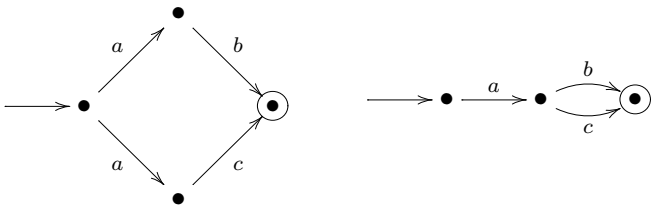
Proposition 2.50 $(\mathcal{A}, p_0) \sim (\mathcal{B}, q_0) \Rightarrow L(\mathcal{A}_{p_0}) = L(\mathcal{B}_{q_0})$

Proof: By assumption player II has a winning strategy in $\mathrm{BG}(\mathcal{A}, p_0, \mathcal{B}, q_0)$. We show $L(\mathcal{A}_{p_0}) \subseteq L(\mathcal{B}_{q_0})$. (The other direction is similar to this one.)

Assume $w \in L(\mathcal{A}_{p_0})$. We apply the winning strategy of player II to a play where player I moves in $\mathcal{A}$ from p_0 to $F_{\mathcal{A}}$ by a path labeled w. Player II realizes a path in $\mathcal{B}$ starting from q_0 and labeled with w. We have to check that the last state of this path is in $F_{\mathcal{B}}$. But this is clear, because otherwise player II would lose in the next step. □

The following example shows that the converse direction of proposition 2.50 fails.

Example 2.51 Consider the following two NFAs:



They accept the same language, but they are not bisimilar (both started on their leftmost state). ⊠

Remark 2.52 *Bisimulation equivalence is finer than language equivalence. This means that it results to a correct but not optimal reduction of an NFA.*

To compute the bisimulation equivalence between two LTS we have to check for all states p, q whether $(\mathcal{A}, p) \sim (\mathcal{B}, q)$. Again, similar to the minimization algorithm for DFAs, we determine the pairs p, q where player I has a winning strategy (showing non-equivalence).

Remark 2.53 *1. If $p \in F_{\mathcal{A}}$ and $q \notin F_{\mathcal{B}}$, or vice versa, player I wins after his first move.*

Input: LTS $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \Delta_\mathcal{A})$, $\mathcal{B} = (Q_\mathcal{B}, \Sigma, \Delta_\mathcal{B})$

1. Mark $p, q \in Q_\mathcal{A} \times Q_\mathcal{B}$ if $p \in F_\mathcal{A} \not\Leftrightarrow q \in F_\mathcal{B}$
2. **while** some states have been marked in the last iteration **do**
3. Mark each pair p, q previously unmarked for which there is a $(p, a, p') \in \Delta_\mathcal{A}$ such that for all $(q, a, q') \in \Delta_\mathcal{B}$ (p', q') is marked
4. Mark each pair p, q previously unmarked for which there is a $(q, a, q') \in \Delta_\mathcal{B}$ such that for all $(p, a, p') \in \Delta_\mathcal{A}$ (p', q') is marked.
5. **od**

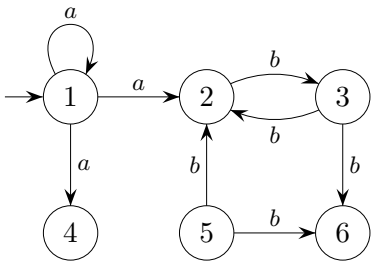
Output: All marked pairs.

Figure 2.3: Bisimulation Marking Algorithm

2. *If player I can pick $(p, a, p') \in \Delta_\mathcal{A}$ such that for all $(q, a, q') \in \Delta_\mathcal{B}$ that player II can choose, player I wins from (p', q'), then from (p, q) player I wins. Similarly, if player I can pick $(q, a, q') \in \Delta_\mathcal{B}$ such that for all $(p, a, p') \in \Delta_\mathcal{A}$ that player II can choose, player I wins from (p', q'), then from (p, q) player I wins.*

The Remark 2.53 motivates the *marking algorithm* for marking pairs (p, q) with $(\mathcal{A}, p) \not\sim (\mathcal{B}, q)$ shown in Figure 2.3.

Example 2.54 We apply the marking algorithm to reduce the NFA shown below. According to our convention, the final states are 4 and 6. That is

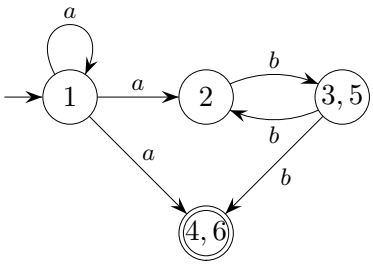


why before entering the loop, we mark all pairs, where exactly one of those two states occurs. In the first loop we mark the pair $(2, 1)$ because we can choose to move from 2 with b, but we cannot move with the same letter from state 1. We also mark the pair $(5, 2)$ because there exists a transition with b from 5 that leads to state 6 that is already marked with every state that is reachable from 2 with the same transition (namely 3, since it is the only state reachable with a b and $(6, 3)$ is marked). Continuing like this, the algorithm terminates leaving the pairs $(5, 3)$ and $(6, 4)$ unmarked.

	1	2	3	4	5
2	1	X	X	X	X
3	1	1	X	X	X
4	0	0	0	X	X
5	1	1		0	X
6	0	0	0		0

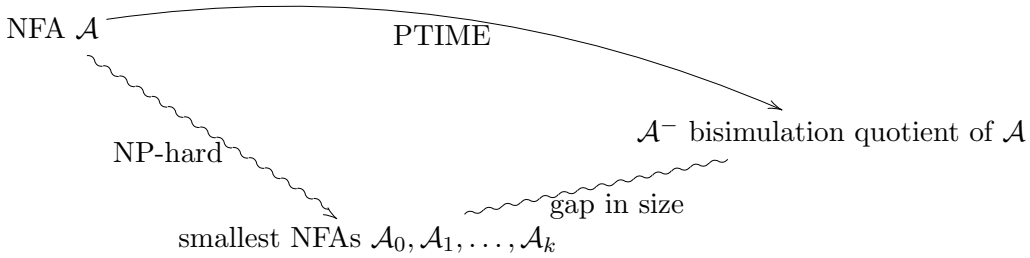
X : It is not necessary to look at this field.
0 : This field was marked before entering the loop (final and non-final states).
i : This field was marked in the i-th loop ($i = 1, 2, \ldots$).

This means that we can merge these states in the reduced automaton. The resulting NFA is shown below and it happens to be also a minimal one.



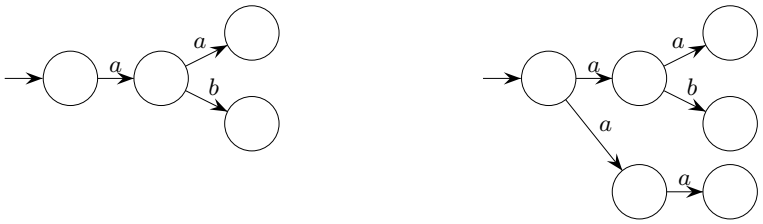
⊠

The algorithm terminates after $\leq |Q_{\mathcal{A}}| \cdot |Q_{\mathcal{B}}|$ loops (all possible pairs of states). Every loop requires the consideration of each previously unmarked pair (p, q) by checking the corresponding outgoing transitions, i.e. a computational effort that is proportional to $|Q_{\mathcal{A}}| \cdot |\Delta_{\mathcal{A}}| + |Q_{\mathcal{B}}| \cdot |\Delta_{\mathcal{B}}|$. Altogether, if $\mathcal{A}, \mathcal{B}$ have $|Q_{\mathcal{A}}| + |Q_{\mathcal{B}}| = n$ states and $|\Delta_{\mathcal{A}}| + |\Delta_{\mathcal{B}}| = m$ transitions, then the marking algorithm terminates in time $O(n^3 m)$. Better algorithms are known working in time $O(n \cdot \log^2(n))$ (Page and Tarjan, 1987).



There is a variant of $BG(\mathcal{A}, p, \mathcal{B}, q)$ that results to wider equivalence classes, and is therefore often considered as more appropriate. It is called the *alternating simulation* and the main difference to bisimulation is that this time player I has to stay in the NFA that he chose in his first move, i.e. he cannot arbitrarily switch between the two NFAs. In this case the quotient NFA is again language equivalent and efficiently computable.

Example 2.55 For the following NFAs the alternating simulation holds, but not the bisimulation.



⊠

Finally, there is an alternative technique that leads not only to the reduced, but even to the minimal NFA. The idea relies on the fact that given an NFA with n states, there are only finitely (of course exponentially) many NFAs with less than n states. The following procedure illustrates, how we can construct a (in most cases huge) NFA that contains all equivalent smaller NFAs and in which we can restrict our attention only to those NFAs who really have the possibility to be minimal ones. Given a regular language L, there is indeed a canonical NFA, in which we can search for the minimal one; it is constructed as follows:

1. Determine the minimal DFA $\mathcal{A}_L$ (powerset construction and DFA-minimization)

2. In $\mathcal{A}_L$ invert all transitions and switch all final states to initial ones and vice versa and thus obtain the NFA $\overline{\mathcal{A}_L}$.

3. Once again apply the powerset construction to obtain $D(\overline{\mathcal{A}_L})$ (deterministic).

4. Invert this DFA too and obtain $\overline{D(\overline{\mathcal{A}_L})}$.

5. The Universal-NFA $\mathcal{C}$ is constructed out of the state sets of the last NFA.

The NFA $\overline{D(\overline{\mathcal{A}_L})}$ has states $A_1, \ldots, A_k$, each of them consisting of $\mathcal{A}_L$-states. The states of the Universal-NFA $\mathcal{C}$ are sets of the form $\{A_{i_1}, \ldots, A_{i_m}\}$, where $A_{i_1} \cap \ldots \cap A_{i_m} \neq \emptyset$ must hold. For further details, please refer to O. Matz, A. Potthoff, "Computing Small Nondeterministic Automata".

Theorem 2.56 *The Universal-NFA of L contains each minimal NFA recognizing L as an induced NFA for an appropriate subset of its states.*

This confirms our initial claim, namely that the minimal NFA can be found within the Universal-NFA.

2.4 The Syntactic Monoid

In this section we introduce a structure that characterizes a regular language but also contains a neutral element and enables us to perform binary operations just like in the case of natural numbers. The reason why we find such a structure very useful to describe a language, is that the set of all words follows the same concept (neutral element, binary operations) but automata don't. In other words we shall see that using this structure we can directly identify characteristics of a language that cannot be directly handled by automata, like for example whether a language is star-free (i.e. FO-definable or LTL-definable)

Exactly as the minimal DFA is a canonical acceptor for a regular language L, we define a further canonical acceptor $M(L)$, the "syntactic monoid" for L, with the same signature as for all words over Σ, namely $(\Sigma^*, \cdot, \epsilon)$.

There is a close relation between the monoid structure and the transition structure of the minimal DFA $\mathcal{A}_L$. This relation is presented in the following. First, we refine the Nerode congruence by making it symmetric:

Definition 2.57 For a language $L \subseteq \Sigma^*$ we define: $u \approx_L v$ if $\forall x, y \in \Sigma^* : xuy \in L \Leftrightarrow xvy \in L$.

Remark 2.58 *$\approx_L$ is a right congruence with respect to concatenation, i.e. $\approx_L$ is an equivalence relation and $u \approx_L v$, $u' \approx_L v' \Rightarrow uu' \approx_L vv'$. If $u \approx_L v$ and $u' \approx_L v'$, then $xuu'y \in L$ iff $xvv'y \in L$ for all words x and y.*

Remark 2.59 *$u \approx_L v$ implies $u \sim_L v$ (take $x = \varepsilon$). The converse does not hold in general.*

Example 2.60 Consider the language $L = bba^*$. $a \sim_L ba$, because for all $w \in \Sigma^*$: $aw \notin L$ and $baw \notin L$. But $a \not\approx_L ba$, because $b \cdot a \cdot \epsilon \notin L$, and $b \cdot ba \cdot \epsilon \in L$. ⊠

Notation:

- The $\approx_L$-class of $u \in \Sigma^*$ is denoted by $\langle u \rangle_L$.
- Let $\Sigma^*/_{\approx_L}$ be the set of all $\approx_L$-classes.

A multiplication (concatenation) between $\approx_L$-classes is defined by representatives: $\langle u \rangle_L \cdot \langle v \rangle_L := \langle uv \rangle_L$. The product is well-defined because $\approx_L$ is a right congruence.

Remark 2.61 *$(\Sigma^*, \cdot, \epsilon)$ is a monoid (i.e. $\cdot$ is associative, and ϵ is the neutral element).*

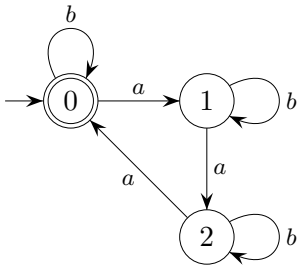
The structure $M(L) := (\Sigma^*/_{\approx_L}, \cdot, \langle\epsilon\rangle)$ is also a monoid, namely the *syntactic monoid* of L. But how can we construct the correct monoid for a language? We do this by applying specific transformations on the automaton (we take the canonical one) that recognizes this language.

Let $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ be a DFA. Every word $w \in \Sigma^*$ induces a state transformation $w^{\mathcal{A}} : Q \to Q$ that is defined by $w^{\mathcal{A}}(q) := \delta(q, w)$. Note that $\epsilon^{\mathcal{A}} = \mathrm{id}_Q$ (identity function). The composition goes as follows: $u^{\mathcal{A}} \circ v^{\mathcal{A}}$ is the function $(uv)^{\mathcal{A}}$: $u^{\mathcal{A}} \circ v^{\mathcal{A}}(q) = v^{\mathcal{A}}(u^{\mathcal{A}}(q))$.

Remark 2.62 *The structure* $T(\mathcal{A}) = (\{u^{\mathcal{A}} \mid u \in \Sigma^*\}, \circ, \mathrm{id}_Q)$ *is a monoid, namely the* transition monoid *of* $\mathcal{A}$.

Note that over Q, there are at most $|Q|^{|Q|}$ many functions. This means that the transition monoid of a DFA is finite. If $Q = \{0, \ldots, n\}$, then we represent the state transformation $u^{\mathcal{A}}$ by the sequence $u^{\mathcal{A}}(0), \ldots, u^{\mathcal{A}}(n)$ of values on the states of the DFA.

Example 2.63 Consider the following automaton and the transformations beside.



$\epsilon^{\mathcal{A}}$: (0 1 2)
$a^{\mathcal{A}}$: (1 2 0)
$aa^{\mathcal{A}}$: (2 0 1)

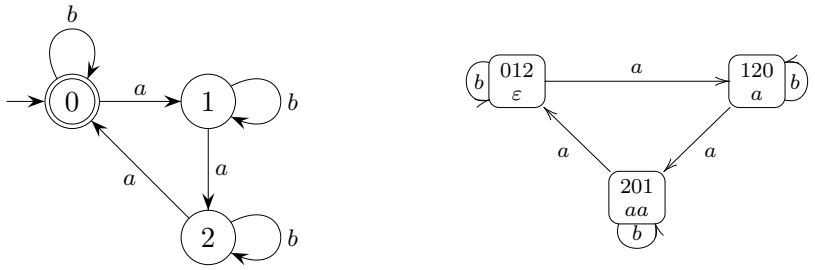
⊠

One can easily see that there are no other (different) transformations on this DFA, e.g. $b^{\mathcal{A}} = \epsilon^{\mathcal{A}}$ or $aabaa^{\mathcal{A}} = a^{\mathcal{A}}$. But of course this is no proof that we have indeed found all possible transformations. There is a formal method to determine all functions $u^{\mathcal{A}}$ and it goes as follows:

1. Go through all words u in canonical order, until no new functions occur.

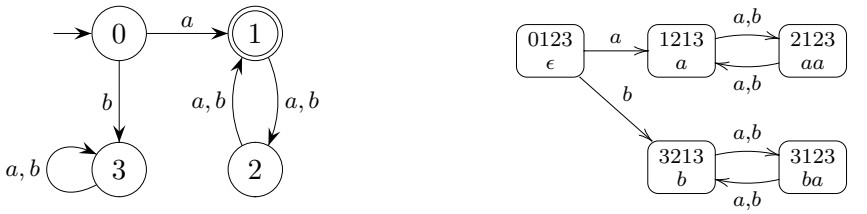
2. Draw an a-labeled arrow from $u^{\mathcal{A}}$ to $(ua)^{\mathcal{A}}$

Example 2.64 Let $L_1 := \{\, w \in \Sigma^* \mid |w|_a \equiv 0 \pmod 3 \,\}$. The DFA $\mathcal{A}_{L_1}$ is shown below:



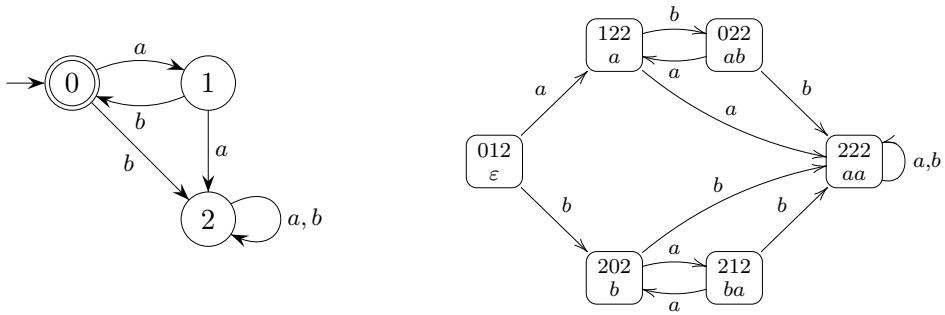
Having determined the functions from the previous example, we draw the transition monoid (on the right). ⊠

Example 2.65 Consider the language $L_2 := \{\, aw \in \Sigma^* \mid |w| \equiv 0 \pmod 2 \,\}$ recognized by the following automaton $\mathcal{A}_{L_2}$:



The corresponding transition monoid $T(\mathcal{A}_{L_2})$ is drawn on the right. ⊠

Example 2.66 $L_3 := (ab)^*$ is recognized by the following automaton $\mathcal{A}_{L_3}$:



The corresponding transition monoid $T(\mathcal{A}_{L_3})$ is drawn on the right. ⊠

Now we can proceed to constructing the syntactic monoid. For this purpose we prove the following theorem.

Theorem 2.67 *Let L be regular and $\mathcal{A}_L$ the minimal automaton recognizing L. The syntactic monoid $M(L)$ is isomorphic to the transition monoid of $\mathcal{A}_L$, short: $M(L) \cong T(\mathcal{A}_L)$.*

Hence, by constructing the transition monoid (by using the previously illustrated method) and by applying the last theorem, we obtain the syntactic monoid.

Proof: Define $f : M(L) \to T(\mathcal{A}_L)$ by $f(\langle u \rangle) := u^{\mathcal{A}_L}$. f is the appropriate isomorphism. f is well-defined and injective because

$$\langle u \rangle = \langle v \rangle \quad \text{iff} \quad u^{\mathcal{A}_L} = v^{\mathcal{A}_L}.$$

Verification: $\langle u \rangle = \langle v \rangle$

iff $\forall x, y\colon xuy \in L \iff xvy \in L$

iff $\forall x\colon [xu] = [xv]$ (by definition)

iff $\forall x\colon u^{\mathcal{A}_L}([x]) = v^{\mathcal{A}_L}([x])$
(Note that $[x]$ is a state of $\mathcal{A}_L$ and $u^{\mathcal{A}_L}([x]) = [xu]$)

iff $u^{\mathcal{A}_L} = v^{\mathcal{A}_L}$

f is surjective, because

$$u^{\mathcal{A}_L} = f(\langle u \rangle) \text{ for all } u^{\mathcal{A}_L} \in T(\mathcal{A}_L)$$

Finally, f is a homomorphism, because

$$f(\langle u \rangle \cdot \langle v \rangle) = f(\langle uv \rangle) = (uv)^{\mathcal{A}_L} = u^{\mathcal{A}_L} \circ v^{\mathcal{A}_L} = f(\langle u \rangle) \circ f(\langle v \rangle)$$

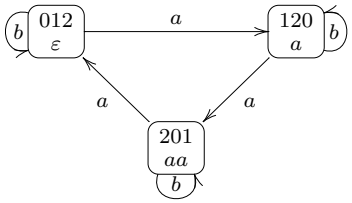
□

Now we can use these results to identify the class, in which a language belongs. For example, in the previous chapter we have shown that the star-free languages are non-counting. In the following we show that it can be identified, whether a language is non-counting, by the property "group-free" of the syntactic monoid. Furthermore, we show that for regular languages, the properties non-counting and star-free coincide.

Definition 2.68 An element m of a monoid M is called *idempotent*, if $m \cdot m = m$.

Every subset of M that forms a group through multiplication $\cdot$, has exactly one idempotent element. This is the neutral element of the group. We call a monoid M *group-free* if the only occurring subsets that form groups are the trivial ones $\{m\}$ with an idempotent m.

Example 2.69 We look at the monoid of a previous example.

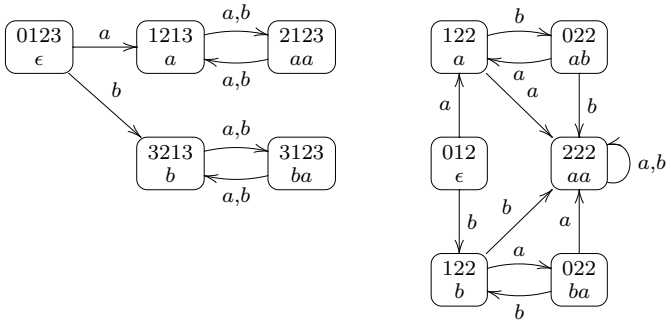


	ϵ	a	aa
ϵ	ϵ	a	aa
a	a	aa	ϵ
aa	aa	ϵ	a

	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Obviously the whole monoid represents a non-trivial group, since it is counting modulo-3. The two multiplication and addition tables show the correspondence between this monoid and the group $(\mathbb{Z}_3, +)$ respectively. ⊠

Example 2.70 Consider the following two monoids.



The left one has two groups, each one of the type $\mathbb{Z}_2$, while the right one does not have any non-trivial groups, i.e. it is group-free. ⊠

Theorem 2.71 (Schützenberger 1965) *Let L be regular. L is star-free if and only if $M(L)$ is group-free.*

According to this theorem, one can algorithmically decide, whether a given regular language is star-free.

Example 2.72 According to the previous examples:

1. $L_1 := \{ w \in \Sigma^* \mid |w|_a \equiv 0 \pmod 3 \}$ is not star-free.
2. $L_2 := \{ aw \in \Sigma^* \mid |w| \equiv 0 \pmod 2 \}$ is not star-free.
3. $L_3 := (ab)^*$ is star-free.

⊠

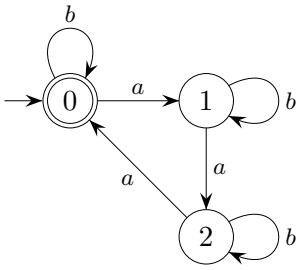
Theorem 2.71 allows even the generation of a star-free expression out of a given group-free monoid. This is implemented in the AMoRE system.

We can also determine whether a monoid $M(L)$ is group-free by directly analyzing the corresponding minimal DFA $\mathcal{A}_L$. The required property is given in the following definition.

Definition 2.73 A DFA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ *allows permutations*, in case there exist a subset $Q_0 \subseteq Q$ and a word $w \in \Sigma^+$, such that $w^{\mathcal{A}}|_{Q_0}$ represents a non-trivial permutation on Q_0. In the opposite case $\mathcal{A}$ is called *permutation-free*.

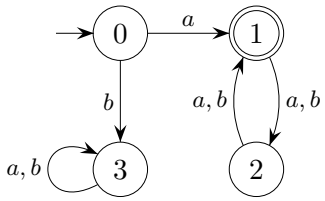
Example 2.74 We examine the minimal deterministic automata recognizing the three languages of the previous examples.

$\mathcal{A}_{L_1}$:



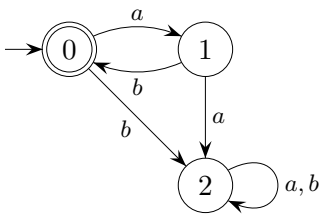
This DFA allows permutations. Take $Q_0 = \{0, 1, 2\}$ and $w = a$. Then $a^{\mathcal{A}_{L_1}}|_{\{0,1,2\}}$ represents a non-trivial permutation on $\{0, 1, 2\}$ (0 1 2 $\mapsto$ 1 2 0).

$\mathcal{A}_{L_2}$:



This DFA allows permutations. Take $Q_0 = \{1, 2\}$ and $w = a$. Then $a^{\mathcal{A}_{L_2}}|_{\{1,2\}}$ represents a non-trivial permutation on $\{1, 2\}$ (1 2 $\mapsto$ 2 1).

$\mathcal{A}_{L_3}$:



This DFA is permutation-free. ⊠

Finally, we gather all the previous properties of a regular language in the following theorem.

Theorem 2.75 *For a regular language L the following conditions are equivalent:*

(a) L is non-counting.

(b) The minimal DFA $\mathcal{A}_L$ is permutation-free.
(c) The syntactic monoid $M(L)$ is group-free.
(d) L is star-free.

According to the theorem of Schützenberger (c) $\Leftrightarrow$ (d). In the following we show

- not (c) $\Rightarrow$ not (b),
- not (b) $\Rightarrow$ not (a),
- not (a) $\Rightarrow$ not (c).

Proof (from group to permutation): Assume $M(L)$ is not group-free, say having a group G. Choose a $g \in G$ other than the neutral element of G and consider the cyclic group $\{g, g^2, \ldots, g^k\}$ with g^k as a neutral element. $g = \langle u \rangle$ corresponds via the isomorphism $M(L) \cong T(\mathcal{A}_L)$ to the transformation $u^{\mathcal{A}_L}$. Then $g^i = \langle u^i \rangle$, and the following holds:

1. $(u^k)^{\mathcal{A}_L} = (u^{2k})^{\mathcal{A}_L}$
2. $(u^k)^{\mathcal{A}_L} \neq (u^{k+1})^{\mathcal{A}_L}$

Now choose a q, such that $\delta(q, u^k) \neq \delta(q, u^{k+1})$. Set $Q_0 = \{\delta(q, u^k), \delta(q, u^{k+1}), \ldots, \delta(q, u^{2k-1})\}$. Then $u^{\mathcal{A}_L}$ is a permutation on Q_0. □

Proof (from permutation to modulo-counting): Let the transformation $v^{\mathcal{A}_L}$ be a permutation π on Q_0. Choose a $q \in Q_0$ and obtain the elements $q, \pi(q), \pi^2(q), \ldots, \pi^m(q) = q$. Then $\delta(q, v^{km}) = q$, $\delta(q, v^{km+1}) = q' \neq q$. Choose u, w such that $\delta(q_0, u) = q$, and not $\delta(q, w) \in F \Leftrightarrow \delta(q', w) \in F$. Now we have to show that for any arbitrarily large n:

$$\exists u, v, w\text{: not } uv^n w \in L \Leftrightarrow uv^{n+1} w \in L$$

Take $n = km$ where k is arbitrary. Then *it does not hold* that

$$\delta(q_0, uv^{km} w) \in F \Leftrightarrow \delta(q_0, uv^{km+1} w) \in F$$

□

Proof (from modulo-counting to group): Note that in every finite monoid each element m has a power m^k that is idempotent. Now set $K =$ the smallest common multiple of these k-values. For every element m of a monoid $m^K = m^{2K}$ $(*)$. By hypothesis there are arbitrarily large n's (also $> K$), such that for appropriate u, v, w:

$$uv^n w \in L \text{ but } uv^{n+1} w \notin L$$

For such an n then $\langle v^n \rangle \neq \langle v^{n+1} \rangle$. Because of $(*)$, $\langle v^K \rangle = \langle v^{2K} \rangle$. Then $\langle v^K \rangle \neq \langle v^{K+1} \rangle$ must hold. Now $\langle v^{K+1} \rangle$ forms a group in $M(L)$. □

2.5 Exercises

Exercise 2.1 Find a family $(L_n)_{n>0}$ of languages over $\{a, b\}$, such that

- L_n is recognized by a DFA with $\mathcal{O}(n)$ states, and
- every DFA that recognizes L_n^{R} has at least 2^n many states.

Hint: The first automaton of Example 2.14 points to the right direction. ($L_n^{\mathrm{R}} = \{b_n \cdots b_1 \mid b_1 \cdots b_n \in L\}$ is the language of the flipped words.)

Exercise 2.2* Consider the following algorithm that is applied on a DFA $\mathcal{A}$ (over $\Sigma = \{a, b\}$):

1. Invert $\mathcal{A}$, i.e. turn the $\mathcal{A}$-transitions to the opposite direction and switch the initial states with the final ones. Obtain $\mathcal{A}^{\mathrm{R}}$. (Note that the automaton may have several initial states.)
2. Apply the powerset construction (construct only the reachable states). Obtain $D(\mathcal{A}^{\mathrm{R}})$. (The initial state of this automaton is the set of initial states of the NFA.)
3. Invert $D(\mathcal{A}^{\mathrm{R}})$, obtain $(D(\mathcal{A}^{\mathrm{R}}))^{\mathrm{R}}$.
4. Apply the powerset construction as in (2), obtain $D((D(\mathcal{A}^{\mathrm{R}}))^{\mathrm{R}})$.

Show that the resulting automaton is the minimal automaton for $\mathcal{A}$.

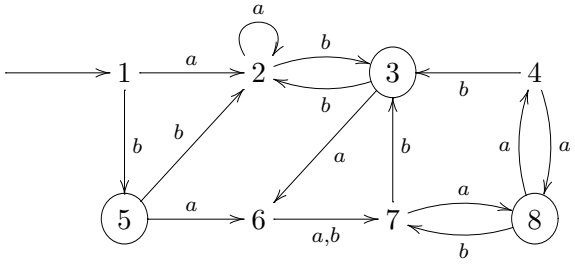
Exercise 2.3 Consider the language $L = b(a + b)^*aa$ over the alphabet $\{a, b\}$.

(a) Give representatives for the five $\sim_L$-classes.

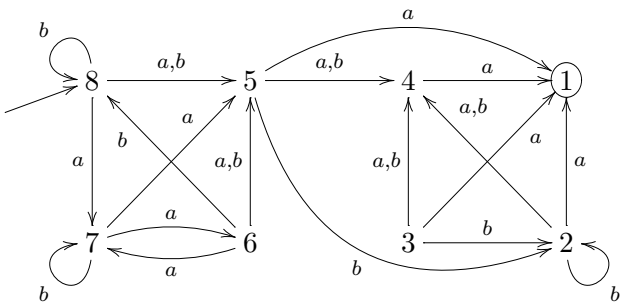
(b) Verify that the representatives are pairwise not language equivalent, i.e. for any pair u, v, $u \not\sim_L v$ holds (construct a table, where for each pair a separating word w is entered).

(c) Using (a) give the minimal automaton $\mathcal{A}_L$ for L.

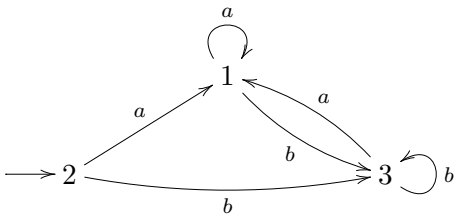
Exercise 2.4 (a) Apply the minimization algorithm (page 54) in order to minimize the following DFA (describe the steps as in Example 2.34):



(b) Construct the bisimulation quotient $\tilde{\mathcal{A}}$ of the following NFA $\mathcal{A}$:



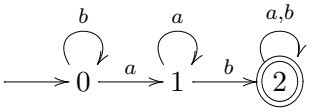
Exercise 2.5 Consider the DFA $\mathcal{A}$ without final states as given by the following transition graph:



Choose final states such that $\mathcal{A}$ is minimal, or show that this is impossible.

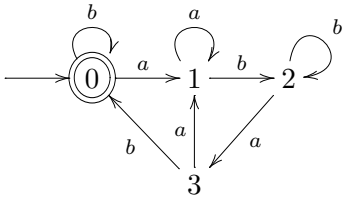
Exercise 2.6 * Show that there exists a family of regular languages languages $(L_n)_{n>0}$ such that L_n is recognized by n different (not isomorphic) minimal NFAs.
Hint: The editors do not know the solution yet.

Exercise 2.7 (a) determine the transition monoid of the following automaton according to the method presented in the lecture:



Mark with a circle the idempotent elements.

(b) Determine the transition monoid (with 13 elements) of the following automaton:



Give a non-trivial group.

Exercise 2.8 * Give a family of DFAs $\mathcal{A}_n = (Q_n, \Sigma, q_0^n, \delta_n, F_n)$, $n > 0$, with $|Q_n| = n$ over the alphabet $\Sigma = \{a, b, c\}$, such that the transition monoid of $\mathcal{A}_n$ has exactly n^n elements. Hence, every mapping $f : Q_n \to Q_n$ has to be implemented by a word in Σ^*.

Chapter 3

Tree Automata

In the last two chapters we dealt with two fundamental issues in the world of finite automata, namely establishing a connection to logical (or "logic-like") expressions and discovering algebraic ideas and methods to solve typical problems on finite automata. In this chapter we are going to generalize the model of a finite automaton itself, and use it for processing generalized inputs (i.e. not only finite words). This can be done in two different dimensions. One is to lift the constraint of finiteness and consider infinite inputs such as the run of a non-terminating procedure. In this case we are not anymore interested in reaching some final state but rather in ensuring stability, e.g. by passing again and again through a certain state. The other is to consider (finite) inputs that are branching rather than being linearly ordered (i.e. letters follow one another).

The first dimension goes beyond the scope of this course and is the core of the next semester's course about automata over infinite objects. The second dimension leads to the tree automata and tree languages, which are our main subject in this chapter.

Our primary motivation relies on the fact that trees model structured objects, which happen to be very frequent in computer science. Even the finite words we were dealing with in the previous chapters are a special case in this category; they represent linear trees (only one successor). The fundamental questions we are going to answer are accordingly a generalization of the ones we considered in the last two chapters, i.e.:

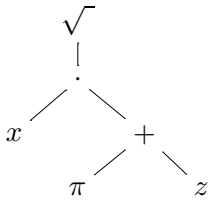
- Which techniques can we borrow from the area of finite words?
- How can we construct automata that describe the properties of trees?
- How can we define equivalent formalisms like regular expressions and logical formulas?
- Are there efficient algorithms for the fundamental problems (non-emptiness, equivalence and minimization of automata)?

Finally, we examine some applications in the area of describing structured objects such as context-free languages and XML-Documents.

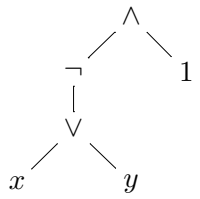
3.1 Trees and Tree Languages

Trees arise naturally as structured data, as terms, or as processes with branching behavior.

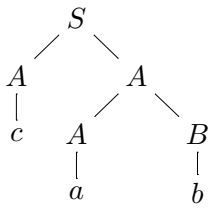
Example 3.1 The term $\sqrt{x \cdot (\pi + z)}$ corresponds to a tree:



A Boolean term $\neg(x \vee y) \wedge 1$ corresponds to a tree:



A context-free grammar (CF): $S \to AA,\ A \to a \mid c \mid AB,\ B \to b$ induces a derivation tree:



⊠

In the case of a context-free syntax of a programming language, a derivation tree captures the structure of a program.

Example 3.2 Consider the mini-language of WHILE-programs. An example program in this language would be the Euclidean algorithm for the greatest common divider:

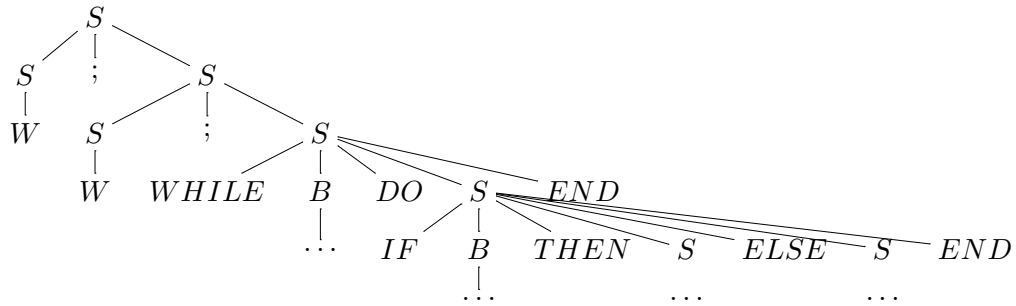
{Input in x_1, x_2}
```
x3 := x1; x4 := x2;
WHILE x3 <> x4 DO
```

```
  IF x3 < x4 THEN x4 := x4 - x3 ELSE x3:= x3 - x4 END
END
```
{the value of x_3 is $\gcd(x_1, x_2)$}

A formal definition of the WHILE-programs includes:

- a terminal alphabet: Σ_{keyboard}
- non-terminal symbols:
  S for ⟨program⟩
  W for ⟨assignmentofvalue⟩
  V for ⟨variable⟩
  D for ⟨decimal number⟩
  Z for ⟨positive digit⟩
  F for ⟨sequence of decimal digits⟩
  B for ⟨condition⟩
  O for ⟨comparison operator⟩
  E for ⟨arithmetic expression⟩
- a grammar
  $S \rightarrow W \mid S;S \mid$ `IF` B `THEN` S `ELSE` S `END` $\mid$
  `WHILE` B `DO` S `END`
  $W \rightarrow V := E$
  $V \rightarrow \mathrm{X}D$
  $D \rightarrow 0 \mid Z \mid ZF$
  $F \rightarrow 0 \mid Z \mid 0F \mid ZF$
  $Z \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
  $E \rightarrow D \mid V \mid (E+E) \mid (E-E) \mid (E \cdot E)$
  $B \rightarrow EOE$
  $O \rightarrow = \mid < \mid > \mid <> \mid =< \mid =>$

According to this definition a possible syntax tree of the gcd-program looks as follows:

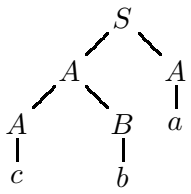


⊠

We know that it is very important to point out the hierarchical structure of a program even in its normal text representation. That is why programmers use indents to show how expressions are nested in each other. A similar

way to represent the structure of a derivation tree in normal text form is the use of special bracketing. For every non-terminal symbol X the brackets $(_X$ and $)_X$ are introduced. Then, the derivation tree t for w is coded by the word w_t that is produced by inserting bracketing symbols into w.

Example 3.3 The derivation tree on the left can be coded by the word on the right.



$$(_S\ (_A\ (_A\ c\ )_A\ (_B\ b\ )_B\ )_A\ (_A\ a\ )_A\ )_S$$

⊠

XML-documents are also coding trees in normal text form. Here, the bracketing notation is slightly modified: $\langle X \rangle$ is used for $(_X$ and $\langle /X \rangle$ is used for $)_X$.

Example 3.4 Consider the following XML-document:

```
<ProfileOfStudies>
  <faculty> Computer Science  </faculty>
  <semester> 3 </semester>
  <courses>
    <title> PROGRAMMING </title>
    <title> COMPUTER STRUCTURES </title>
    <title> DATA STRUCTURES </title>
  </courses>
</ProfileOfStudies>
```

The main difference between the previous examples and XML-documents is that in the latter the branching degree is generally unlimited. For example there is no restriction on the number of titles that the "courses"-bag may contain. This case is considered in further detail later. ⊠

Definition 3.5 A *ranked alphabet* is a non-empty finite set Σ of symbols, where every symbol a is assigned a finite set $\mathrm{rk}(a) \subseteq \mathbb{N}$ (the set of all *ranks* or *arities* of a).

$$\Sigma_i := \{\, a \in \Sigma \mid i \in \mathrm{rk}(a) \,\}$$

$$\Sigma = \Sigma_0 \cup \cdots \cup \Sigma_m \quad (\text{where } m = \max\{\mathrm{rk}(a) \mid a \in \Sigma\})$$

Note that the Σ_i need not be disjoint.

Example 3.6 The ranked alphabet for the term $\sqrt{x \cdot (\pi + z)}$ from Example 3.1 is $\Sigma = \{\sqrt{}, +, \cdot, x, \pi, z\}$ with $\Sigma_2 = \{+, \cdot\}$, $\Sigma_1 = \{\sqrt{}\}$ and $\Sigma_0 = \{x, \pi, z\}$. ⊠

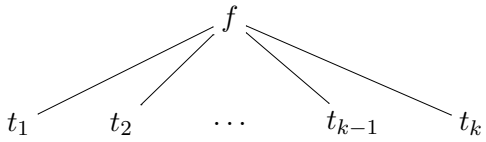
Example 3.7 The ranked alphabet for the term $\neg(x \vee y) \wedge 1$ from Example 3.1 is $\Sigma = \{0, 1, \neg, \wedge, \vee\}$ with $\Sigma_0 = \{0, 1\}$, $\Sigma_1 = \{\neg\}$, $\Sigma_2 = \{\wedge, \vee\}$. ⊠

Example 3.8 The ranked alphabet for the context-free grammar from Example 3.1 is $\Sigma = \{a, b, c, A, B, S\}$ with $\Sigma_0 = \{a, b, c\}$, $\Sigma_1 = \{A, B\}$, $\Sigma_2 = \{S, A\}$. ⊠

We first define trees in a term-like fashion.

Definition 3.9 Let $\Sigma = \Sigma_0 \cup \ldots \cup \Sigma_n$ be a ranked alphabet. The set T_Σ of *trees (or terms)* over Σ is defined inductively by:

- Each symbol $a \in \Sigma_0$ is a tree $a \in T_\Sigma$.
- For $f \in \Sigma_k$ and $t_1 \ldots t_k \in T_\Sigma$, $f(t_1 \ldots t_k)$ is a tree in T_Σ depicted by



A set $T \subseteq T_\Sigma$ is called a *tree-language*.

Remark 3.10 *This definition of trees allows*

- inductive definitions *of functions on the set of* Σ*-trees*
- proofs by induction *over the construction of* Σ*-trees.*

Example 3.11 Inductive definitions:

1. The *frontier word* yield(t) of a tree t (the labelling of the leaves of t read from left to right) is inductively defined by
   yield(a) = a for $a \in \Sigma_0$,
   yield($a(t_1, \ldots, t_i)$) = yield(t_1) $\cdots$ yield(t_i) for $a \in \Sigma_i$ ($i > 0$).
2. *Semantics of WHILE-programs*
   Let WHILE$_n$ be the set of WHILE-programs with the variables `x1`, ..., `xn`. Each such program computes a partial function $f : \mathbb{N}^n - \to \mathbb{N}^n$. Let F_n be the set of these functions. Then, the semantics of a program is a function σ : WHILE$_n \to F_n$ that is defined inductively over the structure of the program.

⊠

Example 3.12 An inductive proof for the arithmetical terms over $\{0, 1, +, \cdot\}$:

Hypothesis: The modulo-3 value of a term t with at most one 1, is either 0 or 1.

Inductive proof:

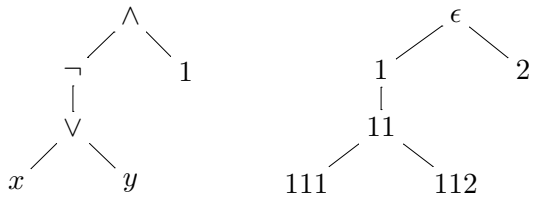
1. Induction start:
   Verify the hypothesis for the terms 0 and 1.

2. Induction step:
   Assume the hypothesis is fulfilled for t_1, t_2. Show that the hypothesis holds for $(t_1 + t_2)$ and $(t_1 \cdot t_2)$ (by simply distinguishing the possible cases).

⊠

An alternative, but equivalent, definition of trees distinguishes the tree-domain, which is the set of underlying nodes, from their values under an evaluation function.

Example 3.13 Below the tree associated to the term $\neg(x \vee y) \wedge 1$ and its underlying domain is shown. The evaluation function val satisfies $val(\epsilon) = \wedge$; $val(1) = \neg, \ldots$



⊠

Definition 3.14 Let $\Sigma = \Sigma_0 \cup \ldots \cup \Sigma_n$ be a ranked alphabet. A Σ-*valued tree* is a tuple $t = (\mathrm{dom}_t, \mathrm{val}_t)$ where dom_t is a prefix closed subset of $\{1, \ldots n\}^*$ satisfying $ui \in \mathrm{dom}_t \Rightarrow uj \in \mathrm{dom}_t$ for every $j < i$, and $\mathrm{val}_t : \mathrm{dom}_t \to \Sigma$ maps a node u with k successors to a symbol in Σ_k.

3.2 Tree Automata

3.2.1 Deterministic Tree Automata

Definition 3.15 A *deterministic tree automaton* (DTA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, F)$ where

- Q is a finite set of states,
- Σ is a ranked alphabet,

- $F \subseteq Q$ is a set of final states, and
- $\delta : \bigcup_{i \geq 0}(Q^i \times \Sigma_i) \to Q$ is a transition function.

By convention $Q^0 \times \Sigma_0 := \Sigma_0$. The *evaluation function* $\delta^* : T_\Sigma \to Q$ is defined inductively by $\delta^*(a) = \delta(a)$ for $a \in \Sigma_0$, and $\delta^*(f(t_1 \dots t_i)) := \delta(\delta^*(t_1), \dots, \delta^*(t_i), f)$ for $f \in \Sigma_i$ and $t_1, \dots, t_i \in T_\Sigma$.

A DTA $\mathcal{A}$ *accepts* a tree $t \in T_\Sigma$ if $\delta^*(t) \in F$. The tree-language *recognized* by $\mathcal{A}$ is $T(\mathcal{A}) = \{t \in T_\Sigma \mid \mathcal{A} \text{ accepts } t\}$.

Example 3.16 We define a tree automaton accepting all true Boolean expressions over $\Sigma = \{\wedge, \vee, \neg, 0, 1\}$. Let $Q = \{q_0, q_1\}$ be the state set, $F = \{q_1\}$ be the set of final states, and δ be defined by

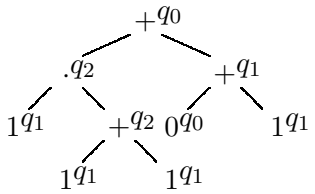
$$\begin{aligned} \delta(0) &= q_0 \\ \delta(1) &= q_1 \\ \delta(q_i, \neg) &= q_{|i-1|} \\ \delta(q_i, q_j, \wedge) &= q_{\min(i,j)} \\ \delta(q_i, q_j, \vee) &= q_{\max(i,j)} \end{aligned}$$

⊠

Definition 3.17 T is called *DTA-recognizable* or *regular*, if there exists a DTA $\mathcal{A}$ with $T = T(\mathcal{A})$.

Example 3.18 The set of the true logical expressions (the ones that can be evaluated to 1) is regular. ⊠

Example 3.19 (on arithmetic expressions) Let $\Sigma = \Sigma_0 \cup \Sigma_2$ with $\Sigma_0 = \{0, 1\}$, $\Sigma_2 = \{+, \cdot\}$. We construct a tree automaton that recognizes the tree language $T = \{t \in T_\Sigma \mid \text{ value of } t = 0 \pmod 3\}$. A tree that belongs to this language is shown in the picture below. On the same picture, the run of the automaton, which we intend to construct, is shown; namely we use 3 states to distinguish the 3 different cases in modulo-3 counting.



Formally: We use a state set $Q = \{q_0, q_1, q_2\}$, where q_i means that "the modulo-3 remainder of the evaluated term is i". Obviously, q_0 is the only final state. The transition function is defined explicitly for each possible value

of its arguments, using a parametric formulation for matters of compactness:

$$\delta(0) = q_0, \quad \delta(1) = q_1$$
$$\delta(q_i, q_j, +) = q_{i+j \bmod 3}, \quad \delta(q_i, q_j, \cdot) = q_{ij \bmod 3}$$

⊠

In the following we consider 3 further examples of languages, in order to get more familiar with constructing DTAs.

Example 3.20 Let $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$ with $\Sigma_0 = \{0, 1\}$, $\Sigma_1 = \{g\}$,and $\Sigma_2 = \{f\}$. We define tree automata for the languages

$$T_1 = \{t \in T_\Sigma \mid \text{ there is an odd number of } f\text{'s in } t\}$$
$$T_2 = \{t \in T_\Sigma \mid \text{ there are at least two } f\text{'s in } t\}$$
$$T_3 = \{t \in T_\Sigma \mid \text{ on the frontier of } t \text{ there is no 1 before a 0}\}$$

For T_1 let $Q = \{q_0, q_1\}$ and $F = \{q_1\}$. q_0 signals an even number of f's and q_1 an odd number of f's in the subtree below the current node. This induces the following transition function:

$$\delta(0) = \delta(1) = q_0$$
$$\delta(q_i, g) = q_i$$
$$\delta(q_i, q_j, f) = q_{(i+j+1) \bmod 2}$$

For T_2 let $Q = \{q_0, q_1, q_2\}$ (signaling respectively no, one and at least two occurrences of an f in the subtree below the current node) and $F = \{q_2\}$. This induces the following transition function:

$$\delta(0) = \delta(1) = q_0$$
$$\delta(q_i, g) = q_i$$
$$\delta(q_i, q_j, f) = q_{\min(i+j+1,2)}$$

For T_3 let $Q = \{q_0, q_1, q_{01}, q_{10}\}$ and $F = \{q_0, q_1, q_{01}\}$. A node of the tree is labeled with state q_{01} if all 0's are before the first 1 in the frontier word of the subtree below this node. q_{10} signals that there is a 1 somewhere left of a 0 on the frontier, and q_0, q_1 signal that only 0's, respectively 1's, appear on the frontier of the corresponding subtrees. The meaning of the states is implemented in the following transition function: $\delta(0) = q_0$, $\delta(1) = q_1$, $\delta(q, g) = q$ for all $q \in Q$,

$$\begin{aligned}
\delta(q_0, q_0, f) &= q_0 & \qquad \delta(q_{01}, q_0, f) &= q_{10} \\
\delta(q_1, q_1, f) &= q_1 & \qquad \delta(q_{01}, q_1, f) &= q_{01} \\
\delta(q_0, q_1, f) &= q_{01} & \qquad \delta(q_0, q_{01}, f) &= q_{01} \\
\delta(q_1, q_0, f) &= q_{10} & \qquad \delta(q_1, q_{01}, f) &= q_{10},
\end{aligned}$$

and $\delta(p, q, f) = q_{10}$ for all other combinations of states $p, q \in Q$. ⊠

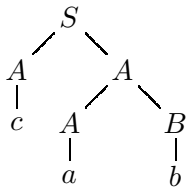
Tree languages can also be defined by the rules of context-free grammars. How do these languages relate to the regular languages we illustrated so far?

Definition 3.21 If $G = (N, T, P, S)$ is a context-free grammar, then let $T(G)$ *be the set of all derivation trees generated by* G (with frontier words consisting of T-symbols).

For this definition we assume that ϵ is not used in the grammar.

Example 3.22 The derivation tree on the right belongs to the language defined by the grammar on the left.

$S \to AA$
$A \to c \mid a \mid AB$
$B \to b$



⊠

The following theorem shows that this kind of tree languages forms a proper subset of the regular ones.

Theorem 3.23 *a) If* G *is a context-free grammar, then* $T(G)$ *is regular.*

b) Not every regular tree language can be represented as a language $T(G)$ *for a context-free grammar* G*.*

Proof: (part a) Given a grammar $G = (N, T, P, S)$ we construct a DTA with state set $N \cup T \cup \{*\}$ and final state S. The tree automaton should be able to copy the tree symbols into its states, as long as the tree is built up (in a bottom-up fashion) according to the rules of G. Hence, the input alphabet consists of $\Sigma_0 = T$ and $\Sigma_i = \{X \in N \mid \text{there exists a rule } X \to \alpha \text{ with } |\alpha| = i\}$ (for $i > 0$). The transition function is defined as follows:

$$\begin{aligned} \delta(a) &= a \quad \text{for } a \in \Sigma_0 \\ \delta(\sigma_1, \ldots, \sigma_i, X) &= X \quad \text{for } X \to \sigma_1 \ldots \sigma_i \in P \text{ for } i > 0, X \in \Sigma_i \\ \delta(\sigma_1, \ldots, \sigma_i, X) &= * \quad \textit{otherwise} \quad (\text{“}\textit{error}'' - \textit{state}) \\ \delta(\sigma_1, \ldots, \sigma_i, *) &= * \end{aligned}$$

□

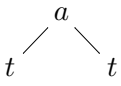
Proof: [part b] It suffices to show that there exists a tree language that cannot be represented as a language of derivation trees. For this reason we consider the set T containing all trees with the following property:

(∗) On every path there are at most two A's and there exists at least one path with two A's.

Looking at this property, on the one hand we observe that T is a regular language because $(*)$ is a conjunction of two properties that can be easily recognized by a DTA. On the other hand we also observe that in case a symbol A occurs twice in a path of a derivation tree t (as a result of a sequence of grammar rules), then it is possible to produce trees with an arbitrary number of A's on the same path (using the same sequence of rules again and again). Consequently, T is recognizable by a DTA but cannot be of the form $T(G)$ for a grammar G. □

Finally, we illustrate an example of a language that belongs to a higher class in the sense that it is not recognizable by a DTA.

Example 3.24 Let $\Sigma = \Sigma_0 \cup \Sigma_2$ with $\Sigma_0 = \{b\}$, $\Sigma_2 = \{a\}$. We show that the language $T := \{\, a(t,t) \mid t \in T_\Sigma \,\}$ is not recognizable. By $a(t,t)$ we mean what is shown in the picture below, where t is a (sub)tree.



When reading the root of a tree, the only information a DTA can get from the two subtrees below, is the states it reaches at the top of each subtree. Hence, the idea to prove this assumption (as well as other similar ones) is to show that the finitely many states of the DTA do not suffice to compare the infinitely many different subtrees.

Proof: Towards a contradiction, we assume that the DTA $\mathcal{A} = (Q, \Sigma, \delta, F)$ recognizes T. Since we do not have a restriction on the length of the tree, there are obviously infinitely many trees over Σ. But because Q is finite, there exist two different trees t, t' with $\delta^*(t) = \delta^*(t')$ and hence also $\delta^*(a(t,t)) = \delta^*(a(t,t'))$ (at this point the automaton cannot distinguish between t and t' anymore, since both lead it to the same state). Now, since $\delta^*(a(t,t)) \in F$ also $\delta^*(a(t,t')) \in F$, which means that $a(t,t') \in T$. Contradiction. ⊠

In case we want to generate a similar repetition of states, this time only on the same path, we use another method that relies on the pumping lemma. To achieve this we first have to formally describe how we can concatenate trees to get longer and longer paths. In the case of finite words no special definition was required, because it is obvious that we append a word at the (only) end of another one. In the case of finite trees we do not have one specific end, but instead a number of leaves; hence we have to always declare at which one of them we append a new tree when concatenating.

Definition 3.25 Let Σ be a ranked alphabet and c a new symbol of rank 0. A *special tree* over Σ is a tree over $\Sigma \cup \{c\}$, where c occurs exactly once (as a leaf). We call S_Σ the set of all special trees over Σ. For $t \in T_\Sigma$ or

$t \in S_\Sigma$ and $s \in S_\Sigma$ let $t \circ s$ be the tree, that results from s by replacing c by t.

Theorem 3.26 (Pumping Lemma) *Let T be a regular tree language. Then, for an appropriate n (namely the number of states of a tree automaton recognizing T) the following holds: If the height of $t \in T$ is $\geq n$, then there are two nodes on a path of t that can split t in three pieces t_0, s_1, s_2, such that an iteration (arbitrarily often) of $s_1 \neq c$ results in trees that also belong to T. In short notation:*

$$t = t_0 \circ s_1 \circ s_2 \text{ with } t_0 \circ (s_1)^i \circ s_2 \in T \text{ for every } i \geq 0 \ .$$

Proof: Assume T is recognized by some tree automaton with n states. If the height of t is $\geq n$, then on some path of the run tree of t there have to be two positions, where a state repetition occurs. Define the split $t = t_0 \circ s_1 \circ s_2$ on these two positions. □

Now we can apply this theorem to show the non-recognizability for further examples of tree languages, where an equality not between subtrees, but between the lengths of all paths in the whole tree (complete trees) is required.

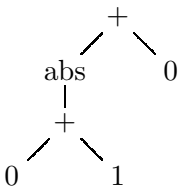
Example 3.27 Let $\Sigma = \Sigma_0 \cup \Sigma_2$ with $\Sigma_0 = \{e\}$, $\Sigma_2 = \{a, b\}$. The language $T = \{t \in T_\Sigma \mid t \text{ is complete } \}$ is not regular. (A tree t is complete iff all its paths have the same length).

Proof: Towards a contradiction, assume that T is regular (i.e. recognized by some DTA). Then, consider a split according to the Pumping Lemma. By repeating the middle part twice, we get a tree that is accepted, although it is not complete. Contradiction. ⊠

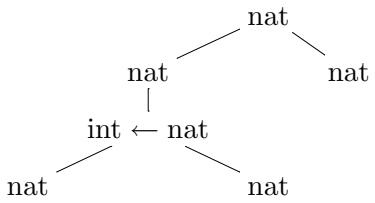
By the previous two examples we realized that DTAs are not powerful enough to solve some fundamental questions over trees. There are still some other interesting problems reducible to recognizing tree properties that can be characterized by finitely many states. The problem of type checking that is embedded in every modern programming language represents such an issue.

Example 3.28 Consider the type information of functions below on the left. A compiler has to check whether a term like the one represented by the tree on the right, is adhering to these rules.

$0 \to \text{nat}$
$1 \to \text{nat}$
$+ : \text{nat} \times \text{nat} \to \text{nat}$
$+ : \text{int} \times \text{int} \to \text{int}$
$\text{abs} : \text{int} \to \text{nat}$
$\text{nat} \leq \text{int}$



To answer this purpose, a DTA is used that has a state for each one of the types (finitely many). The DTA evaluates the term using the type rules as transitions and assumes an error state in case of a type mismatch. The run tree of the automaton is shown below:



Because the abs-function is not defined on nat, the last rule (namely that nat is a subtype of int) has to be applied first. This corresponds to the implicit type casting in programming languages and is implemented by ϵ-transitions in the tree automaton. ⊠

3.2.2 Nondeterministic Tree Automata

Definition 3.29 A *nondeterministic tree automaton* (NTA) is a tuple $\mathcal{A} = (Q, \Sigma, \Delta, F)$ where

- Q is a finite state set,
- $\Sigma = \Sigma_0 \cup \ldots \cup \Sigma_m$ is a ranked alphabet,
- $F \subseteq Q$ is a set of final states, and
- $\Delta \subseteq \bigcup_{i=0}^{m}(Q^i \times \Sigma_i \times Q)$ is a transition relation.

By convention $Q^0 \times \Sigma_0 \times Q = \Sigma_0 \times Q$. A *run* of $\mathcal{A}$ on a tree $t \in T_\Sigma$ is a mapping $\rho : \mathrm{dom}_t \to Q$ such that

- for leaf nodes $u \in \mathrm{dom}_t$ we have $(\mathrm{val}_t(u), \rho(u)) \in \Delta$, and
- for nodes u with successors $u1, \ldots, ui$ we have

$$(\rho(u1), \ldots, \rho(ui), \mathrm{val}_t(u), \rho(u)) \in \Delta.$$

An NTA $\mathcal{A}$ *accepts* a tree $t \in T_\Sigma$ if for some run ρ of $\mathcal{A}$ on t $\rho(\epsilon) \in F$. The tree-language *recognized* by $\mathcal{A}$ is $T(\mathcal{A}) = \{t \in T_\Sigma \mid \mathcal{A} \text{ accepts } t\}$.

Example 3.30 Let $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$ with $\Sigma_0 = \{0, 1\}$, $\Sigma_1 = \{g\}$, and $\Sigma_2 = \{f\}$. Let

$$T = \{t \in T_\Sigma \mid \text{some path through } t \text{ has even length }\}.$$

We construct an NTA that guesses a path through t, counts modulo-2 the length of this path, and labels all other nodes with a dummy symbol.

Let $Q = \{q_0, q_1, *\}$ and $F = \{q_0\}$. The transition relation implements the idea above.

$$\begin{aligned}\Delta := \{\ & (0,*),(1,*),(0,q_0),(1,q_0),(q_0,g,q_1),(q_1,g,q_0),\\ & (*,g,*),(q_0,*,f,q_1),(q_1,*,f,q_0),(*,q_0,f,q_1),(*,q_1,f,q_0)\\ & (q_0,q_0,f,*),(q_1,q_1,f,*),(*,*,f,*)\ \}\end{aligned}$$

⊠

In the case of automata over finite words, nondeterminism does not offer any additional expressive power. The following theorem shows that the same happens in the case of tree automata.

Theorem 3.31 *For each NTA one can construct an equivalent DTA.*

Proof: For a given NTA $\mathcal{A} = (Q, \Sigma, \Delta, F)$ we construct a DTA $\mathcal{A}' = (Q', \Sigma, \delta, F')$ where

- $Q' = 2^Q$,
- $F' = \{P \subseteq Q \mid F \cap P \neq \emptyset\}$,
- $\delta(a) = \{q \in Q \mid (a, q) \in \Delta\}$ for $a \in \Sigma_0$, and
- $\delta(P_1, \ldots, P_i, f) = \{q \in Q \mid \exists q_1 \in P_1 \ldots \exists q_i \in P_i :\ (q_1, \ldots, q_i, f, q) \in \Delta\}$ for $f \in \Sigma_i$.

It remains to show that for every $t \in T_\Sigma$, $\delta^*(t)$ is the set of those states $q \in Q$, for which a run of $\mathcal{A}$ on t exists reaching q at the root of t. By induction on the construction of t, one can easily see that:

$$q \text{ reachable by } \mathcal{A} \text{ on the root of } t \text{ iff } q \in \delta^*(t)\ .$$

Then:

$\mathcal{A}'$ accepts t

iff $\delta^*(t) \in F'$

iff $\exists q \in \delta^*(t) \cap F$

iff $\exists q \exists$ run ρ of $\mathcal{A}$ on t with $\rho(\epsilon) = q$ and $q \in F$

iff $\mathcal{A}$ accepts t.

□

The closure properties also hold, as they do for NFAs.

Theorem 3.32 *The class of regular tree languages is closed under the Boolean operations ($\cup$, $\cap$, complement) and under projection.*

This can be easily shown by copying the following well-known proofs:

- intersection and union by product automaton,
- complement as with normal NFAs,
- for the projection $f : \Sigma \to \Gamma$ (assumed as rank preserving) change from a transition (q_1, q_2, a, q) to $(q_1, q_2, f(a), q)$.

A natural thing to ask is whether there are any results that *cannot* be directly transferred from word automata to tree automata. Unfortunately, the answer to this question is positive. Until now, we were using tree automata that process trees from the leaves to the root (bottom-up). We can also introduce automata that evaluate their input in the opposite direction (top-down) and show that in this case a reduction from nondeterminism to determinism is not possible.

Definition 3.33 A nondeterministic top-down tree automaton ($\downarrow$NTA) over $\Sigma = \Sigma_0 \cup \cdots \cup \Sigma_m$ is of the form $\mathcal{A} = (Q, \Sigma, Q_0, \Delta)$ with

- finite state set Q,
- $Q_0 \subseteq Q$ as a set of initial states,
- $\Delta \subseteq \bigcup_{i=0}^{m}(Q \times \Sigma_i \times Q^i)$
  (The pairs in $\Delta \cap (Q \times \Sigma_0)$ are also called "final combinations").

An accepting run of $\mathcal{A}$ on t is a mapping $\rho : \mathrm{dom}_t \to Q$ with

- $\rho(\epsilon) \in Q_0$,
- for each node u of t with $i > 0$ successors, we have $\big(\rho(u), \mathrm{val}_t(u), \rho(u1), \ldots, \rho(ui)\big) \in \Delta$,
- for each leaf u of t, $\big(\rho(u), \mathrm{val}_t(u)\big)$ is a final combination.

$\mathcal{A}$ accepts t if and only if there exists a run on t.

Remark 3.34 *Normal NTAs can be directly transformed to $\downarrow$NTAs and vice versa.*

A $\downarrow$DTA is a special $\downarrow$NTA that has a singleton set $\{q_0\}$ as an initial state set Q_0, transition functions $\delta_1, \ldots, \delta_m$ with $\delta_i : Q \times \Sigma_i \to Q^i$ instead of Δ and final combinations exactly like a $\downarrow$NTA. A tree t is accepted by the unambiguous run on t. This also means that a state assumed on a position of the run tree can only depend on the nodes of the path back to the root. The example language considered below shows that $\downarrow$DTA can recognize less than $\downarrow$NTA (in contrast to what happens in the case of bottom-up tree automata).

Remark 3.35 *The doubleton set*

$$T := \left\{ \begin{array}{ccc} & f & \\ a & & b \end{array}, \begin{array}{ccc} & f & \\ b & & a \end{array} \right\}$$

is recognizable by a ↓NTA (and an NTA as well), but not by a ↓DTA.

Proof (↓NTA recognizability): The following ↓NTA recognizes T:

$$\begin{aligned} Q &:= \{q_0, q_a, q_b\} \\ Q_0 &:= \{q_0\} \\ \Delta &:= \{(q_0, f, q_a, q_b), (q_0, f, q_b, q_a), (q_a, a), (q_b, b)\} \end{aligned}$$

□

Proof (↓DTA non-recognizability): Towards a contradiction we assume that the ↓DTA $\mathcal{A}$ recognizes T. Then, $\mathcal{A}$ accepts

$$\begin{array}{ccc} & f & \\ a & & b \end{array}, \text{ with the unambiguous run } \begin{array}{ccc} & f^{q_0} & \\ a^{q_1} & & b^{q_2} \end{array}$$

This means that $\delta(q_0, f) = (q_1, q_2)$ and (q_1, a) and (q_2, b) are final combinations. $\mathcal{A}$ likewise accepts

$$\begin{array}{ccc} & f & \\ b & & a \end{array} \text{ with the run } \begin{array}{ccc} & f^{q_0} & \\ b^{q_1} & & a^{q_2} \end{array}$$

Consequently, (q_1, b) and (q_2, a) are also final combinations. Hence $\mathcal{A}$ also accepts $f(a, a)$. Contradiction. □

3.2.3 Emptiness, Congruences and Minimization

Theorem 3.36 *The emptiness problem for nondeterministic tree automata is decidable in polynomial time.*

Proof: Given an NTA $\mathcal{A}$, the main idea for the proof is to consider the *set $E_\mathcal{A}$ of all states that are reachable on the roots of input trees* and note that $T(\mathcal{A}) \neq \emptyset$ iff $E_\mathcal{A}$ contains a final state. Then, $E_\mathcal{A}$ is computed in the following way:

- For $h = 0, 1, 2, \ldots$ determine the states that are reachable on trees of height h, until no new states arise.
- Now verify whether the computed set contains a final state.

Emptiness algorithm:

Input: $\mathcal{A} = (Q, \Sigma, \Delta, F)$
$E_0 := \{q \in Q \mid (a, q) \in \Delta \text{ for some } a \in \Sigma_0\}$
$n := 0$
`Repeat`

$$n := n + 1$$
$$E_n := E_{n-1} \cup \{q \in Q \mid \text{there exists } q_1, \ldots, q_i \in E_{n-1}, a \in \Sigma_i \text{ with } (q_1, \ldots, q_i, a, q) \in \Delta\}$$

`Until` $E_n = E_{n-1}$
`If` $E_n \cap F = \emptyset$ output("empty") `else` output("non-empty")

Note that since Q is finite, the algorithm terminates. The effort in terms of time complexity is $O(|Q| \cdot (|Q| \cdot |\Delta|))$.

Correctness: It has to be shown that $E_n = E_{\mathcal{A}}$ for the smallest n with $E_n = E_{n-1}$.

The inclusion $E_n \subseteq E_{\mathcal{A}}$ can be easily shown by induction over n. To prove the opposite inclusion $E_{\mathcal{A}} \subseteq E_n$, we have to show that if q is reachable on a tree of height i, then $q \in E_i$ (induction). Now $\bigcup_{i \in \mathbb{N}} E_i = E_n$ holds and hence, if $q \in E_{\mathcal{A}}$, then also $q \in E_n$. □

The Nerode congruence for trees is defined using the notion of special trees introduced previously. Hereby, when appending a tree $t \in T_\Sigma$ to a special tree $s \in S_\Sigma$, to construct $t \circ s$, s is called the "context" of t in $t \circ s$.

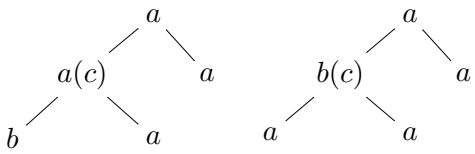
Definition 3.37 (Nerode congruence) Define $\sim_T \subseteq T_\Sigma \times T_\Sigma$ for $T \subseteq T_\Sigma$ by:

$$t_1 \sim_T t_2 \Leftrightarrow \text{for each special tree } s \in S_\Sigma: \; t_1 \circ s \in T \Leftrightarrow t_2 \circ s \in T$$

Example 3.38 Let $\Sigma = \Sigma_0 \cup \Sigma_2$ with $\Sigma_0 = \Sigma_2 = \{a, b\}$ and the language T with

$$T = \{\, t \in T_\Sigma \mid \text{on every branch of } t \text{ there are two consecutive occurrences of an } a\}.$$

Consider the trees $t = a(b, a)$, $t' = b(a, a)$ and $t'' = a(b, b)$. Then, $t \not\sim_T t'$ and $t \sim_T t''$. The first can be shown by choosing the special tree $s = a(c, a)$. Then $t \circ s \in T$ but $t' \circ s \notin T$ as it can be seen below (the letter in brackets is the one that was substituted for the sake of the concatenation).



⊠

Definition 3.39 For a given DTA $\mathcal{A} = (Q, \Sigma, \delta, F)$ (w.l.o.g. assume all states are reachable) define $p_1 \sim_{\mathcal{A}} p_2$ by: If $\delta^*(t_1) = p_1$ and $\delta^*(t_2) = p_2$ then for every special tree s

$$\delta^*(t_1 \circ s) \in F \quad \Leftrightarrow \quad \delta^*(t_2 \circ s) \in F$$

Remark 3.40 *In the special case that $\Sigma = \{c\} \cup \Sigma_1$ the relations $\sim_T$ and $\sim_{\mathcal{A}}$ are identical to the corresponding congruences for the case of finite words.*

Remark 3.41 *a) $\sim_T$ and $\sim_{\mathcal{A}}$ are right congruences.*

b) If $p \sim_{\mathcal{A}} p'$ with $\delta^*(t) = p$ and $\delta^*(t') = p'$, then $t \sim_T t'$.

c) index of $\sim_T \leq$ index of $\sim_{\mathcal{A}}$.

d) If T is recognizable, then $\sim_T$ has a finite index.

e) If $\sim_T$ has a finite index, then T is recognizable; moreover, the $\sim_T$-classes form the state space of the (up to isomorphism unambiguous) minimal DTA $\mathcal{A}_T$ that recognizes T.

The proofs are analogous to the ones in the word case and are therefore skipped. Only for the last remark we give the formal definition of the minimal DTA as well as the method for computing it.

Definition 3.42 For a given $\sim_T$ with finite index we define the DTA $\mathcal{A}_T = (Q_T, \Sigma, \delta_T, F_T)$ with

- $Q_T = T_\Sigma/_{\sim_T}$ (the set of equivalence classes $[t]$)
- $\delta_T(a) = [a]$ for $a \in \Sigma_0$,
- $\delta_T([t_1], \ldots, [t_i], a) = [a(t_1, \ldots, t_i)]$ for $a \in \Sigma_i, i > 0$,
- $F_T = \{\, [t] \mid t \in T \,\}$

Note that δ_T is well-defined, since $t_1 \sim_T t_1', \ldots, t_i \sim_T t_i'$ implies that $a(t_1, \ldots, t_i) \sim_T a(t_1', \ldots, t_i')$.

To compute this minimal DTA, as in the case of finite words, starting from an arbitrary DTA we first eliminate any non-reachable states and then we merge the equivalent states (i.e. two states q, q' with $q \sim_{\mathcal{A}} q'$) into one.

Only the way we identify two equivalent states (given two other states that have been already found to be equivalent) is a little more complicated than in the case of finite words. This is because, as stated before, in trees we have to explicitly state at which leaf we concatenate a tree with another. Formally: $q \sim_{\mathcal{A}} q', a \in \Sigma_i \Rightarrow$
$\delta(q_1, \ldots, q_{j-1}, q, q_{j+1}, \ldots, q_i, a) \sim_{\mathcal{A}} \delta(q_1, \ldots, q_{j-1}, q', q_{j+1}, \ldots, q_i, a)$
for all $j \in \{1, \ldots, i\}$ and all $q_1, \ldots, q_{j-1}, q_{j+1}, \ldots, q_i \in Q$.

Translated into an algorithm, the procedure takes the following steps:

1. Mark all pairs (q, q'), such that exactly one state belongs to F.
2. While a new pair was marked in the last step: mark all yet non-marked pairs (q, q'), for which there exists an $a \in \Sigma_i$ and states $q_1, \ldots, q_{j-1}, q_{j+1}, \ldots, q_i$, such that the pair

   $$(\delta(q_1, \ldots, q_{j-1}, q, q_{j+1}, \ldots, q_i, a), \delta(q_1, \ldots, q_{j-1}, q', q_{j+1}, \ldots, q_i, a))$$

   is already marked.

As in the case of conventional DFAs, this algorithm can also be applied to test the equivalence between two given tree automata by minimizing them and comparing them with respect to isomorphism.

3.3 Logic-Oriented Formalisms over Trees

In this section we look at the two kinds of logic-oriented formalisms that we studied in the first chapter about words, namely regular expressions and MSO-logic. We only focus on issues that have to be extended or refined to meet the special characteristics of trees, skipping what can be directly transferred from the area of finite words.

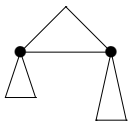
3.3.1 Regular Expressions

Regular expressions are built up using three main operations, namely union, concatenation and star-operation. For the first one, no special explanation needs to be given. For the other two some further consideration is necessary. The way we defined concatenation in the previous section (specifying exactly one leaf where a special tree is appended to the root of another tree) makes it rather weak to express all languages that are recognizable by tree automata.

Definition 3.43 $T_1 \cdot^c T_2$ is the set of all trees that are formed from some $t \in T_1$, by replacing each c-labeled leaf by some tree from T_2.

Note that from now on we are concatenating trees in a top-down manner, in contrast to what we did in Section 3.2. This is a natural thing to do at this point, since we now append more than one tree to an existing one. Also note

that these trees ($\in T_2$) can be different. If we represent a tree by a triangle and mark the c-labeled leaves by a dot, a concatenated tree belonging to $T_1 \cdot^c T_2$ may look like

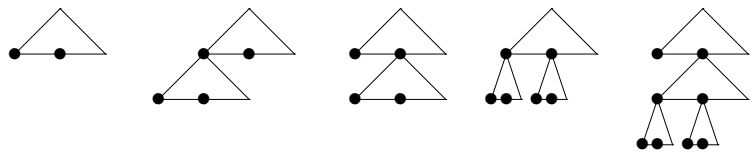


where the upper tree belongs to T_1 and the lower ones to T_2.

Since the star-operator induces a special kind of concatenation it has to be defined in a similar way.

Definition 3.44 $T^{*_c} := T_0 \cup T_1 \cup T_2 \cup \cdots$ with $T_0 := \{c\}$, $T_{i+1} := T_i \cdot^c (T \cup \{c\})$

Example 3.45 Consider the singleton tree language $T := \left\{ \begin{array}{c} f \\ / | \backslash \\ c\ c\ a \end{array} \right\}$. Some of the trees belonging to T^{*_c} are schematically shown below: ⊠

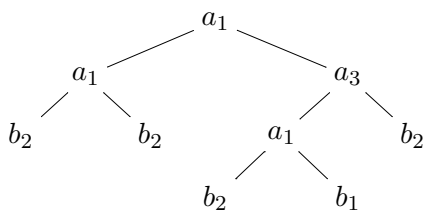
- 

Definition 3.46 (Regular Expressions) $T \subseteq T_\Sigma$ is *definable by a regular expression*, if there exist an $n \geq 0$ and 0-ary symbols $c_1, \ldots, c_n$, such that T is constructed from finitely many trees over $\Sigma \cup \{c_1, \ldots, c_n\}$ by finitely often applying the operators $\cup$, $\cdot^{c_i}$ and $^{*_{c_i}}$ $(i \in \{1, \ldots, n\})$.

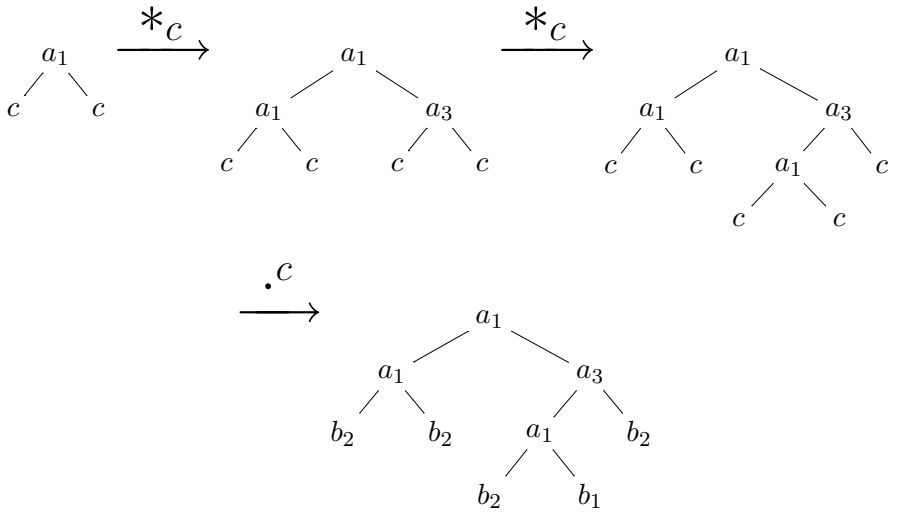
Example 3.47 Let $\Sigma = \Sigma_0 \cup \Sigma_2$ with $\Sigma_2 := \{a_1, \ldots, a_m\}$, $\Sigma_0 := \{b_1, \ldots, b_k\}$. T_Σ is definable by a regular expression:

$$T_\Sigma := \left(\begin{array}{c} a_1 \\ / \backslash \\ c\ \ c \end{array} + \cdots + \begin{array}{c} a_m \\ / \backslash \\ c\ \ c \end{array} \right)^{*_c} \cdot^c (b_1 + \cdots + b_k)$$

Take for example the following tree that obviously belongs to T_Σ:



This tree can be generated according to the above mentioned regular expression by twice applying the star-operator ($*_c$) and once the concatenation ($\cdot^c$):



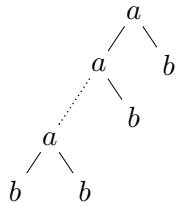
⊠

Example 3.48 Let $\Sigma_2 = \{a\}$, $\Sigma_0 = \{b\}$.

1. If T is the set of all trees with height bigger than 1, then

$$T = \left(\begin{array}{ccccc} & a & & a & \\ a & & c \;+\; c & & a \\ c & c & & c & c \end{array} \right) \cdot^c T_\Sigma$$

2. If T is the set of all "left-combs", then



$$T = \left(\begin{array}{ccc} & a & \\ c & & b \end{array} \right)^{*_c} \cdot^c b$$

3. If T is the set of all trees whose paths are all of even length, then

$$T = \left(\begin{array}{ccccc} & & a & & \\ & a & & a & \\ c & & c & c & c \end{array} \right)^{*_c} \cdot^c b$$

4. If T is the set of all "left-combs" of even length, then

$$T = \left(\begin{array}{ccc} a & & a \\ c_1 \;\; b & \cdot^{c_1} & c_2 \;\; b \end{array} \right)^{*_{c_2}} \cdot^{c_2} b$$

⊠

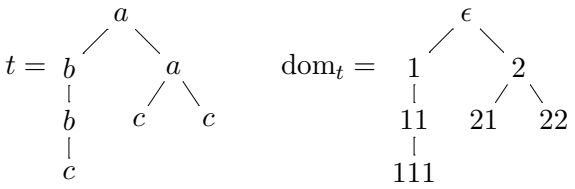
Remark 3.49 *Note that c is considered as a non-terminal symbol that can appear in the construction but not in the trees of the language.*

Theorem 3.50 (Kleene-Theorem for Tree Languages) *A tree language $T \subseteq T_\Sigma$ is recognizable by a tree automaton if and only if it is definable by a regular expression.*

Proof ($\Leftarrow$): We suppose that a we are given a language, defined by a regular expression. By definition the expression contains finitely many trees. Hence, we can proceed by induction on the construction of regular expressions.

Induction start: By the definition of regular expressions, we start with languages containing finitely many trees. These can be built up from singleton languages by finitely often applying the union operator. And since we are going to handle this operator in the induction step anyway, it suffices now to construct an automaton recognizing a singleton language, i.e. exactly one tree.

Example 3.51 Given the following tree



we construct an NTA over the state set dom_t, such that in a successful run state i is assumed on reading node i. The transitions are:
$(c, 111), (c, 21), (c, 22),$
$(111, b, 11), (21, 22, a, 2), (11, b, 1), (1, 2, a, \epsilon)$. ⊠

Generally given a tree t, the automaton recognizing it is $\mathcal{A} = (\mathrm{dom}_t, \Sigma, \Delta, \{\epsilon\})$ with $\Delta := \{\, (u1, \ldots, ui, \mathrm{val}_t(u), u) \mid u \in \mathrm{dom}_t \,\}$.

Induction step: Given automata $\mathcal{A}_i = (Q_i, \Sigma, \Delta_i, F_i)$ for $i = 1, 2$ with $Q_1 \cap Q_2 = \emptyset$,

- $T(\mathcal{A}_1) \cup T(\mathcal{A}_2)$ is recognized by the NTA $\mathcal{A} := (Q_1 \cup Q_2, \Sigma, \Delta_1 \cup \Delta_2, F_1 \cup F_2)$
- $T(\mathcal{A}_1) \cdot^c T(\mathcal{A}_2)$ is recognized by the NTA $\mathcal{A} := (Q_1 \cup Q_2, \Sigma, \Delta, F_1)$ with

$$\begin{aligned}\Delta = (\Delta_1 \setminus \{\, (c, q) \mid q \in Q_1 \,\}) \cup \Delta_2 \\ \cup \{\; (q_1^{(2)}, \ldots, q_i^{(2)}, a, q^{(1)}) \mid \exists q' \in F_2 : \\ (q_1^{(2)}, \ldots, q_i^{(2)}, a, q') \in \Delta_2 \text{ and } (c, q^{(1)}) \in \Delta_1 \,\}\end{aligned}$$

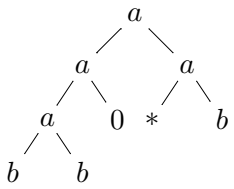
- $T(\mathcal{A}_1)^{+_c}$ is recognized by the NTA $\mathcal{A} := (Q_1, \Sigma, \Delta, F_1)$ with

$$\Delta := \Delta_1 \cup \{\ (q_1, \ldots, q_i, a, q) \mid \exists q' \in F_1 : (q_1, \ldots, q_i, a, q') \in \Delta_1 \text{ and } (c, q) \in \Delta_1\ \}$$

□

Proof ($(\Rightarrow)$): We suppose that T is recognized by the NTA $\mathcal{A} = (Q, \Sigma, \Delta, F)$. First of all we have to choose some distinguished 0-ary symbols to enable concatenation ("c-symbols"). For this purpose we misuse the state set of the automaton and extend Σ to $\Sigma \cup Q$, where the symbols from Q have arity 0.

Example 3.52 Let $\Sigma = \Sigma_0 \cup \Sigma_2$ with $\Sigma_0 = \{b\}$, $\Sigma_2 = \{a\}$ and $Q = \{*, 0, 1\}$. An example tree over this extended alphabet would be



⊠

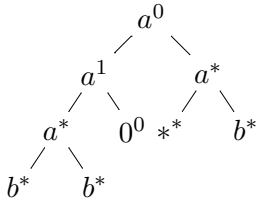
After having extended the alphabet we construct $\mathcal{A}'$ from $\mathcal{A}$ by adding the transitions (p, p) for every $p \in Q$, to allow the automaton to read the new 0-ary symbols on the leaves and synchronously assume the corresponding state (having the same name).

Next we define a set $T(R, S, q)$ for $R, S \subseteq Q$ that contains all trees over $\Sigma \cup Q$, on which there exists a run of $\mathcal{A}'$,

- whose leaves that are labeled by Q-symbols in the input tree, are labeled with states from R
- whose root is labeled by the state q and
- whose remaining nodes are occupied by states from S

We call the elements of R the "source states", q the "target state" and the elements of S the "intermediate states".

Example 3.53



$\in T(\{0, *\}, \{1, *\}, 0)$

⊠

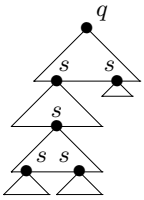
By the definition of $T(R, S, q)$ the following holds:

$$T(\mathcal{A}) = \bigcup_{q \in F} T(\emptyset, Q, q) \ .$$

Hence, it suffices to show that all sets $T(R, S, q)$ are regular, which we prove by induction over the size of S.

Induction start: $S = \emptyset$. In this case $T(R, S, q)$ contains only trees of length 0 or 1. These are finitely many, so $T(R, S, q)$ is regular.

Induction step: $|S| > 0$: Let $S = S_0 \cup \{s\}$ (with $s \notin S_0$), where for S_0 the induction hypothesis holds. A tree $t \in T(R, S, q)$ iff there exists a run on t with source states in R, q as target state and intermediate states in $S_0 \cup \{s\}$. If we decompose t by splitting it on the nodes where s occurs



then by induction hypothesis we can build a regular expression for the decomposed parts, since all their intermediate nodes are in S_0. Using the following equation we can then construct a regular expression for $T(R, S, q)$.

$T(R, S_0 \cup \{s\}, q) \ =$
$T(R, S_0, q) \cup T(R \cup \{s\}, S_0, q) \cdot^s \big(T(R \cup \{s\}, S_0, s)\big)^{*_s} \cdot^s T(R, S_0, s).$ □

3.3.2 MSO-Logic

In this subsection we introduce a way to describe the properties of trees using the Monadic Second-Order Logic. Furthermore, as we did in the case of finite words, we compare the expressive power of this logic to the tree automata, this time without giving a proof.

Definition 3.54 (MSO-Logic over trees) Let $\Sigma = \Sigma_0 \cup \cdots \cup \Sigma_m$ be a ranked alphabet. A tree t corresponds to a structure

$$\underline{t} = \big(\mathrm{dom}_t, S_1^t, \ldots, S_m^t, \sqsubseteq^t, (P_a^t)_{a \in \Sigma}\big)$$

with

- S_i^t the i-th successor relation on dom_t (between two positions, where the second is on the i-th branch starting from the first, when counted "from left to right")

- $\sqsubseteq^t$ the prefix relation on dom_t (between two positions in t that are on the same path)
- $P_a^t := \{u \in \mathrm{dom}_t \mid \mathrm{val}_t(u) = a\}$ (i.e. the set of all positions of t labeled with an a)

The syntax of MSO-Logic over trees follows the one over words, though with the atomic formulas

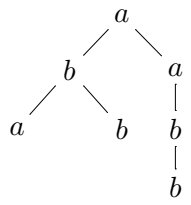
$$x = y, \quad S_1(x,y), \quad \ldots, \quad S_m(x,y), \quad x \sqsubseteq y, \quad P_a(x), \quad X_i(x)$$

Given a formula $\varphi(X_1, \ldots, X_n)$ the expression

$$\big(\underbrace{\mathrm{dom}_t, S_1^t, \ldots, S_m^t, \sqsubseteq^t, (P_a^t)_{a\in\Sigma}}_{\underline{t}}, K_1, \ldots, K_n\big) \models \varphi(X_1, \ldots, X_n)$$

states that "as an interpretation for X_i, $\underline{t}$ with $K_i \subseteq \mathrm{dom}_t$ satisfies φ". A sentence φ defines the tree language $T(\varphi) := \{t \in T_\Sigma \mid \underline{t} \models \varphi\}$. Such languages are called MSO-definable.

Example 3.55 For the tree t over $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$, with $\Sigma_0 = \Sigma_1 = \Sigma_2 = \{a, b\}$ depicted below, $P_a = \{\epsilon, 11, 2\}, P_b = \{1, 12, 21, 211\}$ and obviously $2 \sqsubseteq 211$.



⊠

Example 3.56 We express three main properties of trees by means of MSO-formulas and use these in following examples. Consider an alphabet with maximal rank $m = 2$. Then:

1. We can express that a node x is the root of a tree (i.e. it does not have any predecessors) by

$$\mathrm{Root}(x) = \neg\exists y(S_1(y,x) \vee S_2(y,x))$$

2. We can express that a node x is on the frontier of a tree (i.e. it does not have any successors) by

$$\mathrm{Fr}(x) = \neg\exists y(S_1(x,y) \vee S_2(x,y))$$

3. We can express that a set X of nodes of a tree is "closed under predecessors" (i.e. if some node belongs to X, then so does its predecessor) by
$$\forall y \forall z (X(y) \wedge (S_1(z,y) \vee S_2(z,y)) \rightarrow X(z))$$

⊠

Example 3.57 Now consider some more complicated issues:

1. The prefix operator $\sqsubseteq$ can be eliminated. That is, we can express $x \sqsubseteq y$ by:
$$\forall X (X(y) \wedge \text{ "X is closed under predecessors" } \rightarrow X(x))$$

2. "a occurs twice on the same path":
$$\exists x \exists y (\neg x = y \wedge x \sqsubseteq y \wedge P_a(x) \wedge P_a(y))$$

3. "a occurs twice not on the same path":
$$\exists x \exists y (\neg y \sqsubseteq x \wedge \neg x \sqsubseteq y \wedge P_a(x) \wedge P_a(y))$$

4. "there exists a subtree labeled only with a's":
$$\exists x \forall y (x \sqsubseteq y \rightarrow P_a(y))$$

5. "the frontier word contains an infix ab": To express this property we first express the $\prec$ relation, which is a partial order meaning that a node is "on the left" of another, for the case of a maximal rank $m = 2$:
$$x \prec y = \exists x_0 \exists y_0 \exists z (S_1(z,x_0) \wedge S_2(z,y_0) \wedge x_0 \sqsubseteq x \wedge y_0 \sqsubseteq y)$$
Then, the quoted property can be expressed by:
$$\exists x \exists y (\mathrm{Fr}(x) \wedge \mathrm{Fr}(y) \wedge P_a(x) \wedge P_b(y) \wedge x \prec y \wedge \neg \exists z (\mathrm{Fr}(z) \wedge x \prec z \wedge z \prec y))$$

⊠

The following theorem asserts the equivalence between tree automata and MSO-logic.

Theorem 3.58 (Doner, Thatcher-Wright 1968) *$T \subseteq T_\Sigma$ is regular if and only if T is MSO-definable.*

The proof is completely analogous to the one of the theorem of Büchi, Elgot, Trakhtenbrot, namely:

- for the direction from formulas to tree automata we consider an induction over the construction of MSO-formulas and make use of the closure properties of tree automata
- for the direction from tree automata to formulas we describe the existence of a run by an existential MSO-formula.

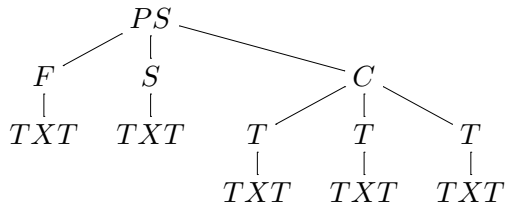
3.4 XML-Documents and Tree Automata

In this section we consider XML-Documents, which are special kinds of trees as we experienced in the very beginning of this chapter. Compared to the conventional trees we were dealing with by now, these ones may be slightly more complicated, but are therefore nowadays extremely significant for applications. The additional complication relies on the fact that arbitrary branching is allowed. But let us recall Example 3.4.

Example 3.59 We know that an XML-Document, like the one shown below, is a word that codes a tree using a bracketing $\langle X \rangle$ and $\langle /X \rangle$ to signal the beginning and the end of a branch.

```
<ProfileOfStudies>
  <faculty> Computer Science  </faculty>
  <semester> 3 </semester>
  <courses>
    <title> PROGRAMMING </title>
    <title> COMPUTER STRUCTURES </title>
    <title> DATA STRUCTURES </title>
  </courses>
</ProfileOfStudies>
```

This XML-Document codes the following tree:



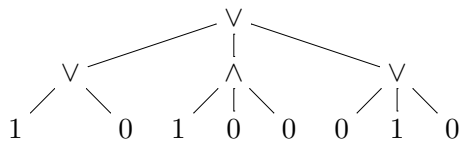
Note that the branching under "Courses" is arbitrary, i.e. we could as well list more than three courses. ⊠

This unlimited branching is already known from other areas. In arithmetics for example, although we formally define "+" as a binary operation, we are used to writing a sum of arbitrarily many numbers $(n_1 + \ldots + n_m)$ because we know that the order is not important. In this sense we could as well say that the "+"-operation has arbitrary arity (at least 2). The following example illustrates a similar case in the area of boolean expressions.

Example 3.60 In *generalized boolean expressions* arbitrarily long conjunctions and disjunctions can be formed ($\beta_1 \wedge \ldots \wedge \beta_n$ and $\beta_1 \vee \ldots \vee \beta_n$ with $n > 0$). In this sense the expression

$$((1 \vee 0) \vee (1 \wedge 0 \wedge 0) \vee (0 \vee 1 \vee 0))$$

represents the arbitrarily (but finitely) branching tree:



⊠

Considering the previous examples it becomes obvious that the ranked alphabets we introduced in the beginning of this chapter are not appropriate anymore. Instead, we introduce a labeling alphabet Σ that does not restrict the branching of a node.

Definition 3.61 A Σ-*valued tree* over the alphabet Σ is of the form $t = (\mathrm{dom}_t, \mathrm{val}_t)$, where

- $\mathrm{dom}_t \subseteq \mathbb{N}_+^*$ is a tree domain
- $\mathrm{val}_t : \mathrm{dom}_t \to \Sigma$ is the labeling function.

A tree domain D satisfies the following conditions:

1. D is closed under prefixes ($wj \in D \Rightarrow w \in D$)
2. D is closed under "left brother" ($wj \in D, 1 \leq i < j \;\Rightarrow\; wi \in D$)

A node with *rank* i must have exactly i successors.

The model of a Σ-valued tree is defined similarly to the one over a ranked alphabet. The main difference is that in this case it is rather inappropriate to use an indexed successor relation, because the index could be arbitrary. Instead, we only use a simple (not indexed) relation of the immediate successor and additionally we introduce an ordering relation between the immediate successors of a node.

Definition 3.62 (Model of a Σ-valued tree) Let t be a Σ-valued tree. Then

$$\underline{t} = (\mathrm{dom}_t, S^t, \sqsubseteq^t, <^t, (P_a^t)_{a\in\Sigma})$$

with

- S^t = a successor relation (edge relation) without indexing
- $\sqsubseteq^t$= a prefix relation on dom_t
- $<^t$= an ordering relation between all immediate successors of the same node, i.e., $u <^t v$ iff u is an "elder brother" of v in t

- $P_a^t = \{u \in \mathrm{dom}_t \mid \mathrm{val}_t(u) = a\}$

MSO-definability is defined as in the previous case.

XML-documents are such Σ-valued trees and their models can be used to interpret MSO-formulas. However, it is quite inconvenient to use MSO-logic to describe (groups of) XML-documents. Instead, another special kind of document description is used.

Definition 3.63 (Document-Type Definition (DTD)) A *Document-Type Definition (DTD)* is a system $\mathcal{S}$ of rules for the creation of a set of XML-documents. Each rule in this system has the form of a rule of a generalized context-free grammar, namely $X \to r$, where r is a regular expression. The resulting derivation trees (if coded into words using the special XML-bracketing) are the XML-documents created by $\mathcal{S}$.

Recall that such systems have exactly the same expressive power as conventional context-free grammars.

Theorem 3.64 *By using generalized context-free grammars, exactly the context-free languages can be constructed.*

Example 3.65 A context-free grammar to create the XML-document (tree) of Example 3.59 is defined by the following rules

$$\begin{aligned}
&\texttt{ProfileOfStudies} \to \texttt{faculty} \cdot \texttt{semester} \cdot \texttt{courses}\\
&\texttt{faculty} \to \texttt{terminal word}\\
&\texttt{semester} \to \texttt{terminal word}\\
&\texttt{courses} \to \texttt{title}^*\\
&\texttt{title} \to \texttt{terminal word}
\end{aligned}$$

The corresponding (equivalent) DTD would be:

```
<!DOCTYPE ProfileOfStudies
 [ <!ELEMENT ProfileOfStudies (faculty, semester, courses)>
   <!ELEMENT faculty (\# PCDATA)>
   <!ELEMENT semester (\# PCDATA)>
   <!ELEMENT courses (title*)>
   <!ELEMENT title (\# PCDATA)>
 ]>
```

where the keyword `\# PCDATA` stands for "parsed character data" to allow words over the the terminal alphabet. We use the keyword "TEXT" instead. ⊠

An alternative way to describe sets of XML-documents is to use a special tree automaton that is able to process them. Of course, such a kind of automaton should be defined in a way that it has exactly the same expressive power as DTDs.

Definition 3.66 (XML Tree Automaton) For an alphabet Γ let $\mathrm{Reg}(\Gamma)$ be the set of regular languages over Γ. An *XML-Automaton* is a nondeterministic tree automaton of the form $\mathcal{A} = (Q, \Sigma, \Delta, F)$ with

- a finite state set Q,
- a labeling alphabet Σ,
- a transition relation $\Delta \subseteq \mathrm{Reg}(Q) \times \Sigma \times Q$,
- a final state set $F \subseteq Q$.

Note that because of the arbitrary branching of the trees, there may be infinitely many different combinations of son-nodes allowed, in order to proceed to a certain parent node. Nevertheless, the transition relation is still presented as a finite set by each time specifying a regular expression or an NFA over Q for the first component of a transition tuple.

Definition 3.67 (Run and acceptance of an XML-automaton) A run of $\mathcal{A}$ on t is a function $\rho : \mathrm{dom}_t \to Q$ with the following conditions:

- for every node $u \in \mathrm{dom}_t$ of rank n there exists $(L, \mathrm{val}_t(u), \rho(u)) \in \Delta$ with $\rho(u1) \dots \rho(un) \in L$
- $\rho(\epsilon) \in F$ (i.e. the state assumed on the root has to be a final one)

$\mathcal{A}$ *accepts* t if there a exists such a run of $\mathcal{A}$ on t (i.e. a run fulfilling both conditions). By $T(\mathcal{A})$ we denote the set of all trees accepted by $\mathcal{A}$. We call such a tree language *regular*.

Note that, according to the first condition of the run function, for an a-valued leaf node the transition relation of an XML-automaton must contain a transition (L, a, q) with $\epsilon \in L$.

Example 3.68 We construct an XML-automaton that accepts the generalized Boolean expressions (as they were illustrated in Example 3.60) that evaluate to true. Not surprisingly we have to use a set of two states $Q = \{q_0, q_1\}$ denoting that the subtree evaluates to false or true, respectively. The automaton accepts if the true state is assumed at the root ($F = \{q_1\}$). The transition relation Δ contains the following transitions:

- $(\epsilon, 0, q_0), (\epsilon, 1, q_1)$
- $(Q^* q_0 Q^*, \wedge, q_0), \quad (q_1^+, \wedge, q_1)$
- $(Q^* q_1 Q^*, \vee, q_1), \quad (q_0^+, \vee, q_0)$

$\boxtimes$

To show that the definition of XML-automata is a correct one, we have to show our primary requirement, which was to create an alternative formalism that has the same expressive power as DTDs.

Theorem 3.69 *Every DTD is equivalent to an XML-automaton.*

Proof: Simply use the non-terminal symbols of the rule system $\mathcal{S}$ as states in the XML-automaton (cf. Theorem 3.23). □

One should note that for the other direction (the translation from XML-automata to DTDs) the situation is as in Theorem 3.23. This can be avoided by allowing several "instances" $A_1, A_2, \ldots, A_k$ of each non-terminal symbol A and considering the derivation trees with the indices of these instances removed (the details are omitted).

Note that the automaton defined in Example 3.68 happens to have at most one run on a given tree. This holds because the transitions are defined in a "mutually exclusive" way, i.e. all languages defined by the $\wedge$-transitions do not have common elements (and the same goes for the $\vee$-transitions). This kind of determinism results from the specific transitions of the automaton, but is still not integrated into its structure. In the following we define a variation of the conventional XML-automaton with a built-in determinism.

Definition 3.70 (Deterministic XML-Automaton) A deterministic XML-automaton has the form $\mathcal{A} = (Q, \Sigma, \delta, (\mathcal{S}_a)_{a\in\Sigma}, F)$, where:

- Q, Σ, F are as in the conventional case,
- $\delta : \Sigma \to Q$ (to initialize the run),
- for each $a \in \Sigma$, $\mathcal{S}_a$ is a DFA over Q with an output in Q.

The unambiguous run ρ of $\mathcal{A}$ on t is specified as follows:

- for a leaf node u: $\rho(u) = \delta(\mathrm{val}_t(u))$,
- for a node u of rank $n > 0$ and $\mathrm{val}_t(u) = a$: $\rho(u)$ is the (unambiguous) output of $\mathcal{S}_a$ on the input $\rho(u1) \ldots \rho(un)$ (i.e. upon reading the ordered sequence of the states of the immediate successors of an a-valued node, $\mathcal{S}_a$ returns the state that has to be assigned to the node)

Now that we have defined both the deterministic and the non-deterministic version of XML-automata it is natural to question ourselves about their expressive power. The following theorem asserts that, as with automata on ranked trees, nondeterminism does not offer the ability to describe more languages.

Theorem 3.71 *A nondeterministic XML-automaton is equivalent to a deterministic XML-automaton.*

Furthermore, all main issues that were considered for the case of automata on ranked trees (and also on finite words) hold in exactly the same way for XML-automata. These issues mainly include the following points:

1. The regular sets of valued trees are closed under the Boolean operations and under projection.

2. A set of valued trees is regular if and only if it is MSO-definable.

3. The emptiness problem, the inclusion problem and the equivalence problem for XML-automata are all decidable.

All these statements can be proved by simply copying and adapting the results that we know from the previous cases. Alternatively, we can code the Σ-valued trees by binary trees and use the results of the previous sections directly.

The mentioned properties of XML-automata can be used for various applications concerning XML-documents. Among these are:

1. Testing DTDs for compatibility: For the XML-automata $\mathcal{A}, \mathcal{B}$ corresponding to the given DTDs, check whether $T(\mathcal{A}) \subseteq T(\mathcal{B})$.

2. Query evaluation: Given a query expressed in logic (e.g. XPath-expression, MSO-formula), extract all documents satisfying the following condition: For the XML-automaton $\mathcal{A}$ construct a query representation $\mathcal{B}$ and check whether $T(\mathcal{A}) \cap T(\mathcal{B}) \neq \emptyset$.

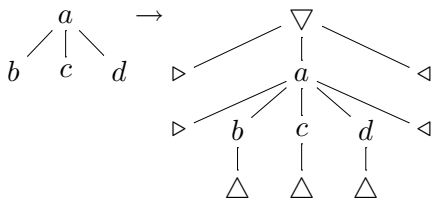
Remark 3.72 *a) Another very essential application in XML technologies is the transformation of an XML-document into another tree structure. This problem cannot be handled by the XML-automata that we introduced in this section; it requires the definition of XML-automata with output, the so-called* tree transducers*, which construct a new tree while processing an XML-document as input.*

b) It is very common in current applications to use identifiers to establish links between XML-documents. This leads to structures that cannot be considered (and processed) as trees anymore. The resulting graphs are much more difficult to handle than the initial XML-documents and require more complex methods.

As stated before, the model that we used for processing XML-documents, actually, was intentionally defined in such a way that it provides exactly the same expressive power as DTDs. In fact, when trying to figure out a way to read and validate a tree with arbitrary (but still finite) branching, another idea comes first into mind; that is, to navigate through the tree in a vertical and horizontal manner by visiting exactly one position of the tree at a time.

(This is similar to the way NFAs read the input of finite words letter by letter, although in this case we cannot really speak of a navigation since the movement takes place only in one direction). In the following we construct and briefly study such a "path-oriented" automaton that is always at one node of a tree at a time, and from there it can move to the parent node ($\uparrow$), to the first immediate successor ($\downarrow$), to the next immediate successor of the immediate predecessor node (i.e. to the next brother node) ($\rightarrow$) and analogously to the previous one ($\leftarrow$), or remain at the same position (stationary move: s). To avoid the case that this automaton gets out of bounds we extend the input tree t to t^+, by adding border nodes labeled by $\triangleright, \bigtriangledown, \triangleleft, \bigtriangleup$.

Example 3.73 The simple tree on the left must be extended to the one on the right before it is given as an input to the new automaton.



⊠

Now, let us describe this new automaton formally.

Definition 3.74 (Tree Walking Automaton) A *Tree Walking Automaton (TWA)* has the form $\mathcal{A} = (Q, \Sigma, q_0, \Delta, q_f)$ with

- a finite state set Q, an initial state q_0, an accepting state q_f,
- a transition set $\Delta \subseteq Q \times (\Sigma \cup \{\triangleright, \bigtriangledown, \triangleleft, \bigtriangleup\}) \times D \times Q$ where $D = \{\leftarrow, \uparrow, \rightarrow, \downarrow, s\}$.

A tuple $(p, a, d, q) \in \Delta$ says that, "when being in state p and currently reading the symbol a, perform movement d and proceed to state q, in case this is possible." A *configuration* of $\mathcal{A}$ on the input tree t is a pair (u, q), where $u \in \mathrm{dom}_t$ is the current position and q the current state of the TWA. The initial configuration is (ϵ, q_0) and the accepting one is (ϵ, q_f). $\mathcal{A}$ accepts the input t, if $\mathcal{A}$ can reach the accepting configuration (ϵ, q_f) from the initial configuration (ϵ, q_0) by processing t. The language $T(\mathcal{A})$ recognized by $\mathcal{A}$ is defined as $T(\mathcal{A}) = \{t \mid t \text{ is a } \Sigma\text{-valued tree and } \mathcal{A} \text{ accepts } t^+\}$.

Example 3.75 Let $\Sigma = \{a, b, c\}$ and consider the language T of all Σ-valued trees having at least one a-labeled node with both a b-labeled and a c-labeled descendant. A *nondeterministic* TWA that recognizes T has a

state set $Q = \{q_a, q_b, q_c, q_r, q_f\}$, where q_a, q_b, q_c mean that the automaton is currently searching for an a, b or c respectively, q_r that the pattern was found and the automaton is on the way back to the root, and q_f is the accepting state. We choose $q_0 = q_b$, i.e. we first navigate down the tree to find a node with label b, from there a node with label a on the path back to the root, and from there a node with label c somewhere in the subtree. Translated into transitions this procedure is coded as follows:

- search for an appropriate b (i.e. the automaton has to guess that this b has an a-labeled predecessor) with transitions $(q_b, \sigma, \downarrow / \rightarrow, q_b)$
- on visiting a b switch to state q_a and search for an a by using transitions (q_b, b, s, q_a) and $(q_a, \sigma, \uparrow, q_a)$
- on visiting an a proceed to q_c and search for a c-labeled node in the subtree with the transitions $(q_a, a, \downarrow, q_c)$ and $(q_c, \sigma, \rightarrow / \downarrow, q_c)$
- having visited a c proceed to q_r and return to the root by applying the transitions (q_c, c, s, q_r) and $(q_r, \sigma, \uparrow, q_r)$
- accept with the transition $(q_r, \bigtriangledown, s, q_f)$

$\boxtimes$

Regarding the relationship between XML-automata and TWAs and between deterministic and nondeterministic TWAs, the following results have been shown up to now.

Theorem 3.76 *a) Any TWA can be translated into an equivalent (nondeterministic) XML-automaton.*

b) There exists a language that is recognizable by an XML-automaton but not by any (nondeterministic) TWA.

c) There exists a language that is recognizable by a nondeterministic TWA but not by any deterministic TWA.

There are, still, some fundamental questions concerning TWAs that remain open to computer science; one of those is concerned with complementation: "Is the complement of the tree language recognized by a nondeterministic TWA also recognizable by a nondeterministic TWA?" We suspect that the answer is negative, though.

3.5 Automata over Two-Dimensional Words (Pictures)

In this last section of the chapter we are going to take one more further step in generalizing our models by considering two-dimensional words as elements of languages and inputs for automata. Strictly speaking, this slightly

generalized model does not really belong to the area of trees. The essential distinction between them is the fact that trees are the kind of directed graphs where two pieces of information flow from the root to the leaves in a "spreading" manner, i.e. without being able to converge again into some node. In other words, in trees a node has always at most one predecessor whereas in two-dimensional words it has at most two. Still, we are going to study pictures as a generalized model in this chapter because this very step of generalization is the one that leads us out of the convenient world of decidability and efficient computability as far as it concerns the most fundamental problems on describing and processing input models. In particular, after introducing automata and MSO-formulas over labeled grids, we are going to verify that

a) the class of recognizable picture languages is not closed under complement and

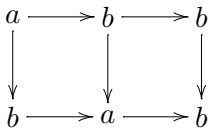
b) the emptiness problem is undecidable.

Definition 3.77 (Picture) A *picture* over Σ is a pair $p = (\mathrm{dom}_p, \mathrm{val}_p)$ with

- $\mathrm{dom}_p = \{1, \ldots, m\} \times \{1, \ldots, n\}$ for $m, n > 1$ and
- $\mathrm{val}_p : \mathrm{dom}_p \to \Sigma$.

In this case we are speaking of a picture of size (m, n).

Example 3.78 Below we can see a picture of size $(2, 3)$.



⊠

Definition 3.79 (Picture Language) Let Σ^{++} be the set of all pictures over Σ. Any $B \subseteq \Sigma^{++}$ is called a *picture language* over Σ. If Σ is a singleton set, then we speak of *grids* $[m, n]$ and *grid languages*.

Example 3.80 Below we list three basic picture languages that we are going to consider in further detail later.

1. for $\Sigma = \{a, b\}$:
   $B_1 = \{p \in \Sigma^{++} \mid p(3, 3) = a, \text{ otherwise } p(i, j) = b\}$
2. for $\Sigma = \{\cdot\}$:
   $B_2 = \{[n, n] \mid n > 0\}$ (square grids)

3. for $\Sigma = \{a, b\}$:
   $B_3 = \{pp \mid p \in \Sigma^{++}, p \text{ is a square}\}$

⊠

As usual we are going to introduce a model over these pictures that allows us afterwards to define MSO-formulas, describing languages like the ones mentioned above. Models of pictures belonging to a language should then serve as an interpretation for the corresponding MSO-formula.

Definition 3.81 (Pictures as Models) For a picture p of size (m, n) over Σ let

$$\underline{p} := \left(\mathrm{dom}_p, S_1^p, S_2^p, \left(P_a^p\right)_{a\in\Sigma}\right)$$

with

- $S_1^p\big((i,j),(i+1,j)\big)$ (lower successor), $S_2^p\big((i,j),(i,j+1)\big)$ (right successor) for all suitable i, j and
- $P_a^p = \{u \in \mathrm{dom}_p \mid \mathrm{val}_p(u) = a\}$.

Definition 3.82 (MSO-Logic over Pictures) The *MSO-Logic over pictures* (over Σ) is as previously defined with the atomic formulas:

$$x = y \quad S_1(x,y) \quad S_2(x,y) \quad P_a(x) \quad X(y)$$

The semantics $(\underline{p}, K_1, \ldots, K_r) \models \varphi(X_1, \ldots, X_r)$ where $K_i \subseteq \mathrm{dom}_p$ is defined as usually. For a sentence φ we call $B(\varphi) = \{p \in \Sigma^{++} \mid \underline{p} \models \varphi\}$ the *picture language defined by* φ.

Example 3.83 Consider the language B_1 of Example 3.80. First we construct an auxiliary formula to specify the symbol at position $(0, 0)$ of the picture:

$$\mathrm{topleft}(x) = \neg\exists y S_1(y,x) \wedge \neg\exists y S_2(y,x)$$

Now B_1 can be defined by the following formula:

$$\exists x_1, y_1, x_2, y_2, x_3(\mathrm{topleft}(x_1) \wedge S_1(x_1,y_1) \wedge S_2(y_1,x_2) \wedge S_1(x_2,y_2) \wedge S_2(y_2,x_3) \wedge P_a(x_3) \wedge \forall y(\neg y = x_3 \rightarrow P_b(y)))$$

i.e. by expressing the fact that in the picture there must be a top left position from which, after "moving" twice right and twice down, we reach an a-labeled position whereas all other positions are labeled by letter b. ⊠

Example 3.84 Consider the language B_2 of Example 3.80. In a square grid, departing from the top left position we can reach the bottom right position by continuously moving once to the right and once down in an

alternating fashion. In this sense we can say that a square grid is actually a picture in which "every set X that contains the top left position and is closed under diagonal successors also contains the bottom right position". This is exactly what we express by the following MSO-formula:

$$\begin{aligned}\forall X \; \big[\; & \forall x(\text{topleft}(x) \rightarrow X(x)) \wedge \\ & \forall x \forall y \forall z(X(x) \wedge S_1(x,y) \wedge S_2(y,z) \rightarrow X(z)) \\ & \rightarrow \forall x(\text{bottomright}(x) \rightarrow X(x)) \; \big]\end{aligned}$$

where $\text{bottomright}(x) := \neg\exists y S_1(x,y) \wedge \neg\exists y S_2(x,y)$ ⊠

Now that we have defined a logical formalism to describe picture languages it is time to construct a corresponding automaton-oriented model. To this purpose we first recall the construction we adopted in the case of conventional trees. The run of a tree automaton is a mapping of states to the nodes of the input tree. This mapping, along with the labels of the nodes, has to comply with the transitions that the automaton is allowed to perform.

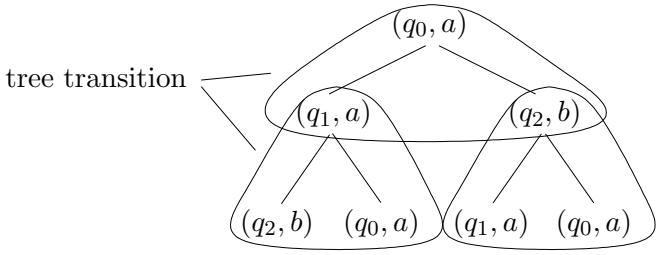


Figure 3.1: Run tree of a tree automaton.

This way, a run is composed by matching the prescribed triangles (allowed transitions) to a triangle of nodes (labeled tree node along with both its immediate successors). Picture acceptors (Tiling-Systems) are nondeterministic systems that function in a similar fashion. They guess the assignment of states to the input picture and then verify the local correctness of the assignment by matching it with the allowed transitions. But this time transitions are represented by (2×2)-squares instead of triangles, as illustrated by Figure 3.2. Each such possible square of the "filled-out" picture has to correspond to a transition of Tiling-System. To be able to distinguish those squares that include some boundary nodes (i.e. belonging to the first or the last row or column) of the picture, we extend the input by marginal symbols #, as illustrated by the following example:

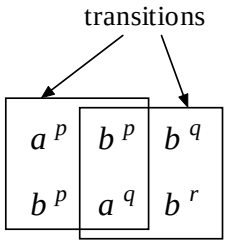


Figure 3.2: Verification of matching transitions on an assignment of states

$$\begin{array}{ccccc} \# & \# & \# & \# & \# \\ \# & a & b & b & \# \\ \# & b & a & b & \# \\ \# & \# & \# & \# & \# \end{array}$$

Definition 3.85 (Tiling-Systems (T-Systems)) A *Tiling-System (T-System)* for pictures over Σ has the form $\mathcal{A} = (Q, \Sigma, \Delta)$ with a finite set of states Q and $\Delta \subseteq \left[(\Sigma \cup \{\#\}) \times Q\right]^4$. In other words, a transition is a tuple of four pairs of the type (input symbol, assigned state). To have a better overview we represent a transition $(a_1, q_1, a_2, q_2, a_3, q_3, a_4, q_4)$ by a matrix $\begin{pmatrix} a_1q_1 & a_2q_2 \\ a_3q_3 & a_4q_4 \end{pmatrix}$. A run of $\mathcal{A}$ on $p = \begin{pmatrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \dots & a_{m,n} \end{pmatrix}$ is a matrix $\rho = \begin{pmatrix} q_{0,0} & \cdots & q_{0,n+1} \\ \vdots & \ddots & \vdots \\ q_{m+1,0} & \cdots & q_{m+1,n+1} \end{pmatrix}$ such that, with $a_{0,j} = a_{i,0} = a_{m+1,j} = a_{i,n+1} := \#$, the following holds, for all suitable i, j:

$$\begin{pmatrix} a_{i,j}q_{i,j} & a_{i,j+1}q_{i,j+1} \\ a_{i+1,j}q_{i+1,j} & a_{i+1,j+1}q_{i+1,j+1} \end{pmatrix} \in \Delta.$$

$\mathcal{A}$ *accepts* p if and only if there exists such a run of $\mathcal{A}$ on p. As usual we call the set $B(\mathcal{A}) := \{p \in \Sigma^{++} \mid \mathcal{A} \text{ accepts } p\}$ the language recognized by the t-system $\mathcal{A}$.

Now we are going to consider once again the languages presented in Example 3.80 and this time describe them by t-systems.

Example 3.86 To recognize B_1 (the set of all pictures with an a at (3,3) and a b everywhere else) we construct a t-system with states $Q = \{0, 1, 2\}$ (1 to be assigned to the first and 2 to the second diagonal element) and

the transitions $\begin{pmatrix} \#0 & \#0 \\ \#0 & b1 \end{pmatrix}$, $\begin{pmatrix} b1 & b0 \\ b0 & b2 \end{pmatrix}$, $\begin{pmatrix} b2 & b0 \\ b0 & a0 \end{pmatrix}$, along with all other ones that are required to "fill-out" the rest of the picture with state 0. An example run could be (states are denoted as exponents, and state 0 is assumed whenever no state is mentioned):

$$\begin{array}{ccccc} \#^0 & \# & \# & \# & \# \\ \# & b^1 & b & b & \# \\ \# & b & b^2 & b & \# \\ \# & b & b & a^0 & \# \\ \# & b & b & b & \# \\ \# & \# & \# & \# & \# \end{array}$$

⊠

Example 3.87 To recognize the square grids required by language $B_2 = \{[m, m] \mid m > 0\}$ from Example 3.80 we use almost the same trick as when we were looking for an appropriate MSO-formula. The idea is to "emit a signal-state" from the top left corner and transmit it through the grid in the diagonal direction. To this purpose we choose a state set $Q = \{0, 1\}$ and require the transitions to fit to the picture below (state 0 is assumed whenever no state is mentioned):

$$\begin{array}{ccccc} \#1 & \# & \# & \# & \# \\ \# & \cdot 1 & \cdot & \cdot & \# \\ \# & \cdot & \cdot 1 & \cdot & \# \\ \# & \cdot & \cdot & \cdot 1 & \# \\ \# & \# & \# & \# & \#1 \end{array}$$

The t-system accepts the grid only if it can match its special verification transition $\begin{pmatrix} \cdot 1 & \#0 \\ \#0 & \#1 \end{pmatrix}$ with some part of the input, i.e. if the 1 that was emitted from the top-left corner is received by the bottom-right corner. ⊠

Remark 3.88 *If two languages B_1, B_2 are both recognizable by t-systems, then the same holds for the languages $B_1 \cup B_2$ and $B_1 \cap B_2$.*

The idea for the proof of $B_1 \cup B_2$ is to construct a union of the two t-systems that should have disjoint state sets. For $B_1 \cap B_2$ the usual product construction has to be applied in order to build a t-system with state set $Q_1 \times Q_2$.

The last example that we study right away is a more complicated one and is based on the language B_3 of Example 3.80.

Example 3.89 Let $\Sigma = \{a, b\}$ and consider the language $\overline{B_3}$, namely the set of all pictures over Σ that do not have the form pp for some square p. A picture p' belongs to this language if either

1. p' is not of size $(n, 2n)$, or

2. p' is indeed of size $(n, 2n)$, but there exist some $1 \leq i, j \leq n$ such that $\mathrm{val}_{p'}(i, j) \neq \mathrm{val}_{p'}(i, j + n)$.

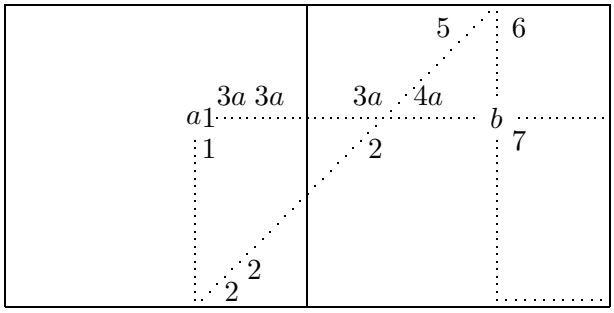
It suffices to construct one t-system for each case because afterwards, by Remark 3.88, we can construct a t-system for the union of these two, which obviously recognizes $\overline{B_3}$.

Case 1: Let us first find out how we can accept pictures of size $(n, 2n)$. This can be done by extending the idea of Example 3.87, namely as soon as the signal state 1 is received by some bottom position it has to be reflected to the up-right diagonal direction (renamed as signal state 2 to avoid any confusion). Now we shall expect to receive the signal state 2 in the top-right corner of the grid. Below we can see an accepting run that also shows the other transitions that are required in order to fill out the grid correctly:

#1	#0	#0	#0	#0	#0	#0	#0	#0	#0	#0	#2
#0	·1	·0	·0	·0	·0	·0	·0	·0	·0	·2	#0
#0	·0	·1	·0	·0	·0	·0	·0	·0	·2	·0	#0
#0	·0	·0	·1	·0	·0	·0	·0	·2	·0	·0	#0
#0	·0	·0	·0	·1	·0	·0	·2	·0	·0	·0	#0
#0	·0	·0	·0	·0	·1	·2	·0	·0	·0	·0	#0
#0	#0	#0	#0	#0	#0	#0	#0	#0	#0	#0	#0

However, in this example we actually want to accept everything but this kind of grids. For this purpose, we keep the same t-system and refuse to receive the signal state 2 at the top-right corner by striking out the transition $\begin{pmatrix} \#0 & \#2 \\ \cdot 2 & \#0 \end{pmatrix}$. Instead, we may receive the signal state 2 everywhere else on the right or top border by adding the transitions $\begin{pmatrix} \#0 & \#0 \\ \cdot 2 & \cdot 0 \end{pmatrix}$, $\begin{pmatrix} \cdot 0 & \#0 \\ \cdot 2 & \#0 \end{pmatrix}$ (in case the number of columns is greater or less than $2n$, respectively), or we may receive only the signal state 1 somewhere on the right border by adding the transitions $\begin{pmatrix} \cdot 1 & \#0 \\ \cdot 0 & \#0 \end{pmatrix}$, $\begin{pmatrix} \cdot 1 & \#0 \\ \#0 & \#0 \end{pmatrix}$, $\begin{pmatrix} \cdot 0 & \#0 \\ \cdot 1 & \#0 \end{pmatrix}$ (in case the number of columns is $\leq n$). Of course we expect state 0 in the critical top-right corner by adding the transition $\begin{pmatrix} \#0 & \#0 \\ \cdot 0 & \#0 \end{pmatrix}$. Finally, some extra consideration has to be done for the sizes $(1, 1)$ and $(1, n)$, which we skip at this point.

Case 2: The second t-system has to accept pictures p of size $(n, 2n)$ (we already know from the previous case how this can be done) that have at least one pair of positions (i, j), $(i, j+n)$ such that $\mathrm{val}_p(i, j) \neq \mathrm{val}_p(i, j+n)$. First of all the t-system has to guess such a position (i, j), which can be done easily by using nondeterminism. Then, it has to count n positions to the right to verify that the symbol at $(i, j+n)$ is different. Of course counting cannot be done by using n different states because n is not fixed. The idea is depicted below (we assume that (i, j) is a-labeled and $(i, j+n)$ is b-labeled):



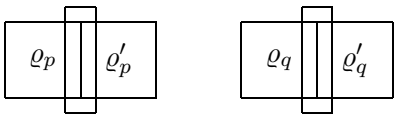
The position (i, j) is guessed by using the transition $\begin{pmatrix} a1 & *3a \\ *1 & *0 \end{pmatrix}$. From there the signal state 1 is emitted vertically towards the bottom border and the signal state $3a$ (the a stands for the letter that has been read; it is perfectly in order to do this because the alphabet is finite and given) is emitted horizontally towards position $(i, j+n)$. How do we know that state $3a$ just reached this position to perform the comparison with (i, j)? As soon as state 1 hits the bottom border, it is renamed to 2 and further transmitted to the upper right diagonal direction, mainly with the transitions $\begin{pmatrix} *1 & *2 \\ \#0 & \#0 \end{pmatrix}$ and $\begin{pmatrix} *0 & *2 \\ *2 & *0 \end{pmatrix}$. When state 2 crosses the horizontal transmission of state $3a$, they are renamed to 5 and $4a$, respectively. The corresponding transitions are $\begin{pmatrix} *3a & *4a \\ *2 & *0 \end{pmatrix}$ and $\begin{pmatrix} *0 & *5 \\ *4a & *4a \end{pmatrix}$. When state 5 hits the top border, it is renamed to 6 and further transmitted vertically towards the bottom border, using transitions $\begin{pmatrix} \#0 & \#0 \\ *5 & *6 \end{pmatrix}$, $\begin{pmatrix} *5 & *6 \\ *0 & *6 \end{pmatrix}$ and $\begin{pmatrix} *0 & *6 \\ *0 & *6 \end{pmatrix}$. Note that this happens at column $j+n$ because the picture has n rows and state 2 (later state 5 and 6) took as many steps to the right as to the top. When state 6 crosses the horizontal transmission of state $4a$, the letter comparison is done by transitions $\begin{pmatrix} *0 & *6 \\ *4a & b7 \end{pmatrix}$, $\begin{pmatrix} *4a & b7 \\ *0 & *7 \end{pmatrix}$. In

case the letters are different an "OK" signal is transmitted (by using state 7) to the bottom and finally to the bottom-right corner with transitions $\binom{*0\ *7}{*0\ *7}$, $\binom{*0\ *7}{\#0\ \#7}$, $\binom{*7\ *7}{\#7\ \#7}$, $\binom{*7\ \#7}{\#7\ \#7}$. ⊠

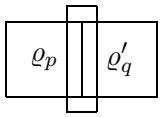
But what about the language B_3 of Example 3.80? Is it also recognizable by a t-system as it is the case with its complement? If not, then we would have proven that the closure property under complement does not hold for picture languages recognizable by t-systems.

Theorem 3.90 *The picture language $B_3 = \{pp \mid p \in \Sigma^{++}, p \text{ is a square}\}$ is not recognizable by a t-system.*

Proof: Towards a contradiction we assume that $\mathcal{A} = (Q, \{a, b\}, \Delta)$ recognizes B_3. Set $k := |Q|$ and consider the runs of $\mathcal{A}$ on pictures of size $(n, 2n)$. The number of all possible central stripes (the part of the run consisting of columns n and $n+1$) in such runs is at most $|(\Sigma \cup \{\#\}) \times Q|^{2(n+2)} = (3k)^{2n+4}$. The number of all possible pictures of size (n, n) over Σ is 2^{n^2}. Because the second term grows faster than the first one (with respect to n), we can find a suitable n such that $2^{n^2} > (3k)^{2n+4}$. For such an n, there exist two pictures pp and qq with $p \neq q$ that are both accepted by $\mathcal{A}$ with two runs that have the same central stripes:



Combining the left part of the left picture with the right part of the right one, we can compose a run that accepts pq:



Contradiction. □

The following theorem summarizes the closure properties that we studied above.

Theorem 3.91 *The class of picture languages recognizable by t-systems is closed under union and intersection, but not under complement.*

As far as it concerns the relation of t-systems to MSO-logic the following holds:

Theorem 3.92 (Giammarresi, Restivo, Seibert, Ths. 1994) *A picture language is t-system recognizable if and only if it is EMSO-definable.*

Theorem 3.93 (Matz, Schweikardt, Ths. 1997) *As the number of alternations between second-order quantifiers grows we obtain an infinite hierarchy of bigger and bigger classes of picture languages.*

Compare the latter result with the case of finite words, where EMSO-logic can provide the same expressive power as MSO-logic.

We conclude this section (and the whole chapter about trees) by proving that even the most fundamental decision problem on automata is not solvable within the area of pictures.

Theorem 3.94 *The emptiness problem for t-systems is undecidable.*

Proof: To prove undecidability, as usual, we consider a reduction of the halting problem for Turing machines. In other words, we transform a Turing machine (TM) M to a t-system $\mathcal{A}_M$, such that

$$M \text{ halts on the empty tape} \quad \Longleftrightarrow \quad B(\mathcal{A}_M) \neq \emptyset \ .$$

This should suffice to prove that it is not possible to have a method to decide the emptiness problem for t-systems because, if we had one, then we would be able to decide the halting problem for TMs as well (through the above mentioned transformation), which has been proven to be impossible.

We construct $\mathcal{A}_M$ in such a way that $B(\mathcal{A}_M)$ codes (in its elements) exactly the terminating computations of M when started on the empty tape. Let $M = (Q, \Sigma, q_0, \Delta, q_f)$ with $\sqcup \in \Sigma$. Δ contains transitions of the form $(q, a, a', l/r, q')$. We suppose that the tape is left bounded. The configuration words are of the form $b_1 \ \ldots \ b_{i-1} \ (qb_i) \ b_{i+1} \ \ldots \ b_n$ (denoting that M is currently in state q, reading letter b and the rest of the symbols are written on the tape to the left and the right of the head). For transitions $(q, b_i, b', l, q') \in \Delta$ and $(q, b_i, b', r, q') \in \Delta$ the successor configurations are $b_1 \ \ldots \ (q'b_{i-1}) \ b' \ b_{i+1} \ \ldots \ b_n$ and $b_1 \ \ldots \ b_{i-1} \ b' \ (q'b_{i+1}) \ \ldots \ b_n$, respectively. If we write the sequence of configurations one after the other into the rows of a table, we obtain a picture of size (m, n), for sufficiently large m, n. This way, a terminating computation of M started on the empty tape can be represented by a picture over $\Sigma \cup (Q \times \Sigma)$:

$$\begin{array}{ccc} a_{1,1} & \ldots & a_{1,n} \\ \vdots & & \vdots \\ a_{m,1} & \ldots & a_{m,n} \end{array}$$

with

- $a_{1,1} \ldots a_{1,n} = (q_0\sqcup)\sqcup \ldots \sqcup$,

- $a_{i+1,1} \dots a_{i+1,n}$ is the word coding a successor configuration of $a_{i,1} \dots a_{i,n}$ for $1 \leq i < m$, and
- in $a_{m,1} \dots a_{m,n}$ a letter $(q_f b)$ occurs.

In the transitions of $\mathcal{A}_M$ no states are needed because the t-system only needs to verify the correctness of the picture by comparing two consecutive rows and two consecutive positions on the tape (because the Turing machine can move the head only by one position). For this comparison the 2×2 tiles of the t-system suffice. Hence, the t-system needs the following transitions (we have to assume that the TM is deterministic and complete; otherwise, we have to check that on each row the control state appears only once, which is possible using 2 states in the system):

For the initial configuration:

$$\begin{pmatrix} \# & \# \\ \# & (q_0 \sqcup) \end{pmatrix}, \begin{pmatrix} \# & \# \\ (q_0 \sqcup) & \sqcup \end{pmatrix}, \begin{pmatrix} \# & \# \\ \sqcup & \sqcup \end{pmatrix}, \begin{pmatrix} \# & \# \\ \sqcup & \# \end{pmatrix}$$

For the intermediate configurations:

$$\begin{pmatrix} b & (qa) \\ b & a' \end{pmatrix}, \begin{pmatrix} \# & (qa) \\ \# & a' \end{pmatrix}, \begin{pmatrix} (qa) & b \\ a' & (q'b) \end{pmatrix}, \qquad \text{for } (q, a, a', r, q') \in \Delta$$

$$\begin{pmatrix} (qa) & b \\ a' & b \end{pmatrix}, \begin{pmatrix} (qa) & \# \\ a' & \# \end{pmatrix}, \begin{pmatrix} b & (qa) \\ (q'b) & a' \end{pmatrix}, \qquad \text{for } (q, a, a', l, q') \in \Delta$$

$$\begin{pmatrix} \# & a \\ \# & (qa) \end{pmatrix}, \begin{pmatrix} a & \# \\ (qa) & \# \end{pmatrix}, \begin{pmatrix} \# & a \\ \# & a \end{pmatrix}, \begin{pmatrix} a & \# \\ a & \# \end{pmatrix}, \qquad \text{for } a \in \Sigma, q \in Q$$

$$\begin{pmatrix} a & a' \\ a & a' \end{pmatrix}, \begin{pmatrix} a & a' \\ a & (qa') \end{pmatrix}, \begin{pmatrix} a & a' \\ (qa) & a' \end{pmatrix}, \qquad \text{for } a, a' \in \Sigma, q \in Q.$$

For the final configuration:

$$\begin{pmatrix} \# & a \\ \# & \# \end{pmatrix}, \begin{pmatrix} a & a' \\ \# & \# \end{pmatrix}, \begin{pmatrix} a & \# \\ \# & \# \end{pmatrix},$$

$$\begin{pmatrix} \# & (q_f a) \\ \# & \# \end{pmatrix}, \begin{pmatrix} (q_f a) & a' \\ \# & \# \end{pmatrix}, \begin{pmatrix} a & (q_f a') \\ \# & \# \end{pmatrix}, \begin{pmatrix} (q_f a) & \# \\ \# & \# \end{pmatrix}$$

for $a, a' \in \Sigma$. □

3.6 Exercises

Exercise 3.1 Let $\Sigma = \Sigma_0 \cup \Sigma_2$ with $\Sigma_0 = \Sigma_2 = \{f, g\}$ be a ranked alphabet. Give tree automata for the following languages:

(a) $\{\, t \in T_\Sigma \mid$ on some path in t there are two consecutive occurrences of $f \,\}$

(b) $\{\, t \in T_\Sigma \mid$ on every path in t there are two consecutive occurrences of $f \,\}$

Remark: a path is always starts at the root and ends at a leaf.

Exercise 3.2 Let the tree languages $T_1, T_2 \subseteq T_\Sigma$ be recognized by the tree automata $\mathcal{A}_1, \mathcal{A}_2$, where $\mathcal{A}_i = (Q_i, \Sigma, \delta_i, F_i)$ for $i = 1, 2$. Give tree automata recognizing the following languages:

(a) $T_\Sigma \backslash T_1$,

(b) $T_1 \cap T_2$.

Exercise 3.3 Let $\Sigma = \Sigma_0 \cup \Sigma_2$ be a ranked alphabet with $\Sigma_0 = \{c\}$, $\Sigma_2 = \{a, b\}$. Show that

(a) $T_1 := \{\, t \in T_\Sigma \mid$ in t there are more a's than b's $\,\}$ is not regular,

(b) $T_2 := \{\, t \in T_\Sigma \mid$ in t there are more c's than a's $\,\}$ is regular.

Exercise 3.4 Let $\Sigma_2 = \{f\}$, $\Sigma_0 = \{a, b\}$, and

$$T := \{\, t \in T_\Sigma \mid \text{the frontier word of } t \text{ is in } (ab)^+ \,\} \ .$$

Give a DTA that recognizes T. Provide also an explanation.
Alternatively: Solve the exercise for any regular language $L \subseteq \{a, b\}^+$ instead of $(ab)^+$.

Exercise 3.5 (a) Which language is generated by the following regular expression over $\Sigma_0 = \Sigma_2 = \{0, 1\}$?

$$\left(\begin{array}{c} 0 \\ / \ \backslash \\ c_0 \ \ c_0 \end{array} \right)^{*_{c_0}} .^{c_0} \left[\begin{array}{c} 1 \\ / \ \backslash \\ c_1 \ \ c_1 \end{array} .^{c_1} \left(\begin{array}{c} 0 \\ / \ \backslash \\ c_0 \ \ c_0 \end{array} \right)^{*_{c_0}} .^{c_0} \{0\} \right]$$

(b) Give a regular expression over $\Sigma = \Sigma_0 \cup \Sigma_2$ with $\Sigma_0 = \{0, 1\}$ and $\Sigma_2 = \{f\}$ that generates the language

$$T := \{\, t \in T_\Sigma \mid \text{the frontier word of } t \text{ contains an infix } 01 \,\} \ .$$

Exercise 3.6 * Let an *almost-deterministic top-down tree automaton* be defined like a deterministic top-down tree automaton, but with a set Q_0 of initial states. A run is successful if one of these states is assumed at the root. Show that there exists a recognizable tree language that cannot be recognized by an almost-deterministic top-down tree automaton.

Exercise 3.7 Consider the generalized arithmetic expressions with constants $0, 1$ and the functions $+$ and $\cdot$ with arbitrary arities $n \geq 2$. Give an XML-automaton that recognizes the set T_0 of all generalized expressions that have the value 0 (modulo 3).

Exercise 3.8 (a) Give a deterministic (!) TWA that accepts those input trees over $\Sigma = \{a, b, c\}$ where an a occurs.
Hint: Program a run through the whole input tree.

(b) Give a deterministic TWA for the language T presented in the lecture, namely the set of all Σ-valued trees having at least one a-labeled node with both a b-labeled and a c-labeled successor.

Provide an explanation for each part.

Exercise 3.9 Give a t-system that accepts exactly those pictures p over the singleton alphabet $\{\cdot\}$ whose domain dom_p contains an even number of elements. Provide an explanation.

Exercise 3.10 * Show that the set of square grids $[m, m]$ where m is a prime number is recognizable by a t-system.

Chapter 4

Pushdown and Counter Systems

In this chapter we are entering the area of systems whose sets of configurations are infinite. This makes it harder to find algorithmic solutions to certain problems, because we cannot anymore hope to gather all required information by taking finitely many steps throughout the transition graph of such a system. Nevertheless, we present some methods that successfully solve the most fundamental problems. We then apply our results to systems that are based on recursive structures and gain some significant results. Finally, we identify and prove decidability limits towards more complex systems.

4.1 Pushdown and Counter Automata

It is well known that pushdown automata accept precisely the context-free languages. Words are accepted using a finite set of control states and a stack as an auxiliary memory. A *global state* or *configuration* of such an automaton is a pair consisting of a control state and a stack content. Hence, different from the finite state systems treated so far, the set of configurations of a pushdown automaton can be infinite, and therefore new questions arise.

Definition 4.1 A *pushdown automaton* (PDA) is a tuple $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta)$ where

- Q is a finite set of control states,
- Σ is a finite input alphabet,
- Γ is a finite stack alphabet,
- $q_0 \in Q$ is the initial state,

- $Z_0 \in \Gamma$ is the initial stack symbol, and
- $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times \Gamma^* \times Q$ is the transition relation.

A transition $(p, a/\epsilon, Z, \beta, q) \in \Delta$ is interpreted in the following way: If $\mathcal{A}$ is in state p, reading letter a or no input letter respectively, with Z on top of the stack, it replaces Z by β and enters control state q.

A *configuration* is a pair (state q, stack content u) and is often represented by a single word qu, where the first letter of u represents the top stack symbol. An *extended configuration* is a triple (state, stack content, remaining input). With a transition $(p, a/\epsilon, Z, \beta, q) \in \Delta$ we reach a successor of the current extended configuration $(p, Zu, aw) \vdash (q, \beta u, w)$ or $(p, Zu, w) \vdash (q, \beta u, w)$ in case the empty word is read. In terms of conventional configurations this is written as $pZu \vdash q\beta u$. Hence, we observe that the configurations of a PDA induce a *prefix rewriting system*.

An input word w is accepted by a PDA $\mathcal{A}$ if there is a computation of $\mathcal{A}$ on w that starts in q_0 with Z_0 on the stack, reads all letters of w, and reaches the empty stack at the end of w. In other words, a PDA accepts w if from (q_0, Z_0, w) an extended configuration (q, ϵ, ϵ) can be reached in finitely many steps:

$$(q_0, Z_0, w) \ \vdash^* \ (q, \epsilon, \epsilon) \text{ for some } q \in Q \ .$$

We call $L(\mathcal{A}) = \{w \in \Sigma^* \mid \mathcal{A} \text{ accepts } w\}$ the language recognized by $\mathcal{A}$.

Theorem 4.2 (Main Theorem) *A language L is context-free (i. e. can be constructed by a context-free grammar) iff L is recognizable by a PDA.*

We recall three fundamental results from the area of context-free grammars. Given a context-free grammar (or two for the third result) G:

1. The word problem "does w belong to $L(G)$?" is decidable (CYK-algorithm).
2. The emptiness problem "Is $L(G) = \emptyset$?" is decidable (marking of the variables from which a terminal word can be derived).
3. The inclusion and the equivalence problem are both undecidable.

Since PDAs describe the same class of languages as context-free grammars, there must be a way to gain analogous results for the former by some reduction to the latter. However, because in computer science PDAs often occur independently (i. e. not related to some grammar), we solve the first two problems directly for PDAs.

Pushdown automata play a central role in various topics in computer science. In compilers the stack serves as the central data structure. In particular, it is used as an auxiliary memory in the implementation of recursive

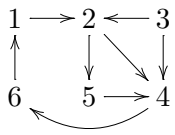
procedures. In the special case that the stack alphabet consists of a single symbol, apart from the initial stack symbol, a stack actually implements a counter. If the stack contains only the initial stack symbol, the number 0 is represented. Any other number is represented by the number of occurrences of the (only) stack symbol on top of the initial one. This again shows that the set of all possible stack contents is infinite, as natural numbers are too.

Example 4.3 The depth-first search in a graph $G = (V, E)$, starting from a node v_0, can conveniently be implemented by the following recursive procedure DFS(v):

```
Mark v
For all non-marked successors w of v:   DFS(w)
```

We illustrate this process on the graph below, where $v_0 = 1$, and denote the nested calls of the algorithm on the left.

```
DFS(1)
    DFS(2)
        DFS(4)
            DFS(6)
        DFS(5)
```



To implement this by an iterative program we have to store the procedure calls in "reverse" direction. This can be done by the following algorithm that uses a stack as an auxiliary memory:

```
Initialize stack with v0
Mark v0
As long as the stack is not empty, assume the top symbol is v:
   if there exists a non-marked neighbor of v:
     mark the first non-marked neighbor w of v
     and push w onto the stack
   otherwise, delete v from the stack
```

If we apply this algorithm to the example graph from above, the stack content evolves as follows:

$$1 \to 21 \to 421 \to 6421 \to 421 \to 21 \to 521 \to 21 \to 1 \to \epsilon \ .$$

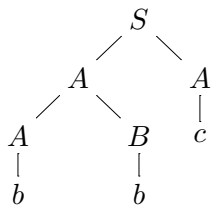
⊠

Example 4.4 According to Theorem 4.2, if a language L can be generated by a context-free grammar, then L is recognizable by a PDA. Suppose L is generated by a grammar G. Then $w \in L$ iff there exists a derivation tree t for w in G. While reading w, this can be verified by guessing a left derivation that corresponds to a depth-first search through the derivation tree. A stack once again serves as an auxiliary memory for this kind of depth-first search.

Given the following grammar

$$S \to AA, \quad A \to AB \mid b \mid c, \quad B \to b \ ,$$

we suppose that the tree shown below is guessed by a nondeterministic automaton. On the right we can see how the stack content evolves after each step of the depth-first search.



$$\begin{aligned} &S\\ &AA\\ &ABA\\ &bBA\\ &BA\\ &bA\\ &A\\ &c \end{aligned}$$

⊠

By defining the configurations of a PDA as states of a transition system, we can extend the PDA to an infinite automaton, since there are in general infinitely many configurations. This shows again that PDAs can accept a wider class of languages than finite automata.

Definition 4.5 [Configuration graph of a PDA] If $\mathcal{A}$ is a PDA, then the *configuration graph* of $\mathcal{A}$ is defined as $G_{\mathcal{A}} = (V, (E_a)_{a\in\Sigma\cup\{\epsilon\}})$, where

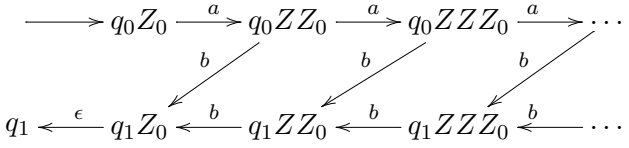
- V is the set of all configurations reachable from q_0Z_0, and
- $(pZu, q\beta u) \in E_a$ or $(pZu, q\beta u) \in E_\epsilon$ if there is a transition $(p, Z, a/\epsilon, \beta, q) \in \Delta$.

A *pushdown graph* is the configuration graph of a PDA without ϵ-transitions.

Example 4.6 We define a PDA $\mathcal{A}$ for the language $L = \{a^n b^n \mid n > 0\}$. Let $\mathcal{A} = (\{q_0, q_1\}, \{a, b\}, \{Z_0, Z\}, q_0, Z_0, \Delta)$ with

$$\Delta = \left\{ \begin{array}{ll} (q_0, a, Z_0, ZZ_0, q_0), & (q_0, a, Z, ZZ, q_0),\\ (q_0, b, Z, \epsilon, q_1), & (q_1, b, Z, \epsilon, q_1),\\ (q_1, \epsilon, Z_0, \epsilon, q_1) & \end{array} \right\}.$$

The configuration graph of $\mathcal{A}$ is shown below.



⊠

The way we defined a PDA at the beginning of the section allows to replace the top symbol of the stack by several ones (i. e. a word β) in one step. We say that a PDA is in *normal form*, if it has only transitions $(p, a/\epsilon, Z, \beta, q)$ with $|\beta| \leq 2$. That is, in one step the length of the stack can be increased or reduced by at most 1.

Lemma 4.7 *For every PDA there exists an equivalent PDA in normal form.*

Proof: For each transition with $|\beta| > 2$ add $|\beta| - 2$ intermediate transitions. □

Every PDA induces a *pushdown system*, where we suppress the special roles of q_0 and Z_0, and ignore the input letters.

Definition 4.8 A *pushdown system* (PDS) $\mathcal{P}$ is a tuple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of states, Γ is a finite stack alphabet, and $\Delta \subseteq (P \times \Gamma \times P \times \Gamma^*)$ is a finite set of transition rules, written $pa \to qv$.

A *configuration* (or global state) of a pushdown system $\mathcal{P}$ is a word $pw \in P \cdot \Gamma^*$, where the first letter of w represents the top symbol of the stack.

We write

- $pw \vdash p'w'$ if $w = aw_0$, $w' = vw_0$, and $pa \to p'v \in \Delta$,
- $pw \vdash^i p'w'$ if there exists $p_0w_0, \ldots, p_iw_i$ with $pw = p_0w_0$, $p_iw_i = p'w'$, and $p_jw_j \vdash p_{j+1}w_{j+1}$ for $0 \leq j < i$, and
- $pw \vdash^* p'w'$ if $pw \vdash^i p'w'$ for some $i \geq 0$.

The *configuration graph* of a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$ is the graph with vertex set $\{c \mid c \in P\Gamma^*\}$ and an edge between c_1 and c_2 if $c_1 \vdash c_2$.

Example 4.9 Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS, where $P = \{p_0, p_1, p_2\}$, $\Gamma = \{a, b, c\}$ and $\Delta = \{(p_0a \to p_1ba), (p_1b \to p_2ca), (p_2c \to p_0b), (p_0b \to p_0\epsilon)\}$. A sequence of configurations reachable from p_2cbba is shown below.

$$p_2cbba \vdash p_0bbba \vdash p_0bba \vdash p_0ba \vdash p_0a \vdash p_1ba \vdash p_2caa \vdash \ldots$$

⊠

In the following we consider an extension of pushdown systems to systems that have more than one stack.

Definition 4.10 [n-Pushdown System] An *n-pushdown system* is defined analogously to a PDS but with n stacks. What needs to be redefined is the way transitions are executed. By $[n]$ we denote the set $\{1, \ldots, n\}$. Then,

the transition relation is $\Delta \subseteq Q \times [n] \times \Gamma \times \Gamma^* \times Q$. A transition (p, i, a, v, q) means that being in state p with a as the top symbol of stack i, the system is allowed to replace this a by v and proceed to state q. The configurations are analogous to the ones of a PDS.

We recall the special case of counter systems and extend them to more than one counters by giving analogous formal definitions.

Definition 4.11 [Counter System] A *counter system* is a pushdown system whose stack alphabet contains one single symbol Z apart from the initial stack symbol Z_0. Hence, the stack content is $Z^m Z_0$ and represents the number m. A transition (p, Z, v, q) with $v \in Z^*$ means the following:

- for $|v| > 1$: addition of $|v| - 1$ and change to state q;
- for $|v| = 1$: change to state q;
- for $|v| = 0$: subtraction of 1 and change to state q.

A counter system is deterministic if the corresponding PDS is.

An *n-counter system* is an n-pushdown system where the stack alphabet for all stacks contains one single symbol Z apart from the initial stack symbol Z_0. A configuration has the form $(p, i_1, \ldots, i_n)$, denoting the current state and the number currently represented by each of the n counters.

Counter systems appear in programming languages, however under a different name and format. To study the computation limits on functions over $\mathbb{N}$, register machines were introduced as a model of programming language.

Definition 4.12 [Register Machines (Shepherdson, Sturgis 1963)] An *n-register machine* is a sequence of instructions $1\ I_1; 2\ I_2; \ldots; k\ I_k$ (i. e. a program) with program variables $X_1, \ldots, X_n$. For $j < k$ an instruction I_j has one of the following forms:

- `INC` X_i (increase the value of X_i by 1 and proceed to the next instruction);
- `DEC` X_i (decrease the value of X_i by 1, if possible, and proceed to the next instruction);
- `IF` $X_i = 0$ `GOTO` m (if the value of $X_i = 0$, jump to instruction I_m, otherwise proceed to I_{j+1}).

I_k is the instruction `STOP`. A configuration $(j, \text{valueOf}(X_1), \ldots, \text{valueOf}(X_n))$ contains the number j of the instruction that is executed in the next step, and the values currently assigned to each of the n variables.

Example 4.13 Consider the register machine R_0:

1 `IF` $X_2 = 0$ `GOTO` 5;
2 `INC` X_1;
3 `DEC` X_2;
4 `IF` $X_3 = 0$ `GOTO` 1;
5 `STOP`

Because $(1, x, y, 0) \vdash^* (5, x+y, 0, 0)$, R_0 can be used to perform the addition of two natural numbers. ⊠

Even if register machines are allowed to perform only the three basic instructions mentioned above, it can be shown that they possess the same expressive power as Turing machines. That is, using only these three instructions and a finite number of variables one can compute all functions in number theory. Instead of proving this directly, we conclude this section by showing that register machines are indeed only a different formulation of counter systems, and in a later section we reduce Turing machines to counter systems.

Proposition 4.14 *Deterministic counter systems $\mathcal{Z}$ in normal form can simulate register machines R and vice versa.*

More specifically, given a counter system $\mathcal{Z}$, there exists a register machine R that comprises the states of $\mathcal{Z}$ such that:

$$\begin{aligned} & \mathcal{Z} : (p, i_1, \ldots, i_n) \vdash^* (q, j_1, \ldots, j_n) \\ \text{iff} \quad & R : (p, i_1, \ldots, i_n) \vdash^* (q, j_1, \ldots, j_n) \end{aligned}$$

The converse direction can be expressed in an analogous way.

Proof (from register machines to counter systems): Given R construct $\mathcal{Z}$ by using the instruction numbers $1, \ldots, k$ as states and add the following transitions:

- for $I_j =$ `INC` X_i: $(j, i, Z, ZZ, j+1)$, $(j, i, Z_0, ZZ_0, j+1)$;
- for $I_j =$ `DEC` X_i: $(j, i, Z, \epsilon, j+1)$, $(j, i, Z_0, Z_0, j+1)$;
- for $I_j =$ `IF` $X_i = 0$ `GOTO` m: (j, i, Z_0, Z_0, m), $(j, i, Z, Z, j+1)$.

□

Proof (from counter systems to register machines): For the sake of simplicity we exclude complex cases in the transitions of the counter system. We assume that transitions either do not depend on whether the current top stack symbol is Z or Z_0, or they leave the stack unchanged (each counter system can easily be transformed into one satisfying these restrictions).

Given a deterministic n-counter system $\mathcal{Z}$ in normal form, we construct an $n+1$-register machine R where we use the last variable as a constant $X_{n+1} = 0$ to be able to express static `GOTO` instructions (i. e. not depending on the "dynamic" value of a variable). Before proceeding to the construction we make the counter system complete by adding transitions such that for every possible control state and top stack symbol the counter system can take a step. If $\mathcal{Z}$ has the states $1, \ldots, k$, introduce the instruction numbers $1, \ldots, 2k+1$ (instruction $2k+1$ is `STOP`). We denote the static instruction "j `IF` $X_{n+1} = 0$ `GOTO` m" by "j `GOTO` m". The transition relation is then transformed to register specific instructions according to the following cases:

- for $(j, i, Z/Z_0, ZZ/ZZ_0, m)$: $2j-1$ `INC` X_i; $2j$ `GOTO` $2m-1$
- for (j, i, Z, ϵ, m) and (j, i, Z_0, Z_0, m): $2j-1$ `DEC` X_i; $2j$ `GOTO` $2m-1$
- for (j, i, Z, Z, m) and (j, i, Z_0, Z_0, m'):
  $2j-1$ `IF` $X_i = 0$ `GOTO` $2m'-1$; $2j$ `GOTO` $2m-1$

□

4.2 The Reachability Problem for Pushdown Systems

Configuration graphs of pushdown systems are (in general) infinite. Hence the question whether one can reach from a configuration c_1 a configuration c_2 becomes more interesting. We call this the *reachability problem*. Let C be a set of configurations. We define

$$post^*(C) := \{c \mid c_0 \vdash^* c \text{ for some } c_0 \in C\}$$

and

$$pre^*(C) := \{c \mid c \vdash^* c_0 \text{ for some } c_0 \in C\}.$$

With this notation the reachability problem for two configurations can be restated in two versions:

- **Forward reachability problem:** Given c and the definition of C, does $c \in post^*(C)$ hold?
- **Backwards reachability problem:** Given c and the definition of C, does $c \in pre^*(C)$ hold?

These formulations can be sharpened to meet a generalized version of the reachability problem. That is, given the definition of some configuration set C, we want to determine the definitions of the configuration sets $post^*(C)$ and $pre^*(C)$. Note that such a result is very helpful for testing certain safety conditions in system design.

Theorem 4.15 (Büchi 1964) *Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS and $C \subseteq P\Gamma^*$ a regular set of configurations. Then $pre^*(C)$ and $post^*(C)$ are again regular.*

Moreover, from an NFA $\mathcal{A}$ accepting C we can compute an NFA $\bar{\mathcal{A}}$ accepting $pre^*(C)$ and an NFA $\tilde{\mathcal{A}}$ accepting $post^*(C)$.

Before proceeding to the proof, we look at some very significant results that emanate from this theorem.

Corollary 4.16 *The stack contents that are reachable from the initial configuration $q_0 Z_0$ of a PDA form a regular set.*

Corollary 4.17 *The point-to-point reachability problem for pushdown systems is decidable.*

Proof: Let c_1, c_2 be two configurations. Obviously $\{c_2\}$ is a regular set. By using Theorem 4.15 we can determine a DFA that recognizes $pre^*(\{c_2\})$. Then we only need to check whether c_1 is accepted by this DFA, in order to decide if c_2 is reachable from c_1. □

Since counter systems (and register machines equivalently) are a specialization of pushdown systems, the same statement as Corollary 4.17 must hold for this case too.

Theorem 4.18 *The point-to-point reachability problem for 1-counter systems (equivalently for 1-register machines) is decidable.*

Corollary 4.19 *The emptiness problem for context-free languages is decidable.*

Proof: We use PDAs (instead of grammars) to prove this corollary. For the language recognized by a PDA $\mathcal{A}$ the following holds:

$$L(\mathcal{A}) \neq \emptyset \text{ iff } q_0 Z_0 \vdash^* (q, \epsilon) \text{ for some } q \in Q \ .$$

This means that the context-free language is non-empty iff $q_0 Z_0 \in pre^*(Q\epsilon)$, which can be decided by Theorem 4.15. □

Proof (of Theorem 4.15): First we have to clarify how we will give definitions for sets of configurations. We use special NFAs to represent such regular sets. Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS and $C \subseteq P\Gamma^*$ a regular set of configurations. A *P-automaton* for C is an NFA $\mathcal{A} = (Q, \Gamma, P, \Delta_{\mathcal{A}}, F)$ that accepts from an initial state $p \in P$ exactly the words $w \in \Gamma^*$ such that $pw \in C$. The control states of $\mathcal{P}$ are used as initial states of $\mathcal{A}$. Additionally, it is not allowed in $\mathcal{A}$ to have a transition to an initial state.

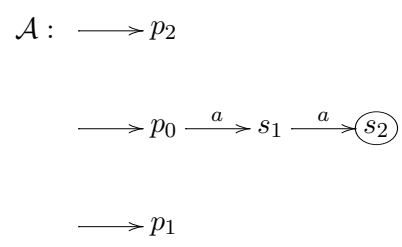
Input: A P-automaton $\mathcal{A}$, a PDS $\mathcal{P} = (P, \Gamma, \Delta)$

1. $\mathcal{A}_0 := \mathcal{A}$, $i := 0$
2. **repeat** :
3. **if** $pa \rightarrow p'v \in \Delta$ and $\mathcal{A}_i : p' \xrightarrow{v} q$
4. **then** add (p, a, q) to $\mathcal{A}_i$ to obtain $\mathcal{A}_{i+1}$
5. $i := i + 1$
6. **until** no more transitions can be added.
7. $\bar{\mathcal{A}} := \mathcal{A}_i$

Output: $\bar{\mathcal{A}}$

Figure 4.1: First Saturation Algorithm

Example 4.20 Consider the pushdown system from Example 4.9 and the configuration $C = \{p_0aa\}$. The P-automaton for C is the following:



⊠

The idea that we use (proposed by Bouajjani, Esparza, Maler, Hansel, Rossmanith, Schwoon) is a transformation of such an NFA. We consider the regular target set C of configurations of a PDS and suppose that the NFA $\mathcal{A}(C)$ accepts qvw with a run $q \xrightarrow{v} r \xrightarrow{w} F$. If there exists a rule $pa \rightarrow qv$ in the pushdown system, then paw must be accepted by the NFA as well. Therefore we add a transition $p \xrightarrow{a} r$ to its transition relation.

Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS and $C \subseteq P\Gamma^*$ a regular set of configurations represented by a P-automaton $\mathcal{A} = (Q, \Gamma, P, \Delta_{\mathcal{A}}, F)$. We present two saturation algorithms to compute P-automata $\bar{\mathcal{A}}$ and $\tilde{\mathcal{A}}$ accepting the sets $pre^*(C)$ and $post^*(C)$, respectively.

The first saturation algorithm is shown in Figure 4.1. Example 4.21 shows the P-automaton computed by the first saturation algorithm with the pushdown system from Example 4.9 and the P-automaton from Example 4.20 as inputs.

Since the number of transitions that can be added to a P-automaton is finite, the algorithm terminates. For the correctness we have to prove that

$$pw \in pre^*(C) \Leftrightarrow \bar{\mathcal{A}} : p \xrightarrow{w} F.$$

For the direction from left to right we show for $n \geq 0$:

$$pw \vdash^n ru \in C \;\Rightarrow\; \bar{\mathcal{A}} : p \xrightarrow{w} F.$$

For $n = 0$ there is nothing to show. Suppose now that $pw \vdash^n ru$ and $ru \in C$. We can decompose this sequence into $paw' \vdash p'vw' \vdash^{n-1} ru$ where $w = aw'$ and $pa \to p'v$ is a transition of the pushdown system. By the induction hypothesis we know that $\bar{\mathcal{A}} : p' \xrightarrow{vw'} F$, i. e. there is a state q such that $\bar{\mathcal{A}} : p' \xrightarrow{v} q \xrightarrow{w'} F$. Hence the saturation rule ensures that there is a transition $(p, a, q) \in \Delta_{\bar{\mathcal{A}}}$, and thus pw is accepted by $\bar{\mathcal{A}}$.

For the converse we have to show a more general claim to make the induction work. We show that for all pw and all q we have

$$\bar{\mathcal{A}} : p \xrightarrow{w} q \;\Rightarrow\; \exists p'w' \;:\; \mathcal{A} : p' \xrightarrow{w'} q \text{ and } pw \vdash^* p'w' \qquad (*)$$

For $q \in F$ this yields the original claim because $\mathcal{A} : p' \xrightarrow{w'} q$ then means that $p'w' \in C$, and thus $pw \vdash^* p'w'$ implies that $pw \in pre^*(C)$.

If $\bar{\mathcal{A}} : p \xrightarrow{w} q$, then there exists i such that $\mathcal{A}_i : p \xrightarrow{w} q$. We prove (*) by induction on i. For $i = 0$ the claim obviously holds with $p'w' = pw$ as $\mathcal{A}_0 = \mathcal{A}$.

For $i > 0$ let (p_1, a, q_1) be the transition that was added by the algorithm to $\mathcal{A}_{i-1}$ to obtain $\mathcal{A}_i$. This transition is used j times in the run $\mathcal{A}_i : p \xrightarrow{w} q$ for some $j \geq 0$. We prove the induction step for i by another induction, this time on j. If $j = 0$, then we get $\mathcal{A}_{i-1} : p \xrightarrow{w} q$ and we can directly apply the induction hypothesis for i.

If $j > 0$, then we can split the run $\mathcal{A}_i : p \xrightarrow{w} q$ at the first position where the new transition is used as follows (where $w = u_1 a u_2$):

$$\underbrace{\mathcal{A}_{i-1} : p \xrightarrow{u_1} p_1}_{(1)}, \quad \underbrace{\mathcal{A}_i : p_1 \xrightarrow{a} q_1}_{(2)}, \quad \underbrace{\mathcal{A}_i : q_1 \xrightarrow{u_2} q}_{(3)}.$$

We can apply the induction hypothesis for i to (1) (i. e. to pu_1 and p_1). This means that there is $p_1'u_1'$ such that $\mathcal{A} : p_1' \xrightarrow{u_1'} p_1$ and $pu_1 \vdash^* p_1'u_1'$. As a P-automaton does not have any transitions leading to an initial state, the only possibility for $\mathcal{A} : p_1' \xrightarrow{u_1'} p_1$ is $u_1' = \epsilon$ and $p_1' = p_1$. We get $pu_1 \vdash^* p_1$.

Considering (2), the transition (p_1, a, q_1) was added by the algorithm because there is a pushdown rule $p_1 a \to p_2 v$ with $\mathcal{A}_{i-1} : p_2 \xrightarrow{v} q_1$. Concatenating this run with (3) we obtain the run $\mathcal{A}_i : p_2 \xrightarrow{v} q_1 \xrightarrow{u_2} q$ in which the transition (p_1, a, q_1) is used only $j - 1$ times. The induction hypothesis on j yields $p'w'$ such that $\mathcal{A} : p' \xrightarrow{w'} q$ and $p_2 v u_2 \vdash^* p'w'$.

Combining this with $pu_1 \vdash^* p_1$ (derived from (1)) and the rule $p_1 a \to p_2 v$ we get $pw \vdash^* p_1 a u_2 \vdash p_2 v u_2 \vdash^* p'w'$, proving (*).

For the part of Theorem 4.15 concerning $post^*(C)$ we use the same idea. We suppose we are given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, this time in normal form,

Input: A P-automaton $\mathcal{A}$ and a PDS $\mathcal{P} = (P, \Gamma, \Delta)$

1. Set $\mathcal{A}_0 := \mathcal{A}$ and $i := 0$
2. For each rule $r : pa \to p'bc$ add a new state r and a transition (p', b, r)
3. **repeat** (Saturation steps):
4. **if** $pa \to p' \in \Delta$ and $\mathcal{A}_i : p \xrightarrow{a} q$ **then** add (p', ϵ, q)
5. **if** $pa \to p'a' \in \Delta$ and $\mathcal{A}_i : p \xrightarrow{a} q$ **then** add (p', a', q)
6. **if** $r : pa \to p'a'a'' \in \Delta$ and $\mathcal{A}_i : p \xrightarrow{a} q$ **then** add (r, a'', q)
7. $i := i + 1$
8. **until** no more transitions can be added.
9. $\tilde{\mathcal{A}} := \mathcal{A}_i$

Output: $\tilde{\mathcal{A}}$

Figure 4.2: Second Saturation Algorithm

i. e. containing only rules $pa \to q$ or $pa \to qb$ or $pa \to qbc$. We use a P-automaton that recognizes the (regular) configuration set C of $\mathcal{P}$. Suppose that the NFA $\mathcal{A}(C)$ accepts paw with a run $p \xrightarrow{aw} F$. If there exists a rule $pa \to qv$ in the pushdown system, then we must be able to construct a run $q \xrightarrow{vw} F$ in the NFA. If the first transition of the run $p \xrightarrow{aw} F$ is $p \xrightarrow{a} r$, then we distinguish three cases for a rule $pa \to qv$ of the PDS.

1. $v = \epsilon$: add a transition $q \xrightarrow{\epsilon} r$ (now $qvw = qw$ is accepted by the run $q \xrightarrow{\epsilon} r \xrightarrow{w} F$);

2. $v = b$: add a transition $q \xrightarrow{b} r$ (now $qvw = qbw$ is accepted by the run $q \xrightarrow{b} r \xrightarrow{w} F$);

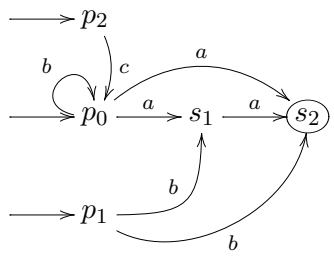
3. $v = bc$: we have to add some transition "$q \xrightarrow{bc} r$". But since the NFA can process only one letter at a time, we need an intermediate state r_ρ for each such rule ρ with $|v| = 2$. In this case, before starting the procedure we should first have extended the NFA by a transition $q \xrightarrow{b} r_\rho$. Now we can add the transition $r_\rho \xrightarrow{c} r$, so that the NFA finally accepts $qvw = qbcw$ by the run $q \xrightarrow{b} r_\rho \xrightarrow{c} r \xrightarrow{w} F$.

The second saturation algorithm is shown in Figure 4.2. Example 4.21 shows the P-automaton computed by this algorithm with the same inputs as above.

Since the correctness proof is similar to the one for the first saturation algorithm, it has been dropped here. □

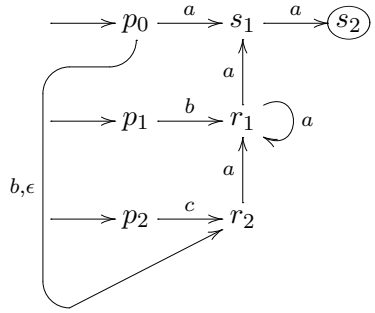
Example 4.21 Let $\mathcal{P}$ be the pushdown system from Example 4.9. Applying the first saturation algorithm for the P-automaton from 4.20, we obtain the automaton $\bar{\mathcal{A}}$ accepting $pre^*(p_0aa)$.

$\bar{\mathcal{A}}$:



The automaton $\tilde{\mathcal{A}}$ obtained by applying the second saturation algorithm and accepting $post^*(p_0aa)$ is shown below.

$\tilde{\mathcal{A}}$:



⊠

Complexity estimation of the first saturation algorithm: For the backwards reachability problem we are given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ in normal form and a P-automaton $\mathcal{A} = (Q, \Gamma, P, \Delta_{\mathcal{A}}, F)$. Hence, the length of the input for the algorithm is $|P| + |\Delta| + |Q| + |\Delta_{\mathcal{A}}|$. In the first step of each loop (see Figure 4.1) we examine the existence of both a rule of the pushdown system (there are $|\Delta|$ many) and a matching run (with some v) in the P-automaton. Because $|v| \leq 2$ (normal form), such a run can comprise at most two transitions. Hence, for each rule in Δ we have to examine at most $|Q|^2$ runs, which in the worst case sums up to $|\Delta| \cdot |Q|^2$ checks in each loop. Now because a transition is a combination of a pair of states and a letter of the input alphabet, the loop can be executed at most $|P| \cdot |Q| \cdot |\Gamma|$ times; more transitions cannot be added to the P-automaton. All in all we get a (polynomial) complexity of $|\Delta| \cdot |P| \cdot |Q|^3 \cdot |\Gamma|$.

Complexity estimation of the second saturation algorithm: The input is the same as before. In this algorithm we have an additional computational effort that must be carried out before entering the loop (see Figure 4.2), in order to extend the P-automaton for each rule that increases the

stack by one symbol. This can be done as many times as there are rules, namely $|\Delta|$ times. In each loop and for each rule in Δ we have to find out in which of the three cases it belongs (at most 3 comparisons) and then find a matching transition (not a run!). In the worst case this can be done in $3 \cdot |\Delta| \cdot |Q|$ steps. For a similar reason as before the loop can be executed at most $(|P| + |\Delta|) \cdot |Q| \cdot |\Gamma|$ times. All in all we get a (polynomial) complexity of $|\Delta| + 3 \cdot |\Delta| \cdot (|P| + |\Delta|) \cdot |Q|^2 \cdot |\Gamma|$.

We conclude this section by recalling what holds about the equivalence tests between two PDAs.

Theorem 4.22 *The problem*

> *"Given two PDAs $\mathcal{A}, \mathcal{B}$, do $\mathcal{A}$ and $\mathcal{B}$ recognize the same language?"*

is undecidable. However, it is decidable for deterministic PDAs.

Theorem 4.23 *The problem*

> *"Given two PDAs $\mathcal{A}, \mathcal{B}$, are $\mathcal{A}$ and $\mathcal{B}$ bisimilar?"*

is decidable.

4.3 Recursive Hierarchical Automata

In the previous section we thoroughly studied the reachability problem for PDSs. In this section we consider a kind of system whose analysis depends on the application of this reachability test. These are the so called "recursive hierarchical systems" which consist of components $K_0, \ldots, K_n$ that can call one another or even themselves (recursion). The states of such a system are either "normal" ones or calls of a component K_j. Figure 4.3 shows the system K_0 that comprises three normal states and two calls, one to itself and one to a component K_1. Calls to components are usually represented as boxes, like in the picture. Unlike the case of Figure 4.3, boxes may have

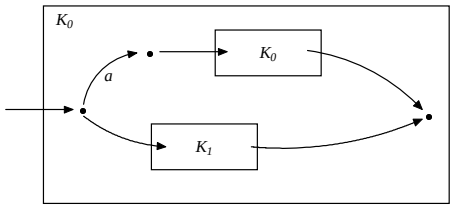


Figure 4.3: A recursive hierarchical system.

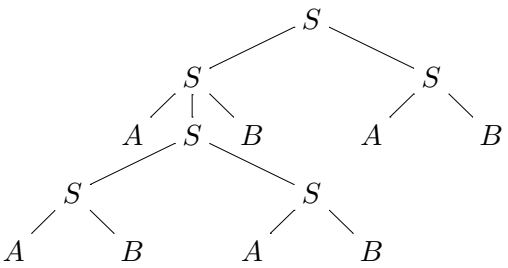
several inputs and outputs. For the sake of simplicity, we only consider the case of a single input and output in the following.

The concept which recursive hierarchical systems rely on is of course well-known to us from the context-free grammars. The idea of replacing a component with several ones (including itself) is represented by the rules that imply a substitution of some letter by several ones (including itself).

Example 4.24 Consider the context-free grammar G, defined by the following rules:

$$S \to AB \mid ASB \mid SS, \quad A \to a \mid cA, \quad B \to b \mid cB \ .$$

S generates the bracket words over $\{A, B\}$, e. g. $AABABBAB$. If we imagine "A" as "(" and "B" as ")", then this word corresponds to "(()())()". A derivation tree of this word is shown below.



Finally A is replaced by words in c^*a and B by words in c^*b.

The rules of this grammar induce a recursive hierarchical system S (Figure 4.4) that can sequentially call the components A, B or the components A, S, B or the components S, S.

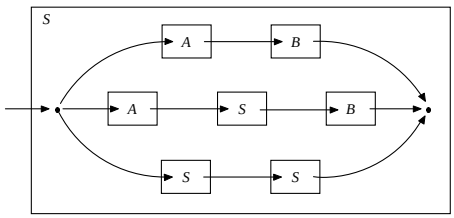


Figure 4.4: A recursive hierarchical system corresponding to the grammar of Example 4.24.

The component A is shown in Figure 4.5, where either a c is selected and A is recursively called, or the call is completed by an a. The component B is constructed similarly. ⊠

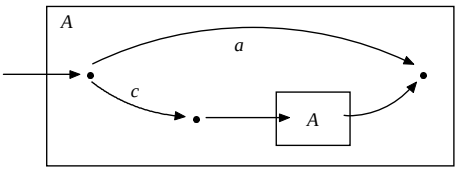


Figure 4.5: A component of the recursive hierarchical system of Figure 4.4.

Definition 4.25 [Recursive Hierarchical Automaton] A *recursive hierarchical automaton* over Σ has the form $\mathcal{A} = (K_0, \ldots, K_n)$, where every component $K_j = (Q_j, BX_j, BI_j, i_j, \Delta_j, o_j)$ consists of the following:

- a finite state set Q_j;
- a finite set of boxes BX_j;
- a specification of the box content $BI_j : BX_j \to \{0, \ldots, n\}$. In particular, $BI_j(B) = i$ means that in the component K_j the box B is a call to the component K_i;
- an input state i_j and an output state o_j;
- a transition relation Δ_j with transitions of the form (p, a, q), (p, ϵ, B), (B, ϵ, q), (B, ϵ, B') for $p, q \in Q_j$, $a \in \Sigma$, and $B, B' \in BX_j$.

Definition 4.26 [Run of a Recursive Hierarchical Automaton] A *run* of $\mathcal{A}$ is a sequence $p_0, b_1, p_1, b_2, p_2, \ldots, b_m, p_m$ of

- states p_i from $Q = \bigcup_{j=0}^{n} Q_j$ and
- labels $b_i \in \Sigma \cup \{\epsilon\}$.

For each segment p_i, b_{i+1}, p_{i+1} of the sequence there is a $j \in \{0, \ldots, n\}$ such that the following holds:

- $(p_i, b_{i+1}, p_{i+1}) \in \Delta_j$ (normal transition), or
- $(p_i, b_{i+1}, B) \in \Delta_j$ with $B \in BX_j$, $BI_j(B) = m$, and $p_{i+1} = i_m$ (a call to component K_m), or
- $(B, b_{i+1}, p_{i+1}) \in \Delta_j$ with $B \in BX_j$, $BI_j(B) = m$, and $p_i = o_m$ (return from component K_m), or
- $(B, b_{i+1}, B') \in \Delta_j$ with $B, B' \in BX_j$, $BI_j(B) = m$, $BI_j(B') = m'$, $p_i = o_m$, $p_{i+1} = i_{m'}$ (call to component $K_{m'}$, while returning from component K_m).

An *accepting run on* w is a run from i_0 to o_0, labeled by the word w. $\mathcal{A}$ accepts w, if there exists an accepting run of $\mathcal{A}$ on w.

Example 4.27 Consider the word *ccab* and the recursive hierarchical systems of Figures 4.4 and 4.5. By following the upper branch of system S and twice the lower and then once the upper branch of component A, we get the following accepting run of *ccab* (of course we also follow once the upper branch of component B, which is analogous to A):

$$i_S \xrightarrow{\epsilon} i_A \xrightarrow{c} q \xrightarrow{\epsilon} i_A \xrightarrow{c} q \xrightarrow{\epsilon} i_A \xrightarrow{a} o_A \xrightarrow{\epsilon} o_A \xrightarrow{\epsilon} o_A \xrightarrow{\epsilon} i_B \xrightarrow{b} o_B \xrightarrow{\epsilon} o_S \ .$$

⊠

Theorem 4.28 *Recursive hierarchical automata recognize exactly the context-free languages.*

We skip the proof, since it merely goes by a direct transformation of grammars to recursive hierarchical automata and vice versa. The transformation of the first direction was illustrated in Example 4.24.

Naturally, the run of the recursive hierarchical automaton illustrated in the last example motivates the question about the reachability problem; namely, given two states of such an automaton, is there a sequence of transitions to reach one from the other? The following theorem answers this question indirectly by giving an equivalence between recursive hierarchical automata and pushdown automata. It is then obvious that we can solve the reachability problem, since we have found an efficient solution for the case of pushdown automata.

Theorem 4.29 *For every recursive hierarchical automaton $\mathcal{A}$ an equivalent pushdown automaton can be constructed.*

Remark 4.30

1. *Theorem 4.29 is a variant of the translation from grammars to pushdown automata.*
2. *With some additional effort we obtain variants of Theorem 4.29 with several input and output states.*

Proof (of Theorem 4.29): Given a recursive hierarchical automaton $\mathcal{A} = (K_0, \ldots, K_n)$ with $K_j = (Q_j, BX_j, BI_j, i_j, \Delta_j, o_j)$, we construct a pushdown automaton with state set $Q = \bigcup_{j=0}^{n} Q_j$ and stack alphabet $\{Z_0\} \cup \bigcup_{j=0}^{n} BX_j$. The main idea of the construction is to store the current K_j-state in the control state of the PDA, and to store on the stack the boxes that are called in a LIFO fashion. This means that the box that is called is stored on top of the stack. This way, the correct information about "where to return" when a call is completed is always available. If only the initial stack symbol Z_0

is stored on the stack, then no box call is currently active, and furthermore the current state is in Q_0.

A PDA-transition (p, b, Z, v, q) is added in the following cases:

- For a normal transition $(p, b, q) \in \Delta_j$, we set $v = Z$ (the stack remains unchanged).
- For an entrance transition $(p, \epsilon, B) \in \Delta_j$ into box B from $p \in Q_j$, we set $b = \epsilon$, $v = BZ$ (the box that is called is added on top of the stack for later use), and q is the initial state of K_m, where $m = BI_j(B)$.
- For a return transition $(B, \epsilon, q) \in \Delta_j$ from box $B \in BX_j$, we set $b = \epsilon$, $Z = B$, $v = \epsilon$ (the box that finished its call is erased from the stack), and p is the final state of K_m where $BI_j(B) = m$.
- For a "jump"-transition $(B, \epsilon, B') \in \Delta_j$ from box B to box B', both in BX_j, we set $b = \epsilon$, $Z = B$, $v = B'$, p is the final state of K_m, where $BI_j(B) = m$, and q is the initial state of $K_{m'}$, where $BI_j(B') = m'$.

□

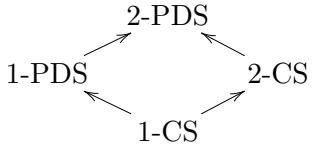
Remark 4.31 *a) We are often not interested in the recognized language, so we suppress the alphabet Σ and merely study recursive hierarchical systems.*

b) As discussed before, the theorem leads to the solution of the (forward and backward) reachability problem for recursive hierarchical systems.

c) The theorem holds even when using bisimulation equivalence instead of language equivalence.

4.4 Undecidability Results

Up to this point we have shown that the reachability problem is decidable both for single pushdown and counter systems. We have also introduced n-pushdown systems that make use of more than one auxiliary memory to perform their computations. To proceed further we can take a generalization step towards two different directions. One way is to extend the 1-CS with a second counter and the other way is to add a second stack to the 1-PDS. The following schema illustrates this idea by representing a generalization by an arrow.



As far as the reachability problem is concerned, the system that is most likely to lead us to an undecidability result is the most complex one, namely the 2-PDS. In the following we first show that this statement indeed holds, and then proceed to proving the same even for a 2-CS, by showing that it has the same expressive power as a 2-PDS. Hence, we gain the insight that both extensions cross the boundaries of decidability and lead us to Turing complete systems.

Theorem 4.32 *The reachability problem is undecidable for both 2-pushdown systems and 2-counter systems.*

Proof: The proof involves three main steps:

1. *From TM to 2-PDS:* Given a TM M we construct a 2-PDS $\mathcal{P}_M$ and configurations c_0, c_1, such that M halts when started on the empty tape iff $\mathcal{P}_M$: $c_0 \vdash^* c_1$.

2. *From 2-PDS to 4-counter systems:* Given a 2-PDS $\mathcal{P}$ and configurations c_0, c_1, we construct a 4-counter system $\mathcal{Z}$ and configurations c_0', c_1' that simulate $\mathcal{P}$, namely: $\mathcal{P}$: $c_0 \vdash^* c_1$ iff $\mathcal{Z} : c_0' \vdash^* c_1'$.

3. *From 4-counter systems to 2-counter systems:* Given a 4-counter system $\mathcal{Z}$ and configurations c_0, c_1, we construct a 2-counter system $\mathcal{Z}'$ and configurations c_0', c_1' that simulate $\mathcal{Z}$, namely: $\mathcal{Z} : c_0 \vdash^* c_1$ iff $\mathcal{Z}' : c_0' \vdash^* c_1'$.

1. To simulate a TM we use the two stacks of a 2-PDS to store the tape content on the left and on the right of the current working cell. Only the immediate (left and right) neighbors of the working cell will be directly accessible, since they will be the current top symbols of the stacks.

Lemma 4.33 *Given a TM M one can construct a 2-PDS $\mathcal{P}_M$ and configurations c_0, c_1, such that M halts when started on the empty tape iff $\mathcal{P}_M$: $c_0 \vdash^* c_1$.*

Proof: Consider a TM $M = (Q, \Gamma, q_0, \Delta, q_f)$ with transitions of the form $(p, a, b, R/L/N, q)$, expressed in colloquial language: "If in state p and reading an a on the working tape, print letter b there, move the head to the right / to the left / don't move, and assume state q". An M-configuration is of the form $[a_1 \cdots a_m p b_1 \cdots b_n]$ with left and right end markers "[" and "]" respectively. The corresponding configuration of the 2-PDS would then be $(p, b_1 \cdots b_n], a_m \cdots a_1[)$. Hence, we construct a 2-PDS $\mathcal{P}_M = (Q \cup (Q \times \Gamma), \Gamma, \Delta')$ with "]" and "[" as the initial stack symbol of the first and the second stack respectively, and allow the following transitions:

- for each non-moving transition (p, a, b, N, q) add a PDS transition $(p, 1, a, b, q)$;
- for each right-moving transition (p, a, b, R, q) add the PDS transitions $(p, 1, a, \epsilon, (q, b))$ and $((q, b), 2, c, bc, q)$ for all $c \in \Gamma$;
- for each left-moving transition (p, a, b, L, q) add the PDS transitions $(p, 2, c, \epsilon, (p, c))$ and $((p, c), 1, a, cb, q)$ for all $c \in \Gamma$.

Furthermore we have to ensure that in the end both stacks of the PDS are empty, so that it accepts exactly when the TM reaches a final state. Hence, we have to add transitions that force the PDS to repeatedly erase the top symbols in both stacks, until it reaches the initial ones ("]" and "["), namely:

$$(q_f, 1, a, \epsilon, q_f) \text{ for } a \neq] \text{ and } (q_f, 2, a, \epsilon, q_f) \text{ for } a \neq [\ .$$

Now we have ensured that M can reach a certain configuration from another one if and only if $\mathcal{P}_M$ can do the same for the corresponding configurations. Formally:

$$M : [upv] \vdash^* [u'qv'] \ \text{ iff } \ \mathcal{P}_M : (p, v\,], u^R[) \vdash^* (q, v'\,], (u')^R[) \ .$$

This holds also if one configuration is the initial one (with an empty tape) and the other is a final one:

$$M : [q_0 \sqcup] \vdash^* [u'q_f v'] \text{ for some } u', v' \ \text{ iff } \ \mathcal{P}_M : (q_0, \sqcup\,], [) \vdash^* (q_f,], [).$$

In other words, M terminates when started on the empty tape if and only if $\mathcal{P}_M$ can reach the configuration $c_1 := (q_f,], [)$ from the configuration $c_0 := (q_0, \sqcup\,], [)$. □

2. To simulate a 2-PDS by a 4-counter system, we simulate each pushdown stack by a 2-counter system. The following lemma proves that this is possible.

Lemma 4.34 *Given a PDS $\mathcal{P}$ and configurations c_0, c_1, one can construct a 2-counter system $\mathcal{Z}_\mathcal{P}$ and configurations c_0', c_1', such that $\mathcal{P} : c_0 \vdash^* c_1$ iff $\mathcal{Z}_\mathcal{P} : c_0' \vdash^* c_1'$.*

Proof: To simulate a pushdown stack by two counters we code a stack content over a stack alphabet with k elements by a $k+1$-adic 0-free number n, carrying the code of the bottom most symbol at the leading position. We only illustrate the proof for the case $k = 9$. Then we can relate each of the letters in Γ to a (different) decimal digit (except 0) and map the stack content $i_0 \cdots i_r$ to the number with the decimal representation $i_r \cdots i_0$.

Example 4.35 Consider an initial stack content 3511. Then

- the stack content 3511 is coded by the number $z = 1153$;
- the top symbol is the remainder of z after dividing it by 10;
- erasing the top symbol of the stack corresponds to a division by 10 without remainder;
- replacing the top symbol of the stack e. g. by 7 corresponds to: division by 10 without remainder, multiplication by 10 and addition of 7;
- replacing the top symbol of the stack e. g. by 27 corresponds to: division by 10 without remainder, multiplication by 10 and addition of 7, multiplication by 10 and addition of 2.

⊠

These arithmetic operations can easily be carried out by using a second counter. We assume that the PDS $\mathcal{P} = (P, \Gamma, \Delta)$ is given in normal form. To simplify the construction of the 2-counter system $\mathcal{Z}$, we continue using $|\Gamma| = 9$. Every $\mathcal{P}$-transition is carried out by a block of $\mathcal{Z}$-transitions (which we will represent by a pseudo program p):

- Erasing the top symbol of the stack (with transition $pa \to q$) is simulated by:
  p: if $\text{rem}(z_1) = a$ then $z_1 := z_1$ div 10, goto q
- Replacing the top symbol of the stack by another symbol (with transition $pa \to qb$) is simulated by:
  p: if $\text{rem}(z_1) = a$ then $z_1 := z_1$ div 10; $z_1 := z_1 * 10$; $z_1 := z_1 + b$; goto q
- Replacing the top symbol of the stack by two symbols (with transition $pa \to qbc$) is simulated by:
  p: if $\text{rem}(z_1) = a$ then $z_1 := z_1$ div 10; $z_1 := z_1 * 10$; $z_1 := z_1 + c$; $z_1 := z_1 * 10$; $z_1 := z_1 + b$; goto q

We give the counter operations only for the second case. The other two are analogous. We describe (almost) all of the four operations of this case below:

- The computation of $\text{rem}(z_1)$ can be done with the auxiliary counter y_1 as follows:
  - Subtract 1 from z_1 and add 1 to y_1, going through the states $0, \ldots, 9$ in a cyclic manner.

  - z_1 becomes 0 on reaching state i iff $\mathrm{rem}(z_1) = i$. Hence we get the remainder in the index of the state.
  - Now the initial value of z_1 is stored in y_1, and we can write it back to z_1 with the opposite operation, but without changing the state.

- Multiplication by 10:
  - We use 10 auxiliary states and repeat the following until $z_1 = 0$: $z_1 := z_1 - 1$ and ten times $y_1 := y_1 + 1$. In other words, we decrease the first counter one by one until it reaches 0, and for each decrease we perform 10 increases of the second counter.
  - Again, we can copy the value of y_1 back to z_1, simultaneously resetting the auxiliary counter to 0.

- Addition of b is easy.

- Division by 10 without remainder is left as an exercise to the reader.

Finally, for the initial configuration $c = (p, i_0 \cdots i_r)$ of the PDS, the corresponding configuration of the 2-counter system is $c' = (p, (i_r \cdots i_0)_{10}, 9)$, where 9 is used as the initial stack (counter) symbol. Then, since we have coded all possible transitions of the PDS by corresponding ones for the 2-counter system, the following holds:

$$\mathcal{P} : c_0 \vdash^* c_1 \text{ iff } Z_{\mathcal{P}} : c_0' \vdash^* c_1' .$$

□

It follows directly from Lemma 4.34 that any 2-PDS $\mathcal{P}$ can be simulated by a 4-counter system $\mathcal{Z}_{\mathcal{P}}$ (actually 3 counters would suffice, since we only need one auxiliary counter). For the initial configuration $c = (p, i_0 \cdots i_r, j_0 \cdots j_s)$ of $\mathcal{P}$, the corresponding configuration of $\mathcal{Z}_{\mathcal{P}}$ is then $c' = (p, (i_r \cdots i_0)_{10}, 9, (j_s \cdots j_0)_{10}, 9)$.

3. The following lemma shows that we can in fact further reduce the number of counters we need.

Lemma 4.36 *For every 4-counter system $\mathcal{Z}$ and configurations c_0, c_1, one can construct a 2-counter system $\mathcal{Z}'$ and configurations c_0', c_1', such that $\mathcal{Z} : c_0 \vdash^* c_1$ iff $\mathcal{Z}' : c_0' \vdash^* c_1'$.*

Proof: To simulate four counters (each with content k_i) by two, we code $(k_1, \ldots, k_4)$ by $\hat{k} := 2^{k_1} \cdot 3^{k_2} \cdot 5^{k_3} \cdot 7^{k_4}$ and store this number in one of the counters. Adding, respectively deleting, 1 to k_1 then corresponds to multiplying, respectively dividing, $\hat{k}$ by 2, and analogously for $k_2, \ldots, k_4$.

Again these operations can be computed using the second counter as an auxiliary memory. □

Now that we can simulate a TM by a counter system with two counters, this suffices to show that the reachability problem is undecidable even for 2-counter systems. □

A consequence of Theorem 4.32 is that a programming language with two integer variables X_1, X_2, instructions $X_i := X_i + 1$, $X_i := X_i - 1$, and IF $X_i = 0$ THEN JUMP TO (line) s is already Turing complete, i. e. powerful enough to simulate every Turing machine.

4.5 Retrospection: The Symbolic Method

In this section we look back at the way we solved the reachability problem for the different models of automata theory, but this time in a more abstract sense. First of all, it is interesting to note why the reachability problem is so important in system design. The typical problem that we face in such cases is that we are given a transition system (e. g. a graph) and some set of states or configurations (e. g. vertices) in this system. Then, we want to find the transitive closure under predecessor of this set, i. e. we want to extend this set in order to additionally include all those states from which, after a finite number of steps, we can reach the initial designated set. Imagine that the designated set includes all those states that we want a run to avoid (deadlocks, endless loops, conflicts, etc.). Then of course we also want to avoid the states from which a visit to the "malicious" states can be forced. This is a typical kind of safety constraint that is checked in the framework of *Model Checking*.

Returning once again to the example of a graph, we know that there is a very efficient way to solve this problem, namely by a backwards breadth-first search in time $\mathcal{O}(|V|+|E|)$, where V and E are the sets of vertices and edges of the graph respectively. In fact we cannot hope for a more efficient result. However, if we consider for example a circuit with 100 flip-flops (nothing extraordinary nowadays), the number of different states in this system is 2^{100}, which is also the number of steps that our very efficient algorithm has to take before issuing the result. Hence, even if our method is as good as it gets, it is still not satisfactory enough for large systems.

Since we cannot improve the algorithm itself, our only way out is to improve the input. Strictly speaking, we have to find a way to improve the method by which the systems are coded before they are given as an input to an algorithm. This is exactly the idea of the *symbolic method*; instead of enumerating a set of states we can describe it (e. g. by some logic formalism or an automaton) using significantly less space. Then, we can perform a reachability analysis by successively computing the descriptions of C, $pre(C)$, $pre^2(C)$ etc., until no new states can be added anymore. Note

that this is definitely going to be the case sometime, because the overall number of available states in the system is finite anyway.

In the algorithm that is presented below we actually manipulate a formula or an automaton until it describes exactly the set that we are looking for. We denote

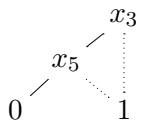
- by β an abstract description (formula or automaton), and
- by $[\beta]$ the set defined by β.

The symbolic algorithm for backwards reachability takes the following steps:

1. $\beta :=$ description of C;
2. $\beta' :=$ description of $[\beta] \cup \{q \in Q \mid$ there is a transition from q to $[\beta]\}$;
3. if $[\beta] = [\beta']$, terminate with output β as a description of $pre^*(C)$; otherwise $\beta := \beta'$ and back to 2.

We already know that in the case of finite systems on the state space $\{0,1\}^n$, it is very convenient to describe sets by boolean formulas $\beta(x_1, \ldots, x_n)$.

Example 4.37 Consider the state set $Q = \{0,1\}^5$ and the designated set of configurations $C = \{q \in Q \mid$ 3rd or 5th component $= 0\}$. The set C can be described by $\beta = \neg x_3 \vee \neg x_5$. In Section 2.2 we studied a way to construct a minimal acyclic automaton for such a boolean formula, namely OBDDs. The OBDD for (the description of) C is shown below.



⊠

It is obvious that it is much easier to apply the reachability algorithm on the level of OBDDs than to apply it on enumerated sets. To achieve this, two steps of the algorithm presented above still remain to be implemented:

1. $\beta' :=$ description of $[\beta] \cup \{q \in Q \mid$ there is a transition from q to $[\beta]\}$;
2. the equivalence test ("$[\beta] = [\beta']$?").

At this point let us recall that this is exactly what we were striving towards when we were trying to solve the reachability problem for pushdown systems. To describe the given set C we used a P-automaton. To implement the extension to the transitive closure (under predecessor) of C, we

gradually exhausted the states belonging strictly to $pre^*(C)$ by adding new transitions to the P-automaton. Furthermore, because we used such a structure, we were not concerned with an equivalence test. The termination of the algorithm depended on whether it was possible to add new transitions to the automaton. What we finally observe is that the reachability analysis of pushdown systems is performed according to the symbolic method, by using NFAs for the representation of state sets.

We conclude this chapter about pushdown systems by noting that these are a first essential step in the verification of infinite systems. In such systems we look at two words u, v, e. g. representing two configurations, and we want to examine all ways that may lead from the first to the second. To change the first letter of one word and replace it by another (or two) is only a special case.

4.6 Exercises

Exercise 4.1 Show that every PDA $\mathcal{A}$ is (language) equivalent to a PDA $\mathcal{B}$ that uses only purely expanding and purely reducing transitions.

Exercise 4.2 Let $\mathcal{P} := (P, \Gamma, \Delta)$ be a pushdown system with $P := \{p_0, p_1\}$, $\Gamma := \{a, b, c\}$ and the following set Δ of transitions:

$$p_0 a \to p_0 \qquad p_0 b \to p_1 c$$
$$p_0 c \to p_1 \qquad p_1 c \to p_0 bb$$

Consider the configuration $c_1 = p_0 aac$. As in the lecture $\{c_1\}$ is accepted by an NFA $\mathcal{A}$ with states p_0, p_1, s_1, s_2, s_3. Apply the saturation procedures to compute the automata $\bar{\mathcal{A}}$ and $\tilde{\mathcal{A}}$ accepting $pre^*\{c_1\}$ and $post^*\{c_1\}$, respectively.

Exercise 4.3 A 1-register machine R *decides* a set $M \subseteq \mathbb{N}$, if R, initialized with $x \in \mathbb{N}$, always terminates and moreover with value 1 assigned to X_1 in case $x \in M$, and with value 0 assigned to X_1 in case $x \notin M$.

Show that if M is eventually periodic, then M is decided by a 1-register machine.

(M is eventually periodic, if for suitable k_0, p the following holds:

$$\text{for all } x \geq k_0 : x \in M \Leftrightarrow x + p \in M \ .)$$

Exercise 4.4 * Show that the following problem is decidable: Given a pushdown automaton $\mathcal{A}$, does there exist a sequence $q_0 Z_0 \vdash q_1 w_1 \vdash q_2 w_2 \vdash \ldots$ of configurations such that the length of the configurations is unbounded, i. e. for every $n \in \mathbb{N}$ there is an i such that $|q_i w_i| > n$?

Hint: Consider the repetitions and the following parameters: The number of states n, the number of symbols in the stack alphabet m, and the maximal length of a word v in a transition $(p, a/\epsilon, Z, v, q)$.

Exercise 4.5 In the lecture we applied the saturation algorithm to compute the set $pre^*(C)$ to P-automata whose states from P have no incoming transitions. Show that this assumption is necessary.

For this purpose, find a pushdown system $\mathcal{P} := (P, \Gamma, \Delta)$, a set C of configurations and a P-automaton $\mathcal{A}$ that recognizes C and has incoming transitions into states in P, such that the saturation algorithm computes no automaton for the set $pre^*(C)$ when applied to $\mathcal{A}$.

Exercise 4.6 Construct a recursive automaton that recognizes the language generated by the following context-free grammar:

$$\begin{array}{lcl} S & \to & aB \mid bA \\ A & \to & a \mid aS \mid bAA \\ B & \to & b \mid bS \mid aBB \end{array}$$

Give a run of the automaton on the word *bbabaa*.

Exercise 4.7* Show that the following problem is undecidable.

Given: A 2-pushdown system $\mathcal{P}$ and a configuration c of $\mathcal{P}$.

Question: Does a number n exist, such that $|c'| < n$ holds for all configurations $c' \in post^*(c)$?

The length of the configuration is the sum of the lengths of both stack contents.

Hint: For a reduction the halting problem for offline Turing machines is recommended, which can be assumed to be undecidable. (An offline Turing machine has a read-only input tape and a read-write working tape.)

Chapter 5

Communicating Systems

In this chapter different types of automata are introduced which allow to model parallel and distributed computations directly rather than encoding the behavior of such a system into a single finite state device. We gain a succinct representation of parallel processes, but also show that one has to be careful not to include too powerful communication possibilities to maintain the nice properties of finite state machines.

5.1 Synchronized Products

Recall the product construction for NFAs. We generalize this construction by distinguishing which actions are relevant for which automata.

Definition 5.1 (Synchronized product) Let $\mathcal{A}_i = (Q_i, \Sigma_i, q_{0i}, \Delta_i, F_i)$ be NFAs for $1 \leq i \leq n$.. The *synchronized product* of $\mathcal{A}_1, \ldots, \mathcal{A}_n$ is the automaton

$$\mathcal{A}_1 \circ \ldots \circ \mathcal{A}_n := (Q_1 \times \ldots \times Q_n, \Sigma_1 \cup \ldots \cup \Sigma_n, (q_{01}, \ldots, q_{0n}), \Delta, F_1 \times \ldots \times F_n)$$

with

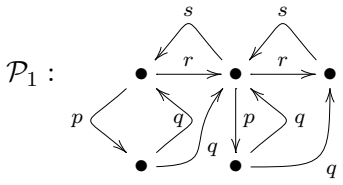
$$\big((p_1, \ldots, p_n), a, (q_1, \ldots, q_n)\big) \in \Delta \;:\Leftrightarrow\; \begin{cases} (p_i, a, q_i) \in \Delta_i & \text{for } a \in \Sigma_i \text{ and} \\ p_i = q_i & \text{for } a \notin \Sigma_i. \end{cases}$$

In binary products we usually emphasize the synchronizing alphabet $\Sigma_0 := \Sigma_1 \cap \Sigma_2$ by writing $\mathcal{A}_1 \circ_{\Sigma_0} \mathcal{A}_2$.

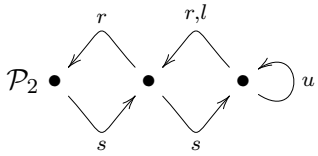
Example 5.2 We model a producer–consumer system by a synchronized product.

1st process: This process models a shipbuilder with two sites for storing ships. Possible actions are: **p**roduce ship, check the **q**uality, **s**ell a ship, and **r**etake a ship. This process can be implemented by the following

NFA, where the upper three states indicate whether there are 0 (left), 1 (middle), or 2 ships (right) ready to be sold.



2nd process: This process models a very cautious ship owner who uses a ship only if she possesses two. Possible actions are: buy (**s**ell for 1st process), return a ship (**r**etake for 1st process), **u**se ship, and **l**ose a ship. This process can be implemented by the following NFA.



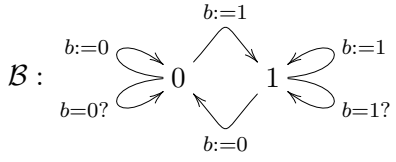
The synchronized product $\mathcal{P}_1 \circ_{\{r,s\}} \mathcal{P}_2$ of these automata models the interaction between the shipbuilder and the ship owner. It has 15 states and thus cannot be represented as succinct as the component automata. ⊠

Example 5.3 In this example we model a mutual exclusion protocol called *Peterson's Protocol* where two processes need access to a critical section. The processes are the following ones:

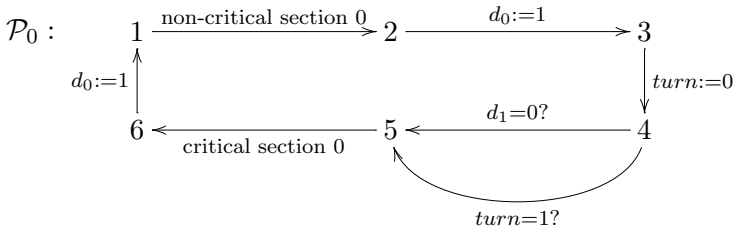
	repeat		**repeat** :
	1. (non-critical section)		*1.* (non-critical section)
	2. $d_0 := 1$		*2.* $d_1 := 1$
$\mathcal{P}_0$:	*3.* $turn := 0$	$\mathcal{P}_1$:	*3.* $turn := 1$
	4. wait ($d_1 = 0$ or $turn = 1$)		*4.* wait ($d_0 = 0$ or $turn = 0$)
	5. (critical section)		*5.* (critical section)
	6. $d_0 := 0$		*6.* $d_1 := 0$

The protocol uses the Boolean variables d_i and $turn$. Process P_i sets d_i to 1 and $turn$ to i iff it request the critical section. The variable $turn$ ensures that access to the critical section alternates if both processes repeatedly pose requests.

We represent the protocol by a synchronized product using a two-state automaton $\mathcal{B}$ for each Boolean variable b

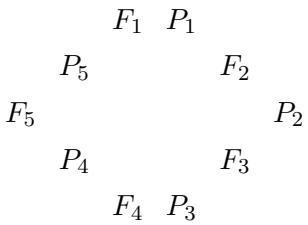


and the following automaton for $\mathcal{P}_0$ ($\mathcal{P}_1$ is defined analogously):

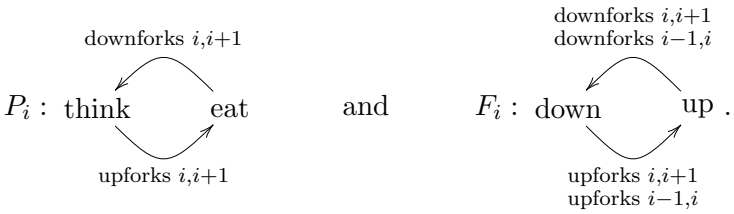


Peterson's Protocol is modeled by the synchronized product $\mathcal{P}_0 \circ \mathcal{P}_1 \circ \mathcal{D}_0 \circ \mathcal{D}_1 \circ \mathcal{T}$. Note that the expanded product automaton has $2^3 \cdot 6^2 = 288$ states! ⊠

Example 5.4 We model the famous *Dining Philosophers Problem* as a synchronized product. The problem is the following: 5 philosophers are sitting at a table, and there is a fork in between two of them (as shown below). Each philosopher either thinks or eats, and to eat she needs both of her neighboring forks, i.e. philosopher i needs forks i and $i+1 \pmod 5$.



A simplified version where we assume that every philosopher picks both forks simultaneously can be modeled by the synchronized product of



⊠

The synchronized product of the transition systems presented in the previous example is only a model for all possible actions (transitions) that can occur and for all possible situations (states), in which the group of philosophers can find itself at any moment in time. To really describe *the problem* of the dining philosophers (there are several versions of it) we have to construct a protocol that specifies a code of behavior between them by allowing and forbidding certain actions. Such a protocol would for example force the following constraints:

- A deadlock must not occur; that is from every reachable situation, an action of at least one philosopher is allowed.
- Every philosopher will again and again obtain the forks that are accessible by him (liveliness condition).
- By introducing an intermediate state "hungry" we may require that whenever a philosopher gets hungry, he will eventually obtain his forks.

Conditions like the ones above are not modeled in the transition rules of the system. They are posed on the behavior of the system, i.e. on the possible paths through the graph. A path can be coded by a word, and thus the condition will specify a subset of all possible words, i.e. a language. The first condition stated above is for example a typical reachability condition and hence can be expressed by the techniques on finite words that we have learned so far. For the last condition above one has to examine infinite executions of the system and thus this cannot be expressed by the same techniques.

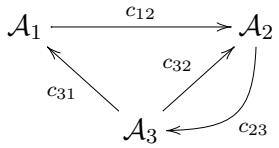
In the general case the solution and verification of such problems is realized in three different levels. The synchronized product is constructed to describe a language L over the action alphabet representing the possible behavior. The required behavior of the communicating components is described by an automaton for some language K. A solution (protocol) is another automaton recognizing a sub-language $L_0 \subseteq K$. Usually L and K are given, and we have to follow an efficient method to determine L_0. Although the synchronized product offers a very compact description of a system with say k components of n states each, the "state space explosion" to the number n^k is generally inevitable. Techniques like the symbolic method are unfortunately not always helpful, and this is one of the main problem of system verification nowadays.

5.2 Communication via FIFO Channels

In this section we introduce finite state machines which communicate with each other via channels. A channel implements a FIFO queue, i.e messages are processed in first-in first-out order. We represent the content of a queue by a word $a_1 \dots a_n$ where letters added to the queue are appended to the left and read from the right.

The basic idea for two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ and channels c_{12} and c_{21} is the following: $\mathcal{A}_1$ sends messages to $\mathcal{A}_2$ via channel c_{12} and, vice versa, $mathcal A_2$ sends messages to $\mathcal{A}_1$ using channel c_{12}. The content of a channel c_{ij} is a word over a message alphabet Γ. In general automata $\mathcal{A}_1, \dots, \mathcal{A}_n$ communicate with each other via a set of channels $CN \subseteq CN_n := (\{1, \dots, n\} \times \{1, \dots, n\}) \setminus \{(1,1), \dots, (n,n)\}$. For example, if $CN =$

$\{(1,2),(3,1),(3,2)(2,3)\}$ we obtain the following communication structure:



Definition 5.5 A *communicating finite state machine* (CFSM) is a tuple $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, \Gamma, CN)$ where $CN \subseteq CN_n$, Γ is a message alphabet, and for $1 \leq i \leq n$ $\mathcal{A}_i = (Q_i, \Sigma_i, q_{0i}, \Delta_i, F_i)$ is a finite automaton with transitions of the form

- $(p, a, q) \in Q_i \times \Sigma_i \times Q_i$, (local)
- $(p, m!j, q)$ where $p, q \in Q_i,\ m \in \Gamma,\ (i,j) \in CN$, and (write)
- $(p, m?j, q)$ where $p, q \in Q_i,\ m \in \Gamma,\ (j,i) \in CN$. (read)

A *configuration* of a CFSM $\mathcal{A}$ is a tuple $\big((p_1, \ldots, p_n), (w_{ij})_{(i,j)\in CN}\big)$ where $p_i \in Q_i$ and $w_{ij} \in \Gamma^*$ is content of channel c_{ij}. A configuration of $\mathcal{A}$ is also called a *global state*.

The semantics of the transitions is defined as follows:

$$(\bar{p}, (w_{ij})) \vdash (\bar{q}, (u_{ij})) :\Leftrightarrow \text{ there exists a } k \text{ such that}$$

$$\begin{cases} (p_k, a, q_k) \in \Delta_k \text{ and} & p_i = q_i \text{ for } i \neq k, \\ & u_{ij} = w_{ij} \text{ for } (i,j) \in CN \\ (p_k, m!j, q_k) \in \Delta_k \text{ and} & p_i = q_i \text{ for } i \neq k, \\ & u_{kj} = m w_{kj}, u_{ii'} = w_{ii'} \text{ for } (i,i') \neq (k,j) \\ (p_k, m?j, q_k) \in \Delta_k \text{ and} & p_i = q_i \text{ for } i \neq k, \\ & u_{jk} m = w_{jk}, u_{ii'} = w_{ii'} \text{ for } (i,i') \neq (j,k) \end{cases}$$

A transition (p, a, q) of an automaton $\mathcal{A}_i$ is interpreted as usual, the transitions $(p, m!j, q)$ and $(p, m?j, q)$ are used to write, respectively read, a letter from the corresponding channel. Note that with this definition of a CFSM only one component automaton $\mathcal{A}_i$ is active at a time. However this can be easily generalized to real parallel processing.

Remark 5.6 *IF a CFSM has* bounded channels*, i.e. in every execution of the system the length of the channel contents is bounded by some $l \in \mathbb{N}$, then the number of configurations is bounded by $|Q_1| \cdot \ldots \cdot |Q_n| \cdot (|\Gamma| + 1)^{|CN| \cdot l}$. Hence such a CFSM is equivalent to an NFA.*

A configuration c of a CFSM $\mathcal{A}$ is *reachable* from the initial configuration $(\bar{q_0}, \bar{\epsilon})$ (where $\bar{q_0} = (q_{01}, \ldots, q_{0n})$) if $(\bar{q_0}, \bar{\epsilon}) \vdash \ldots \vdash c$. Let C be a set of configurations of a CFSM $\mathcal{A}$. If the languages of the channel contents from

C are regular, then C can be finitely represented by tuples consisting of a control state and a regular expression r_{ij} for the content of channel c_{ij}. We call such a set a *regular set* of configurations.

The *reachability problem* for CFSMs is the following:

Input:	A CFSM $\mathcal{A}$, a regular set C of configurations.
Problem:	Is a configuration $c \in C$ reachable from $(\bar{q}_0, \bar{\epsilon})$?

Theorem 5.7 *The reachability problem for CFSMs is undecidable.*

Proof: We reduce the halting problem for Turing Machines to the reachability problem for CFSMs. Remember that instructions of a Turing machine M are of the form $(s, a, a', *, s')$ where s, s' are states of M, a, a' are symbols from the tape alphabet, and $* \in \{L, R, N\}$ is the direction in which the head of M moves. A configuration of M is a word $\flat usv\$$ where $\flat$ and $\$$ are the left and right end markers of the tape inscription, u is the word to the left of the head of M, v is the word to the right of it (including the symbol beneath the head), and s is the current state of M. We assume that M only stops if a special state s_{fin} is reached.

We define an effective transformation of a Turing machine M into a CFSM $\mathcal{A}_M$ where $\mathcal{A}_M = (\mathcal{A}_1^M, \mathcal{A}_2^M, \{(1,2),(2,1)\})$ such that M stops when started on the empty tape iff $\mathcal{A}_M$ reaches a configuration in a set C to be defined later on.

The idea is that $\mathcal{A}_1^M$ puts the current configuration of M into channel c_{12}. $\mathcal{A}_2^M$ reads this configuration, applying changes to it according to the instructions of M and sends the new configuration back to $\mathcal{A}_1^M$ using channel c_{21}. More precisely:

To start the simulation of M, $\mathcal{A}_1^M$ puts the initial configuration $\flat s_0 \hbar\$$ into channel c_{12}, and proceeds to state p_0. From thereon $\mathcal{A}_1^M$ will write every letter read from c_{21} to channel c_{12}. $\mathcal{A}_2^M$ behaves as follows

1. Starting from the initial state read the first three letters a_1, a_2, a_3 from c_{12} and store them in a state $[a_1, a_2, a_3]$.

2. If a_2 is not a state symbol then

   - if $a_3 = \$$ write a_1, a_2, a_3 to c_{21} and enter the initial state again.
   - else write a_1 to c_{21}, read a letter b from c_{12} and update the state to $[a_2, a_3, b]$.

3. If $[a_1, a_2, a_3] = [a, s, b]$ then 0

   - if $b = \$$ write a to c_{21} and enter state $[s, \hbar, \$]$
   - else if

     (i) $(s, b, b', N, s') \in \Delta_M$ proceed to state $[a, s', b']$

(ii) $(s, b, b', R, s') \in \Delta_M$ proceed to state $[a, b', s']$
(iii) $(s, b, b', L, s') \in \Delta_M$ proceed to state $[s', a, b']$ if $a \neq \flat$, else write $\flat$ to channel c_{21} and enter state $[s', \hbar, b']$.

Let now C be the set of configurations of the communication state machine $\mathcal{A}_M$ in which s_{fin} occurs in channel c_{12}. This set is obviously regular. We obtain that the TM M started on the empty tape reaches the stop state s_{fin} iff $\mathcal{A}_M$ reaches from the initial configuration $((q_{01}, q_{02}), \epsilon, \epsilon)$ a configuration in C. Hence the reachability problem for CFSMs is undecidable. □

This proof can be easily modified such that one single automaton with a channel to itself suffices to simulate Turing machines. We therefore conclude that the reachability problem is decidable for systems with an unbounded LIFO memory (stack), but undecidable for systems with an unbounded FIFO channel.

For the boundedness problem we experience a likewise situation. In exercise 4.4 we proved that this problem is decidable for pushdown systems. This is unfortunately not the case for a FIFO-communicating automaton.

Theorem 5.8 *The problem, whether a FIFO-communicating automaton has bounded channels starting from a configuration c, is undecidable.*

The idea is to consider the decidability of the boundedness problem for the case of a Turing machine. Since we have already found a way to simulate TMs by CFSMs, the problem is decidable for the former if and only if it is decidable for the latter.

Lemma 5.9 *The boundedness problem for Turing Machines "Given a TM M and a TM-configuration c, is M bounded starting from c?" is undecidable.*

Proof: We call a TM M bounded starting from c, in case M reaches only finitely many cells of the working tape, when starting from c. This may happen because of two reasons; either M will eventually halt if started from c, or it will not halt but will move in a bounded area by visiting again and again the same cells. We know that the first case in undecidable, but we are not sure about the second. For this reason we construct a new TM M' that simulates M in such a way that M terminates starting from c iff M' is bounded starting from c. This can be done easily by instructing M' to work exactly as M and additionally replace the right end marker \$ by $\hbar$\$ after each step. Then obviously M halts starting from c iff M' is bounded starting from c. □

Proof of Theorem 5.8: We use the previous lemma and the construction from the proof of Theorem 5.7. Thus, given M and c we can construct a FIFO-communicating automaton $\mathcal{A}_M$ such that M is bounded starting from

c iff $\mathcal{A}_M$ is bounded starting from a configuration where initially c is stored in channel c_{12}. □

5.3 Message sequence charts

Message sequence charts are a simple formalism for the notation of asynchronous communicating processes, their syntax settled in the International Standard Z120.

Definition 5.10 A *Message Sequence Chart* (MSC) C is given by

- a finite set $\{P_1, \ldots, P_n\}$ of processes,
- a finite set $M = \{m_1, \ldots, m_k\}$ of messages, where each m_i involves a send event s_i and a receive event r_i and $E = \{s_1, \ldots, s_k, r_1, \ldots, r_k\}$ is the set of all events,
- a label function $l : E \to \{P_1, \ldots, P_n\}$, assigning to each event a process such that $l(s_i) \neq l(r_i)$, and
- for every $1 \leq i \leq n$ a total order $<_i$ defined on the set $E_i := \{e \in E \mid l(e) = i\}$ of all events belonging to process i (called *visual order*)

Example 5.11 An MSC is shown in Figure 5.1. In this MSC $P = \{P_1, P_2, P_3\}$ and e.g. $E_2 = \{r_1, s_2, s_3, r_4, s_5\}$ with $r_1 <_2 s_2 <_2 s_3 <_2 r_4 <_2 s_5$.

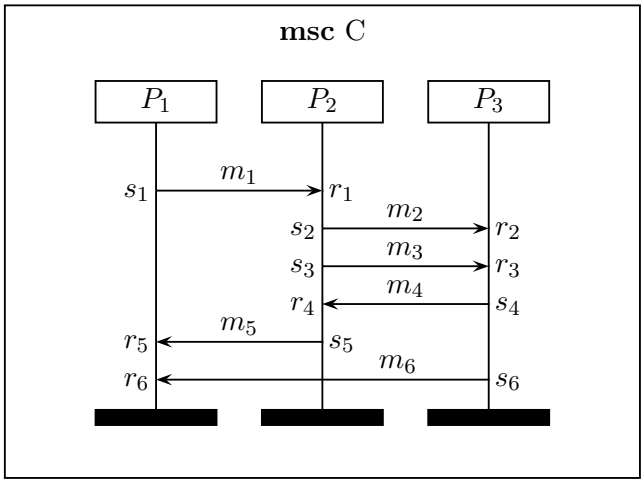


Figure 5.1: MSC from Example 5.11

⊠

Our aim is to assign a semantics to MSCs by defining a partial order on the send and receive events. To do this we need some facts on partial orders as a preparation.

Definition 5.12 A *partial order* over a set A is a relation $< \subseteq A \times A$ such that

- $\neg(a < a)$ for all $a \in A$ ($<$ is irreflexive),
- $(a < b) \wedge (b < c) \Rightarrow a < c$ for all $a, b, c \in A$ ($<$ is transitive), and
- $\neg\big((a < b) \wedge (b < a)\big)$ for all $a, b \in A$ ($<$ is acyclic).

Obviously every partial order gives rise to an acyclic graph. Conversely, every acyclic graph $(A, \rightarrow)$ induces a partial order $(A, <)$ with $a < b \ :\Leftrightarrow$ there is a non-empty $\rightarrow$-path from a to b.

Definition 5.13 Let $(A, <)$ be a partial order. The *Hasse diagram* $(A, \lessdot)$ of $(A, <)$ is the acyclic graph defined by

$$a \lessdot b :\Leftrightarrow \ a < b \text{ and there exists no } c \in A : a < c < b$$

A *linearization* of a partial order $(A, <)$ is a total order $(A, \prec)$ such that

$$a < b \ \Rightarrow \ a \prec b.$$

A linearization of a partial order can be obtained algorithmically by a topological sort based on depth-first search.

Example 5.14 Let $A :=$ {pants, trousers, socks, shoes, watch, belt, shirt, tie, jacket}. Figure 5.2 shows a Hasse diagram for A such that the associated partial order allows to dress correctly. The associated partial order is defined by $a < b$ iff there exists a nonempty path from a to b in the Hasse diagram for A.

A possible linearization of this partial order is the following:

$$\text{watch} < \text{shirt} < \text{tie} < \text{pants} < \text{trousers} < \text{belt} < \text{jacket} < \text{socks} < \text{shoes}.$$

⊠

Definition 5.15 Let C be an MSC. The *semantics* of C is the partial order on the send and receive events of C induced by the acyclic graph $(E, \rightarrow)$ defined by

$$\begin{aligned} e \rightarrow e' \ :\Leftrightarrow \ & e = s_j \text{ and } e' = r_j, \text{ or} \\ & e <_i e' \text{ and at least one of them is send event, or} \\ & e = r_j,\ e' = r_k \text{ with } l(r_j) = l(r_k) \text{ and } s_j <_i s_k. \end{aligned}$$

A *race* of an MSC C is a pair of receive events r, r' such that $r <_i r'$ for some i, but not $r < r'$ in the partial order induced by $(E, \rightarrow)$.

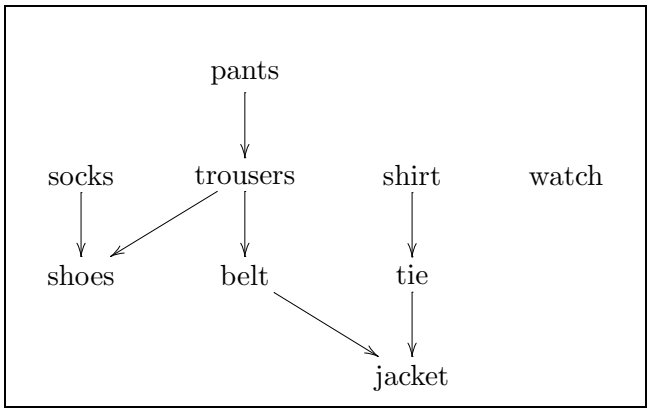


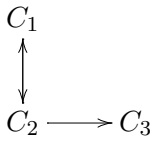
Figure 5.2: Hasse Diagram for Example 5.14

Example 5.16 Figure 5.3 shows the Hasse diagram of the partial order defined by the MSC C from Figure 5.1. Note that r_5 and r_6 are not ordered in the partial order induced by $(E, \rightarrow)$, but $r_5 <_1 r_6$, i.e. r_5 and r_6 are a race of C. ⊠

An important task when dealing with MSCs is to find all races. The standard procedure is to construct the acyclic graph $(E, \rightarrow)$ from Definition 5.15, then compute its transitive closure $(E, <)$, and compare this with the visual orders $<_i$.

MSCs can be generalized to *higher level message sequence charts* (HMSCs), where a HMSC is a directed graph composed of MSCs. A HMSC defines a family of MSCs, obtained from paths in the HMSC graph by concatenating the component MSCs.

Example 5.17 Let a HMSC be given by the following graph with component MSCs as shown in Figure 5.4:



An expansion of this HMSC is shown in Figure 5.5, where the dashed lines indicate that the part between label 1 and 2 can be repeated arbitrarily often afterwards. We obtain a family of MSCs with an unbounded set of races (for process P_3).

⊠

The previous example leads us to partial orders with a larger and larger number of nodes. There is a special theory that deals with such partial

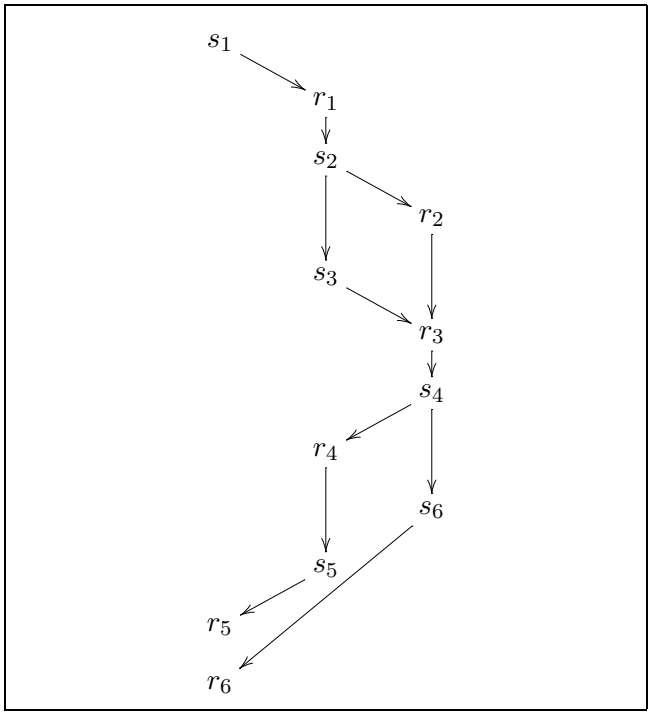


Figure 5.3: Hasse Diagram for the MSC of Figure 5.1

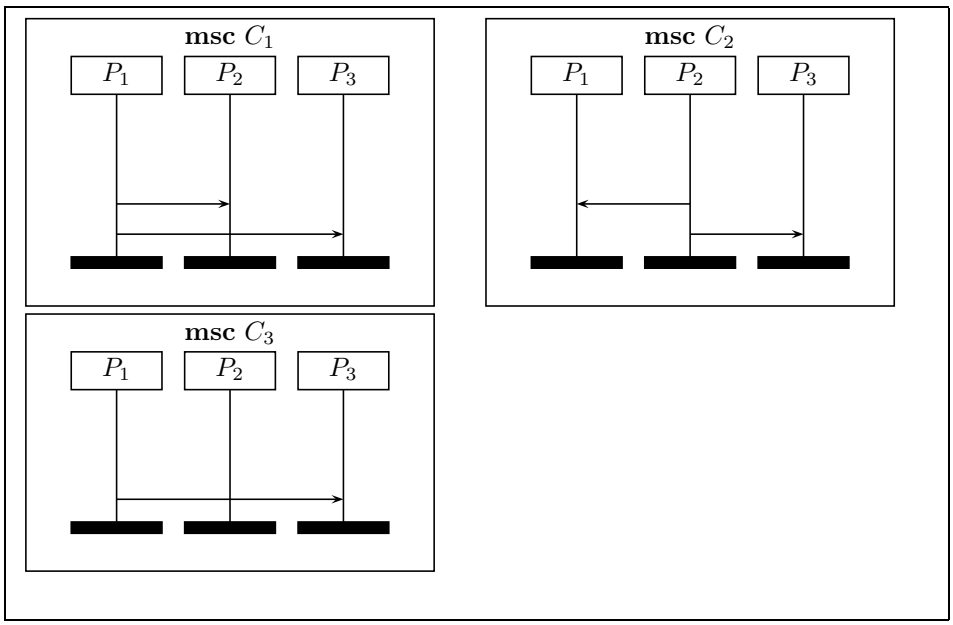


Figure 5.4: Component MSCs for Example 5.17

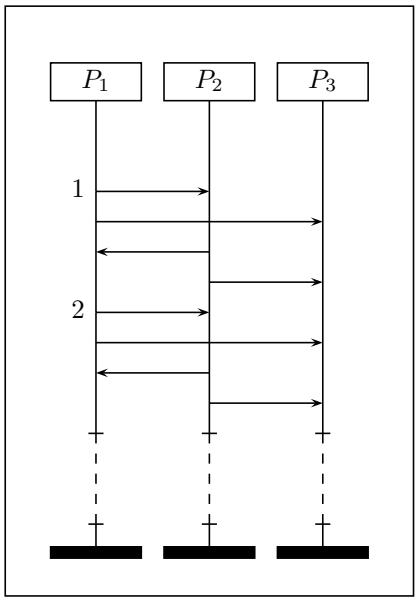


Figure 5.5: Expanded HMSC from Example 5.17

orders as *trace languages*. These represent a generalization of words to labeled partial orders. They are defined by an *independence relation* on a *concurrency alphabet*. Such an alphabet is of the form (Σ, I) with a set of letters Σ and an irreflexive relation $I \subseteq \Sigma \times \Sigma$. A pair $(a, b) \in I$ implies that two consecutive occurrences of a and b can be switched. Then, every word $w \in \Sigma^*$ defines an equivalence class $[w]$ which is the set of words which can be obtained from w by a sequence of switches of neighboring letters a, b with $(a, b) \in I$. We call $[w]$ a *trace* ("Mazurkiewicz trace").

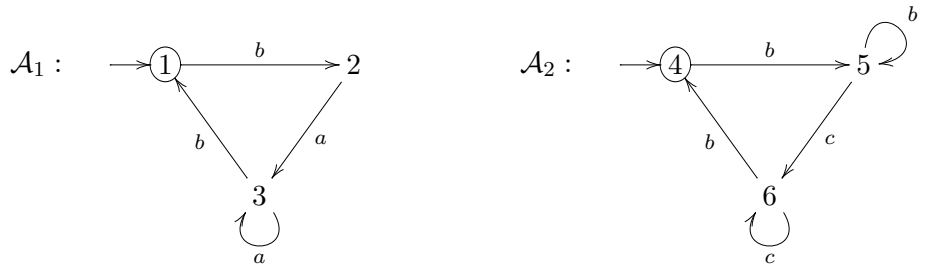
Example 5.18 Let D be the relation $(\Sigma \times \Sigma) \setminus I$. D describes the "dependency relation" and contains all pairs (a, b) whose order must not be switched. (Σ, D) can by given as an undirected graph, e.g. by

$$\begin{array}{cc} a & d \\ | & | \\ b & - c \end{array}$$

In this case, if $w = adabcd$, then $aadbcd \in [w]$ and $aabdcd \in [w]$. ⊠

5.4 Exercises

Exercise 5.1 Consider the following automata $\mathcal{A}_1$ over $\{a.b\}$ and $\mathcal{A}_2$ over $\{b, c\}$.



(a) Draw the graph of the synchronized product $\mathcal{A}_1 \circ_b \mathcal{A}_2$, disregarding the states that are not reachable from the initial state $(1, 4)$.

(b) Describe (informally or by a regular expression) the language accepted by $\mathcal{A}_1 \circ_b \mathcal{A}_2$.

Exercise 5.2 Consider the synchronized product modeling the five philosophers, with automata $\mathcal{P}_1, \ldots, \mathcal{P}_5$ and $\mathcal{F}_1, \ldots, \mathcal{F}_5$ as explained in this chapter. Determine the number of states of this synchronized product. Give an upper bound (as small as you can justify) on the number of reachable states from the initial states "think" for the philosophers and "down" for the forks.

Exercise 5.3 (a) Consider the language L defined by the regular expression $(b^+ac^+a)^*$. Find an NFA $\mathcal{A}_1$ over $\{a, b\}$ and an NFA $\mathcal{A}_2$ over $\{a, c\}$ such that L is accepted by the synchronized product $\mathcal{A}_1 \circ_a \mathcal{A}_2$.

(b) Show that no NFAs $\mathcal{A}_1, \mathcal{A}_2$ as in (a) exist such that the language $(b^+ac^+a)^* + (c^+ab^+a)^*$ is accepted by their synchronized product $\mathcal{A}_1 \circ_a \mathcal{A}_2$.

Exercise 5.4 Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be synchronized products of NFAs, each consisting of m factor NFAs with n states each. Describe a procedure to test the equivalence of $\mathcal{A}_1$ and $\mathcal{A}_2$ and give a bound on the time complexity in terms of m and n.

Exercise 5.5 (a) Develop a CFSM $\mathcal{A}$ that models the "Five Philosophers Problem". $\mathcal{A}$ shall consist of component automata $\mathcal{P}_1, \ldots, \mathcal{P}_5$ for the philosophers and $\mathcal{F}_1, \ldots, \mathcal{F}_5$ for the forks, which are lifted separately here. $\mathcal{P}_i$ shall be connected via forward and backward channels to both $\mathcal{F}_i$ and $\mathcal{F}_{i+1}$. In particular your model has to ensure that

(i) no fork can be lifted by two philosophers simultaneously,

(ii) a philosopher can eat iff she possesses both of her forks,

(iii) a philosopher can think iff she possesses no fork, and

(iv) once a fork is requested, a philosopher is not in state "think" anymore.

Explain your construction.

(b) Modify your CFSM such that a philosopher is allowed to drop her request for a fork (because the fork is in use) and go back to state "think".

Exercise 5.6 In the course it was shown that CFSMs consisting of a single component automaton $\mathcal{A}$ with a channel to itself induces an undecidable reachability problem. ("Given such an $\mathcal{A}$, can it reach from an initial configuration (q, ϵ) a configuration (p, w) with control state p and channel content w?")

Analyze this problem for the restricted case of a CFSM with an one-letter message alphabet $\Gamma = \{m\}$. Justify your answer.

Exercise 5.7 Consider the message sequence chart from Figure 5.6.

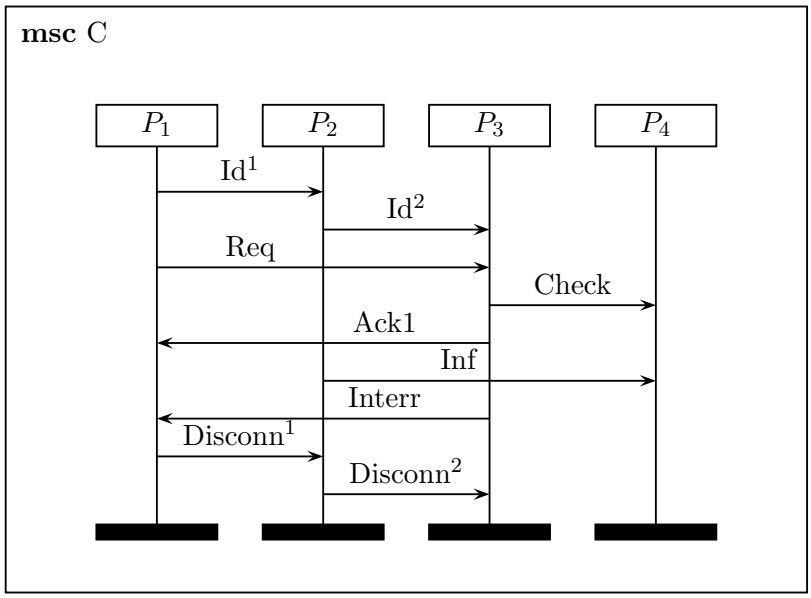


Figure 5.6: MSC for Exercise 5.7

We denote the send events by $s(Id^1), s(Id^2), \ldots$ and the receive events by $r(Id^1), r(Id^2), \ldots$.

(a) Draw the Hasse diagram for the partial order associated with C.

(b) List the races occurring in C.

(c) Give three linearizations of the events of C.

Chapter 6

Petri Nets

6.1 Basic Definitions

Petri nets are models for concurrent systems and their behavior.

Definition 6.1 A *Petri net* is a tuple $N = (P, T, F)$ where P and T are nonempty, finite and disjoint sets of *places* and *transitions*, respectively, and $F \subseteq P \times T \cup T \times P$ is a *flow relation*.

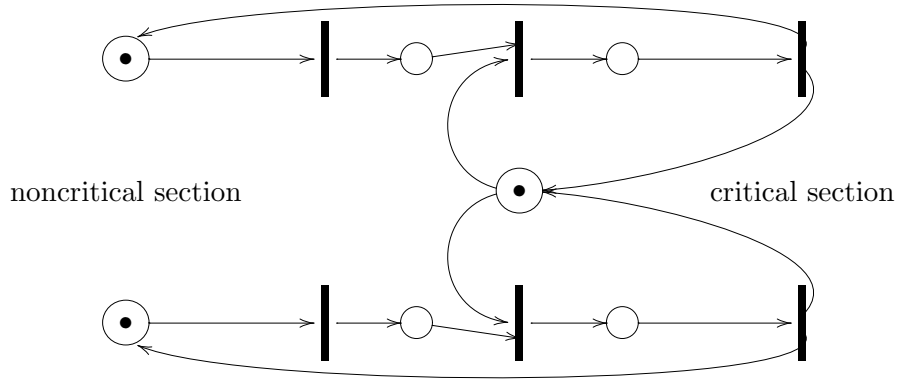
A *marking* of a Petri net $N = (P, T, F)$ is a function $m : P \to \mathbb{N}$ that assigns a number of *tokens* to a place. If $P = \{p_1, \ldots, p_n\}$ we also write $\bar{m} = (m_1, \ldots, m_n)$ for the marking m where m_i is the number of tokens on place p_i.

A transition t can be *fired* (or executed) if there is a token on every place with an incoming edge to t. When t is fired, one token is removed from every place with an incoming edge to t, and one token is put on every place connected to t by an outgoing edge.

In figures we represent places by circles and transitions by bars. Tokens are represented by dots inside a place. In the context of concurrent systems the places are called resource holders, and tokens are resources.

Example 6.2 The following Petri net models a mutual exclusion protocol

where two processes access a critical section.



⊠

Definition 6.3 Let $N = (P, T, F)$ be a Petri net. The *preset* of a transition $t \in T$ is the set ${}^\bullet t := \{p \in P \mid (p,t) \in F\}$, the *postset* is the set $t^\bullet := \{p \in P \mid (t,p) \in F\}$. The preset ${}^\bullet p$ and postset $p^\bullet$ of a place $p \in P$ are defined analogously. The definitions above generalize to sets of transitions and sets of places in the usual way, e.g., for a set $S \subseteq T$ of transitions we set $S^\bullet := \{p \in P \mid (t,p) \in F \text{ for some } t \in S\}$.

Let m, m' be markings of a Petri net $N = (P, T, F)$. We write

$$\begin{aligned} m \geq m' &:\Leftrightarrow \forall p \in P\ m(p) \geq m'(p), \text{ and} \\ m > m' &:\Leftrightarrow \forall p \in P\ m(p) \geq m'(p) \text{ and } \exists p:\ m(p) > m'(p). \end{aligned}$$

Let m be a marking of a Petri net $N = (P, T, F)$ and $t \in T$. We define

$$m \rhd_t m' :\Leftrightarrow \forall p \in {}^\bullet t:\ m(p) > 0 \text{ and}$$

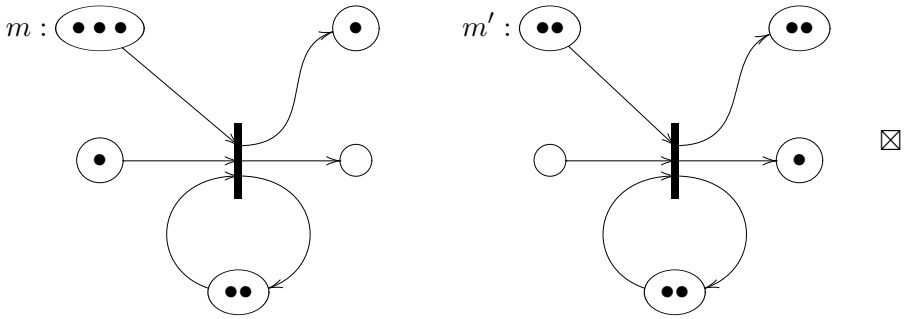
$$\forall p \in P:\ m'(p) = \begin{cases} m(p) - 1 & \text{if } p \in {}^\bullet t, p \notin t^\bullet \\ m(p) + 1 & \text{if } p \notin {}^\bullet t, p \in t^\bullet \\ m(p) & \text{otherwise} \end{cases}$$

We write $m \rhd m'$ if there is a transition t with $m \rhd_t m'$. A *firing sequence* from m to m' is a sequence $\tau = t_1, \ldots, t_n$ of transitions such that there are markings $m_0, \ldots m_n$ with $m_0 = m$, $m_n = m'$ and $m_{i-1} \rhd_{t_i} m_i$ for $1 \leq i \leq n$. We write $m \rhd_\tau m'$ if τ is a firing sequence from m to m' and $m \rhd^* m'$ if such a sequence exists.

We call a marking m a *deadlock* for N if for all $t \in T$ there exists a $p \in {}^\bullet t$ with $m(p) = 0$, i.e., no transition of N can be fired.

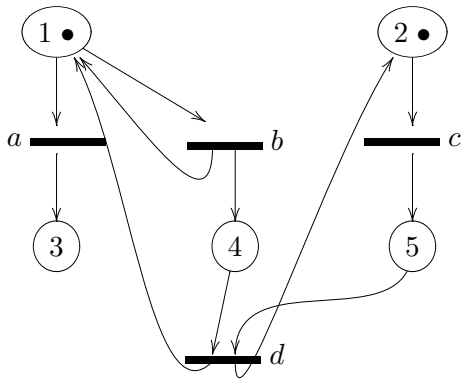
A Petri net with initial marking m_0 is called *unbounded* (or *unsafe*) if for all $K \in \mathbb{N}$ there exists a sequence $m_0 \rhd m_1 \rhd m_2 \rhd \ldots$ of markings such that there is an i with $\sum_{p \in P} m_i(p) > K$.

Example 6.4 The following picture illustrates the firing of a transition.



⊠

Example 6.5 The following Petri net illustrates the modeling power of Petri nets.



Transitions a and b are in conflict, only one of them can be fired at a time. We call such transitions *nondeterministic*. Transition c can be fired independently of a and b. We call c *concurrent* to a and b. If transitions a, c fire, the Petri net ends up in a deadlock. If b, c and d fire repeatedly, more and more tokens are collected at place 1, i.e., the Petri net is unbounded. ⊠

Example 6.6 The following Petri net models two production lines A and B with two intermediate steps each; the main manufacturing phase and the reset back to the idle state. Production line A requires resources p and q,

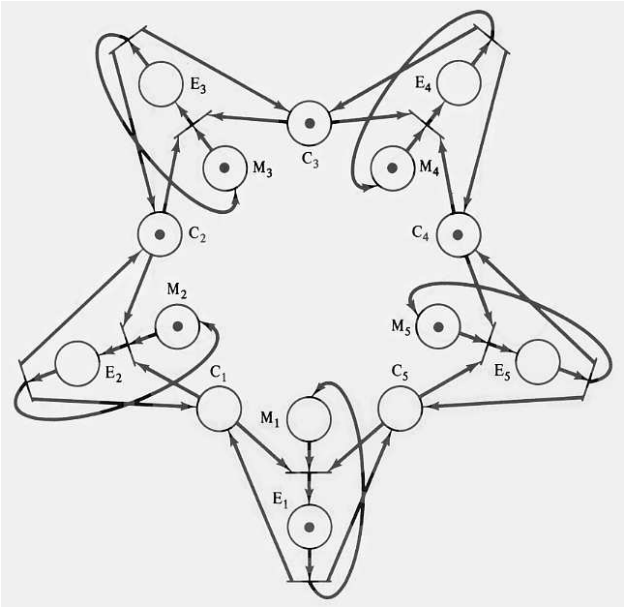
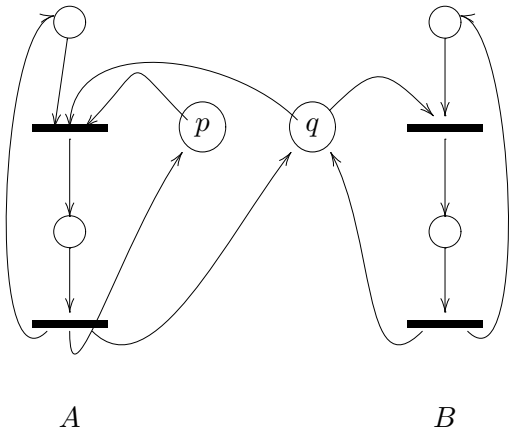


Figure 6.1: A Petri net modeling the problem of five dining philosophers.

whereas production line B needs only q.

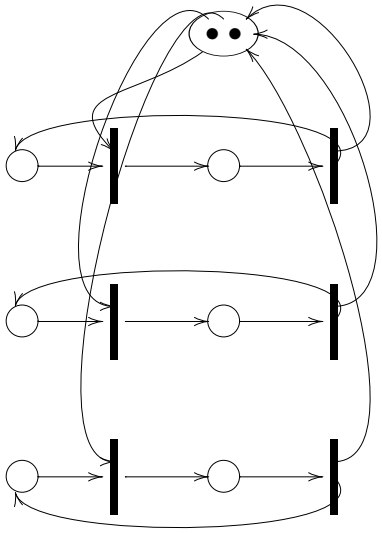


⊠

Example 6.7 Consider the problem of the dining philosophers as it was described in Example 5.4. The Petri net depicted in Figure 6.1 represents a snapshot of a possible modeling of the system, when only the first philosopher holds his forks and eats. ⊠

Example 6.8 Consider a small computer network where three PCs have to share two printers. These two printers can be represented by an auxiliary

place that possesses two tokens in the idle case.



If any of the PCs wants to use a printer it must obviously be in an non-printing state and furthermore it has to find at least one of the two tokens in the auxiliary place. After finishing the print job, the token is put back to the auxiliary place. ⊠

A Petri net with n places $p_1, \ldots, p_n$ and markings in $\mathbb{N}^n$ models a *vector addition system*, where a transition t leads from some marking m to a new marking $m' = m+v$, with $v \in \{-1, 0, 1\}^n$. In particular, the i-th component of v is

- -1, in case p_i is in the preset but not in the postset of t,
- 1 in case p_i is in the postset but not in the preset of t, and
- 0 otherwise.

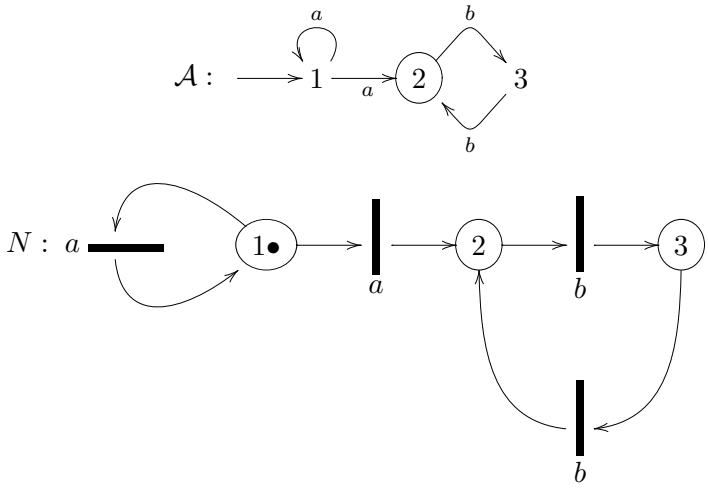
In the special case of *condition/event nets* the markings are also 0-1-vectors. That is, we are only interested whether there is a token at some place or not, and even if there is more than one token at some place, the number is normalized to 1, after each update of the marking. The mutual exclusion protocol in Example 6.2 and the production lines in Example 6.6 are such special cases.

6.2 Petri Nets as Language Acceptors

With a small extension of the original definition Petri nets can be used as language acceptors.

Definition 6.9 Let $N = (P, T, F)$ be a Petri net and M^-, M^+ be finite sets of *initial markings* and *final markings*, respectively. Let Σ be an alphabet and $l : T \to \Sigma$ be a labeling function that associates with each transition a label from Σ. $\mathcal{N} = (N, M^-, M^+, l)$ *accepts* a word $w \in \Sigma^*$ if there exists a firing sequence $\tau = t_1, \ldots, t_n$ from $m^- \in M^-$ to $m^+ \in M^+$ such that $l(\tau) := l(t_1) \ldots l(t_n) = w$. We denote by $L(\mathcal{N})$ the language recognized by the Petri net $\mathcal{N}$.

Example 6.10 The following pictures illustrate how to construct from an NFA $\mathcal{A}$ a Petri net N with initial marking $m^- = (1, 0, 0)$ and final marking final marking $m^+ = (0, 1, 0)$ accepting the same language.

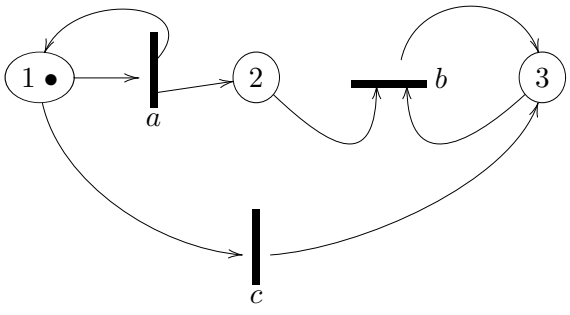


⊠

Lemma 6.11 *For every regular language L there exists a Petri net $\mathcal{N}$ such that $L = L(\mathcal{N})$.* □

The power of Petri nets as language acceptors goes beyond the regular languages.

Example 6.12 Let $L = \{a^n c b^n \mid n \geq 0\}$. L is a Petri net language accepted by the following Petri net with initial marking $m^- = (1, 0, 0)$ and final marking $m^+ = (0, 0, 1)$.



Reading a^n yields the marking $(1, n, 0)$, from $(1, n, 0)$ we proceed to $(0, n, 1)$ via c, and via b^n from $(0, n, 1)$ to $(0, 0, 1)$. This is the only way to reach $(0, 0, 1)$ from $(1, 0, 0)$. Hence this Petri net accepts L. ⊠

In Figure 6.2 the power of Petri nets as language acceptors is shown. Example languages are shown that separate the different classes.

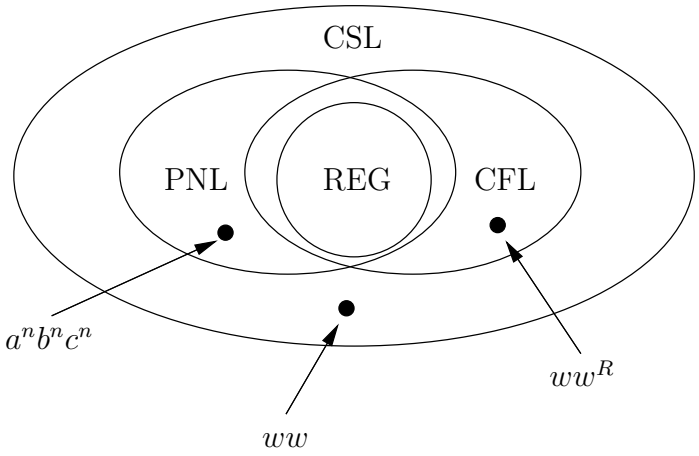
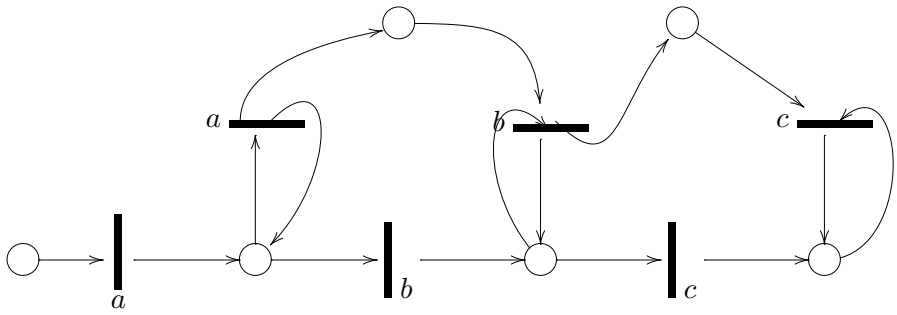


Figure 6.2: On the power of Petri nets as language acceptors

Example 6.13 We know that the language $L = \{a^n b^n c^n \mid n > 0\}$ is context-sensitive, but not context-free. To prove its position in the inclusion diagram of Figure 6.2 we construct the following Petri net:



This Petri net accepts L with initial marking $m^- = (1, 0, 0, 0, 0, 0)$ and final marking $m^+ = (0, 0, 0, 0, 0, 1)$. ⊠

The previous examples demonstrate the power of Petri nets in checking the correct number of occurrences of a letter. However, Petri nets are weak when it comes to checking also the order of occurrences. In the following we realize this situation on the language consisting of the palindromes of even length.

Theorem 6.14 *The context-free mirror-language L consisting of words ww^R with $w \in \{a,b\}^*$ is not a Petri net language.*

Proof: Towards a contradiction we assume that there exists a Petri net N with r transitions that recognizes L. After reading s letters (and firing s transitions), N must be able to reach as many different markings as there are words of length s. This must hold because N has to "remember" the first half of the word ww^R in order to compare it with the (reversed) second half. Since we chose an alphabet with 2 letters, N has to reach at least 2^s different markings upon reading s letters. If it cannot reach that many states, then there are two different words w, w' that lead N to the same marking. This means that N is not be able to distinguish between them and consequently, it accept ww^R if and only if it accepts $w'w^R$. Contradiction.

Hence, it suffices to prove the following lemma.

Lemma 6.15 *For every Petri net that recognizes L the following holds: For a sufficiently large s, only less than 2^s markings are reachable.*

Suppose that the Petri net has r transitions. Consider a sequence of s transitions. Each transition t_j modifies a marking m by adding a vector $v_j \in \{-1,0,1\}^n$. If t_j is fired k_j times during the whole sequence ($1 \leq j \leq r$), then m is modified to the marking $m + \Sigma_{j=1}^{r} k_j \cdot v_j$ where $\Sigma_j k_j = s$. Hence, with s transitions one can reach at most as many markings as there are such tuples $(k_1, \ldots, k_r)$. These r numbers have to sum up exactly to s and therefore $0 \leq k_i \leq s$ for all $i \in \{1, \ldots, r\}$. Thus, there are at most $(s+1)^r$ such tuples. This is a polynomial in s because r is a characteristic of the Petri net. Hence, for a sufficiently large s there are less than 2^s markings reachable by the Petri net. □

6.3 Matrix Representation of Petri Nets

An alternative method to analyze Petri nets is provided by a matrix calculus.

Definition 6.16 Let $N = (P, T, F)$ be a Petri net with $P = \{p_1, \ldots, p_n\}$ and $T = \{t_1, \ldots, t_r\}$. Let D^- and D^+ be the $(r \times n)$-matrices defined by

$$D^-(i,j) = \begin{cases} 1 & p_j \in {}^\bullet t_i \\ 0 & \text{otherwise} \end{cases} \text{ and } D^+(i,j) = \begin{cases} 1 & p_j \in t_i^\bullet \\ 0 & \text{otherwise} \end{cases}.$$

Let $D := D^+ - D^-$.

As before we view a marking as a vector $(m_1, \ldots, m_n)$. Transition t_i is is now coded by vector $e_i = (0, \ldots, 0, \overset{i}{1}, 0, \ldots, 0) \in \{0,1\}^r$. The matrix D

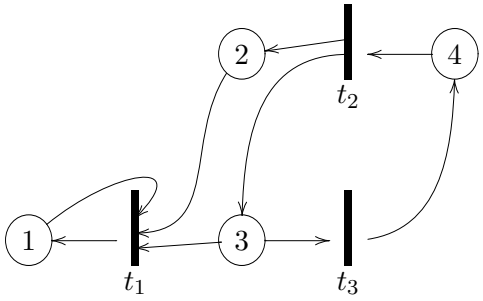
contains all the information about the "net effect" of transition t_i in its i-th row. Suppose t_i can be fired from a marking $\bar{m} = (m_1, \ldots, m_n)$. Then

$$\bar{m} \rhd_{t_i} \bar{m}' \Rightarrow \bar{m}' = \bar{m} + e_i \cdot D.$$

Note that the converse does not hold, i.e.,

$$\bar{m}' = \bar{m} + e_i \cdot D \not\Rightarrow \bar{m} \rhd_{t_i} \bar{m_i}'.$$

Example 6.17 Let N be the following Petri net.



Then

$$D^- = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \text{ and } D^+ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

So we obtain

$$D = \begin{pmatrix} 0 & -1 & -1 & 0 \\ 0 & 1 & 1 & -1 \\ 0 & 0 & -1 & 1 \end{pmatrix}.$$

Let $\bar{m} = (1, 0, 1, 0)$ and fire t_3. Then we get

$$(1, 0, 1, 0) + (0, 0, 1) \cdot D = (1, 0, 1, 0) + (0, 0, -1, 1) = (1, 0, 0, 1)$$

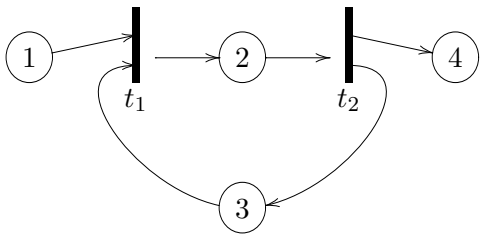
⊠

Lemma 6.18 *If $\bar{m}'$ can be reached from $\bar{m}$ by firing i_1 times t_1, …, i_r times t_r, then $\bar{m}' = \bar{m} + (i_1, \ldots, i_r) \cdot D$.* □

As a consequence of the previous lemma we obtain that if $\bar{m}'$ is reachable from $\bar{m}$, then the equation $\bar{m}' = \bar{m} + (x_1, \ldots, x_r) \cdot D$ has a solution in $\mathbb{N}^r$.

As mentioned above, the converse does not hold:

Example 6.19 Let N be the following Petri net:



If $\bar{m} = (1, 0, 0, 0)$ and $\bar{m}' = (0, 0, 0, 1)$, then $\bar{m}'$ is not reachable from $\bar{m}$, but

$$(0, 0, 0, 1) = (1, 0, 0, 0) + (x_1, x_2) \cdot \begin{pmatrix} -1 & 1 & -1 & 0 \\ 0 & -1 & 1 & 1 \end{pmatrix}$$

$(x_1, x_2) = (1, 1)$ is a solution for the equation. ⊠

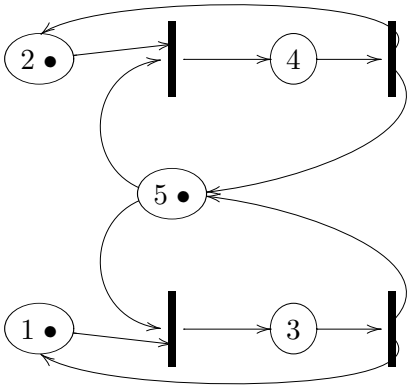
However, we obtain: If $\bar{m}' = \bar{m} + \bar{x} \cdot D$ has no solution $\bar{x}$, then $\bar{m}'$ cannot be reached from $\bar{m}$.

Another application of the matrix representation of Petri nets is motivated by applications for balanced distributed systems, where the total of resources is neither consumed nor expanded.

Definition 6.20 A Petri net $N = (P, T, F)$ with initial marking m^- is said to be *strictly conservative* if for all markings m reachable from m^- we have $m_1^- + \ldots + m_n^- = m_1 + \ldots + m_n$. We may write this as $m^- \cdot \mathbf{1} = m \cdot \mathbf{1}$, where $\mathbf{1} = (1, \ldots, 1) \in \mathbb{N}_+^n$.

N is called *conservative* if there is a weight vector $w = (w_1, \ldots, w_n) \in \mathbb{N}_+^n$ such that $m^- \cdot w = m \cdot w$ for every marking m reachable from m^-.

Example 6.21 Let N be the following Petri net:



N is conservative for $w = (1, 1, 2, 2, 1)$ and $m^- = (1, 1, 0, 0, 1)$. ⊠

We have the following characterization of conservative Petri nets:

Lemma 6.22 *Let $N = (P, T, F)$ be a Petri net with initial marking m^- and let $P = \{p_1, \ldots, p_n\}$ and $T = \{t_1, \ldots, t_r\}$. We assume that for every transition t_i there is a marking m^i reachable from m^- such that t_i can be fired from m^i. Let D be the transition matrix of N and let $\mathbf{0} = (0, \ldots, 0) \in \mathbb{N}^r$. Then N is conservative iff the linear equation $D \cdot w = \mathbf{0}$ has a solution $w \in \mathbb{N}_+^n$.*

Proof: Assume that $D \cdot w = \mathbf{0}$ for some $w \in \mathbb{N}_+^n$. Let m be some marking reachable from m^-, say, by the firing sequence τ. Let $[\tau]$ be the vector $(i_1, \ldots, i_r)$ where i_j is the number of occurrences of t_j in τ. By Lemma 6.18, we have $m = m^- + [\tau] \cdot D$. It follows that $m \cdot w = (m^- + [\tau] \cdot D) \cdot w = m^- \cdot w + [\tau] \cdot D \cdot w = m^- \cdot w$. Hence, N is conservative.

Conversely, assume that N is conservative with weight vector $w \in \mathbb{N}_+^n$ and let $1 \leq i \leq r$. By assumption, there are $m^i, \bar{m}^i$ with $m^- \rhd^* m^i \rhd_{t_i} \bar{m}^i$. Since both $m^i, \bar{m}^i$ are reachable from m^-, we obtain $m^- \cdot w = m^i \cdot w = \bar{m}^i \cdot w$. By Lemma 6.18, we have $\bar{m}^i = m^i + e_i \cdot D$. It follows that $\bar{m}^i \cdot w = (m^i + e_i \cdot D) \cdot w = m^i \cdot w + e_i \cdot D \cdot w$ and thus, $e_i \cdot D \cdot w = (D \cdot w)_i = 0$. Since i was chosen arbitrarily, we obtain $D \cdot w = \mathbf{0}$. □

6.4 Decision Problems for Petri Nets

In this section we investigate important algorithmic issues for Petri nets.

Definition 6.23 The *boundedness problem* for Petri nets is the following:

Input:	A Petri net N, an initial marking m^-.
Problem:	Is there a number K such that for all m reachable from m^- we have $\sum_{p \in P} m(p) \leq K$?

The *reachability problem* for Petri nets is the following:

Input:	A Petri net N, an initial marking m^- and a final marking m^+.
Problem:	Is m^+ reachable from m^- in N?

The *covering problem* for Petri nets is the following:

Input:	A Petri net N, an initial marking m^- and a final marking m^+.
Problem:	Is there a marking m reachable from m^- such that $m \geq m^+$?

The boundedness of a Petri net N with respect to some initial marking m^- actually guarantees that N can be represented by a finite automaton. This holds, because a Petri net with n places that is bounded by some number

K has only markings $(m_1, \ldots, m_n)$ with $\Sigma_{i=1}^{n} m_i \leq K$. This means that also $m_i \leq K$ for every i and hence there are less than K^n different markings reachable in N. These can be represented by the states of a huge, but still finite automaton.

In the following we first solve the boundedness problem and then the covering problem. Unfortunately, we have to skip the proof for the decidability of the reachability problem because it is far too complex to be presented here.

Theorem 6.24 (Karp, Miller 1968) *The boundedness problem for Petri nets is decidable.*

The proof of this theorem needs some preparation. We first define reachability trees over a Petri net and make some important observations about them.

Definition 6.25 [Reachability Tree of a Petri net] Given a Petri net N and an initial marking m^-, the *reachability tree* $t[N, m^-]$ is defined as follows:

- the root of $t[N, m^-]$ is m^-, and
- the immediate successors of a node m are all markings that are reachable from m by firing one transition.

Remark 6.26 *The reachability tree $t[N, m^-]$ contains exactly the markings that are reachable from m^-.*

However, since a place of a Petri net can have arbitrarily many markers, the reachability tree can be infinite. Is there some way to break the (possibly never ending) construction of a reachability tree in order to be able to perform the boundedness test? The construction of a reachability tree for a Petri net relies on the following monotonicity property. If m is reachable from m^- and there is a nonempty firing sequence τ such that $m \rhd_\tau m'$ and $m \leq m'$, then τ can be fired infinitely often. Two cases have to be distinguished:

1. If $m = m'$, then m can be reproduced infinitely often by executing τ again and again.
2. If $m < m'$, then $m_i < m'_i$ for some i, and m_i can be increased arbitrarily by executing τ again and again. In this case we obtain a monotonically increasing marking sequence $m < m' < m'' < \ldots$ and consequently the unboundedness of the Petri net.

Lemma 6.27 (Expansion Lemma) *If $m^- \rhd^* m \rhd^* m'$ with $m' > m$, then the following holds: there exist $m'', m''' \ldots$ with $m^- \rhd^* m \rhd^* m' \rhd^* m'' \rhd^* m''' \rhd^* \ldots$ and $m < m' < m'' < \ldots$. Hence, the Petri net is unbounded.*

The algorithm from Figure 6.3 uses the previous observations to compute a truncated reachability tree $t(N, m^-)$ for a Petri net N, the *Karp-Miller tree*. For simplicity we assume that the initial marking m^- is not a deadlock. Note that the marking m' is possibly changed in line 7 and thus, the conditions in lines 8 and 9 are checked for this possibly new marking.

Input: A Petri net $N = (P, T, F)$, an initial marking m^-

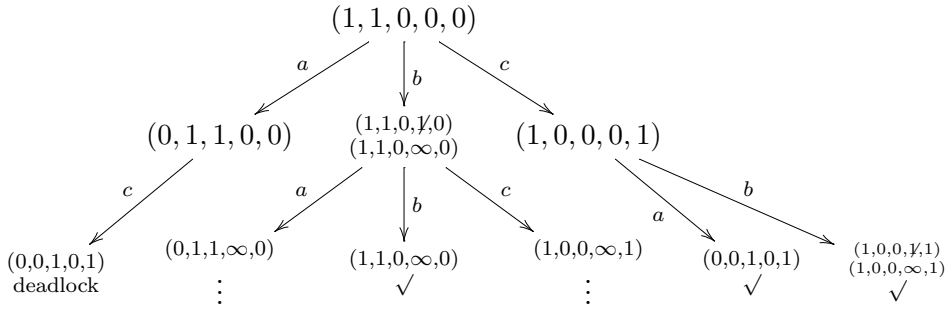
1. Let m^- be an unmarked leaf of $t(N, m^-)$
2. **while** there are unmarked leafs in $t(N, m^-)$ **do**
3. Choose an unmarked leaf m
4. **for** m' such that $m \rhd m'$ **do**
5. Add m' as a new leaf below m
6. **if** there is a $m^1 < m'$ on the path from m' to the root
7. **do** change every $m'_i > m^1_i$ to ∞ **od**
8. **if** m' already occurs in $t(N, m^-)$ **do** mark it **od**
9. **if** m' is a deadlock **do** mark it **od**
10. **od**
11. **od**

Output: $t(N, m^-)$

Figure 6.3: Algorithm computing $t(N, m^-)$

As we shall prove in the following the Karp-Miller tree contains all the information we need. Note that the structure of the output tree of the algorithm may depend on the order of chosen leaves in line 3. Further, there may be $m^1 < m'$ and $m^2 < m'$ with $m^1 \not\leq m^2$ and $m^2 \not\leq m^1$. The structure of the output tree may also depend on the choice of such a marking in line 6. But nevertheless, the set of all leaf labels for different output trees is always the same, and this suffices for our purpose.

Example 6.28 The Karp-Miller tree for the Petri net from Example 6.5 has the following form:



⊠

Now we have all the knowledge we need to decide whether a Petri net is bounded when started from some given marking. The procedure goes as follows: Given a Petri net N and an initial marking m^- we construct the Karp-Miller tree $t(N, m^-)$ by using the algorithm of Figure 6.3. If no ∞ sign occurs in the tree then N is bounded when started from m^-. Otherwise, N is unbounded when started from m^-.

To prove the correctness of this method we proceed in two stages. First, we show that the algorithm of Figure 6.3 terminates, i.e., that the Karp-Miller tree is finite, and second that N is unbounded when started from m^- if and only if a ∞ occurs in $t(N, m^-)$.

Theorem 6.29 *The algorithm from Figure 6.3 computing $t(N, m^-)$ terminates, i.e., $t(N, m^-)$ is finite.*

For the proof we need two intermediate results. The first one is the well known König's Lemma:

Lemma 6.30 (König) *Every finitely branching infinite tree has an infinite path.*

Proof: Let t be a finitely branching infinite tree. Starting from the root r of t we find an infinite path through t by repeatedly picking a son s of the current node such that the subtree of t rooted at s is infinite. Since t is assumed to be a finitely branching but infinite, this is always possible. □

Lemma 6.31 (Dixon) *Let $V \subseteq \mathbb{N}^r$ be infinite. Then there is an infinite subset $V_0 \subseteq V$ that is totally ordered by $\leq$.*

Proof: We prove this by induction on r. For $r = 1$ the order $\leq$ is total, hence there is nothing to show. Assume now that V is an infinite subset of $\mathbb{N}^{r+1}$. Let $V' := \{\bar{m} \in \mathbb{N}^r \mid \exists x : \ (\bar{m}, x) \in V\}$. Two cases have to be distinguished:
Case 1: V' is finite. Then there exists some $\bar{m_0}$ such that for infinitely many x we have $(\bar{m_0}, x) \in V$. These vectors form the desired totally ordered set.
Case 2: V' is infinite. By induction hypothesis there is a totally ordered infinite subset $V_0' \subseteq V'$.

a) For some x_0 we have $x \leq x_0$ for all $(\bar{m}, x) \in V$ with $\bar{m} \in V_0'$. Then there exists $x_1 \leq x_0$ such that $(\bar{m}, x_1) \in V$ for infinitely many $\bar{m} \in V_0'$. These vectors form the desired infinite totally ordered set.

b) For every $(\bar{m}_0, x_0) \in V$ with $\bar{m}_0 \in V_0'$ there exists $\bar{m} \in V_0'$ and $x \in \mathbb{N}$ with $\bar{m} > \bar{m}_0$ and $x > x_0$ such that $(\bar{m}, x) \in V$. In this case the existence of an infinite totally ordered set is trivial.

□

Proof of Theorem 6.29: Assume that $t(N, m^-)$ is infinite. By construction $t(N, m^-)$ is finitely branching. Hence, by König's Lemma, $t(N, m^-)$ contains an infinite path π. For markings $(m_1, \ldots, m_n)$ appearing on π let $i_1, \ldots, i_s$ be those components where eventually ∞ appears. We know that $\{i_1, \ldots, i_s\} \neq \{1, \ldots, n\}$ because otherwise $(\infty, \ldots, \infty)$ would be reached, then repeated, and therefore π would be finite. So w.l.o.g. assume that after the last ∞ sign has been introduced (this happens after a finite number of steps!) only markings $(m_1, \ldots, m_r, \infty, \ldots, \infty)$ $(r \geq 1)$ appear on π.

By Dixon's Lemma there exists an infinite totally ordered set V_0 of markings of the form $(m_1, \ldots, m_r, \infty, \ldots, \infty)$ on π. Let $m \in V_0$ be the first such marking appearing on π. Consider now the markings in V_0 appearing after m on π. Only finitely many of these can be smaller than m. Since V_0 is infinite there must be a m' appearing after m on π with $m' > m$. Hence another ∞ sign would be introduced by the algorithm, contradicting the assumption above. □

Now that we proved that the algorithm terminates we have to show that N is unbounded when started from m^- if and only if a ∞ occurs in the Karp-Miller tree $t(N, m^-)$.

Direction $\Leftarrow$: If ∞ occurs in the Karp-Miller tree then $m^- \rhd^* m \rhd^* m'$ with $m < m'$. According to the expansion lemma we obtain markings $m^- \rhd^* m' \rhd^* m'' \rhd^* m''' \ldots$ with $m' < m'' < m''' \ldots$ Hence, N is unbounded when started from m^-.

Direction $\Rightarrow$: We argue that, if ∞ does not occur in $t(N, m^-)$, then N is bounded when started from m^-. According to the construction of the tree, the markings that occur in $t(N, m^-)$ are exactly the ones that are reachable from m^- (see Remark 6.26), i.e., we have $t(N, m^-) = t[N, m^-]$. Since the Karp-Miller tree is finite (Theorem 6.29), there are only finitely many different reachable markings. Hence, N is bounded when started from m^-. □

The tree $t(N, m^-)$ computed by the Algorithm 6.3 can be applied to decide the covering problem mentioned above, the reachability problem if N is bounded, and other properties like whether there exists a deadlock, or whether N is life (we say that a Petri net is *life* if for every marking m reachable from m^- and for every transition t there is a marking m_t that can be reached from m and where t can be fired).

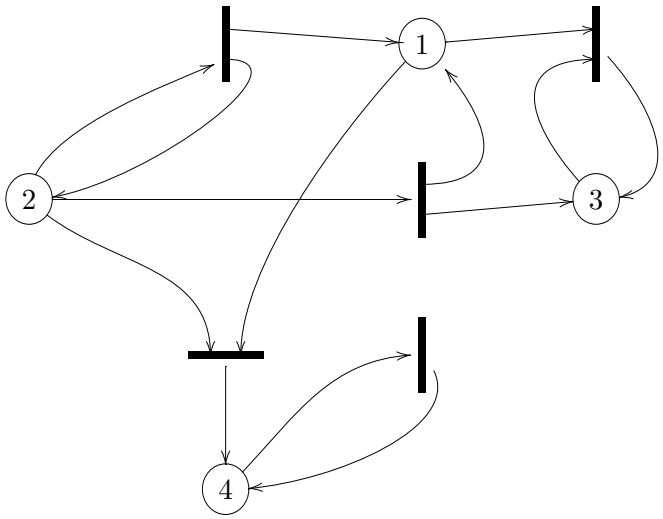
However, a major drawback of the above mentioned algorithm is that its complexity cannot be bounded even by a primitive recursive function.

The result on the reachability problem can be strengthened:

Theorem 6.32 *The reachability problem is decidable even for unbounded Petri nets.* □

6.5 Exercises

Exercise 6.1 *Consider the following Petri net N where the numbers denote the names of the places.*



(a) Construct the reachability tree of N with initial marking $(0, 1, 0, 0)$.

(b) Determine the deadlock markings reachable from $(0, 1, 0, 0)$ and give a description of the reachable states.

Index