Diskrete Ereignissysteme

Daniel Hösli

8. Dezember 2005

Inhaltsverzeichnis

1 Automaten und Sprachen

1.1 Deterministische endliche Automaten (FA)

1.1.1 Nicht formale Definition

Die nicht formale Definition von endlichen Automaten geschieht mittels des **Zustandsdiagramms** indem beliebig viele aber endlich viele Zustände definiert werden können. Dabei muss genau ein **Anfangszustand**, mindestens ein **akzeptierender Zustand** und alle entsprechend möglichen **Übergänge** definiert sein.

1.1.2 Formale Definition eines deterministischen endlichen Automaten

Diese Definition sagt uns, dass wir einen endlichen Automaten mittels dem folgenden 5-tuple beschreiben können.

- Q: eine endliche Menge von Zuständen
- $\sum$: das endliche Alphabet (Symbole, Buchstaben, Zahlen)
- δ: $Q \times \sum \rightarrow Q$ definiert die Übergangsfunktionen (bsp. $\delta(q_0, 0) = \{q_1, q_2\}$ heisst, dass von q_0 aus bei einer Eingabe '0' wir in die Zustände q_1 und q_2 kommen können. Dies muss für alle Übergänge definiert werden!)
- $q_0 \in Q$: Startzustand
- $F \subseteq Q$: eine Menge von akzeptierenden Zuständen

Weiter definieren wir als einen **String** oder ein **Wort** eine Sequenz von Symbolen ($\in \sum$). Der leere String wird mit dem Symbol ε gekennzeichnet. Mit den Betragstrichen wird auf die Länge des Wortes bzw auf die Anzahl der Symbole verwiesen so, dass $|\varepsilon| = 0$.

Wir einigen uns darauf, dass ein Eingangswort immer von links nach rechts gelesen wird, also so wie wir es lesen!

Es folgt logisch, dass ein Wort nur dann akzeptiert wird wenn es im Zustand q_0 beginnt und in einem akzeptierenden Zustand endet! Als **Sprache** bezeichnen wir die Menge aller Wörter die welche vom Automaten akzeptiert werden.

Das Ziel der endlichen Automaten ist mit endlicher Anzahl Memory (Zuständen) zu jeder Zeit sagen zu können, ob man akzeptierend ist oder nicht.

Ein FA hat bei jedem Zustand genau ein Übergang für jedes Symbol des Alphabets

TIPP: Es kann nötig sein bei endlichen Automaten einen "Fehler Zustand" einzufügen, bei dem man bei einer Abweichung hin gelangen kann und für immer bleibt. So kann man zum Teil aus einem nicht deterministischen endlichen Automaten einen deterministischen endlichen Automaten machen.

1.2 Reguläre Sprache

Eine Sprache L nennt man regulär, wenn ein endlicher Automat existiert, welcher diese Sprache erkennen kann.

Theorem: Alle endlichen Sprachen sind Regulär (aber nicht umgekehrt)

1.3 Reguläre Operationen

Wenn wir reguläre Operationen auf reguläre Sprachen anwenden, dann ist das Resultat wiederum eine reguläre Sprache!

1.3.1 Union

'Union' bezeichnet die 'oder'-Beziehung! So liefert der UNIX Befehl:

egrep -i '$a|e|i|o|u$'

alle Wörter die **mindestens einmal einen** dieser Vokale enthalten.

1.3.2 Concatenation

'Concatenation' bezeichnet die 'und'-Beziehung. So liefert der UNIX Befehl:

egrep -i '$(a|e|i|o|u)(a|e|i|o|u)$'

alle Wörter welche mindestens einen doppel-Vokal beinhalten, d.h. jegliche Verknüpfungen von Menge 1 und Menge 2 (bsp. ei, ou, uu,...) werden anerkannt.

ACHTUNG: $L \cdot \{\varepsilon\} = L$ und $L \cdot \emptyset = \emptyset$

1.3.3 Kleene-(*)

Diese Operation liefert anerkennt im gehabten Beispiel nur Zeilen, welche nur aus Vokalen besteht! Dies wird anerkannt falls das Wort aus einer Kombination genau dieser Vokalen ist (aeeeeeeei, uoiaeeeea,...) Dabei dürfen keine anderen Symbole (keine Konsonanten) auftretten, aber jeder Vokal darf 0 oder mehrmals vorkommen.

1.3.4 Kleene-(+)

Diese Operation ist genau gleich wie Kleene-* ausser, dass die leere Menge ε nicht enthalten ist!

1.4 Kartesisches Produkt

Das kartesische Produkt $A \times B$ von zwei Mengen A und B ist die Menge aller geordneten Paare (a, b) mit $a \in A$ und $b \in B$. (=alle Kombinationen von A und B)

Gegeben seien zwei deterministische endliche Automaten $M_1 = (Q_1, \sum, \delta_1, q_1, F_1)$ und $M_2 = (Q_2, \sum, \delta_2, q_2, F_2)$ der beiden Sprachen L_1 und L_2. Diese können nun über verschiedenen Operationen zusammengefügt werden:

- Vereinigung: Entweder M_1 oder M_2 muss akzeptierend sein: $M_\cup = (Q_1 \times Q_2, \sum, \delta_1 \times \delta_2, (q_1, q_2), F_\cup)$
- Schnitt: Beide müssen akzeptierend sein: $M_\cap = (Q_1 \times Q_2, \sum, \delta_1 \times \delta_2, (q_{0,1}, q_{0,2}), F_\cap)$
- Differenz: Die Differenz von zwei Zuständen ist gegeben durch $A - B = \{x \in A | x \; not \; \in B\}$ (=Akzeptiere die Zustände von M_1, ohne die von M_2): $M_- = (Q_1 \times Q_2, \sum, \delta_1 \times \delta_2, (q_{0,1}, q_{0,2}), F_-)$
- Symmetrische Differenz: (XOR) Akzeptiere genau dann wenn nur ein Automat akzeptiert: $M_\oplus = (Q_1 \times Q_2, \sum, \delta_1 \times \delta_2, (q_{0,1}, q_{0,2}), F_\oplus)$

Das **Komplement** $\bar{L}$ einer Sprache L kann gebildet werden, indem wir die akzeptierenden mit den nicht akzeptierenden Zuständen vertauschen.

1.5 Nicht deterministische endliche Automaten (NFA)

Ein Beispiel Problem das mit endlichen Automaten nur schwer gelöst werden kann ist, zu erkennen ob das viert letzte Bit eine 1 ist. Dazu müsste man wissen wie lange das Eingangswort ist, was wir aber nicht wissen können.
Es gilt neu:

- Wir dürfen beliebig viele Pfeile mit gleichen Übergängen von dem selben Zustand aus haben.
- Es müssen nicht alle möglichen Übergänge (Alphabet) von einem Zustand aus benutzt werden.
- Wir definieren neu den ε-Pfad als einen gratis Pfad, den wir gehen können ohne dass eine neue Eingabe erfolgt.

Am einfachsten ist es sich das ganze mittels 'Agenten' vorzustellen. Dabei dupliziert sich jeder Agent bei einem ε-Pfad, wovon einer zurück bleibt und einer weiter geht. Kann ein Agent in einem Zustand nicht weiter gehen bei einer neuen Eingabe, dann stirbt er. Kann er mehrere Pfad nehmen, vervielfacht er sich zu jeder möglichen Destination.

Es folgt daraus, dass ein NFA in seinem Alphabet auch ε enthält, weswegen diese auch als $\sum_\varepsilon$ bezeichnet wird. Die Übergangsfunktion eines NFA sieht folgendermassen aus:

$$\delta : Q \times \sum_\varepsilon \rightarrow P(Q)$$

wobei $P(Q)$ die Potenzmenge von Q ist.
Weiter ist somit klar, dass ein NFA in mehreren akzeptierenden Zuständen, ein einem Moment, gleichzeitig sein kann! Somit sind NFA's **parallele Maschinen**.

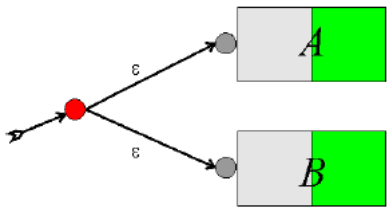
Ein Wort u ist akzeptierend von einem NFA, wenn und nur wenn irgend ein Pfad beginnend in q_0 exisitert, bei dem mindestens ein Agent am Schluss des Wortes in einem akzeptierenden Zustand ist.

ACHTUNG: ε als Eingang bewirkt nichts! und meint einfach, dass der String leer ist!

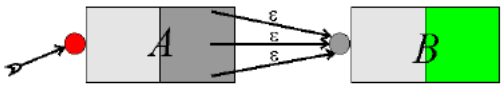
1.5.1 NFA's und reguläre Operationen

Folgend markiert die rechte (dunkle / grüne) Seite jeweils die akzeptierenden Zustände. Die linke (helle / hellgraue) Seite jeweils die nicht akzeptierenden Zustände.

- Werden 2 Automaten per Union (parallel) miteinander verbunden ergibt dies:

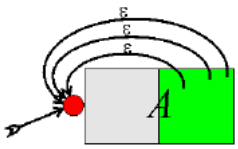


- Werden 2 Automaten per Concatenation miteinander verbunden ergibt dies:

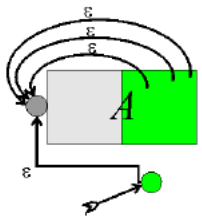


  Dabei geht von allen akzeptierenden Zuständen vom Automaten A ein Agent zum Anfang von Automat B.

- Kleene-+



- Kleene-*



Der einzige Unterschied zu Kleene-+ ist, dass hier auch der leere String ε enthalten ist!

Theorem: Wenn L_1 und L_2 von NFA's akzeptiert werden, dann auch $L_1 \bullet L_2$, L_i^+ und L_i^*

1.6 Reguläre Ausdrücke (REX)

Mit den regulären Ausdrücken können wir eine Sequenz von regulären Operationen ausdrücken. Beispielsweise wollen wir $\{banana, nab\}^*$ ausdrücken:

$$((\{b\} \bullet \{a\} \bullet \{n\} \bullet \{a\} \bullet \{n\} \bullet \{a\}\bullet) \cup (\{n\} \bullet \{a\} \bullet \{b\}\bullet))$$

Operation	Notation	Language	UNIX
Union	$r_1 \cup r_2$	$L(r_1) \cup L(r_2)$	$r_1 \mid r_2$
Concatenation	$(r_1)(r_2)$	$L(r_1) \bullet L(r_2)$	$(r_1)(r_2)$
Kleene-*	$(r)^*$	$L(r)^*$	$(r)*$
Kleene-+	$(r)^+$	$L(r)^+$	$(r)+$
Exponentiation	$(r)^n$	$L(r)^n$	$(r)\{n\}$

Man einigt sich auf folgende Ordnungsreihenfolge: $^*, \bullet, \cup$
Dies führt im oben erwähnten Beispiel zu der Vereinfachung: $(banana \cup nab)^*$

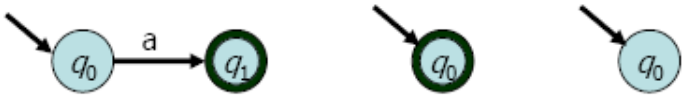
ACHTUNG: $L(\varepsilon) = \{\varepsilon\}$, aber $L(\emptyset) = \{\} = \emptyset$

Folgend einige Beispiel für die Verständlichkeit:

- 0^*10^* = irgendwo genau eine 1 in beliebig vielen 0-en.
- $(\sum\sum)^*$ = gerade Anzahl Buchstaben
- $1^*\emptyset = \emptyset$
- $\sum = \{0,1\}, \{$ w$|$ startet und endet mit dem selben Symbol $\} = 1\sum^* 1 \cup 0\sum^* 0 \cup \varepsilon \cup 1 \cup 0$

1.7 REX → NFA

Da alle NFA's aus regulären Operationen aufgebaut sind folgt, dass aus jeder Regulären Operation ein NFA gmacht werden kann! Es sind die folgenden 3 Basisoperationen definiert:
$a \in \sum, \quad \varepsilon, \quad \emptyset$



Alle weiteren Ausdrücke können mit den zuvor gesehenen regulären Operationen dargestellt werden.
Der einfachste Weg dabei ist einfach, Stur diese Kästchen einzusetzen und am Schluss oder nach den einzelnen Teiloperationen zu vereinfachen.

Es folgt, dass Sprachen, welche von einem NFA akzeptiert werden, regulär sind!

1.8 NFA → FA

Zustände von NFA's können entweder zuviele Ausgänge, zuwenig Ausgänge oder ε-Ausgänge haben. Bei zu wenigen Ausgängen können wir einfach ein CrashZustand einführen, und haben dann bereits aus einem NFA einen FA gemacht, bei den anderen geht das nicht so einfach.
Wir können mit folgendem vorgehen, aber jeden NFA in einen FA umwandeln:

- Anstelle der n Zustände des NFA kommen neu 2^n Zustände des FA. (Bsp. NFA hat Zustände 1 und 2, dann hat der FA die Zustände $\emptyset$, 1, 2, 12).
- Wir markieren die Start und Endzustände. (Bsp. Ist 1 der Startzustand des NFA und über ε kann der Zustand 2 erreicht werden, dann ist der Startzustand des FA 12. Ist der Endzustand im NFA der Zustand 2, dann sind die Endzustände im FA die Zustände 2 und 12, also überall wo 2 vorkommt).
- Wir zeichnen die Übergänge ein. Dabei muss immer bei ε-Pfeilen aufgepasst werden!
- Schlussendlich kann der FA meistens noch vereinfacht werden. So werden meistens nur wenige der 2^n-Zustände gebraucht, oder einige Zustände können zusammengelegt werden!

zu vereinfachen.

Es folgt, dass für jede Sprache L, welche von einem NFA akzeptiert wird, auch ein FA existiert!

1.9 NFA → REX

Um aus einem NFA die REX zu bekommen müssen wir einen Umweg machen über einen generalisierten nichtdeterministischen endlichen Automaten (GNFA). Ein GNFA ist ein Graph dessen Kanten mit regulären Ausdrücken beschriftet sind! Dabei muss gelten:

- Der Startzustand muss eindeutig sein, d.h. er darf keine eingehenden Kanten besitzen.
- Der Endzustand muss eindeutig sein, d.h. er darf keine ausgehenden Kanten besitzen.

Der Ablauf ist dementsprechend, dass wir den Automaten Schritt für Schritt zusammenfassen:

- Mehrere Eingänge werden mit $\cup$ zusammengefasst.
- Evt. müssen der eindeutigen Start- und Endzustand erst erstellt werden (mittels ε davor oder danach!
- Wenn keine Kante von einem Zustand zu einem anderen existiert, dann füge eine Kante ein und beschrifte sie mit $\emptyset$
- Dies wird solange gemacht, bis wir nur noch den Start- und Endzustand haben und somit die REX direkt von der Verbindungskante ablesen können!

Definition: Ein Automat ist nicht weiter reduzierbar, wenn er keine nutzlosen Zustände hat und wenn keine Zustände equivalent sind.

Mit diesen 2 Regeln, lässt sich ein global minimaler Automat bilden!

1.10 Pumping Lemma

Taubenschlag-Prinzip (Pigeonhole principle) Wenn m Tauben auf $n < m$ Taubenschläge verteilt werden müssen, müssen in mindestens einem Taubenschlag 2 Tauben sein.

Idee für FA's: Ein Automat der Sprachen erlaubt mit unendlich langen Strings, muss mindestens eine Schlaufe enthalten! Das heisst, wir teilen den FA auf in 3 Teile, einen Anfang x, eine Schlaufe y und ein Ende z. Die Schlaufe kann nun zwischen 0 und ∞-Mal durchlaufen werden, was zur folge hat, dass die Sprache folgende Worte enthält: $[xz, xyz, xy^2z, xy^3z, ...]$.

Können wir nun beweisen, dass solch eine Schlaufe nicht existieren kann für eine Sprache mit unendlich langen Wörtern, dann haben wir bewiesen, dass es sich um keine reguläre Sprache handelt!

Theorem: Gegeben sei eine reguläre Sprache L. Dann existiert eine Zahl p (=pumping number), so dass alle Worte in L mit Länge $\geq p$ in drei Teile eingeteilt werden können (x,y,z) und dann folgende Bedingungen erfüllen:

- $|y| \geq 1$
- $|xy| \leq p$

- $xy^iz \in L$ für alle $i \geq 0$

Um nun zu beweisen, dass eine bestimmte Sprache nicht regulär ist, muss man zeigen, dass für alle möglichen Typen von y die es geben kann, keiner die genannten Eigenschaften erfüllt, dass für ein $i \geq 0$ das neue Wort wieder von L akzeptiert wird.

2 Intelligentere Automaten

2.1 Kontext-freie Grammatik (CFG)

Kontext-freie Sprachen sind mächtiger als reguläre Sprachen. So können diese durch rekursive Strukturen auch Wörter erkennen, wo die reguläre Sprache versagen würde. Solche Wörter müssen einfach ein Muster aufweisen, wie beispielsweise das Palindrom. Eine wichtige Anwendung ist die Sprachanalyse.

Eine Kontext-freie Grammatik besteht aus einem 4-Tupel $(V, \sum, R, S)$, wobei:

- V: eine endliche Menge von Variablen (Symbolen) ist. (Normalerweise mit Grossbuchstaben gekennzeichnet).
- $\sum$: eine endliche Menge von 'Terminals'. (Dürfen nicht in V vorkommen und sind üblicherweise mit Kleinbuchstaben und Sonderzeichen definiert)
- R: eine endliche Menge von Regeln (oder 'Productions') der Form $A \rightarrow w$. Wobei links (A) immer eine Variable sein muss, und w eine Zusammenstellung von Variablen und/oder Terminals sein kann.
- S: ist die Startvariable.

Jede Grammatik definiert mit ihren Regeln eine Sprache. Um diese Sprache zu finden, generieren wir am einfachsten einen Ableitungsbaum (parse tree), indem wir einfach beim Startpunkt anfangen und dann jeweils für alle weiterführenden Möglichkeiten einen neuen Ast öffnen. Dabei gilt, dass wir die Blätter von links nach rechts ableiten! Dies kann ewig gemacht werden, oder solange bis wir die Struktur der Sprache erkannt haben :-)

Die gefundene Sprache der Grammatik G_1 bezeichnen wir als $L(G_1)$.
Jede Sprache die von einer Kontext-freien Grammatik erzeugt werden kann, nennen wir eine **Kontext-freie Sprache (CFL)**. Diese beinhaltet dementsprechend alle Terminalen-Strings welche wir von der entsprechenden Kontext-freien Grammatik herleiten können!

Definition: Wenn u, v und w Strings mit Variablen und Terminals sind UND $A \rightarrow w$ eine Regel der Grammatik ist, dann sagen wir, dass uAv, uwv herleitet oder produziert. Geschrieben: $uAv \Rightarrow uwv$.

Definition: Wir schreiben $u \Rightarrow^* v$ wenn $u = v$ oder wenn wir v aus u über eine Sequenz von Substitutionen herleiten können. $(u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow ... \Rightarrow v)$

Siehe Beispiel: Infix Expressions in Kapitel 2 S.7 ff

2.2 Mehrdeutigkeit

Gerade wenn wir richtige Sprachen betrachten und diese nach den Regeln der Nomen, Verben, Präpositionen usw. aufteilen, dann können Sätze entstehen wie zum Beispiel "Hannibal ist Reiss mit Clarice" was bedeuten kann, dass Hannibal Reiss ist und Clarice ist, oder dass Clarice und Hannibal Reiss essen.
Solche Mehrdeutigkeiten entstehen, da wir mehrere Möglichkeiten haben diesen Satz zu bilden. D.h. es existieren mehrere Ableitungsmöchglichkeiten dieses Satzes bzw. verschiedene Ableitungsbäume, welche genau diesen Satz ergeben!
Bei Grammatiken welche Regeln der Form $A \to T|E+T$ enthalten, haben wir die Möglichkeit den vorgegebenen String entweder nach Rechts oder nach Links aufzulösen, wodurch bereits 2 verschiedene Ableitungsbäume entstehen!

2.3 Prüfen, dass $L = L(G)$

Falls man eine CFG G und eine Sprache L gegeben hat und beweisen muss, dass $L(G) = G$, so muss man zeigen, dass:

1. $L \subseteq L(G)$: alle Worte in L können mit G generiert werden.
2. $L \supseteq L(G)$: G generiert nur Worte, die in L sind (meistens einfach)

Um Nummer 1 zu beweisen müssen wir induktiv vorgehen!

Beispiel: $G = (S \to \varepsilon|ab|ba|aSb|bSa|SS)$
Wir vermuten, dass $L(G) = \{x \in \{a,b\}^* | n_a(x) = n_b(x)\}$

- Beweis, dass es für einen Wert stimmt: Für ε stimmt es über $S \to \varepsilon$
- Induktive Hypothese: Wir nehmen an, dass x ein solcher String mit gleich vielen a's wie b's ist. Desweiteren nehmen wir an, dass u der kleinste Präfix von $x \in L$ ist.
  Entweder ist nun $|u| < |x|$, dann gilt dass $x = uv$ wobei $v \in L$ sein muss, da auch $x, u \in L$ sind. Hier können wir dann $S \to SS$ ausnutzen um dieses Argument zu wiederholen.
  Oder $x = u$. Hier kann u nicht mit dem gleichen Buchstaben beginnen und enden, da wir sonst $u = avb$ schreiben könnten wobei v mindestens zwei b's enthalten müsste, was wiederum heissen würde, dass u nicht der kleinste Präfix wäre! Deswegen können wir für einen beliebigen String $v \in L$ schreiben: $u = avb$ oder $u = bva$ was wir wiederum mit $S \to aSb$ oder $S \to bSa$ realisieren können.

2.4 Kontext-freie Grammatiken Designen

Kontext freie Grammatiken zu kreieren, ist ein kreativer Prozess. Trotzdem gibt es einige Tricks, die eine das ganze einfacher machen!

- Viele CFG's sind Vereinigungen von einfacheren CFG's. Es ist also empfehlenswert das Ganz in so einfach CFG's wie möglich zu implementieren und dann diese mittels dem 'ODER' Operator (|) zu verknüpfen!. (Bsp. $S \to S_1|S_2|S_3...$)

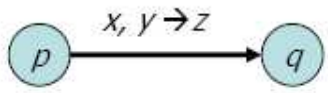
- Eine CFG für einen reguläre Automaten zu kreieren ist einfach. Man muss nur zuerst den deterministischen FA konstruieren und danach wie folgt vorgehen:
  - Aus jedem Zustand wird eine Variable gemacht. (aus q_i wird R_i)
  - Wir erzeugen für die Grammatik die Regel $R_i \rightarrow aR_j$ falls $\delta(q_i, a) = q_j$.
  - Wir erzeugen für die Grammatik die Regel $R_i \rightarrow \varepsilon$, wenn q_i ein akzeptierender Zustand ist.
  - Für den Startzustand setzen wir S für q_0 ein.
- Rekursive Strukturen können via $R \rightarrow uRv$ implementiert werden.

2.5 Push Down Automaten (PDA)

PDA's sind Automaten welche CFG's realisieren können! Der PDA erlaubt im Gegensatz zu den FA's und NFA's einen **Stack**. Mit dem Stack können wir die folgenden Basisoperationen ausführen:

- **Push**: ein neues Element zuoberst anfügen.
- **Pop**: das oberste Element entfernen.
- **Peek**: das oberste Element überprüfen, ohne zu entfernen.

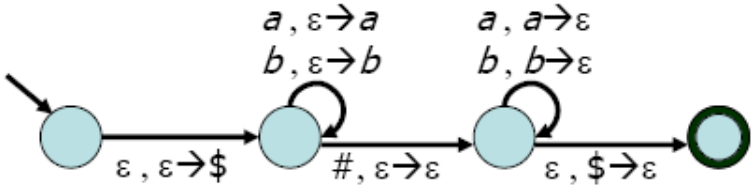
Für die Beschriftung verwenden wir Sipser's Version:



Wenn beim Zustand p der nächste Input x ist und wenn das oberste Element vom Stack y ist, dann gehe zu Zustand q und ersetze y durch z auf dem Stack. Oder anders ausgedrückt: x=Input, y=Pop, z=Push
Dabei gilt:

- $x = \varepsilon$: ignoriere den Input und lese nicht!
- $y = \varepsilon$: ignoriere das oberste Element auf dem Stack und 'push' z
- $z = \varepsilon$: pop y

Als ein Beispiel ist unten der Palindrom-PDA abgebildet:



Merke: Zuerst (wenn der Stack leer ist) wird das Symbol $ drauf gepusht! Am schluss kann man dann einfach wieder lesen, ob das oberste Zeichen $ ist!

2.5.1 Formale Definition des PDA's

Ein PDA besteht aus dem 6-Tupel $M = (Q, \sum, \Gamma, \delta, q_0, F)$, wobei:

- Q: eine endliche Menge von Zuständen
- $\sum$: das endliche Alphabet (Symbole, Buchstaben, Zahlen)
- $q_0 \in Q$: Startzustand
- $F \subseteq Q$: eine Menge von akzeptierenden Zuständen
- Γ: ist das 'tape'-Alphabet
- δ: ist die Übergansfunktion, bestimmt durch den aktuellen Zustand, den nächsten Input und das oberste Symbol auf dem Stack: $Q \times \sum_{\varepsilon} \times \Gamma_{\varepsilon} \to P(Q \times \Gamma_{\varepsilon})$

TIPP: Wenn man 2 Buchstaben gegeneinander abzählen soll, dann kann man einfach für den ersten ein neues Element pushen und für den Zweiten eins poppen. Da wir aber im Stack nicht ins negative gehen können, bedienen wir uns einer Zusätzlichen Variablen: Beispiel wir wollen Strings untersuchen welche gleich viele a wie b haben. Wir zählen $+1$ wenn ein a kommt und -1 wenn ein b kommt. Ist der String leer, und es kommt ein b, dann setzen wir jetzt einfach b drauf und wechseln die push- mit der pop-Beziehung!

2.6 Rechtslineare Grammatiken:

Definition: Eine rechtslineare Grammatik ist eine CFG bei der jede Produktion in der Form $A \to uB$ oder $A \to u$ ist, wobei wobei u ein Terminalsymbol und A, B Variablen sind.

Für jeden NFA existiert eine rechtslineare Grammatik, welche dieselbe Sprache hat!

Nicht jede CFG kann in eine rechtslineare Grammatik umgeformt werden!

2.7 Chomsky Normal Form

Die Chomsky Normal Form ist eine einfache Darstellung einer CFG, wobei alle CF-Sprachen dargestellt werden können.

Definition: Eine CFG ist in CNF falls jede Regel der CFG eine der folgenden Formen hat:

- $S \to \varepsilon$
- $A \to BC$
- $A \to a$

2.7.1 CFG → CNF

Um nun eine beliebige CFG in CNF umzuformen gehe wie folgt vor:

1. Die Startvariable darf nur auf der linken Seite stehen. (D.h. evt eine neue Startvariable bilden!)
2. Entferne alle ε ausser jener von der Startvariable aus.
3. Entferne alle reinen Variablen-Übergänge der Form $A \to B$
4. Kürze vorhandene Regeln nach den Regeln der CNF

2.7.2 CFG → GPDA → PDA

Um eine CFG in einen PDA umzuwandeln gehen wir über einen Generalisierten PDA (GPDA). Der GPDA hat den einzigen Unterschied, dass es erlaubt ist einen ganzen String auf einmal zu pushen. Um dann den dazugehörigen PDA zu bekommen, muss nur noch dieser String in seine Einzelteile zerlegt und diese einzel gepusht werden! Gehe dazu folgendermassen vor:

- Pushe das Markierungssymbol \$ auf den Stack
- Wiederhole folgende Schritte für immer:
  - Wenn zuoberst auf dem stack eine Variable liegt, wähle eine Regel zur Ersetzung der Variablen und pushe den String auf der rechten Seite der Ersetzungsregel auf den Stack. (z.B. $\varepsilon, A \to aTb$)
  - Liegt ein Terminalsymbol a zuoberst auf dem Stack, dann lese das nächste Symbol vom Input. Ist es a, dann fahre fort, sonst liegt ein Fehler vor. $(a, a \to \varepsilon)$
  - Wenn \$ zuoberst lieght, gehe in den akzeptierenden Zustand $(\varepsilon, \$ \to \varepsilon)$

2.7.3 Non-Context-Free-Grammars

Einen NCFG unterscheidet sich von der CFG nur, dass auf der linken Seite neu nicht nur eine Variable, sondern mehrere Variablen stehen können. Somit können gleich ganze Strings ersetzt werden! (bsp. $AB \to BC$)

2.7.4 Context Sensitive Grammars

Für CSG's gilt einfach, dass die Länge der Linken Seite vom Pfeil immer kleiner oder gleich lang als die rechte Seite sein muss.

2.7.5 PDA → CFG

Für jeden PDA gibt es eine CFG, welche die Sprache akzeptiert. Somit sind der PDA und die CFG gleich mächtig. (kein Rezept)

2.8 Tandem Pumping

Analog zum Pumping Lemma mit dem man beweisen kann, dass eine Sprache nicht-regulär ist, gibt es das Tandem Pumping um zu beweisen, dass eine Sprache nicht kontextfrei ist.

Gegeben sei eine kontextfreie Sprache L. Dann existiert eine Zahl p, so dass jeder String in L der Länge $\geq p$ tandem pump-bar ist innerhalb eines Substrings der Länge p.

- $w = uvxyz$
- $|vy| \geq 1$
- $|vxy| \leq p$
- $uv^ixy^iz \in L$ für alle i grösser 0

2.9 Transducers

Definition: Ein endlicher Zustands-Transducer (FST) ist ein endlicher Automat welcher einen String als Ausgang hat und nicht nur akzeptierend/nicht akzeptierend.
Dazu werden alle Übergänge mit zwei Symbolen beschriftet (bsp. 1/2 heisst, bei Input=1 wird Output=2 dem String angehängt).

2.10 Turing Maschine

Eine TM ist ein Gerät mit einer endlichen Grösse von read-only-memory (=Zustände) und einer unendlichen Grösse von read/write tape-memory. Dabei wird angenommen, dass der Input auf dem tape-memory bereit steht, wenn die Maschine angelassen wird.

Ein PDA mit 2 Stacks ist so mächtig wie eine TM und diese ist gleich mächtig wie jeder noch so gute Computer (nur etwas langsamer)

3 Specification Models

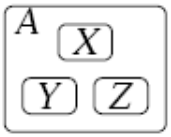
3.1 State Charts

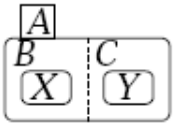
Mit den State Charts sollen die Nachteile von endlichen Automaten behoben werden. Dies sind, dass nur ein einzelner Prozess sequentiell ablaufen kann und dass keine hierarchischen Strukturen

möglich sind.

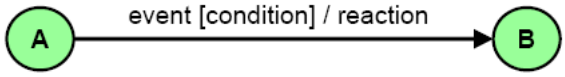
Wir können nun einzelne Zustände (Basic-states) zusammenfassen zu Super-states. Super-states welche Basic-states enthalten werden Ancestor-states genannt.

Grundsätzlich gibt es 2 verschiedene Arten von Super-states: Den OR-Super-state und den AND-Super-state. Beim OR-Super-state ist genau ein Zustand aktiv, wobei im AND-Super-state alle direkten Nachfolgezustände aktiv sind.





Die Übergänge von einem zum nächsten Zustand sind wie folgt beschriftet:



Dabei kann das Ereignis intern oder extern generiert werden. Wenn keine Bedingung angegeben wird, dann gehen wir zu B sobald das Ereignis geschieht. Ansonsten wenn beides steht, gehen wir nur weiter, wenn das Ereignis eintritt und die Bedingung erfüllt ist. Die Reaktion oder Aktion kann Variablen verändern oder Ereignisse auslösen!
Die Simulation läuft dann in 3 Schritten ab:

1. Auswirkung von Veränderungen der Ereignisse und Bedingungen werden ausgewertet.
2. Es wird geschaut welche Übergänge gemacht werden und es werden die rechten Seiten der Zuweisungen berechnet und temporär gespeichert (Dies ist wichtig, denn so kann beispielsweise ein Variablenaustausch stattfinden, so dass gleichzeitig a=b und b=a einander zugewiesen werden können!).
3. Übergänge werden gemacht, Variablen erhalten die neuen Werte.

Betrachte 3/14 bis 3/17 nochmals eingehend!

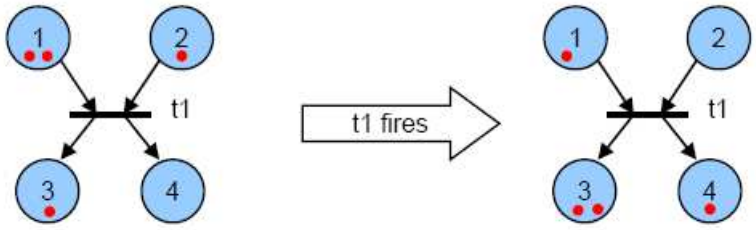
3.2 Petri-Netze

In Petri-Netzen finden die Zustandsübergänge asynchron statt und können somit zum modellieren von gleichzeitigen Prozessen gebraucht werden.
Ein Petri-Netz ist ein bipartiter[1] Graph definiert durch folgendes 4-Tupel:

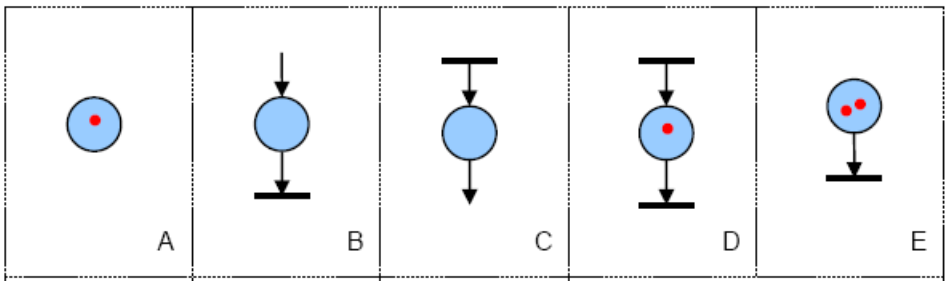
- S ist eine Menge von Orten

[1]Ein einfacher Graph G = (V,E) (V Menge der Knoten, E Menge der Kanten) heißt in der Graphentheorie bipartit (auch paar), falls sich seine Knoten in zwei disjunkte Teilmengen A und B aufteilen lassen, so dass zwischen den Knoten innerhalb beider Teilmengen keine Kanten verlaufen.

- T ist eine Menge von Übergängen
- F ist eine Menge von Kanten (Flussverhältnisse)
- M_0 ist die Anfängliche Verteilung der Tokens



Ein Übergang in einem Petri-Netz ist dann aktiv, falls bei allen eingehenden Verbindungen, mindestens ein Token liegt. Ist dies der Fall, dann kann dieser Übergang abgefeuert werden, wobei von allen Eingehenden Zuständen ein Token abgezogen wird, und zu allen Nachfolgenden Zuständen ein Token zugefügt wird.

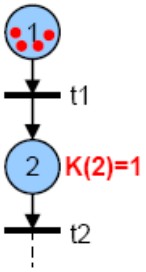


Wir sehen im obigen Bild, dass A,B und C keine gültigen Petri-Netze sind, D und E aber schon. D zeigt dabei einfach ein Zustand mit einer **Quelle** und einer **Senke**.
Zusätzlich können die Übergänge noch gewichtet werden. Steht neben dem QuellPfeil eine Zahl z.B. 2, dann werden vom entsprechenden Quellzustand 2 Token abgezogen, oder falls die 2 beim Zielpfeil steht, dann werdem dem Zielzustand 2 Token beigefügt!

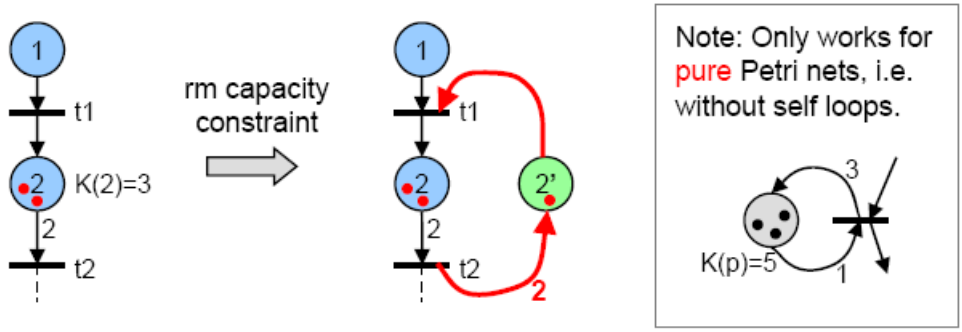
3.2.1 Kapazitätsbeschränkungen

Manchmal möchte man einen Zustand nur eine gewisse Kapazität gewähren. Dies kann auf 2 verschiedene Varianten gemacht werden.

Methode 1: Wir schreiben ganz einfach die Kapazität des entsprechenden Zustandes hin!



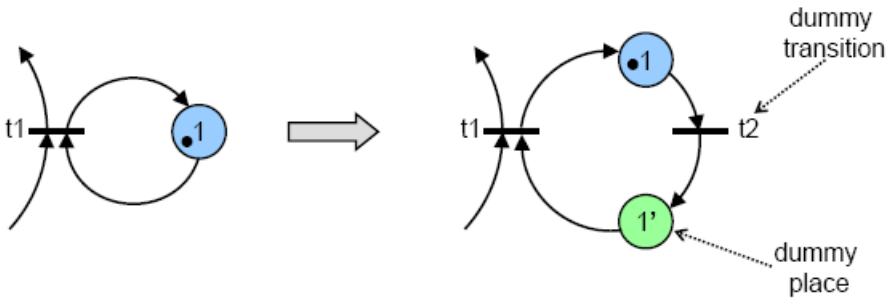
Methode 2: Wir führen einen Zusätzlichen Zustand ein, und verknüpfen diesen so, dass es einfach nicht mehr möglich ist zu feuern, wenn die Kapazitätsgrenze erreicht worden ist.



Gehe dazu wie folgt vor:

- Füge einen Komplementären Zustand ein, und fülle ihn mit sovielen Tokens, dass die Anzahl der Tokens der beiden Zustände (zu Beginn) gleich der Kapazität ist.
- Füge einen Übergang vom Komplementär Zustand zu jedem 't' welches einen Übergang zum Original Zustand hat.
- Füge von jedem 't' das vom Originalzustand ein Token erhält einen Übergang zum Komplementären Zustand hinzu.

ACHTUNG: Bei Eigenschleifen funktioniert das ersetzen der Kapazität nach der genannten Methode nicht! Deswegen ist es nötig bei solchen Schleifen einen 'Dummy'-Zustand einzufügen!



3.2.2 Petri-Netz-Sprachen

Die 'Transitions' (t's) können nun mit Buchstaben / Symbolen beschriftet werden und können somit Sprachen generieren. Dabei wird ein Symbol dem Wort hinzugefügt, wenn ein Token durch t durchläuft!
Es gilt:

Jede endliche Maschine kann als Petri-Netz modelliert werden. Und jede Reguläre Sprache ist eine Petri-Netz-Sprache

K-Boundedness: Wir sagen, dass ein Petri-Netz K-Gebunden ist, falls die Anzahl der Tokens in jedem Zustand, K nicht überschreitet.

Safety: =1-Boundedness = Jeder Zustand hat mindestens ein Token zu jeder Zeit!

Liveness: Wir unterscheiden folgende 'Transitions' (t's):

- **dead**: Wenn t niemals abgefeuert werden kann.
- **L1-live**: Wenn t mindestens einmal im Verlauf abgefeuert werden kann.
- **L2-live**: Wenn t beliebig viele Male abgefeuert werden kann bei einer entsprechenden Sequenz.
- **L3-live**: Wenn t unendlich viele Male in einer bestimmten Sequenz erscheint.
- **L4-live**: Wenn t L1-live ist, für jede mögliche Kombination von M_0 aus.

Merke: L4 $\Rightarrow$ L3 $\Rightarrow$ L2 $\Rightarrow$ L1

3.3 Analyse Methoden

3.3.1 Coverability Tree

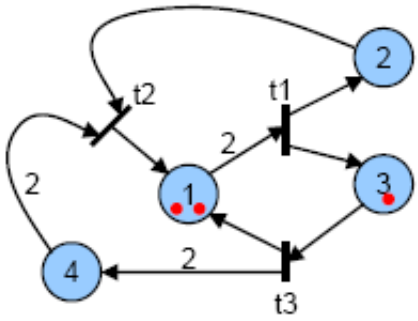
Mit diesem Baum können wir analysieren, welche Token-Verteilungen möglich sind. Dazu schreiben wir die Anfangsverteilung der Tokens in einen Zeilenvektor bsp. [100202] dabei bezeichnet jede Spalte einen Zustand und die Zahl, die anfänglichen Anzahl der Tokens.
Wir schauen nun, welche Transitionen aktiviert werden können und zeichnen somit den aufspannenden Baum auf mit den Übergangsbeschriftung der entsprechenden Transition (bsp. $t1$). Wenn wir irgendwo eine beliebige Anzahl von Tokens in einem Zustand haben können, markieren wir dies mit dem Symbol ω also bsp. [01ω301].

Wir sagen dann:

- dass das Netz beschränkt (bounded) ist, wenn ω in keinem Knoten des Baumes auftaucht.
- dass das Netz sicher (safe) ist, wenn und nur wenn nur 0-en und 1-en im Vektor erscheinen.
- dass eine Transition tot (dead) ist wenn wir von einem Vektor aus keine Transition mehr aktivieren können.
- dass ein Vektor v erreichbar (reachable) ist, wenn er von M_0 aus über eine bestimmte Transitionsfolge erreichbar ist.

3.3.2 Incidence Matrix

Durch die Incidence Matrix können wir ein Petri-Netz über eine Matrix beschreiben. Dabei sind die Spalten die Transitionen und die Zeilen die Zustände. (siehe folgendes Beispiel)



$$\mathbf{A} = \begin{bmatrix} -2 & 1 & 1 \\ 1 & -1 & 0 \\ 1 & 0 & -1 \\ 0 & -2 & 2 \end{bmatrix}$$

$$\mathbf{M_0} = \begin{bmatrix} 2 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

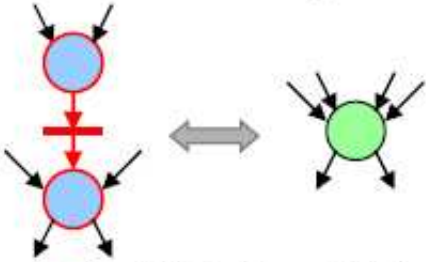
Dadurch können wir einen Spaltenvektor u beschreiben, welche sagt welche Transitionen abfeuern sollen. Es gilt dann die '**state equation**'

$$M_k = M_0 + A \sum_{i=1}^{k} u_i$$

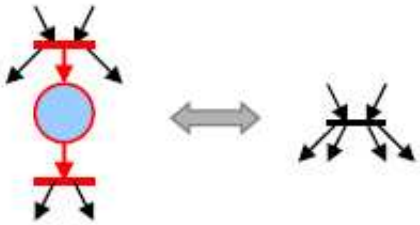
Wobei A die Incidence Matrix ist und M_0 die Startverteilung der Tokens.
Hat M_k negative Werte, dann wurde eine nicht genemigte Transition aktiviert. Es folgt, dass wenn M_k nur positive Werte hat dies keine genügende aber eine notwendige Bedingung für Erreichbarkeit ist!

3.3.3 Reduktions Regeln

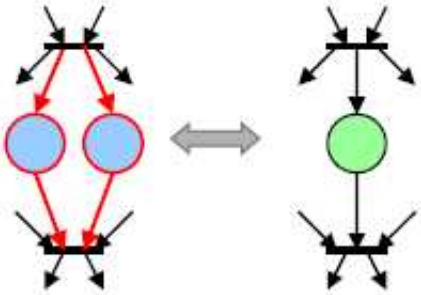
Folgende Regeln können gebraucht werden um ein Petri-Netz zu vereinfachen:



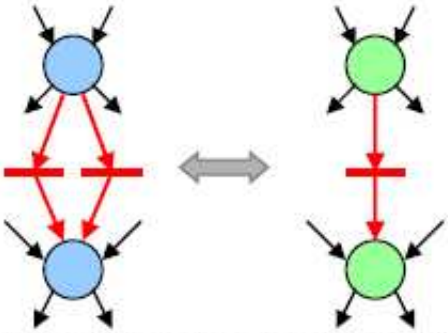
Fusion of Series Places (FSP)



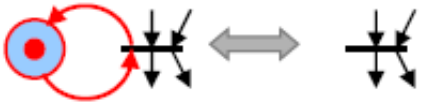
Fusion of Series Transformations (FST)



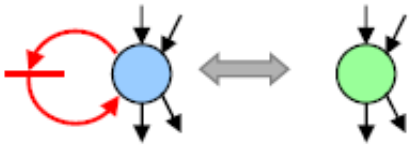
Fusion of Parallel Places (FPP)



Fusion of Parallel Transformations (FPT)



Elimination of Self Loop Places (ESP)



Elimination of Self Loop Transitions (EST)