Grundbegriffe der Theoretischen Informatik

Sommersemester 2018 - Thomas Schwentick

Teil A: Reguläre Sprachen

6: Anwendungen und Erweiterungen

Version von: 3. Mai 2018 (14:08)

Übersicht

- Reguläre Sprachen haben unzählige Anwendungen in der Informatik
- In diesem Kapitel betrachten wir eine kleine Auswahl davon
- Außerdem werfen wir einen Blick auf einige Erweiterungen von endlichen Automaten

Inhalt

Anwendung: Lexikalische Analyse (1/3)

- Phasen eines Compilers (schematisch):
  - Lexikalische Analyse
  - Syntaktische Analyse
  - Semantische Analyse
  - Zwischencode-Erzeugung
  - Zwischencode-Optimierung
  - Code-Erzeugung

- Die lexikalische Analyse wird von einem *Scanner* durchgeführt
- Dafür ist die Ausdruckskraft regulärer Sprachen ausreichend

- Bei der Syntaxanalyse wird die Struktur eines Programmes überprüft und in Form eines Syntax-Baumes repräsentiert
- Sie wird vom *Parser* durchgeführt
- Dazu werden *kontextfreie Sprachen* verwendet, die wir in Teil B kennenlernen werden

Anwendung: Lexikalische Analyse (2/3)

- Was macht ein Scanner?

- Der Scanner fasst zusammengehörige Zeichen des Programmtextes zu *Token* zusammen und ordnet sie *Token-Klassen* zu

Beispiel

```
Zaehler := Zaehler + 3 * Runde;
```

Token	Klasse
`Zaehler`	`Identifikator`
`:=`	`Zuweisungs-Operator`
`Zaehler`	`Identifikator`
`+`	`Additions-Operator`
`3`	`Zahl`
`*`	`Multiplikations-Operator`
`Runde`	`Identifikator`
`;`	`Semikolon`

- Tokenklassen lassen sich durch reguläre Ausdrücke beschreiben (hier in UNIX-Schreibweise):
  - Identifikator: $[\boldsymbol{A}-\boldsymbol{Z}\boldsymbol{a}-\boldsymbol{z}][\boldsymbol{A}-\boldsymbol{Z}\boldsymbol{a}-\boldsymbol{z}\mathbf{0}-\mathbf{9}]*$
  - Zahl: $[\mathbf{0}-\mathbf{9}]+$
  - Multiplikations-Operator: $*$
  - Zuweisungs-Operator: $:=$
  - usw.

- Aus diesen regulären Ausdrücken lässt sich ein *endlicher Automat mit Ausgabe* konstruieren, der die Zerlegung und Zuordnung vornimmt

✎ Automaten mit Ausgabe betrachten wir in diesem Kapitel auch noch

Anwendung: Lexikalische Analyse (3/3)

- Scanner müssen nicht eigenhändig von Grund auf programmiert werden
- Es gibt Scanner-Generatoren, denen nur die regulären Ausdrücke und die entsprechenden Aktionen mitgeteilt werden müssen

- Zum Beispiel `lex`:

if	`{return(if)}`
$[\boldsymbol{A}-\boldsymbol{Z}\boldsymbol{a}-\boldsymbol{z}][\boldsymbol{A}-\boldsymbol{Z}\boldsymbol{a}-\boldsymbol{z}\mathbf{0}-\mathbf{9}]*$	{Code, um den zugehörigen Identifikator in der **Symboltabelle** zu finden; `return(ID)`}
$[\mathbf{0}-\mathbf{9}]*$	{Code, um Wert der Zahl zu berechnen; `return(val)`}
:=	`{return(assignment)}`
$\dots$	$\dots$

- Für praktische Anwendungen mit großen Alphabeten ist es sinnvoll, Mengen von Zeichen kompakter zu beschreiben als durch Aufzählung oder Intervalle ☞ Symbolic Automata

Inhalt

Anwendung: Zeichenkettensuche

- Endliche Automaten können auch für die Suche nach Zeichenketten in Texten verwendet werden
- Wir betrachten dazu das Beispiel der Suche nach mehreren gegebenen Wörtern in einem (typischerweise großen) Text

Definition (MULTISEARCH)

Gegeben: Menge $M = \{w_1, \ldots, w_n\}$ von nicht leeren Zeichenketten, String v

Frage: Kommt einer der Strings $w_1, \ldots, w_n$ in v vor?

- v repräsentiert also den Text, in dem gesucht wird
- In der Praxis ist v viel länger ist als die Strings in M

- Wir gehen im Folgenden davon aus, dass kein String w_i Teilstring eines anderen Strings w_j ist

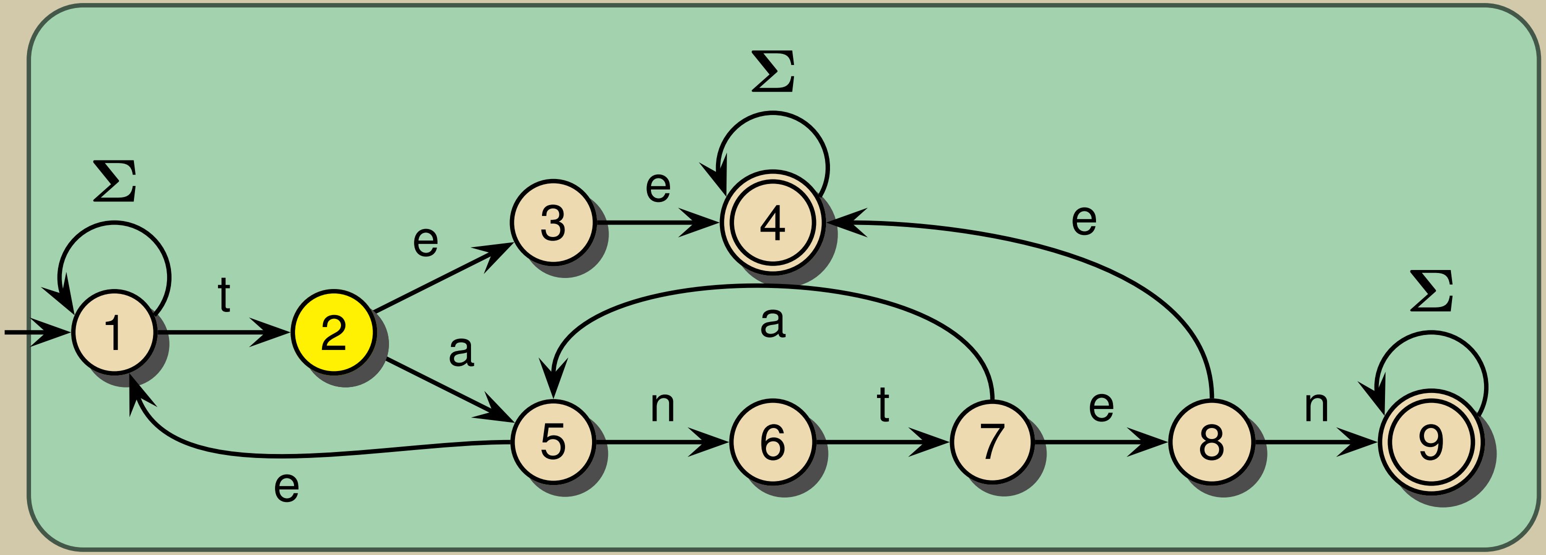
  ✎ Sonst kann w_j einfach weggelassen werden

Zeichenkettensuche: Beispiel (1/2)

- Aus der Suchmenge M lässt sich sehr leicht ein NFA konstruieren, der einen Text genau dann akzeptiert, wenn er ein Wort aus M enthält

Beispiel

- $M = \{\text{tee}, \text{tanten}\}$



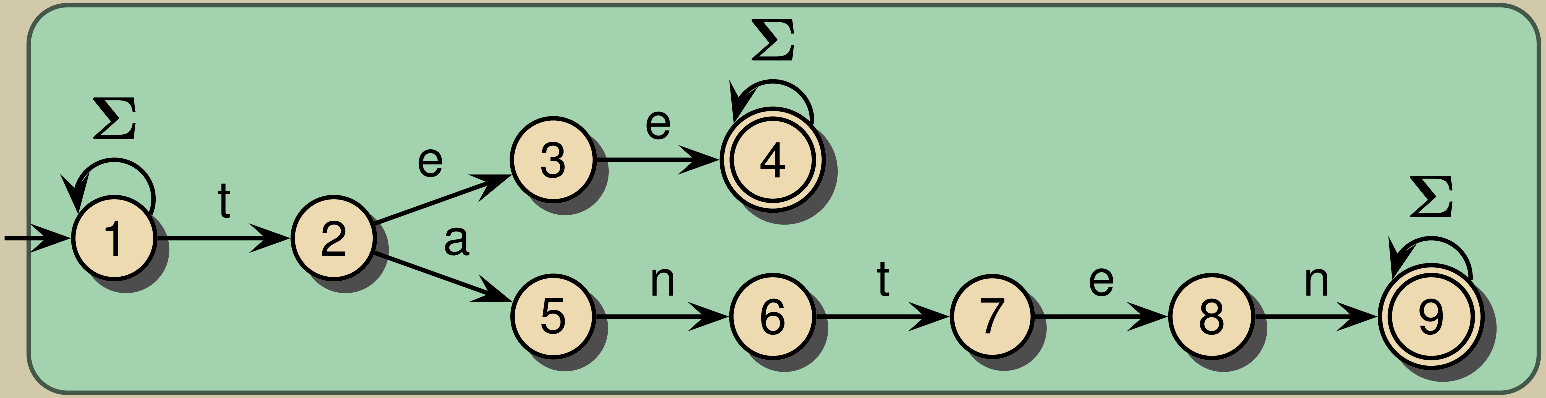
- Wie lässt sich dieser Automat deterministisch machen?
  - Klar: mit der Potenzmengenkonstruktion, aber geht es vielleicht direkter?
- Dazu betrachten wir die Eingabe: **taet|antanteeta**

Zeichenkettensuche: Beispiel (2/2)

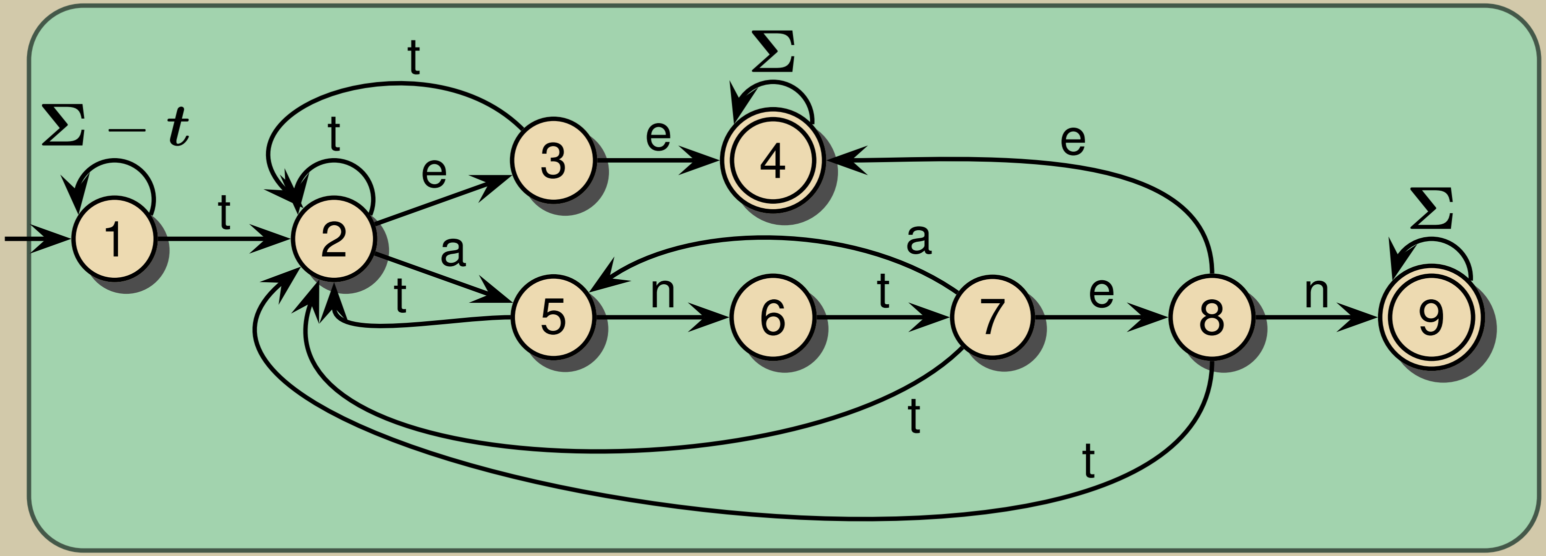
- Insgesamt erhalten wir zu dem NFA den unten angegeben DFA

  ☞ Alle nicht gezeigten Übergänge des DFA führen in Zustand 1

Beispiel: NFA für $M = \{\text{tee}, \text{tanten}\}$



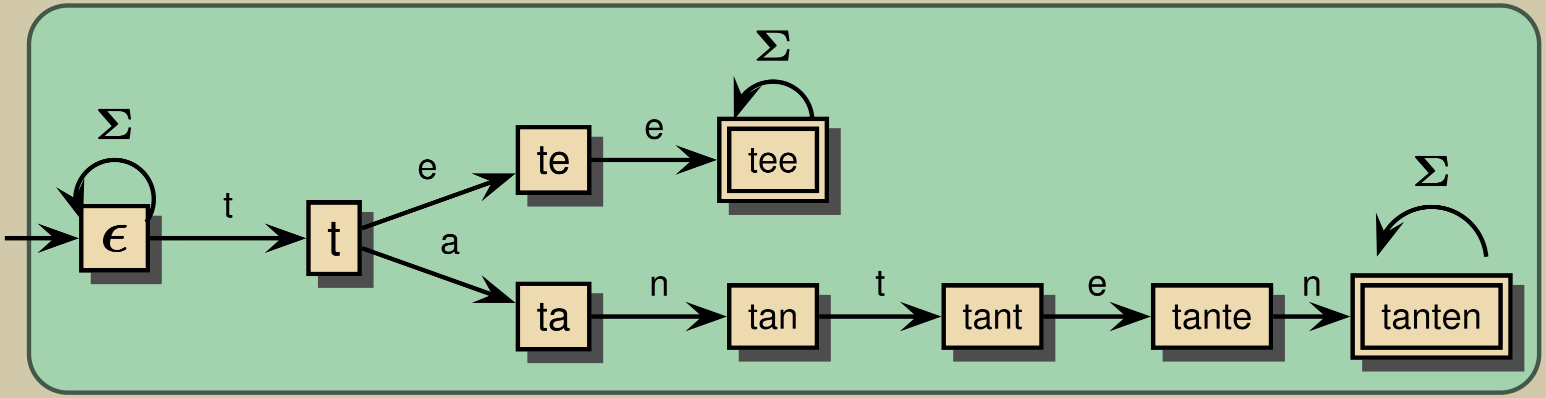
Beispiel: DFA für $M = \{\text{tee}, \text{tanten}\}$



NFA zur Zeichenkettensuche: allgemein

- Wie können wir allgemein zu einer gegebenen Menge $M = \{w_1, \ldots, w_n\}$ von Zeichenketten den NFA und den DFA formal definieren?

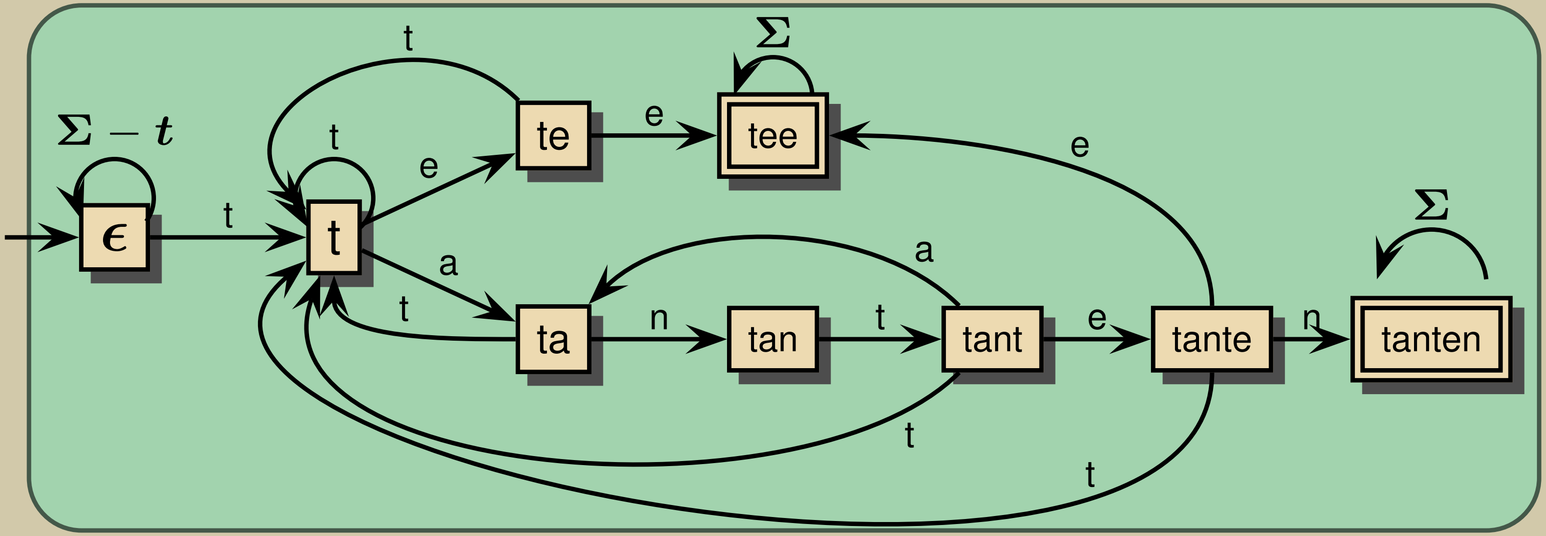
Beispiel: NFA für $M = \{\text{tee}, \text{tanten}\}$



- NFA $\mathcal{A}_M \stackrel{\text{def}}{=} (Q, \Sigma, \delta, s, F)$ mit
  - $Q \stackrel{\text{def}}{=} \{u \in \Sigma^* \mid u$ ist Präfix von w_i, für ein $i \leqslant n\}$
  - $s \stackrel{\text{def}}{=} \epsilon$
  - $F \stackrel{\text{def}}{=} M$
  - δ enthält die folgenden Transitionen für $u \in Q$ und $\sigma \in \Sigma$:
    * $(\epsilon, \sigma, \epsilon)$
    * $(u, \sigma, u\sigma)$, falls $u\sigma \in Q$
    * (u, σ, u), falls $u \in M$

DFA zur Zeichenkettensuche: allgemein

Beispiel: DFA für $M = \{\text{tee}, \text{tanten}\}$



- Der DFA $\mathcal{A}'_M \stackrel{\text{def}}{=} (Q, \Sigma, \delta', s, F)$ hat die selbe Zustandsmenge, den selben Startzustand, und die selbe akzeptierende Menge wie der NFA $\mathcal{A}_M = (Q, \Sigma, \delta, s, F)$

- Wie ergeben sich die zusätzlichen Transitionen?
- Beispielsweise gilt $\delta'(\text{tant}, \text{a}) = \text{ta}$, weil
  „ta" das längste gelesene Suffix von „tanta" ist, das auch Präfix eines Suchstrings ist

- Allgemein definieren wir:

$$\delta'(u, \sigma) \stackrel{\text{def}}{=} \begin{cases} \text{maximales Suffix } y \text{ von } u\sigma \text{ mit } y \in Q & \text{falls } u \notin F \\ u & \text{falls } u \in F \end{cases}$$

- Die Korrektheit kann durch Induktion bewiesen werden

Anwendung: Zeichenkettensuche (Forts.)

- Bei der Zeichenkettensuche bringt also die Umwandlung des NFA in einen DFA keinerlei Größenzuwachs mit sich

- Wie das Beispiel des „Borussia-Automaten“ schon gezeigt hat, kann es durchaus sein, dass sich gegenüber der hier angegebenen DFA-Konstruktion durch Minimierung sogar noch Zustände einsparen lassen

- Der Automat $\mathcal{A}'_M$ lässt sich leicht in einen **Algorithmus** umwandeln, der dann in **linearer Zeit** testet, ob ein Text einen String aus M enthält

- Durch eine einfache Modifikation lassen sich auch alle passenden Textstellen ausgeben

- In vielen Fällen ist es praktisch, zur Spezifikation des Suchmusters die Flexibilität regulärer Ausdrücke zur Verfügung zu haben

- Der reguläre Ausdruck wird dann in einen endlichen Automaten (mit Ausgabe) umgewandelt, der die Stellen des Textes, an denen das Suchmusters passt, ausgibt

- Beispiel:
  - **grep** (**G**lobal search for a **Re**gular **E**xpression and **P**rint out matched lines)
  - Varianten: egrep, fgrep
  - „**egrep** $\boldsymbol{"m(e|a)(i|y)e?r"}$ **adressen.txt**“ gibt alle Zeilen der Datei adressen.txt aus, die einen **meier-artigen** String enthalten

- Um Zeichenketten in riesigen Datenmengen zu finden (Suchmaschinen), sind natürlich erheblich ausgeklügeltere Datenstrukturen und Algorithmen nötig

Inhalt

Anwendung: Automaten-basiertes Model Checking (1/3)

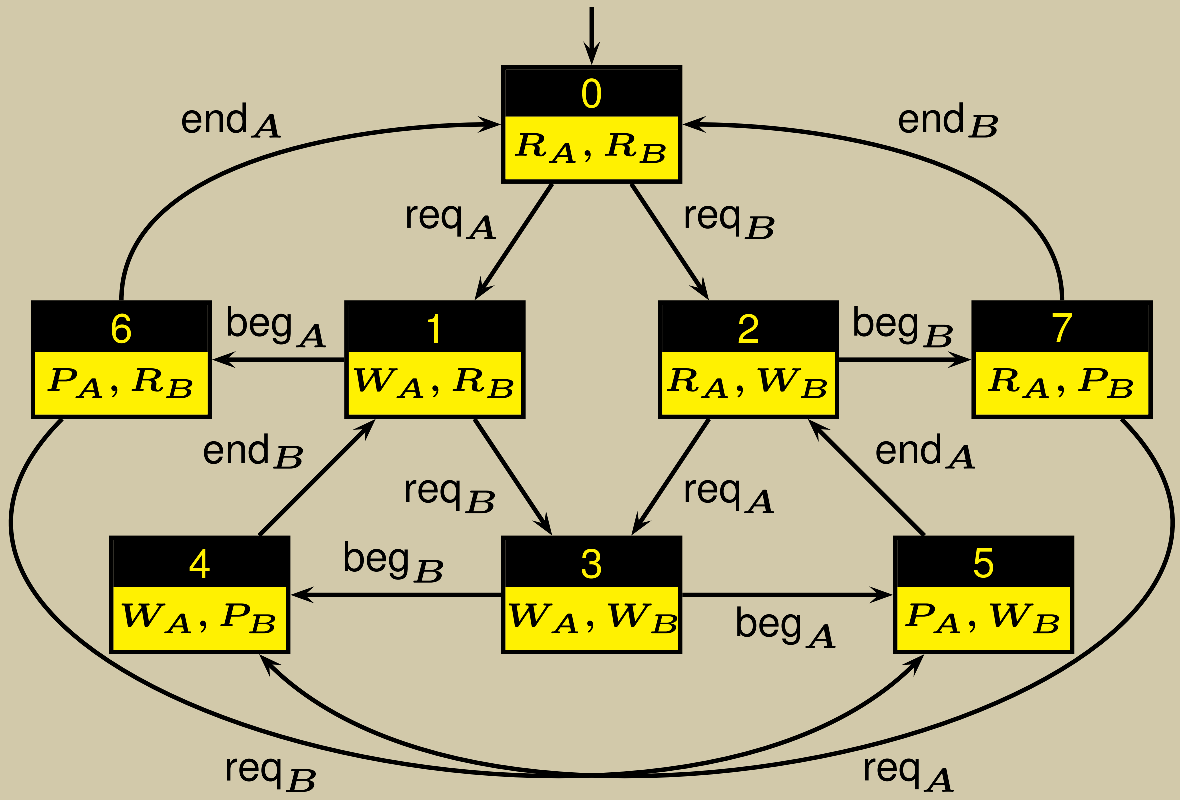
Beispiel: Drucker-Server

- 2 Benutzer (A,B), 1 Drucker
- Jeder Benutzer kann
  - auf Drucker warten (W)
  - drucken (P)
  - nichts tun (R)
- Die Aktionen des Druckers:
  - req_X: Benutzer X startet Druckauftrag
  - beg_X: Beginn des Drucks für Benutzer X
  - end_X: Ende des Drucks für Benutzer X

- Systemzustände entsprechen Zuständen eines NFA
- Transitionen entsprechen Übergängen eines NFA

➞ *endliches Transitionssystem*

Beispiel

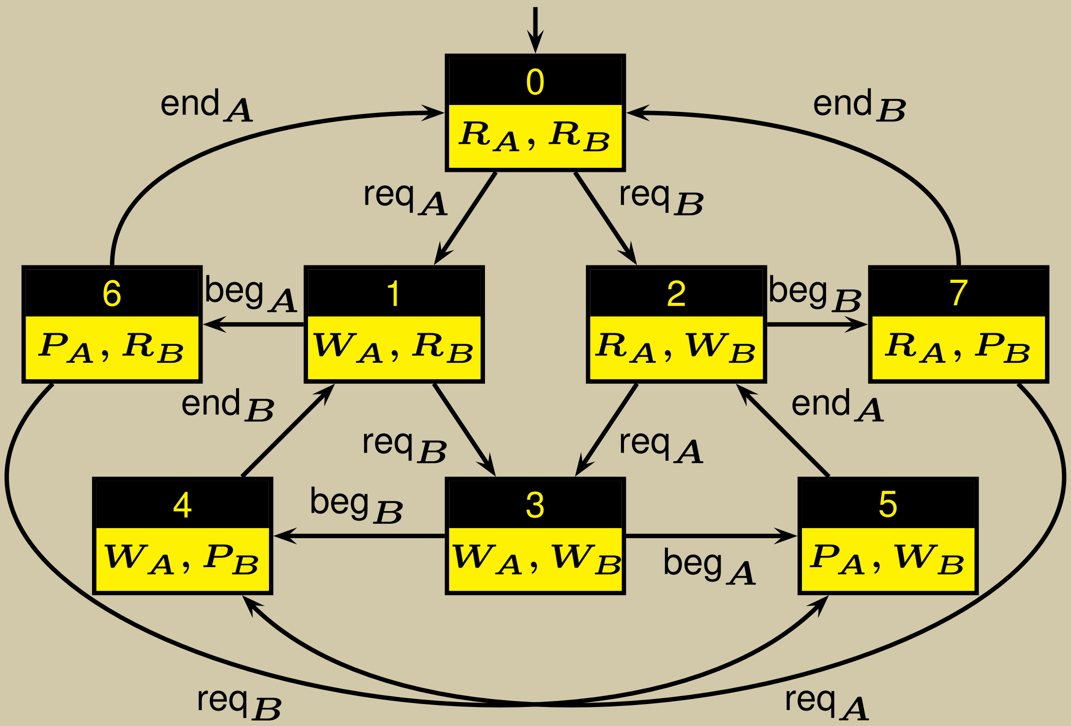


- Wie in Kripkestrukturen haben die Zustände hier jeweils noch eine Menge von Propositionen (aus: $\{W_A, P_A, R_A, W_B, P_B, R_B\}$)

✎ Wir werden uns hier aber auf Eigenschaften beschränken, die sich auf Aktionen beziehen

Anwendung: Automaten-basiertes Model Checking (2/3)

Beispiel



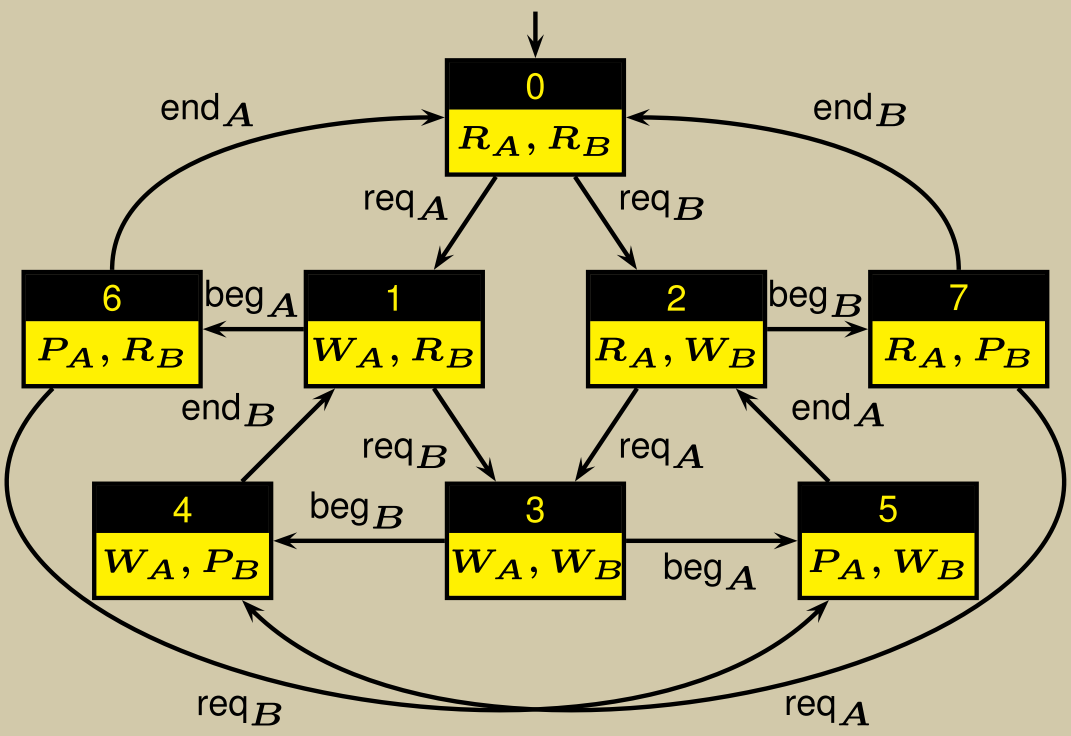
- Wir betrachten an Hand dieses sehr (!) einfachen Beispiels die automatische Verifikation von System-Eigenschaften
- Wir wollen testen, ob die Menge der möglichen Läufe des Transitionssystems eine bestimmte Eigenschaft hat

- Beispiel-Eigenschaft:
  - „Wer zuerst kommt, druckt zuerst“
  - Präziser (und eingeschränkter) sei P die Eigenschaft:

  P: Falls das erste req$_A$ vor dem ersten req$_B$ vorkommt, so soll das erste beg$_A$ vor dem ersten beg$_B$ vorkommen

Anwendung: Automaten-basiertes Model Checking (3/3)

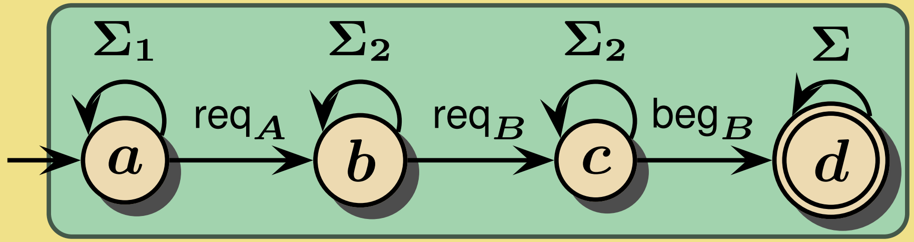
Beispiel



P: Falls das erste req$_A$ vor dem ersten req$_B$ vorkommt, so soll das erste beg$_A$ vor dem ersten beg$_B$ vorkommen

- Sei $\mathcal{A}_\mathcal{T}$ der NFA, der aus $\mathcal{T}$ entsteht, indem alle Zustände als akzeptierend gewählt werden

- Wir konstruieren einen NFA $\mathcal{A}_{\neg P}$, der alle Strings akzeptiert, die P **nicht** erfüllen:



wobei $\Sigma_1 \stackrel{\text{def}}{=} \Sigma - \{\text{req}_A, \text{req}_B\}$ und $\Sigma_2 \stackrel{\text{def}}{=} \Sigma - \{\text{beg}_A\}$ sei

- P gilt in $\mathcal{T} \iff$
$$L(\mathcal{A}_\mathcal{T}) \cap L(\mathcal{A}_{\neg P}) = \varnothing$$

- Das können wir in zwei Schritten testen:
  1. Wir konstruieren den NFA $\mathcal{B}$ für $L(\mathcal{A}_\mathcal{T}) \cap L(\mathcal{A}_{\neg P})$
  2. Wir prüfen, ob $L(\mathcal{B}) = \varnothing$ ☞ Nein!

- Aber: meistens werden Eigenschaften **unendlicher Berechnungen** spezifiziert
  → Es werden Automaten für unendliche Strings verwendet: **ω-Automaten**

Inhalt

Anwendung: UML (1/2)

- **UML** (Unified Modeling Language):
  - Beschreibungssprache für die System-Analyse und System-Entwurf

- UML verwendet verschiedene Diagrammtypen:
  - Klassendiagramm
  - Use-Case-Diagramm
  - Aktivitätsdiagramm
  - Sequenzdiagramm
  - **Zustandsdiagramm**
  - Kollaborationsdiagramm
  - Komponentendiagramm
  - Verteilungsdiagramm

Inhalt

Automatic Planning

- Beim **Planen** geht es darum, eine Schrittfolge zu finden, die eine gegebene Situation in eine angestrebte Zielsituation überführt
- Beim **automatischen Planen** soll eine solche Schrittfolge automatisch gefunden werden
- Situationen können dabei durch **Zustände** eines Transitionssystems modelliert werden
- Entsprechend werden einzelne Schritte durch Transitionen modelliert

- Ein **klassisches Planungs-Szenario** besteht aus:
  - einer Menge S von Situationen,
  - einer Anfangssituation i,
  - einer Menge G von Zielsituationen,
  - einer Menge A von Aktionen,
  - einer Menge $A(s) \subseteq A$ von möglichen Aktionen für jede Situation s
  - einer Transition $\delta(s, a)$ für jede mögliche Aktion $a \in A(s)$
  - einer Kostenfunktion $c : A^* \to \mathbb{R}_+$

- Herausforderung: Zustandsmenge wird riesig, deshalb Leerheitstest schwierig

Inhalt

XML & Document Type Declarations (DTDs)

Beispiel: XML-Dokument

⟨Composer⟩
  ⟨Name⟩ Claude Debussy ⟨/Name⟩
  ⟨Vita⟩
    ⟨Born⟩ ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
    ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
    ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
    ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩ ⟨/Died⟩
  ⟨/Vita⟩
  ⟨Piece⟩
    ⟨PTitle⟩ La Mer ⟨/PTitle⟩
    ⟨PYear⟩ 1905 ⟨/PYear⟩
    ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
    ⟨Movements⟩ 3 ⟨/Movements⟩ ...
  ⟨/Piece⟩...
⟨/Composer⟩...

Beispiel: DTD

```
<!DOCTYPE Composers [
    <!ELEMENT Composers (Composer*)>
    <!ELEMENT Composer (Name, Vita, Piece*)>
    <!ELEMENT Vita (Born, Married*, Died?)>
    <!ELEMENT Born (When, Where)>
    <!ELEMENT Married (When, Whom)>
    <!ELEMENT Died (When, Where)>
    <!ELEMENT Piece (PTitle, PYear,
         Instruments, Movements)>
    ]>
```

Eine Einschränkung von DTDs ...

- Alles, was mit XML zu tun hat, wird vom **World Wide Web Consortium (W3C)** normiert (www.w3.org)

- Zum Thema **reguläre Ausdrücke in DTDs** sagt das W3C:
  - „The content of an element matches a content model if and only if it is possible to trace out a path through the content model, obeying the sequence, choice, and repetition operators and matching each element in the content against an element type in the content model.“

    ☞ Also: reguläre Ausdrücke

  - „For compatibility, it is an error if the content model allows an element to match more than one occurrence of an element type in the content model“

    ☞ Also: spezielle reguläre Ausdrücke

  - „For more information, see **E Deterministic Content Models**“

Und in E steht dann:

- „As noted in 3.2.1 Element Content, it is required that content models in element type declarations be deterministic. This requirement is for compatibility with SGML (which calls deterministic content models ”unambiguous”); “

- „For example, the content model $((\boldsymbol{b},\boldsymbol{c})|(\boldsymbol{b},\boldsymbol{d}))$ is non-deterministic, because given an initial b the XML processor cannot know which $\boldsymbol{b}$ in the model is being matched without looking ahead to see which element follows the $\boldsymbol{b}$. In this case, the two references to $\boldsymbol{b}$ can be collapsed into a single reference, making the model read $(\boldsymbol{b},(\boldsymbol{c}|\boldsymbol{d}))$. An initial $\boldsymbol{b}$ now clearly matches only a single name in the content model. The processor doesn't need to look ahead to see what follows; either $\boldsymbol{c}$ or $\boldsymbol{d}$ would be accepted“

...und was das W3C über die Umsetzung verrät

- Und bei der genauen Erklärung kommen dann Automaten ins Spiel:
  - „More formally: a finite state automaton may be constructed from the content model using the standard algorithms, e.g. algorithm 3.5 in section 3.9 of Aho, Sethi, and Ullman [Aho/Ullman].“
  - „In many such algorithms, a follow set is constructed for each position in the regular expression (i.e., each leaf node in the syntax tree for the regular expression); if any position has a follow set in which more than one following position is labeled with the same element type name, then the content model is in error and may be reported as an error.“
  - „Algorithms exist which allow many but not all non-deterministic content models to be reduced automatically to equivalent deterministic models; see Brüggemann-Klein 1991 [Brüggemann-Klein]“

Fragen

- Was ist eine sinnvolle Definition von *deterministischen regulären Ausdrücken (DREs)* ?
- Wie lassen sich DREs erkennen?
- Kann jede reguläre Sprache von einem DRE beschrieben werden?
- Wenn nicht, wie lässt sich für eine Sprache herausfinden, ob es geht?

Beispiel

- $(a+b)^*a$ ist kein DRE, es gibt aber einen dazu äquivalenten DRE:
  $$b^*a(b^*a)^*$$
- $(a+b)^*a(a+b)$ ist kein DRE, es gibt aber keinen dazu äquivalenten DRE

Deterministische reguläre Ausdrücke: Definition

- Die folgende Definition von DREs soll garantieren, dass beim Scannen eines Wortes, jedes Zeichen des Wortes immer in eindeutiger Weise **einem** Zeichen des regulären Ausdrucks entspricht

- Also:
  - $bc + bd$ ist kein DRE
  - $b(c + d)$ ist ein DRE

- Für ordnen jedem regulären Ausdruck α einen **nummerierten regulären Ausdruck** $\alpha^{\#}$ zu:
  - Dazu nummerieren wir die Positionen von α, die Zeichen aus Σ tragen, von links nach rechts durch, beginnend mit 1:
    * $(bc + bd)^{\#} = b_1c_2 + b_3d_4$
    * $(b(c + d))^{\#} = b_1(c_2 + d_3)$

✎ Solche REs beschreiben also Strings über Alphabeten mit nummerierten Symbolen

- Ist x ein String mit nummerierten Zeichen, so bezeichnet $x^{\natural}$ den String der durch Entfernen der Nummern entsteht: $(b_1d_3)^{\natural} = bd$

- Ein RE α heißt **deterministisch** wenn es keine erweiterten Strings $x\sigma y$ und $x\tau z$ in $L(\alpha^{\#})$ gibt mit:
  - $\sigma \neq \tau$
  - $\sigma^{\natural} = \tau^{\natural}$

- Also:
  - $b_1c_2 = \epsilon \cdot b_1 \cdot c_2 = x\sigma y$ und
  - $b_3d_4 = \epsilon \cdot b_3 \cdot d_4 = x\sigma z$

  zeigen, dass $bc + bd$ nicht deterministisch ist

Pingo

PINGO-Frage: pingo.upb.de

- Welche der folgenden regulären Ausdrücke sind deterministisch?

(A) $b(ba + ca)^*a + ac$

(B) $b(ab + bc)^*a + ac$

(C) $b(ab + ac)^*b + ac$

(D) $b[(ba + ca)^*a + ac]$

Erkennen von DREs

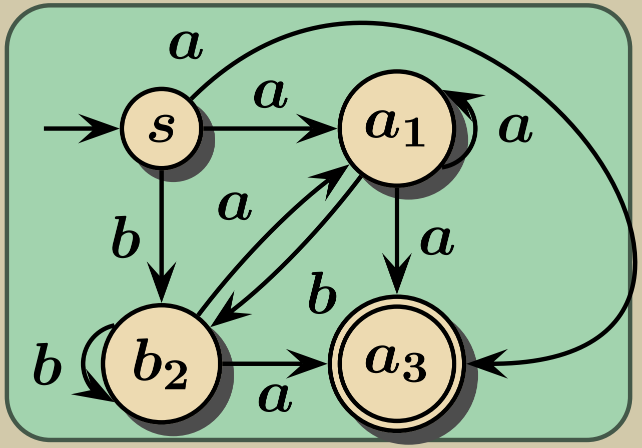
- Um zu erkennen, ob ein RE α deterministisch ist, wandeln wir ihn auf eine neue Weise in einen NFA $\mathcal{A}_\alpha$ (**Glushkov-Automat**) um:
  - Die Zustände des NFAs sind die Zeichen von $\alpha^\#$ plus ein Startzustand s
  - $\delta(s,\sigma) = \{\sigma_i \mid \sigma_i x \in L(\alpha^\#),$ für einen erweiterten String $x\}$
  - $\delta(\tau_i,\sigma) = \{\sigma_j \mid x\tau_i\sigma_j y \in L(\alpha^\#),$ für erweiterte Strings $x, y\}$
  - $F = \{\sigma_i \mid x\sigma_i \in L(\alpha^\#),$ für einen erweiterten String $x\}$

Proposition 6.1

- Ein RE α ist genau dann deterministisch, wenn $\mathcal{A}_\alpha$ deterministisch ist

Beispiel

- Beispielausdruck:
  - $\alpha = (a+b)^*a$
- Glushkov-Automat zu α:



➡ α ist nicht deterministisch

- Eine reguläre Sprache hat genau dann einen DRE, wenn die Zusammenhangskomponenten ihres Minimalautomaten eine (ziemlich komplizierte) Bedingung erfüllen [Brüggemann-Klein, Wood 97]

Inhalt

Automaten mit Ausgabe (1/3)

- Bisher haben wir nur Automaten betrachtet, die Sprachen definieren
  - Diese Automaten haben nur zwei mögliche „Ausgaben“: Akzeptieren oder Ablehnen

- Für den Einsatz in einem Compiler und für viele andere Zwecke ist es nützlich, Automaten um eine reichhaltigere Ausgabe-Komponente zu erweitern
  - ✎ Dafür werden wir als erstes ein Beispiel betrachten

Automaten mit Ausgabe (2/3)

Beispiel

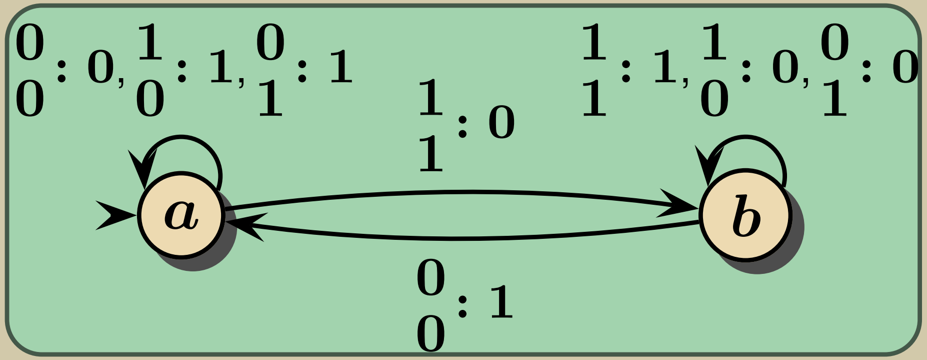
- Als Beispiel betrachten wir einen Automaten mit Ausgabe für die Addition zweier Binärzahlen $\boldsymbol{x}$ und $\boldsymbol{y}$
- Idee: der Automat liest jeweils ein Bit beider Zahlen und gibt ein Ergebnisbit aus

- Dazu werden $\boldsymbol{x}$ und $\boldsymbol{y}$ zusammen als String über $\boldsymbol{\Sigma} = \{\genfrac{}{}{0pt}{}{0}{0}, \genfrac{}{}{0pt}{}{0}{1}, \genfrac{}{}{0pt}{}{1}{0}, \genfrac{}{}{0pt}{}{1}{1}\}$ kodiert
- Jedes Zeichen kodiert ein Bit von $\boldsymbol{x}$ **und** $\boldsymbol{y}$
- Da die Addition mit den niederwertigen Stellen beginnt, wird der Eingabe-String für den Automaten umgekehrt
- Damit die Eingabe wohlgeformt ist, werden $\boldsymbol{x}$ und $\boldsymbol{y}$ mit führenden Nullen aufgefüllt
- Um Platz für einen Übertrag zu haben, werden beide Zahlen um eine führende Null erweitert

- Für $\boldsymbol{x} = \mathbf{26} = \mathbf{11010_2}$ und $\boldsymbol{y} = \mathbf{79} = \mathbf{1001111_2}$ ergibt sich also der Eingabestring: $\begin{matrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \end{matrix}$

Beispiel

$$\boxed{\genfrac{}{}{0pt}{}{0}{1}}\boxed{\genfrac{}{}{0pt}{}{1}{1}}\boxed{\genfrac{}{}{0pt}{}{0}{1}}\boxed{\genfrac{}{}{0pt}{}{1}{1}}\boxed{\genfrac{}{}{0pt}{}{1}{0}}\boxed{\genfrac{}{}{0pt}{}{0}{0}}\boxed{\genfrac{}{}{0pt}{}{0}{1}}\boxed{\genfrac{}{}{0pt}{}{0}{0}}$$



$$\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{0}$$

- Dies ist ein Beispiel für einen *Mealy-Automaten*

- Er arbeitet deterministisch
- Für jedes Eingabezeichen gibt er ein Zeichen aus

Automaten mit Ausgabe (3/3)

- Ein *Mealy-Automat* ist definiert wie ein DFA mit den folgenden Modifikationen:
  - Es gibt keine akzeptierende Menge F
  - Es gibt ein *Ausgabe-Alphabet* Γ und eine Funktion $\lambda : Q \times \Sigma \to \Gamma$, die die auszugebenden Zeichen berechnet

- Mealy-Automaten können also nur String-Funktionen f berechnen, die, für alle Strings w, die Bedingung $|f(w)| = |w|$ erfüllen
- Beispiel:
  Homomorphismen h mit $h : \Sigma \to \Gamma$

- Ein alternatives Modell sind *Moore-Automaten*:
  - Bei ihnen hängt die Ausgabe nur vom jeweiligen Zustand ab: $\lambda : Q \to \Gamma$
  - Bei Moore-Automaten ist
    $$|f(w)| = |w| + 1$$

- Es gibt viele weitere Automatenmodelle zur Definition von Stringfunktionen

- Ein sehr allgemeines Modell stellen *String-Transducer* dar:
  - Sie arbeiten nichtdeterministisch
  - Sie können in jedem Schritt einen String ausgeben (also: $\lambda(q, \sigma) \subseteq \Gamma^*$)

- Um einen Scanner zu modellieren wäre also ein deterministischer String-Transducer ein geeignetes Modell

Inhalt

Weitere Automatenmodelle: Zelluläre Automaten

- Neben einer Vielzahl von Varianten endlicher Automaten werden für viele Modellierungsaufgaben auch Modelle vernetzter Automaten verwendet
- Durch Kommunikation zwischen den Einzelautomaten können dabei äußerst komplizierte Berechnungsmodelle entstehen

- Als erstes Beispiel eines solchen Automatennetzes betrachten wir *zelluläre Automaten*

- In einem Gitter (typischerweise: $\mathbb{Z} \times \mathbb{Z}$) sind „Automaten“ verteilt
- Der Zustand jedes Automaten zum nächsten Zeitpunkt hängt von seinem aktuellen Zustand sowie von den Zuständen seiner Nachbarautomaten ab

- Einfachstes Beispiel: *Conways Spiel des Lebens* (siehe: Wikipedia):
  - Jeder Automat kann zwei Zustände annehmen: □ oder ■
  - Ein Automat nimmt den Zustand ■ genau dann an, wenn zuvor genau drei seiner Nachbarn im Zustand ■ waren, andernfalls geht er in den Zustand □

- Zelluläre Automaten eignen sich zur Simulation natürlicher und künstlicher Prozesse wie Verhalten von Gasen, Verkehrssimulation, Wachstumsprozesse,...

Inhalt

Weitere Automatenmodelle: Message Sequence Charts

- *Message Sequence Charts (MSCs)* bestehen aus endlichen Automaten, die sich über FIFO-Kommunikationskanäle Nachrichten senden können
- Damit können verteilte Systeme simuliert werden, wie zum Beispiel Telekommunikationssysteme oder auch Web Services
- Von internen Berechnungen der beteiligten Prozessoren wird üblicherweise abstrahiert

- Statt Zeichen werden Aktionen „konsumiert“, genauer:
  - Es gibt zwei Arten von Aktionen:
    - Senden einer Nachricht: „$!a$“
    - Empfangen einer Nachricht: „$?a$“
  - Nachrichten werden in den Kanälen zwischengespeichert

- Ein Beispiel findet sich im Buch von Vossen und Witt, Kapitel 4, siehe auch: http://dbms.uni-muenster.de/publications/books/Grundkurs_Theoretische_Informatik/

- Gewisse Eigenschaften solcher Systeme lassen sich dann automatisch testen (zumindest unter einschränkenden Annahmen)

Zusammenfassung

- Reguläre Sprachen und insbesondere endliche Automaten haben unzählige Anwendungen
- Es gibt viele Varianten endlicher Automaten
- Es gibt viele weitere Automatenmodelle, die das Grundprinzip endlicher Automaten in verschiedener Hinsicht verallgemeinern

Literatur für dieses Kapitel

- Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible markup language (XML) 1.0 (fifth edition). Technical report, W3C, 2008. www.w3.org/TR/2008/REC-xml-20081126
- A. Brüggemann-Klein and Wood D. Deterministic regular languages. Technical Report Bericht 38, Universität Freiburg, Institut für Informatik, 1991
- A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998
- Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, San Francisco, CA, 2004
- G. Vossen and U. Witt. *Grundlagen der Theoretischen Informatik mit Anwendungen*. Vieweg, 2000