Automata Theory 101

Ralf Huuck

Outline

- Introduction
- Finite Automata
- Regular Expressions
- ω-Automata

Acknowledgement

Some slides are based on Wolfgang Thomas' excellent lecture on "Automatentheorie and Formale Sprachen".

Introduction

Basics

Basic objects in mathematics

- number (number theory, analysis)
- shapes (geometry)
- sets and transformation on such objects

Basic objects in computer science

- words
- stet of words (language)
  and their transformations
- defining and describing words

Why words?

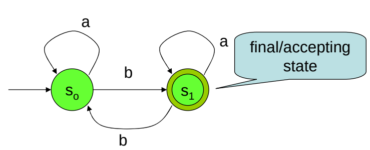
- Every IT system is about **data** and transformation of data

  101010101000001010101111110
  word from alphabet {0,1}

- program is also just a finite word
- every terminating execution is a finite word
- a programming language is the set of all permissible words (i.e., accepted programs)

Automata and Grammars are all about accepting/generating words and defining a language.

Automaton



defines language of
all words over alphabet {a,b} with an odd number of b's.

Finite Automata

Words & Languages

An alphabet is a non-empty set of symbols/letters.

$\Sigma_b = \{0,1\}$ $\Sigma_{lat} = \{a,\ldots,z,A,\ldots,Z\}$

A word is a sequence of symbols from an alphabet

$01111010 \in \Sigma^*_b$ $hello \in \Sigma^*_{lat}$

A language is the set of all possible words

Σ^*_b (all finite Boolean words) Σ^*_{lat} (all finite words of latin characters)

A grammar/automaton restricts to meaningful languages

all 8-bit words all English words

Operations on Words

Concatenation of words: $u=a_1\ldots a_m$ and $v=b_1\ldots b_n$ $(m,n\geq 0)$

$u \cdot v = a_1\ldots a_m b_1\ldots b_n$

Note: empty word ε, word of length 0, but not $\emptyset$

$u \cdot \varepsilon = u = \varepsilon \cdot u$

We often write **uv** instead of **u · v**.

Operations on Languages

Concatenation of languages K and L:

$K \cdot L = \{uv \in \Sigma^* \mid u \in K, v \in L\}$

Example: K={follow, me} L={follow, you}
K · L= {followfollow, meyou, followyou}

Kleene Star

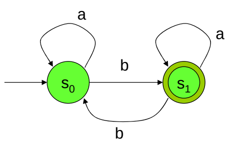
Iterating a language L

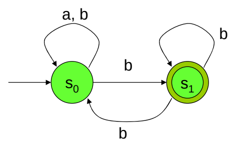
$L^0=\{\varepsilon\}$
$L^1=L$
$L^2=L \cdot L$
$L^{n+1}=L^n \cdot L$

Kleene star: $L^*=\bigcup_{n\geq 0} L^n$

Example: $\{a,b\}^* = \{\varepsilon,a,b,aa,bb,ab,ba,aab,\ldots\}$
all finite sequences over {a,b}.

Deterministic vs Non-Deterministic





Deterministic Finite Automaton (DFA)

from every state every symbol leads to a unique state

To model algorithms.

Non-deterministic Finite Automaton (NFA)

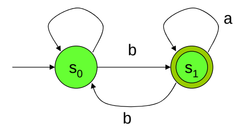
from a state the same symbol might lead t o different states or no state

To model a systems or environment.

Definition DFA

A DFA is of the form

$A=(S,\Sigma,s_0,\delta,F)$



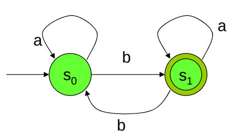
where

- S finite set of states
- Σ alphabet
- s_0 initial state
- $\delta : S\times\Sigma \rightarrow S$ transition function
- $F\subseteq S$ set of final states

Accepting Run of DFA

A run over a word $w=a_0,\ldots,a_n$ ($n\geq 0$) of an DFA is a sequence of states

$q_0,\ldots, q_{n+1}$



such that

- $q_0 = s_0$ and
- $\delta(q_i,a_i)=q_{i+1}$ $(0\leq i \leq n)$

We also write $\delta(q_0,w)=q_{n+1}$.

A run is accepting iff $q_{n+1} \in F$.

Language of DFA

The language accepted by DAF A is

$L(A) = \{w\in \Sigma^* \mid \delta(s_0,w)\in F\}$

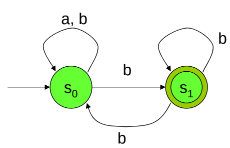
A language K is called DFA accepting if there is a DFA such that L(A)=K.

Two DFAs A and B are equivalent of L(A)=L(B).

Definition NFA

A NFA is of the form

$A=(S,\Sigma,s_0,\Delta,F)$



where

- S finite set of states
- Σ alphabet
- s_0 initial state
- $\Delta : S\times\Sigma\times S$ transition relation
- $F\subseteq S$ set of final states

no longer function

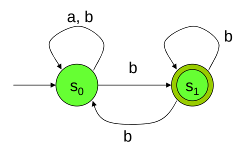
Accepting Run of NFA

A run over a word $w=a_0,\ldots,a_n$ ($n\geq 0$) of an NFA is a sequence of states

$q_0,\ldots, q_{n+1}$

such that

- $q_0 = s_0$ and
- $(q_i,a_i,q_{i+1})\in \Delta$ ($0\leq i \leq n$)



We also write $q_0\to^w q_{n+1}$.

A run is accepting iff $q_{n+1} \in F$.

Language of NFA

The language accepted by NFA A is

$L(A) = \{w\in \Sigma^* \mid s_0\to^w s \text{ and } s\in F \}$

A language K is called NFA accepting if there is a NFA such that L(A)=K.

Two NFAs A and B are equivalent of L(A)=L(B).

Question

Is there a language that is either a DFA or an NFA accepting, but not both?

Answer: No

Claim: L DFA accepting $\Leftrightarrow$ L NFA accepting

Proof:

L DFA accepting $\Rightarrow$ L NFA accepting

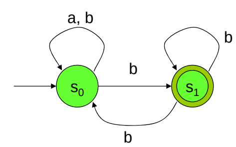
easy: every DFA is in particular an NFA, just relax δ to be a relation

L NFA accepting $\Rightarrow$ L DFA accepting

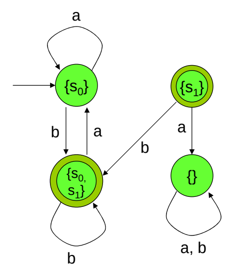
slightly harder, idea see next slide

Idea: NFA to DFA

NFA



DFA



Power set construction

- state space power set
- keep initial state
- for every symbol and state have a transition to the exact reachable subset
- all states which contain a final state are final states

Definition Power Set Automaton

Given NFA $A=(S,\Sigma,s_0,\Delta,F)$.

Define power set DFA $A'=(S',\Sigma,s_0',\delta,F')$ as follows:

- $S' := 2^S$
- $s_0' := \{s_o\}$
- $\delta(P,a) = \{q \in S \mid \text{there is } p \in P : (p,a,q) \in \Delta\}$
- $F' := \{P \subseteq S \mid P \cap F \neq \emptyset\}$

Proof NFA $\Rightarrow$ DFA

Lemma: We show A and A' are equivalent by showing

$A: s_0 \to^w s$ iff $s \in \delta(\{s_0\},w)$

Proof: $A: s_0 \to^w s$

iff $s \in Reach_A(w)$ reachable states for w

iff $s \in \delta(\{s_0\},w)$.

This implies:

$A: s_0 \to^w s$ with $s \in F$

iff $s \in \delta(\{s_0\},w) \cap F \neq \emptyset$

Hence: A is w accepting iff A' is w accepting.

The imagination driving Australia's ICT future.

Regular Expression

The imagination driving Australia's ICT future.

Another Way of Defining a Language

Example

- all words starting with 1 or 3 a's
- followed by a possible sequence of ab's
- followed by at least 1 b

Regular Expression

$(a + aaa) \cdot (a \cdot b)^* \cdot \; b \cdot b^*$

Brackets and concatenation symbols are sometimes omitted when clear from the context.

The imagination driving Australia's ICT future.

RE Syntax

Definition: The set of regular expressions RE_Σ over $\Sigma = \{a_1,\ldots,a_n\}$ is defined inductively by:

Base elements: $\emptyset$, ε, $a_1,\ldots,a_n$

Constructors: if r and s are regular expression so are $(r+s)$, $(r \cdot s)$, and r^*

Alternatively this can be defined in terms of a BNF grammar.

The imagination driving Australia's ICT future.

RE Semantics

We define a language $L(r) \subseteq \Sigma^*$ (set of words) for every regular expression $r \in RE_\Sigma$ as follows:

$L : RE_\Sigma \rightarrow 2\Sigma$ is defined inductively:

1. $L(\emptyset)=\emptyset$, $L(\varepsilon)=\{\varepsilon\}$, $L(a_i)=\{a_i\}$
2. $L(r+s) = L(r) + L(s)$
   $L(r \cdot s) = L(r) \cdot L(s)$
   $L(r^*) = (L(r))^*$

A language is regular if it is definable by a regular expression.

Regular Expressions in UNIX

- $[a_1, a_2, \ldots, a_n]$ instead of $a_1 + a_2 + \ldots + a_n$
- "." instead of Σ (any letter)
- | instead of +
- r? instead of ε + r
- r+ instead of r^*r
- r{4} instead of rrrr

Question

Is there a language that can be expressed
either by an NFA/DFA or an RE
but not both?

Answer

Kleene's Theorem

who also braught us the Kleene algebra, the Kleene star,
Kleene's recursion theorem and the Kleene fixpoint theorem

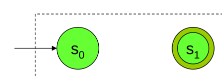
For every RE there is an equivalent NFA and
for every NFA there is an equivalent RE.

We give the proof (sketch) by
a) presenting an inductive construction from RE to NFA and
b) the idea of a transformation algorithm from NFA to RE

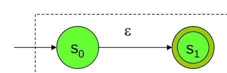
RE to NFA: Thompson Construction

Induction Base

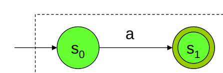
case r=∅: define A_r as



case r=ε: define A_r as



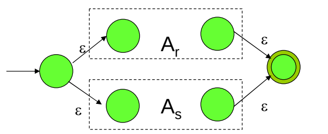
case r=a ($a \in \Sigma$): define A_r as



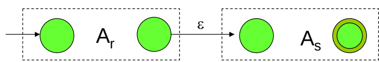
RE to NFA: Thompson Construction

Induction Step

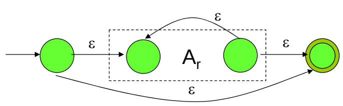
case r+s:
define A_r + A_s as



case r · s:
define $A_r \cdot A_s$ as



case r*:
define A_r*as



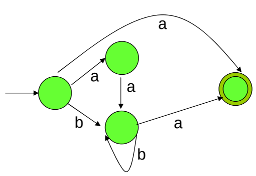
Idea: NFA to RE

Claim: For every NFA we can construct an equivalent RE.
Proof (idea): Create RE from transition labels of NFA.

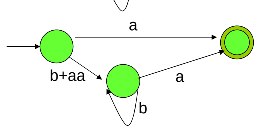
There is a graph transformation algorithm that does exactly this. It is know as the elimination algorithm.

Example

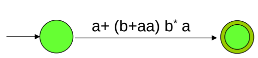
Start



transform



to



Closure Properties, Product Automaton

If $K,L \in \Sigma^*$ regular then $K \cap L$, $K \cup L$ and $K_{comp} := \Sigma^* \setminus K$ regular.

E.g.:$K \cap L$ can be obtained by synchronous product automaton
$A_\times$: For NFA $A_k=(S_k,\Sigma,s_{0k},\Delta_k,F_k)$ for K and $A_L=(S_L,\Sigma,s_{0L},\Delta_L,F_L)$ for L we define:

$A_\times := (S_k \times S_L,\Sigma,(s_{0k},s_{0L}),\Delta,F)$ where

- $((s_k,s_l),a,(s'_k,s'_l)) \in \Delta$ iff $(s_k,a,s'_k) \in \Delta_k$ and $(s_L,a,s'_L) \in \Delta_L$
- $F := F_k \times F_L$

Idea: Run A_k, A_L in parallel and only accept if both accept.

Example

Synchronized Product

A synchronized product on NFAs
$A_1=(S_1,\Sigma_0\cup\Sigma_1,s_{01},\Delta_1,F_1)$, $A_2=(S_2,\Sigma_0\cup\Sigma_2,s_{02},\Delta_2,F_2)$
with disjoint Σ_0, Σ_1, Σ_2 is defined by:

A_{sync}:= $(S_1\times S_2,\Sigma,(s_{01},s_{02}),\Delta,F)$ where

- $((s_1,s_2),a,(s'_1,s'_2))\in \Delta$ iff
  - $a\in\Sigma_1$, $(s_1,a,s'_1)\in\Delta_1$, $s_2=s'_2$ or
  - $a\in\Sigma_2$, $(s_2,a,s'_2)\in\Delta_2$, $s_1=s'_1$ or
  - $a\in\Sigma_0$, $(s_1,a,s'_1)\in\Delta_1$ and $(s_2,a,s'_2)\in\Delta_2$
- $F := F_1 \times F_2$

Means: A_1, A_2 can move independently on Σ_1, Σ_2, but must synchronize on Σ_0

Example

Good To Knows

For any NFAs A,B:

- emptiness problem: $L(A)=\emptyset$?
- infinity problem: Is $|L(A)|$ infinite?
- inclusion problem: $L(A)\subseteq L(B)$?
- equivalence problem: $L(A)=L(B)$?

are all decidable.

Model Checking as Inclusion Problem

Model Checking Problem:

$M \models \phi$?

System | satisfies | property ?

Special case:

NFA A | satisfies | RE B

Solving by: Transform RE B in NFA and check if $L(A) \subseteq L(B)$

which is checking: $L(A) \cap (\Sigma^* \setminus L(B)) = \emptyset$

Model Checking as Inclusion Problem

Model Checking Problem:

$M \models \phi$?

System | satisfies | property ?

Typical: Model checking is not only concerned about finite runs but also infinite, e.g., for non-terminating processes.

This requires more powerful frameworks:
ω–Automata instead of NFAs, temporal logic instead of RE.

Something to Remember

Programmer
- Regular expressions powerful for pattern matching
- Implement regular expressions with finite state machines.
- example: lexer

Theoretician
- Regular expression is a compact description of a set
- DFA is an abstract machine that solves pattern match
- equivalence DFA/NFA and regular expressions
- model checking as inclusion problem

ω – Automata

From Finite to Infinite Systems

So far:
- DFA/NFA and regular expressions define finite systems
- terminating programs, algorithms etc.

Now:
- infinite systems, i.e., systems with infinite runs
- non-terminating programs, operating systems, etc.

Infinite words are called ω words and the automata generating them ω automata.

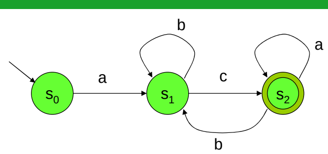
Buchi Automata

A (non-deterministic) Buchi automaton $\langle \Sigma, S, s_0, \Delta, F \rangle$
- Σ is a finite alphabet
- S is a finite set of states
- $s_0 \in Q$ is a subset of initial states
- Δ: $Q \times \Sigma \times Q$ is a transition relation
- $F \subseteq S$ is a subset of accepting states

For an infinite run r let Inf(r) = { s | $s=s_i$ for infinitely many i }.

A run r of a Buchi automaton is accepting iff $Inf(r) \cap F \neq \emptyset$, i.e., some final state occurs infinitely often.

Example



$r_1 = s_0 s_1 s_2 s_2 s_2 s_2 \ldots$ ACCEPTED

$r_2 = s_0 s_1 s_2 s_1 s_2 s_1 \ldots$ ACCEPTED

$r_3 = s_0 s_1 s_2 s_1 s_1 s_1 \ldots$ REJECTED

ω–regular Languages

An ω word has a finite prefix from s_0 to s and then revisits s infinitely often.

For automaton A, if U_s is the regular set of all finite words s_0 to s and V_s the regular set of all finite "revisits". An ω word is

$\alpha = uv_0v_1\ldots$ where $u \in U_s$, $v_i \in V_s$, $i \geq 0$

We write $\alpha \in U_s V^\omega_s$.

The ω regular language of A is $L_\omega(A) = \bigcup_{s \in F} U_s V^\omega_s$.

A language is ω regular iff Buchi recognizable.

Other ω–Automata

There are different types of ω-automata. They typically only differ in their acceptance conditions.

Buchi: $Inf(r) \cap F \neq \emptyset$,

Muller: $\bigvee_{F \in \mathcal{F}} Inf(r)=F$ for $\mathcal{F} \subseteq 2^S$ (must match one set)

Rabin: $\bigvee^n_{i=1}$ $(Inf(r) \cap E_i=\emptyset$ and $Inf(r) \cap F_i \neq \emptyset)$ for $E_i,F_i \subseteq S$ and acceptance set $\{(E_1,F_1),\ldots,(E_n,F_n)\}$, i.e., all states of E_i only visited finitely often, but some states of F_i infinitely

Street: $\bigwedge^n_{i=1}$ $(Inf(r) \cap E_i \neq \emptyset$ and $Inf(r) \cap F_i = \emptyset)$ for $E_i,F_i \subseteq S$ and acceptance set $\{(E_1,F_1),\ldots,(E_n,F_n)\}$ (dual to Rabin)

Equivalence

For non-deterministic ω-automata the following are equivalent (recognize the same language):

Buchi
⇔ Muller
⇔ Rabin
⇔ Street

McNaughton's Theorem

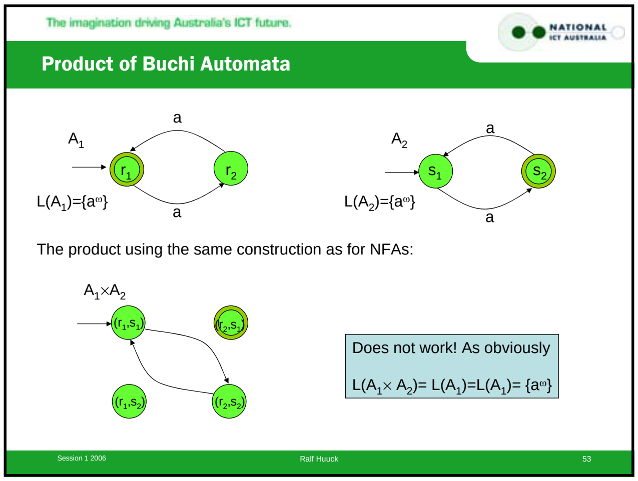
McNaughton's Theorem:
Buchi can be transformed into equivalent deterministic Muller.

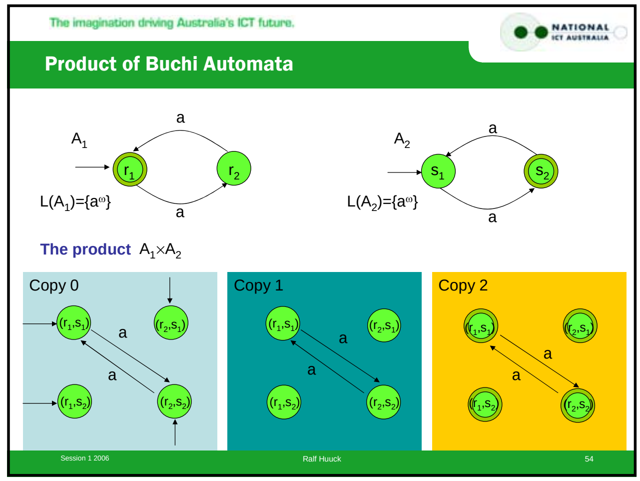
From its proof (Safra's construction) follows:

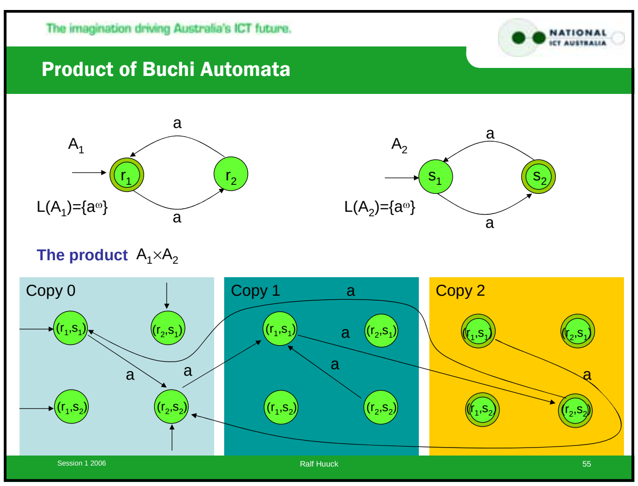
deterministic Muller,
⇔ deterministic Rabin,
⇔ deterministic Street and
⇔ non-deterministic Buchi

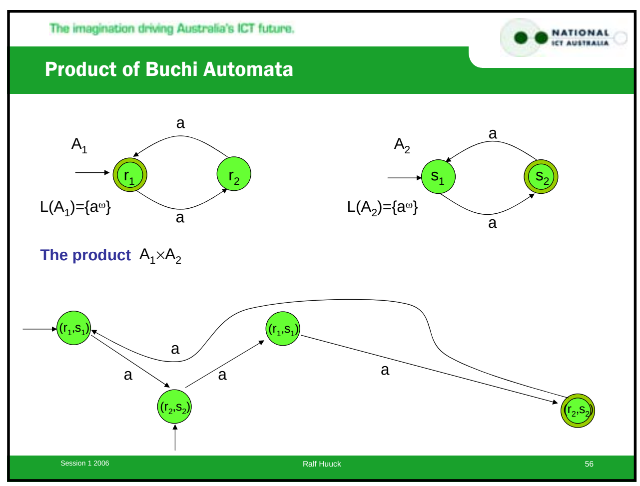
Conclusion

non-deterministic Buchi
⇔ Muller (deterministic/non-deterministic)
⇔ Street (deterministic/non-deterministic)
⇔ Rabin (deterministic/non-deterministic)


The imagination driving Australia's ICT future.
NATIONAL ICT AUSTRALIA
Product of Buchi Automata
A1
L(A1)={aω}
A2
L(A2)={aω}
The product using the same construction as for NFAs:
A1×A2
Does not work! As obviously
L(A1× A2)= L(A1)=L(A1)= {aω}
Session 1 2006
Ralf Huuck
53


The imagination driving Australia's ICT future.
NATIONAL ICT AUSTRALIA
Product of Buchi Automata
A1
L(A1)={aω}
A2
L(A2)={aω}
The product A1×A2
Copy 0
Copy 1
Copy 2
Session 1 2006
Ralf Huuck
54


The imagination driving Australia's ICT future.
NATIONAL ICT AUSTRALIA
Product of Buchi Automata
A1
L(A1)={aω}
A2
L(A2)={aω}
The product A1×A2
Copy 0
Copy 1
Copy 2
Session 1 2006
Ralf Huuck
55


The imagination driving Australia's ICT future.
NATIONAL ICT AUSTRALIA
Product of Buchi Automata
A1
L(A1)={aω}
A2
L(A2)={aω}
The product A1×A2
Session 1 2006
Ralf Huuck
56

Product of Buchi Automata

Strategy

- “multiply” the product automaton by 3
  ($S = S_1 \times S_2 \times \{0,1,2\}$)
- ‘0’ copy initial states, ‘2’ copy final states
- transition relation like “normal” product automaton, but **redirect arcs** such that
  - transition to the ‘1’ copy if in ‘0’ copy and visiting final state from A_1
  - transition to the ‘2’ copy if in ‘1’ copy and visiting final state from A_2,
  - all transitions from ‘2’ copy lead to ‘0’ copy

The product of A_1, A_2 gives us the intersection of their two languages.

Lessons Learned

- DFA vs NFA
- regular vs DFA/NFA
- product of NFAs (intersection of languages)
- ω automata
- product of ω automata

Next Lecture

Model Checking Problem:

$M \models \phi$?

System satisfies property ?

Have a nice language to specify ϕ: use temporal logic.