

Formale Grundlagen 3

#342215

SS 2007

Johannes Kepler Universität
Linz, ÖsterreichUniv. Prof. Dr. Armin Biere
Institut für Formale Modelle und Verifikation<http://fmv.jku.at/fg3>

Formale Grundlagen 3 – #342215 – SS 2007 – Armin Biere – JKU Linz

Sprache eines EA

Definition Ein EA A akzeptiert ein Wort $w \in \Sigma^*$ gdw. es s_i und a_i gibt mit

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \dots \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s_n,$$

wobei $n \geq 0$, $s_0 \in I$, $s_n \in F$ und $w = a_1 \dots a_n$ ($n = 0 \Rightarrow w = \varepsilon$).**Definition** Die Sprache $L(A)$ von A ist die Menge der Wörter die er akzeptiert.

- benutze Reguläre Sprachen zur Spezifikation von Syntax
z.B. Scanner eines Parsers
- benutze EA oder reguläre Sprachen zur Beschreibung von Ereignisströmen!

Formale Grundlagen 3 – #342215 – SS 2007 – Armin Biere – JKU Linz

Endlicher Automat (EA)

Motivation: Automaten für die Modellierung, Spezifikation und Verifikation verwenden!

Definition Ein *Endlicher Automat* $A = (S, I, \Sigma, T, F)$ besteht aus

- Menge von Zuständen S (normalerweise endlich)
- Menge von Initialzuständen $I \subseteq S$
- Eingabe-Alphabet Σ (normalerweise endlich)
- Übergangsrelation $T \subseteq S \times \Sigma \times S$
schreibe $s \xrightarrow{a} s'$ gdw. $(s, a, s') \in T$ gdw. $T(s, a, s')$ "gilt"
- Menge von Finalzuständen $F \subseteq S$

Formale Grundlagen 3 – #342215 – SS 2007 – Armin Biere – JKU Linz

Produkt-Automat

Definition Der Produkt Automat $A = A_1 \times A_2$ von zwei EA A_1 und A_2 mit gemeinsamen Eingabealphabet $\Sigma_1 = \Sigma_2$ hat folgende Komponenten:

$$S = S_1 \times S_2 \quad I = I_1 \times I_2$$

$$\Sigma = \Sigma_1 = \Sigma_2 \quad F = F_1 \times F_2$$

$$T((s_1, s_2), a, (s'_1, s'_2)) \text{ gdw. } T_1(s_1, a, s'_1) \text{ und } T_2(s_2, a, s'_2)$$

Satz Seien A, A_1 , und A_2 wie oben, dann $L(A) = L(A_1) \cap L(A_2)$ **Beispiel:** Konstruktion eines Automaten, der Wörter mit Prefix ab und Suffix ba akzeptiert.
(als regulärer Ausdruck: $a \cdot b \cdot \mathbf{1}^* \cap \mathbf{1}^* \cdot b \cdot a$, wobei $\mathbf{1}$ für alle Buchstaben steht)

Formale Grundlagen 3 – #342215 – SS 2007 – Armin Biere – JKU Linz

Definition Zu $s \in S, a \in \Sigma$ bezeichne $s \xrightarrow{a}$ die Menge der Nachfolger von s definiert als

$$s \xrightarrow{a} = \{s' \in S \mid T(s, a, s')\}$$

Definition Ein EA ist *vollständig* gdw. $|I| > 0$ und $|s \xrightarrow{a}| > 0$ für alle $s \in S$ und $a \in \Sigma$.

Definition ... *deterministisch* gdw. $|I| \leq 1$ und $|s \xrightarrow{a}| \leq 1$ für alle $s \in S$ und $a \in \Sigma$.

Fakt ... deterministisch und vollständig gdw. $|I| = 1$ und $|s \xrightarrow{a}| = 1$ für alle $s \in S, a \in \Sigma$.

Komplement eines Automaten

Definition Der Komplementär-Automat $A = K(A_1)$ eines endlichen Automaten A_1 hat dieselben Komponenten wie A_1 , bis auf $F = S \setminus F_1$.

Satz Der Komplementär-Automat $A = K(A_1)$ eines deterministischen und vollständigen Automaten A_1 akzeptiert die komplementäre Sprache $L(A) = \overline{L(A_1)} = \Sigma^* \setminus L(A_1)$.

Beispiel: Spam-Filter basierend auf der Black-List “abb”, “abba”, und “abacus”!

(Regulärer Ausdruck: $\overline{\text{“abb”} \mid \text{“abba”} \mid \text{“abacus”}}$)

Definition Der Power-Automat $A = \mathbb{P}(A_1)$ eines EA A_1 hat folgende Komponenten

$$\begin{aligned} S &= \mathbb{P}(S_1) \quad (\mathbb{P} = \text{Potenzmenge}) & I &= \{I_1\} \\ \Sigma &= \Sigma_1 & F &= \{F' \subseteq S_1 \mid F' \cap F_1 \neq \emptyset\} \\ T(S', a, S'') & \text{ gdw. } S'' = \bigcup_{s \in S'} s \xrightarrow{a} \end{aligned}$$

Satz A, A_1 wie oben, dann $L(A) = L(A_1)$ und A ist deterministisch und vollständig.

Beispiel: Spam-Filter basierend auf der White-List “abb”, “abba”, und “abacus”!

(Regulärer Ausdruck: $\text{“abb”} \mid \text{“abba”} \mid \text{“abacus”}$)

Orakel-Konstruktion

Idee: ersetze Nicht-Determinismus durch Orakel

Definition Der Orakelisierte EA $A = \text{Orakel}(A_1)$ eines EA A_1 hat die Komponenten:

- $S = S_1$
- $I = I_1$
- $\Sigma = \Sigma_1 \times S_1$
- $T(s, (a, t), s') \text{ gdw. } s' = t \text{ und } T_1(s, a, t)$
- $F = F_1$

Fakt $\pi_1(L(Oracle(A))) = L(A_1)$ (π_1 Projektion auf erste Komponente)

Fakt $Oracle(A_1)$ ist deterministisch gdw. $|I_1| \leq 1$.

Fakt $Oracle(A_1)$ ist fast immer unvollständig (z.B. $T_1 \neq S_1 \times \Sigma_1 \times S_1$ und $|S_1| > 1$).

Hinweis Vollständigkeit lässt sich erreichen, wenn schon A_1 vollständig ist und man statt S_1 die Menge $\{0, \dots, n-1\}$ zu Σ_1 hinzufügt, wobei n die maximale Anzahl der Nachfolger darstellt: $n = \max_{s \in S, a \in \Sigma} |s \xrightarrow{a}|$.

$$T(s, (a, i), s') \text{ gdw. } s' = s_j, \quad s \xrightarrow{a} = \{s_0, \dots, s_{m-1}\}, \quad j \equiv i \pmod{m}$$

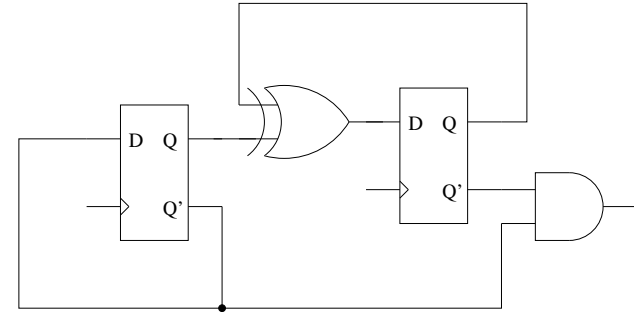
Übung Man führe beide Orakel-Konstruktionen für $a \cdot b \cdot \mathbf{1}^* \cap \mathbf{1}^* \cdot b \cdot a$ durch.

Ein/Ausgabe-Automat

Implementierungen von Automaten müssen deterministisch sein.

Definition Ein/Ausgabe Automat $A = (S, i, \Sigma, T, \Theta, O)$ besteht aus Folgendem:

- einer (endlichen) Menge von Zuständen S ,
- genau einem Initialzustand i ,
- einem Eingabealphabet Σ ,
- einer Übergangsfunktion $T \subseteq S \times \Sigma \rightarrow S$
- einem Ausgabealphabet Θ , mit
- Ausgabefunktion $O: S \times \Sigma \rightarrow \Theta$ (Moore-Maschine: $O: S \rightarrow \Theta$)



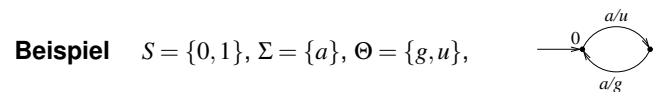
Verhalten eines Ein/Ausgabe-Automaten

Sei $w \in \Sigma^*$ und $a \in \Sigma$.

Definition Man interpretiere T als erweiterte Übergangsfunktion $T \subseteq S \times \Sigma^* \rightarrow S$ wie folgt: es gelte $s = T(s, \varepsilon)$ und $s' = T(s, a \cdot w) \Leftrightarrow \exists s'' [s'' = T(s, a) \wedge s' = T(s'', w)]$.

Definition Ebenso interpretiere man O als erweiterte Ausgabefunktion $O: S \times \Sigma^* \rightarrow \Theta^*$: es gelte $O(s, \varepsilon) = \varepsilon$ und $O(s, a \cdot w) = b \cdot w'$, mit $s' = T(s, a)$ und $w' = O(s', w)$.

Definition Das Verhalten $V: \Sigma^* \rightarrow \Theta^*$ eines Ein/Ausgabe-Automaten ist definiert durch $V(w) = O(i, w)$.



$$T(0, a^{2n}) = 0, \quad T(0, a^{2n+1}) = 1, \quad T(1, a^{2n}) = 1, \quad T(1, a^{2n+1}) = 0$$

$$V(a^{2n}) = (ug)^n, \quad V(a^{2n+1}) = (ug)^n u$$

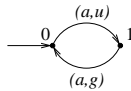
Gegeben ein Ein/Ausgabe-Automat $A = (S, i, \Sigma, T, \Theta, O)$.

Definition Der EA zu A sei definiert als $A' = (S, \{i\}, \Sigma \times \Theta, T', S)$ mit

$$T'(s, (a, b), s') \text{ gdw. } s' = T(s, a) \text{ und } b = O(s, a).$$

Fakt $V(w) = w'$ gdw. $(w, w') \in L(A')$

Fortsetzung des Beispiels:



(graphisch fast kein Unterschied)

- Modellierung *nebenläufiger* Systeme
 - Calculus of Communicating Systems (CCS) [Milner80]
 - Communicating Sequential Processes (CSP) [Hoare85]
 - genauer: **asynchron** kommunizierende Prozesse (Protokolle/Software)
- Synthese: Prozess Algebra (PA) als Programmiersprache (z.B. Occam, Lotos)
- Verifikation von (abstrakteren) PA Modellen ist einfacher
- **Theorie:** Mathematische Eigenschaften nebenläufiger Systeme
 - Wie kann man nebenläufige Systeme vergleichen?
 - Simulation, Bisimulation, Beobachtbarkeit, Divergenz (\Rightarrow Systemtheorie 1)

Gegeben ein EA $A = (S, I, \Sigma, T, F)$.

Definition Der Ein/Ausgabe-Automat zu A sei definiert als $A' = (\mathbb{P}(S), I, \Sigma, T', \{0, 1\}, O)$ mit T' die Übergangsrelation von $\mathbb{P}(A)$ und $O(S', a) = 1$ gdw. $S' \cap F \neq \emptyset$.

Fakt $w \in L(A)$ gdw. $V(w \cdot x) \in \mathbf{1}^{|w|} \cdot 1$ für ein $x \in \Sigma$

Fazit des Vergleiches von Ein/Ausgabe-Automat mit EA:

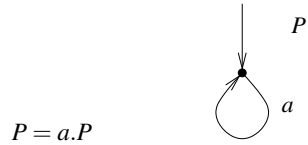
Im wesentlichen stellen beide die gleiche mathematische Struktur dar.

Wir konzentrieren uns auf die kompaktere und elegantere Version des EA.

Insbesondere Nicht-Determinismus lässt sich mit EA besser darstellen.

- Rechtslineare Grammatik = Reguläre Sprache = Chomsky 3 Sprache
 Grammatik G : $N = \varepsilon \mid aM \mid bM \quad M = cN \mid dN$ Startsymbol N
 \Rightarrow Sprache $L(G) = ((a \mid b)(c \mid d))^*$ (als regulärer Ausdruck)
- Syntax bei PA:
 - gleiche Idee: Gleichungen über Nichtterminalen = Prozesse
 - Konkatenation nicht durch Hintereinanderschreiben sondern mit '.' Operator
 - Auswahl dargestellt durch '+' Operator (nicht durch '|')
- Semantik
 - nur interessiert an den möglichen Sequenzen (= Ereignisströme)

Graphische Darstellung



$$R_1 \frac{}{a.P \xrightarrow{a} P}$$

Gleichung

Regel Operationale Semantik
(hier ist P eine Metavariablen)

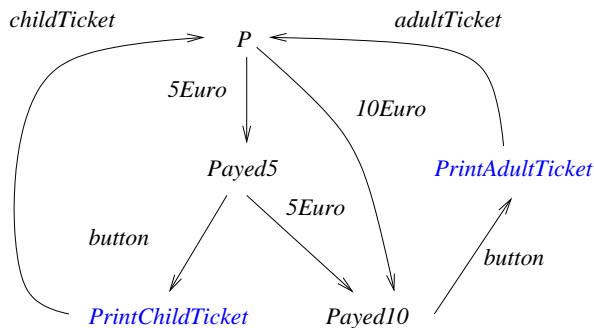
'.' Operator bedeutet sequentielle Konkatenation

Beispiel Eintrittskarten-Automat

$$P = 5Euro.Payed5 + 10Euro.Payed10$$

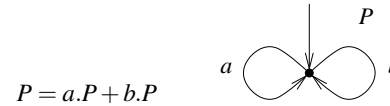
$$Payed5 = button.childTicket.P + 5Euro.Payed10$$

$$Payed10 = button.adultTicket.P$$



Graphische Darstellung

$$R_+^1 \frac{P \xrightarrow{a} P'}{(P+Q) \xrightarrow{a} P'}$$



$$R_+^2 \frac{Q \xrightarrow{a} Q'}{(P+Q) \xrightarrow{a} Q'}$$

Gleichung

Regeln Operationale Semantik
(hier sind P, Q Metavariablen)

'+' Operator bedeutet nicht-deterministische Auswahl

Labelled Transition Systems (LTS)

- LTS als **operationalen Semantik** von PAE
- fast wie Automaten, aber ...
 - keine Finalzustände bzw. im Prinzip sind alle Zustände Finalzustände
 - man ist nur an möglichen Ereignissequenzen interessiert
- LTS $A = (S, I, \Sigma, T)$ mit
 - Zustandsmenge S
 - Aktionen Σ
 - Übergangsrelation $T \subseteq S \times \Sigma \times S$ definiert durch operationale Regeln
 - Anfangszuständen $I \subseteq S$

- Divergente Selbstzyklen
 - $P = a.P + P$ ist **keine** gültige PAE
 - es gibt keine ϵ -Übergänge im Gegensatz zu EA (ϵ ist keine Aktion, da es ja “keine Zeit” braucht)
- Vermeidung von Selbstzyklen
 - Term T heißt **bewacht** bzw. **guarded** wenn T nur in der Form $a.T$ vorkommt (wobei a natürlich unterschiedlich für jedes Vorkommen von T sein kann)
 - einfachste Einschränkung:

Prozessvariablen auf rechter Seite (RHS) einer PAE sind bewacht
 - oder komplexer: jeder “Zyklus” beinhaltet mindestens eine Aktion

$$\begin{array}{l}
 \mathbf{R_{then}} \quad \frac{P \xrightarrow{a} P'}{\mathbf{if } B \mathbf{ then } P \mathbf{ else } Q \xrightarrow{a} P'} \quad B \\
 \\
 \mathbf{R_{else}} \quad \frac{Q \xrightarrow{a} Q'}{\mathbf{if } B \mathbf{ then } P \mathbf{ else } Q \xrightarrow{a} Q'} \quad \neg B
 \end{array}$$

(und ähnliche Regeln für **if-then** alleine)

$$\begin{array}{l}
 \text{Payed}(X) = \text{euro}(Y).\text{Payed}(X + Y) + \text{button}.\text{Print}(X) \\
 \text{Print}(X) = \mathbf{if } (X = 5) \mathbf{ then } \text{childTicket}.P + \mathbf{if } (X = 10) \mathbf{ then } \text{adultTicket}.P
 \end{array}$$

- Aktionen und Zustände können **parametrisiert** sein
 - somit auch parametrisierte Gleichungen
- voriges Beispiel im neuen Gewand ($x \in \{5, 10\}$):

$$\begin{array}{l}
 P = \text{euro}(x).\text{Payed}(x) \\
 \text{Payed}(5) = \text{button}.\text{print}(\text{childTicket}).P + \text{euro}(5).\text{Payed}(10) \\
 \text{Payed}(10) = \text{button}.\text{print}(\text{adultTicket}).P
 \end{array}$$

- möglicherweise zusätzliche Operationen auf den Daten erlaubt:

$$\text{Payed}(X) = \text{euro}(Y).\text{Payed}(X + Y) + \text{button}.\text{ticket}(X).P$$
 - damit insgesamt Beschreibung von *unendlichen Systemen* möglich
 - wird dadurch auch zur echten Programmiersprache

Synchronisation durch Rendezvous wie in CSP

$$\begin{array}{l}
 \Theta \subseteq \Sigma \\
 \mathbf{R}_{||\Theta} \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P ||_{\Theta} Q \xrightarrow{a} P' ||_{\Theta} Q'} \quad a \in \Theta \quad \text{Rendezvous} \\
 \mathbf{R}_{||\Theta}^1 \quad \frac{P \xrightarrow{a} P'}{P ||_{\Theta} Q \xrightarrow{a} P' ||_{\Theta} Q} \quad a \notin \Theta \quad \text{Interleaving} \\
 \mathbf{R}_{||\Theta}^2 \quad \frac{Q \xrightarrow{a} Q'}{P ||_{\Theta} Q \xrightarrow{a} P ||_{\Theta} Q'} \quad a \notin \Theta \quad \text{Interleaving}
 \end{array}$$

Beim Rendezvous ist Sender und Empfänger nicht genauer spezifiziert!

$$\mathbf{R}_{||} \quad \frac{P ||_{\Theta} Q \xrightarrow{a} P' ||_{\Theta} Q'}{P || Q \xrightarrow{a} P' || Q'} \quad \Theta = \Sigma(P) \cap \Sigma(Q)$$

$\Sigma(P)$ ist die Teilmenge der Aktionen von Σ die in P syntaktisch vorkommen

Fakt \parallel ist kommutativ: $P \parallel Q \xrightarrow{a} P' \parallel Q'$ gdw. $Q \parallel P \xrightarrow{a} Q' \parallel P'$

Beweis folgt unmittelbar aus den Regeln

Fakt \parallel ist assoziativ

Beweis: Sei $P = P_1 \parallel (P_2 \parallel P_3)$, $P' = P'_1 \parallel (P'_2 \parallel P'_3)$, $Q = (P_1 \parallel P_2) \parallel P_3$, $Q' = (P'_1 \parallel P'_2) \parallel P'_3$

Zu Zeigen: $P \xrightarrow{a} P' \Leftrightarrow Q \xrightarrow{a} Q'$

Genauer Beweis: 8 Fälle der Zugehörigkeit von $a \in \Sigma(P_i)$ für beide Richtungen.

Intuition:

- $a \in \Sigma(P_i) \Rightarrow P_i \xrightarrow{a} P'_i$
- P_i mit $a \notin \Sigma(P_i)$ ändern sich nicht ($P'_i = P_i$)
- dasselbe gilt für jede "parallele Zusammenschaltung" von P_i

- Verstecken bzw. Abstraktion von internen, **unbeobachtbaren** Aktionen

- Abstraktion zur "stillen" Aktion τ

– Annahme: $\tau \notin \Sigma$

- * formal betrachtet hat man nun Aktionen $\Sigma \cup \{\tau\}$
- * damit kann auch nie auf τ synchronisiert werden

– τ verbraucht trotzdem einen Zeitschritt

$$R \notin \Theta \quad \frac{P \xrightarrow{a} Q}{P \setminus \Theta \xrightarrow{a} Q \setminus \Theta} \quad a \notin \Theta \qquad R \in \Theta \quad \frac{P \xrightarrow{a} Q}{P \setminus \Theta \xrightarrow{\tau} Q \setminus \Theta} \quad a \in \Theta$$

- typische Verwendung für interne Synchronisationen $R = (\parallel_{i=1}^n Q_i) \setminus \{x_1, \dots, x_n\}$

- Klammerung bei \parallel kann weggelassen werden:

$$P \parallel (Q \parallel R) \text{ verhält sich wie } (P \parallel Q) \parallel R \text{ verhält sich wie } P \parallel Q \parallel R$$

- weiter kann Anordnung ignoriert werden

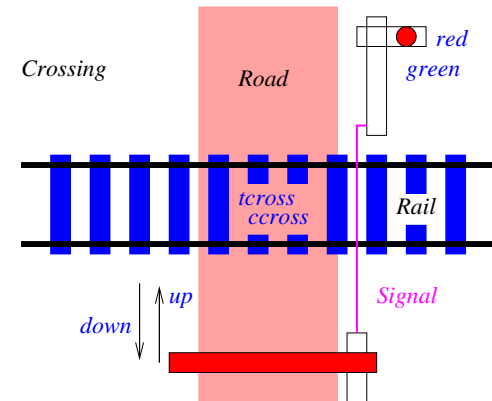
$$P \parallel Q \parallel R \text{ verhält sich wie } P \parallel R \parallel Q \text{ verhält sich wie } Q \parallel P \parallel R \text{ etc.}$$

- Parallel-Schaltung $\parallel_{i \in J} P_i$ bel. Prozesse P_i über Indexmenge J :

$$R \parallel \frac{\forall P_i, a \in \Sigma(P_i) \quad P_i \xrightarrow{a} P'_i \quad \forall P_i, a \notin \Sigma(P_i) \quad P'_i = P_i}{\parallel P_i \xrightarrow{a} \parallel P'_i} \quad \exists P_i \quad P_i \xrightarrow{a} P'_i$$

[BradfieldStirling]

$$\begin{aligned} \text{Road} &= \text{car.up.ccross.down.Road} \\ \text{Rail} &= \text{train.green.tcross.red.Rail} \\ \text{Signal} &= \text{green.red.Signal} + \text{up.down.Signal} \\ \text{Crossing} &= (\text{Road} \parallel \text{Rail} \parallel \text{Signal}) \setminus \{\text{green, red, up, down}\} \end{aligned}$$



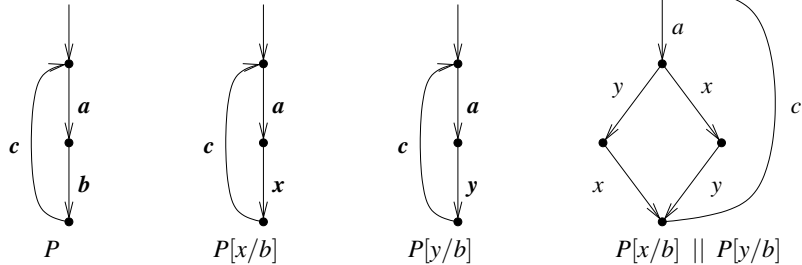
Linking als Substitution von Aktionen

$$R[\] \frac{P \xrightarrow{a} Q}{P[b/a] \xrightarrow{b} Q[b/a]}$$

Beispiel: $(a.P)[b/a] \xrightarrow{b} P[b/a]$

wird benötigt um Prozesse zusammenzubinden oder Templates zu instanziierten:

$$P = a.b.c.P \quad P[x/b] \parallel P[y/b]$$

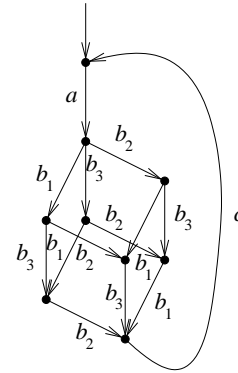


Fragestellung Milner's Scheduler

- klassisches Beispiel aus der Prozessalgebra
 - Modellierung eines Round-Robin-Schedulers **möglichst allgemein**
- Scheduling von n Prozessen $\parallel P_i$ mit $P = a.z.b.P$ und $P_i = P[a_i/a, z_i/z, b_i/b]$
 - a modelliert Starten eines Durchlaufes eines Prozesses
 - z modelliert interne Aktion bzw. interne Aktionen
 - b modelliert Ende des Durchlaufes eines Prozesses
- **Einschränkungen:**
 - Prozesse werden im Round-Robin-Stil gestartet in der Reihenfolge P_1, P_2, \dots
 - Prozess kann nicht zweitesmal gestartet werden ohne beendet worden zu sein
 - **es wird nichts über die Reihenfolge der b_i gesagt!**

$$P = a.b.c.P$$

$$\prod_{i=1}^3 P[b_i/b]$$



Falsche Lösung Milner's Scheduling

- Lösungsansatz: **Proxy für jeden zu kontrollierenden Prozess**
- Zerlegung des Schedulers R' in Token-Ring von n parallelen zyklischen Prozessen Q'
- jedes Q'_i kontrolliert Starten (a_i) und Beenden (b_i) von P_i, \dots
- ... übergibt Ausführungserlaubnis x_i an nächsten $Q'_{i+1} \dots$
- und wartet dann auf Ausführungserlaubnis x_{i-1} vom vorigen Q'_{i-1} im Ring

$$Q' = a.x.b.y.Q'$$

$$Q'_1 = Q'[a_1/a, x_1/x, b_1/b, x_n/y]$$

$$Q'_i = (y.Q'_i)[a_i/a, x_i/x, b_i/b, x_{i-1}/y] \quad i \in \{2, \dots, n\}$$

$$R' = \prod_{i=1}^n Q'_i$$

- falsche Lösung akzeptiert folgende legale Sequenz **nicht**:

– Beenden von P_2 vor P_1 :

$$a_1 a_2 b_2 b_1 \dots$$

- Entkopplung des Beendens (b) und der Berechtigungsannahme (y)

$$Q = a.x.(b.y + y.b).Q$$

$$Q_1 = Q[a_1/a, x_1/x, b_1/b, x_n/y]$$

$$Q_i = (y.Q)[a_i/a, x_i/x, b_i/b, x_{i-1}/y] \quad i \in \{2, \dots, n\}$$

$$R = \parallel_{i=1}^n Q_i$$

- Implementierung durch Warten auf zwei unterschiedliche Nachrichten

$$Road = car.up.ccross.down.Road$$

$$Rail = train.green.tcross.red.Rail$$

$$Signal = green.red.Signal + up.down.Signal$$

$$Crossing = (Road \parallel Rail \parallel Signal) \setminus \{green, red, up, down\}$$

bzw. in CCS

$$Road = car.up.\overline{ccross}.down.Road$$

$$Rail = train.green.\overline{tcross}.red.Rail$$

$$Signal = \overline{green}.red.Signal + \overline{up}.down.Signal$$

$$Crossing = (Road \parallel Rail \parallel Signal) \setminus \{green, red, up, down\}$$

- Aktionen: $\Sigma \dot{\cup} \bar{\Sigma} \dot{\cup} \{\tau\}$
 - gestrichene Aktionen Ausgaben, ungestrichene Eingaben

- anderes Hiding-Prinzip (Doppel-Schrägstrich zur syntaktischen Unterscheidung)

$$R_{\parallel} \frac{P \overset{a}{\rightarrow} Q}{P \parallel \Theta \overset{a}{\rightarrow} Q \parallel \Theta} \quad a \notin \Theta \cup \bar{\Theta}$$

- paarweise **explizite** Synchronisation

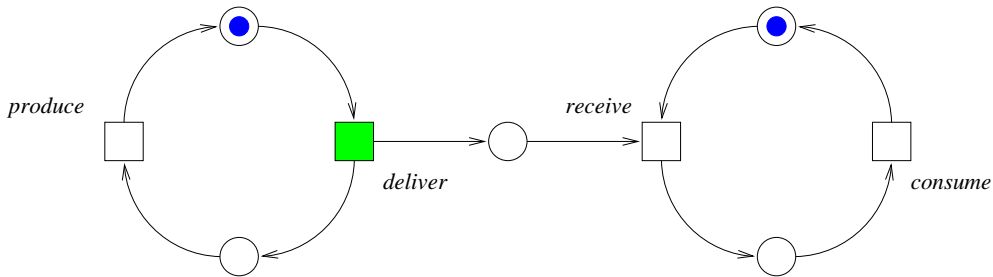
$$R_{\parallel} \frac{P \overset{a}{\rightarrow} P' \quad Q \overset{\bar{a}}{\rightarrow} Q'}{P \parallel Q \overset{\tau}{\rightarrow} P' \parallel Q'} \quad a \in \Sigma \dot{\cup} \bar{\Sigma}$$

$$R_{\parallel}^1 \frac{P \overset{a}{\rightarrow} P'}{P \parallel Q \overset{a}{\rightarrow} P' \parallel Q} \quad R_{\parallel}^2 \frac{Q \overset{a}{\rightarrow} Q'}{P \parallel Q \overset{a}{\rightarrow} P \parallel Q'}$$

- Originalversion Kanäle mit Daten bei CSP
 - Eingabe: $channel ? datain$, Ausgabe: $channel ! dataout$
- π -Kalkül nach [MilnerParrowWalker]
 - Kanäle/Verbindungen werden selbst Daten
 - Beispiel: $TimeAnnounce = ring(caller).\overline{caller}(CurrentTime).\overline{hangup}.TimeAnnounce$
- Probabilistisches Verhalten
 - Übergänge sind mit einer Übergangswahrscheinlichkeit versehen
- Prozess Algebra mit Zeit
 - Übergänge *brauchen* explizit angegebene Zeit

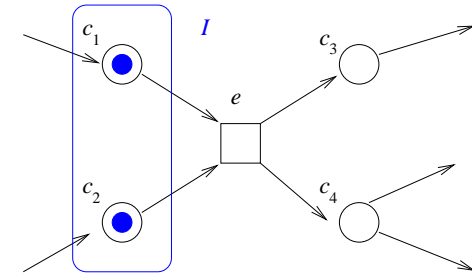
- Neben Prozess-Algebra geläufigste Modellierung *nebenläufiger* Systeme
 - seit 60er Jahren untersucht, in Form von **Activity Diagrams** in UML
 - auch wieder: **asynchron** kommunizierende Prozesse (Protokolle/Software)
- Modellierungs- und Verifikationstools existieren
- **Theorie:** Umfangreiche theoretische Untersuchungen
 - Endlichkeit, Deadlock, ...
- Aus der Praxis kommende Erweiterungen
 - Daten, Färbung, Hierarchie, quantitative Aspekte, ...

Producer Consumer CEN: Anfangsmarkierung



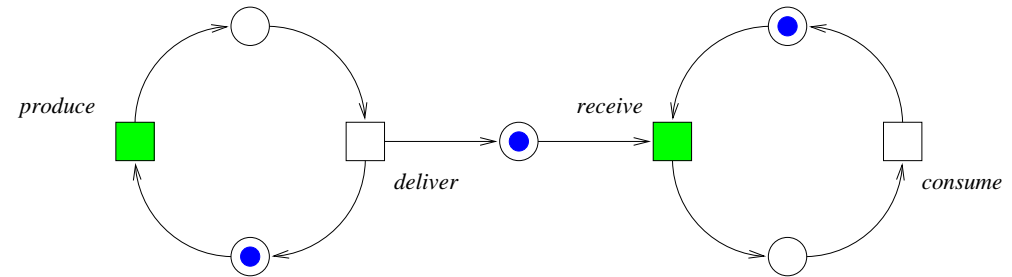
nur ein Event/Transition kann **feuern**

Definition Ein CEN $N = (C, I, E, G)$ besteht aus Bedingungen C , Anfangsmarkierung $I \subseteq C$, Events E und Abhängigkeiten $G \subseteq (C \times E) \dot{\cup} (E \times C)$

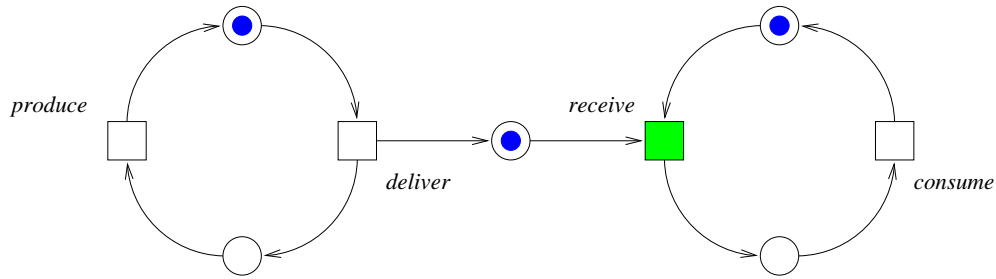


- oftmals \rightarrow statt G
- kann interpretiert werden als *bipartiter* Graph oder ...
- ... Hypergraph mit Multi-Quellen- bzw. Multi-Ziel-Kanten E

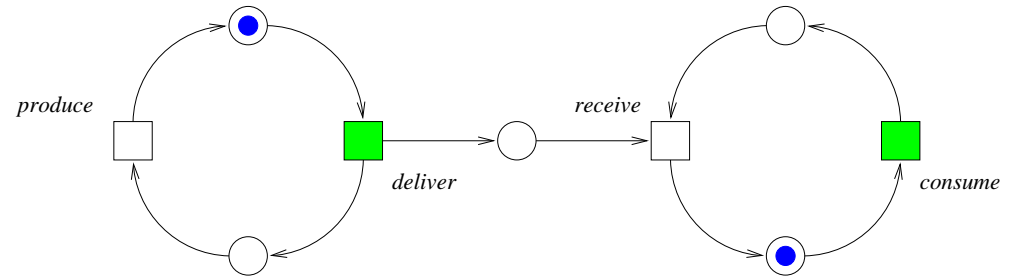
Producer Consumer CEN: Ausgeliefert



zwei Events/Transitionen können **feuern**



Ziel-Condition von *deliver* besetzt



wiederum Auswahl aus zwei möglichen **Events**

Semantik eines CEN als LTS

Definition Geg. CEN $N = (C, I, E, G)$. Das LTS $L = (S, \{I\}, \Sigma, T)$ zu N ist definiert durch

$$S = \mathbb{P}(C) \quad \Sigma = E$$

$$T(C_1, e, C_2) \text{ gdw. } G^{-1}(e) \subseteq C_1 \quad \text{Vorbildungen erfüllt} \quad (1)$$

$$G(e) \cap C_1 = \emptyset \quad \text{Nachbedingungen unerfüllt} \quad (2)$$

$$C_2 = (C_1 \setminus G^{-1}(e)) \cup G(e) \quad \text{Zustandsänderung}$$

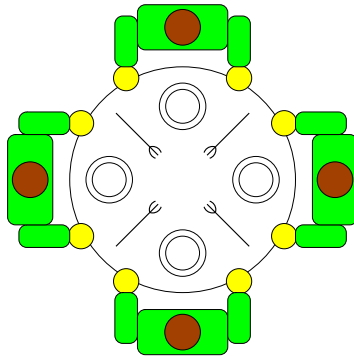
$$G(e) = \text{Nachbedingungen von Event } e \text{ (oder auch } e \rightarrow)$$

$$G^{-1}(e) = \text{Vorbedingungen von Event } e \text{ (oder auch } \rightarrow e)$$

Bemerkungen zur Semantik von CEN

- die Zustände $M \in \mathbb{P}(C)$ des LTS werden auch als **Markierungen** des CEN bezeichnet
- ein Event e ist **ausführbar** in M wenn $M \xrightarrow{e} \neq \emptyset$
- eine Markierung $M \in \mathbb{P}(C)$ ist ein **Deadlock**
 - gdw. M eine *Sackgasse* im LTS darstellt
 - gdw. kein Event in M ausführbar ist
 - gdw. alle Events *disabled* sind
 - gdw. $\forall e \in E [M \xrightarrow{e} = \emptyset]$
- ein CEN hat einen Deadlock gdw. ein Deadlock erreichbar ist

n Philosophen, n Gabeln, n Teller

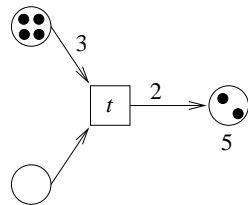


Philosophen Denken und Essen abwechselnd
Essen geht nur mit zwei Gabeln,
welche nicht gleichzeitig vom Tisch genommen werden können

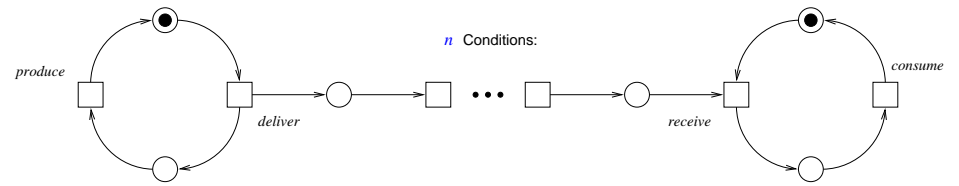
Formale Grundlagen 3 – #342215 – SS 2007 – Armin Biere – JKU Linz

Place Transition Net (PTN)

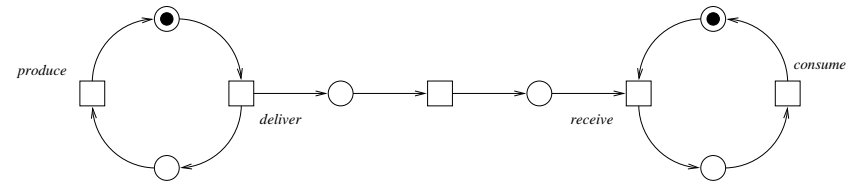
Definition Ein PTN $N = (P, I, T, G, C)$ besteht aus Stellen P , Anfangsmarkierung $I: P \rightarrow \mathbb{N}$, Transitionen T , Verbindungsgraph $G \subseteq (P \times T) \dot{\cup} (T \times P)$, und Kapazität $C: P \dot{\cup} G \rightarrow \mathbb{N}_{\infty}$.



- Kapazität der Kanten G ist endlich und sogar 1 falls nicht explizit angegeben
- Kapazität der Stellen P kann auch ∞ sein und ist es auch falls nicht explizit angegeben
- GEN lassen sich als PTN interpretieren mit konstanter Kapazität $C \equiv 1$



Puffer der Kapazität n

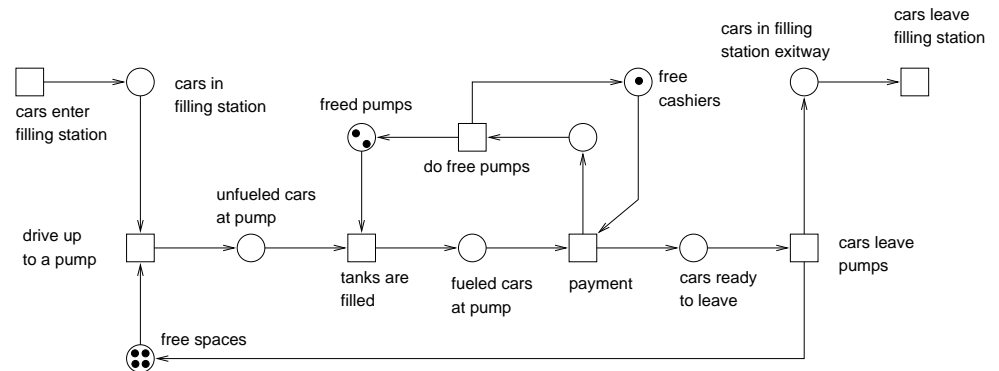


Puffer der Kapazität 2

Formale Grundlagen 3 – #342215 – SS 2007 – Armin Biere – JKU Linz

Filling Station

nach [W. Reisig, *A Primer in Petri Net Design*, 1992]



Geg. PTN $N = (P, I, T, G, C)$.

Definition Transition $t \in T$ kann in Zustand/Markierung $M: P \rightarrow \mathbb{N}$ **feuern** gdw.

$C((p, t)) \leq M(p)$ für alle $p \in G^{-1}(t)$ und

$C((t, q)) + M(q) \leq C(q)$ für alle $q \in G(t)$.

Definition Transition $t \in T$ **führt $M_1: P \rightarrow \mathbb{N}$ nach $M_2: P \rightarrow \mathbb{N}$ über** gdw.

t kann in M_1 feuern und $M_2 = M_1 - M_- + M_+$ mit

$$M_-(p) = \begin{cases} C((p, t)) & p \in G^{-1}(t) \\ 0 & \text{sonst} \end{cases} \quad M_+(p) = \begin{cases} C((t, p)) & p \in G(t) \\ 0 & \text{sonst} \end{cases}$$

Definition Das LTS $L = (S, \{I\}, \Sigma, T_L)$ zu N ist definiert durch

$$S = \mathbb{N}^P \quad \Sigma = T \quad \text{und} \quad T_L(M_1, t, M_2) \text{ gdw. } t \text{ führt } M_1 \text{ nach } M_2 \text{ über}$$

Formale Grundlagen 3 – #342215 – SS 2007 – Armin Biere – JKU Linz

Vereinfachte Hennessy-Milner Logik (HML)

zunächst betrachten wir HML als Bsp. für Temporale Logik für LTS

Geg. Alphabet von Aktionen Σ .

Definition **Syntax** besteht aus booleschen Konstanten $\{0, 1\}$, booleschen Operatoren $\{\wedge, \neg, \rightarrow, \dots\}$ und unären **modalen Operatoren** $[a]$ und $\langle a \rangle$ mit $a \in \Sigma$.

Lese $[a]f$ als für **alle** a -Nachfolger des momentanen Zustandes gilt f

Lese $\langle a \rangle f$ als für **einen** a -Nachfolger des momentanen Zustandes gilt f

Abkürzung $\langle \ominus \rangle f$ steht für $\bigvee_{a \in \ominus} \langle a \rangle f$ bzw. $[\ominus]f$ für $\bigwedge_{a \in \ominus} [a]f$

\ominus kann auch weiterhin als boolescher Ausdruck über Σ geschrieben werden

$$\text{z.B. } [a \vee b]f \equiv [\{a, b\}]f \quad \text{oder} \quad \langle \neg a \wedge \neg b \rangle f \equiv \langle \Sigma \setminus \{a, b\} \rangle f$$

Formale Grundlagen 3 – #342215 – SS 2007 – Armin Biere – JKU Linz

Gebrauch in der Informatik geht auf A. Pnueli zurück

- Häufig zur Spezifikation von Nebenläufigen und Reaktiven System gebraucht
- Erlaubt Verknüpfung von Aussagen zu verschiedenen Zeitpunkten
 - “Morgen ist das Wetter schön”
 - “Die Reaktorstäbe werden nie überhitzt”
 - “Die Zentralverriegelung öffnet sich unmittelbar nach einem Unfall”
 - “Der Airbag löst nur aus, wenn ein Unfall passiert ist”
 - “Einer Bestätigung (Ack) muss eine Anforderung (Req) vorausgehen”
 - “Wenn der Aufzug gerufen wird, dann kommt er auch irgendwann”
- Granularität der Zeitschritte muss natürlich festgelegt werden

Formale Grundlagen 3 – #342215 – SS 2007 – Armin Biere – JKU Linz

Beispiele zur Vereinfachten HML

- | | | |
|----|---|--|
| 1. | $[a]1$ | für alle a -Nachfolger gilt 1 (immer wahr) |
| 2. | $[a]0$ | für alle a -Nachfolger gilt 0
(a darf nicht möglich sein) |
| 3. | $\langle a \rangle 1$ | für einen a -Nachfolger gilt 1
(es muss a möglich sein) |
| 4. | $\langle a \rangle 0$ | für einen a -Nachfolger gilt 0 (immer falsch) |
| 5. | $\langle a \rangle 1 \wedge [b]0$ | es muss a aber es darf nicht b möglich sein |
| 6. | $\langle a \rangle 1 \wedge [\neg a]0$ | es muss und darf nur genau a möglich sein |
| 7. | $[a \vee b] \langle a \vee b \rangle 1$ | nach a oder b muss wiederum a oder b möglich sein |
| 8. | $\langle a \rangle [b] [b] 0$ | a ist möglich und danach aber b keinesfalls zweimal |
| 9. | $[a] (\langle a \rangle 1 \rightarrow [a] \langle a \rangle 1)$ | ist nach a wiederum möglich, dann auch ein zweitesmal |

Formale Grundlagen 3 – #342215 – SS 2007 – Armin Biere – JKU Linz

Geg. LTS $L = (S, I, \Sigma, T)$.

Definition Semantik ist rekursiv definiert als $s \models f$ (lese "f gilt in s"), mit $s \in S$ und f einer vereinfachte Formel in Hennessy-Milner Logik.

$$s \models 1$$

$$s \not\models 0$$

$$s \models [\Theta]g \text{ gdw. } \forall a \in \Theta \forall t \in S: \text{ wenn } s \xrightarrow{a} t \text{ dann } t \models g$$

$$s \models \langle \Theta \rangle g \text{ gdw. } \exists a \in \Theta \exists t \in S: s \xrightarrow{a} t \text{ und } t \models g$$

Definition es gilt $L \models f$ (lese "f gilt in L") gdw. $s \models f$ für alle $s \in I$

Definition Expansion von f ist die Menge der Zustände $[[f]]$ in denen f gilt.

$$[[f]] = \{s \in S \mid s \models f\}$$

Computation Tree Logic für HML (CTL/HML)

zunächst nur in Verbindung mit HML

Definition Syntax aufbauend auf der von HML und zusätzlich

unäre temporale Pfad-Operatoren **X**, **F**, **G** und ein **binärer** temporaler Pfad-Operator **U**.

Pfad-Operatoren müssen einen Pfad-Quantor **E** oder **A** als Präfix haben.

EX f	in einem unmittelbaren Folgezustand gilt f	$\equiv \langle \Sigma \rangle f$
AX f	in jedem Nachfolger muss f gelten	$\equiv [\Sigma] f$
EF f	in einer Zukunft gilt f irgendwann	<i>exists finally</i>
AF f	in allen möglichen Abläufen gilt f irgendwann	<i>always finally</i>
EG f	in einer Zukunft gilt f konstant	<i>exists globally</i>
AG f	es gilt f immer	<i>always globally</i>
E $[f \text{ U } g]$	möglicherweise gilt f solange bis schließlich g gilt (uns g muss für diesen Trace irgendwann gelten)	<i>exists until</i>
A $[f \text{ U } g]$	f gilt immer solange bis schließlich g gilt (uns g muss für jeden Trace irgendwann gelten)	<i>always until</i>

Geg. LTS $L = (S, I, \Sigma, T)$.

Definitionen

Ein **Trace** π von L ist eine endliche oder unendliche Folge von Zuständen

$$\pi = (s_0, s_1, \dots)$$

wobei es für jedes Paar (s_i, s_{i+1}) in π ein $a \in \Sigma$ gibt, mit $s_i \xrightarrow{a} s_{i+1}$. Also gibt es a_0, a_1, \dots mit

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \dots$$

$|\pi|$ ist dessen **Länge**, z.B. $|\pi| = 2$ für $\pi = (s_0, s_1, s_2)$, und $|\pi| = \infty$ für unendliche Traces.

$\pi(i)$ ist der i -te Zustand s_i von π falls $i \leq |\pi|$

$\pi^i = (s_i, s_{i+1}, \dots)$ ist das Suffix ab und inklusive dem i -ten Zustand s_i falls $i \leq |\pi|$

Bemerkung: $|\pi| = \infty$ dann $|\pi^i| = \infty$ für alle $i \in \mathbb{N}$

Beispiele zu CTL/HML

$$\neg \mathbf{EX}f \equiv \mathbf{AX}\neg f \quad \neg \langle \Theta \rangle f \equiv [\Theta]\neg f \quad \neg \mathbf{EF}f \equiv \mathbf{AG}\neg f \quad \neg \mathbf{EG}f \equiv \mathbf{AF}\neg f$$

(De Morgan für $\mathbf{E}[\cdot \text{ U } \cdot]$ benötigt weiteren temporalen Pfad-Operator)

AG $[\neg \text{safe}]0$	es ist immer unmöglich unsichere Aktionen auszuführen
EF $\langle \neg \text{safe} \rangle 1$	möglicherweise kann unsichere Aktionen ausgeführt werden
$\neg \mathbf{E}[\neg \langle \text{req} \rangle 1 \text{ U } \langle \text{ack} \rangle 1]$	es gibt keinen Ablauf auf dem irgendwann <i>ack</i> möglich wird und zuvor <i>req</i> nie möglich war
AG $[\text{req}]\mathbf{AF}[\neg \text{ack}]0$	immer nach einem <i>req</i> muss ein Punkt erreicht werden, ab dem keine Aktion ausser <i>ack</i> möglich ist

CTL/HML erlaubt die Kombination von Zustands- und Aktionsspezifikation

dies ist aber auch notwendig und leider oftmals unelegant

Geg. CTL/HML Formel f, g , ein LTS L . π sei immer ein Trace von L , und $i, j \in \mathbb{N}$.

Definition Semantik $s \models f$ (lese “ f gilt in s ”) ist rekursiv definiert

(nur noch für die neuen CTL Operatoren)

$$s \models \mathbf{EX}f \quad \text{gdw.} \quad \exists \pi[\pi(0) = s \wedge \pi(1) \models f]$$

$$s \models \mathbf{AX}f \quad \text{gdw.} \quad \forall \pi[\pi(0) = s \Rightarrow \pi(1) \models f]$$

$$s \models \mathbf{EF}f \quad \text{gdw.} \quad \exists \pi[\pi(0) = s \wedge \exists i[i \leq |\pi| \wedge \pi(i) \models f]]$$

$$s \models \mathbf{AF}f \quad \text{gdw.} \quad \forall \pi[\pi(0) = s \Rightarrow \exists i[i \leq |\pi| \wedge \pi(i) \models f]]$$

$$s \models \mathbf{EG}f \quad \text{gdw.} \quad \exists \pi[\pi(0) = s \wedge \forall i[i \leq |\pi| \Rightarrow \pi(i) \models f]]$$

$$s \models \mathbf{AG}f \quad \text{gdw.} \quad \forall \pi[\pi(0) = s \Rightarrow \forall i[i \leq |\pi| \Rightarrow \pi(i) \models f]]$$

$$s \models \mathbf{E}[f \mathbf{U} g] \quad \text{gdw.} \quad \exists \pi[\pi(0) = s \wedge \exists i[i \leq |\pi| \wedge \pi(i) \models g \wedge \forall j[j < i \Rightarrow \pi(j) \models f]]]$$

$$s \models \mathbf{A}[f \mathbf{U} g] \quad \text{gdw.} \quad \forall \pi[\pi(0) = s \Rightarrow \exists i[i \leq |\pi| \wedge \pi(i) \models g \wedge \forall j[j < i \Rightarrow \pi(j) \models f]]]$$

Kripke Strukturen

Geg. Menge von Atomaren Aussagen \mathcal{A} (boolesche Prädikate).

Definition eine Kripke Struktur $K = (S, I, T, \mathcal{L})$ besteht aus folgenden Komponenten:

- Zustandsmenge S .
- Anfangszuständen $I \subseteq S$ mit $I \neq \emptyset$
- einer *totalen* Übergangsrelation $T \subseteq S \times S$ (T total gdw. $\forall s[\exists t[T(s, t)]]$)
- Labelling/Markierung/Annotation $\mathcal{L}: S \rightarrow \mathbb{P}(\mathcal{A})$.

Labelling bildet Zustand s auf Menge atomarer Aussagen ab, die in s gelten:

$$\mathcal{L}(s) = \{\text{grau, warm, trocken}\}$$

Kripke Strukturen

- Klassisches Semantisches Modell für Temporale Logik
- reine Zustandssicht, keine Aktionen
 - im Prinzip LTS mit genau einer Aktion ($|\Sigma| = 1$)
 - zusätzlich Annotation von Zuständen mit atomaren Aussagen
- Ursprünge aus der modalen Logik:
 - verschiedene Welten aus S sind über \rightarrow bzw. T verbunden
 - $[]f$ gdw. für alle unmittelbar erreichbaren Welten gilt f
 - $\langle \rangle f$ gdw. es gibt eine unmittelbar erreichbare Welten, in der f gilt

LTS als Kripke Struktur

Definition Kripke Struktur $K = (S_K, I_K, T_K, \mathcal{L})$ zu einem vollständigen LTS $L = (S_L, I_L, \Sigma, T_L)$ ist definiert durch folgende Komponenten

$$\mathcal{A} = \Sigma \quad S_K = S_L \times \Sigma \quad I_K = I_L \times \Sigma \quad \mathcal{L}: (s, a) \mapsto a$$

$$T_K((s, a), (s', a')) \quad \text{gdw.} \quad T_L(s, a, s') \text{ und } a' \text{ beliebig}$$

Ähnliche Konstruktion wie beim Orakel-Automat!

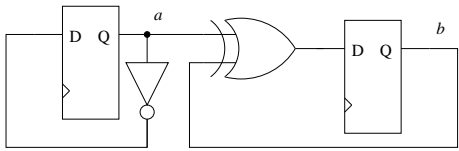
Fakt

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} s_n \quad \text{in } L$$

gdw.

$$(s_0, a_0) \rightarrow (s_1, a_1) \dots \rightarrow (s_n, a_n) \quad \text{in } K$$

Anmerkung oftmals $S \subseteq \mathbb{B}^n$, $\Sigma = \{a_1, \dots, a_n\}$, und $\mathcal{L}((s_1, \dots, s_n)) = \{a_i \mid s_i = 1\}$

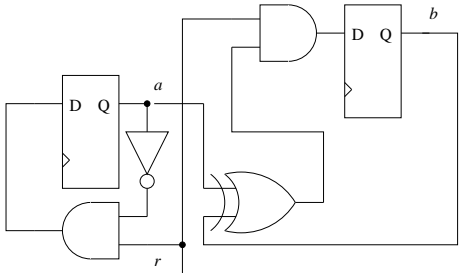


$$S = \mathbb{B}^2$$

$$I = \mathbb{B}^2$$

$$T = \{((0,0), (0,1)), ((0,1), (1,0)), \dots\}$$

$a \in L(s)$ gdw. $s \in \{(0,1), (1,1)\}$
 $b \in L(s)$ gdw. $s \in \{(1,0), (1,1)\}$



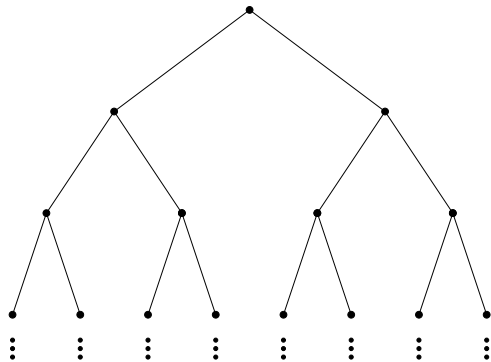
$$S = \mathbb{B}^3$$

$$I = \mathbb{B}^3$$

$$T = \dots$$

$a \in L(s)$ gdw. $s \in \{(-,-,1)\}$
 $b \in L(s)$ gdw. $s \in \{(-,1,-)\}$
 $r \in L(s)$ gdw. $s \in \{(1,-,-)\}$

Netzlisten, also Schaltkreise auf dieser Abstraktionsebene, haben keinen Initialzustand



Alle Mögliche Abläufe werden in einem (unendlichen) Baum dargestellt
 CTL betrachtet die Verzweigungsstruktur (Branching) des Computation Tree
 und hat eine lokale Zustandssicht

von jedem betrachteten Zustand verzweigen neue Pfade

klassische Version der CTL für Kripke Strukturen

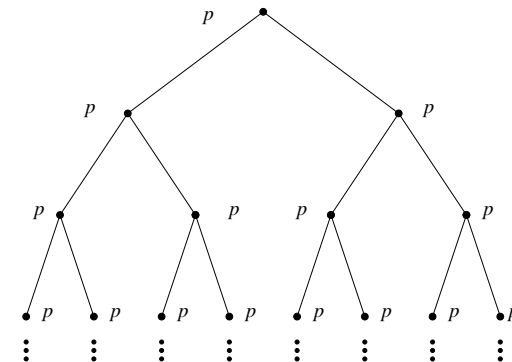
Definition Syntax von CTL enthält alle $p \in \mathcal{A}$, alle booleschen Operatoren $\wedge, \neg, \vee, \rightarrow, \dots$ und die temporalen Operatoren **EX, AX, EF, AF, EG, AG, E[· U ·]** und **A[· U ·]**.

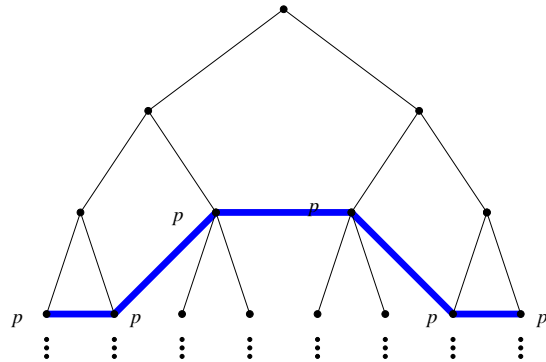
Definition CTL Semantik $s \models f$ (lese "f gilt in s") für Kripke Struktur $K = (S, I, T, \mathcal{L})$ ist genauso rekursiv definiert wie bei CTL/HML wobei zusätzlich $s \models p$ gdw. $p \in \mathcal{L}(s)$.

Beispiele zum 2-Bit Zähler mit Reset

- $AG(\bar{r} \rightarrow AX(\bar{a} \wedge \bar{b}))$
- $AG EX(\bar{a} \wedge \bar{b})$
- $AG EF(\bar{a} \wedge \bar{b})$
- $AG AF(\bar{a} \wedge \bar{b})$ unendlich oft $\bar{a} \wedge \bar{b}$
- $AG(\bar{a} \wedge \bar{b} \wedge r \rightarrow AX A[(a \vee b) U (\bar{a} \wedge \bar{b})])$
- $(AG r) \rightarrow AF(a \wedge b)$

Definition es gilt f in K schreibe $K \models f$ gdw. $s \models f$ für alle $s \in I$ (**generische Definition**)





Vergleich LTL und CTL

- LTL betrachtet jeweils genau einen möglichen linearen Ablauf.
- damit macht $(Gr) \rightarrow F(a \wedge b)$ plötzlich Sinn! (Erster Teil Annahme/Einschränkung)
- LTL ist kompositional (bez. synch. Produkt von Kripke-Strukturen):
 - $K_1 \models f_1, K_2 \models f_2 \Rightarrow K_1 \times K_2 \models f_1 \wedge f_2$
 - $K_1 \models f \rightarrow g, K_2 \models f \Rightarrow K_1 \times K_2 \models g$

Fakt CTL und LTL haben unterschiedliche Ausdrucksmächtigkeit:

z.B. lässt sich $AXEXp$ nicht in LTL ausdrücken, ebenso hat $AFAGp$ kein LTL Pendant

Definition Syntax von LTL ist genauso wie die von CTL, nur dass die temporalen Operatoren keine Pfadquantoren besitzen, also aus X, F, G und U bestehen.

Definition Semantik $\pi \models f$ von LTL Formel f ist auf unendlichen Pfaden π in K definiert:

- $\pi \models p$ gdw. $p \in \mathcal{L}(\pi(0))$
- $\pi \models \neg g$ gdw. $\pi \not\models g$
- $\pi \models g \wedge h$ gdw. $\pi \models g$ und $\pi \models h$
- $\pi \models Xg$ gdw. $\pi^1 \models g$
- $\pi \models Fg$ gdw. $\pi^i \models g$ für ein i
- $\pi \models Gg$ gdw. $\pi^i \models g$ für alle i
- $\pi \models g U h$ gdw. es gibt ein i mit $\pi^i \models h$ und $\pi^j \models g$ für alle $j < i$

Definition $K \models f$ gdw. $\pi \models f$ für alle unendlichen Pfade π in K mit $\pi(0) \in I$

ACTL Formeln als LTL Formeln

[Clarke and Draghicescu'88]

ACTL ist Teilmenge von CTL ohne E Pfadquantor und Negation nur vor $p \in \mathcal{A}$.

Definition zu einer ACTL Formel f definiere $f \setminus A$ als die LTL Formel, die aus f durch Wegstreichen aller A Pfadquantoren entsteht.

Definition f und g sind äquivalent gdw. $K \models f \Leftrightarrow K \models g$ für alle Kripke-Strukturen K . (f und g können aus unterschiedlichen Logiken stammen)

Satz falls ACTL Formel f zu LTL Formel g äquivalent, dann auch zu $f \setminus A$.

Beweis $K \models f \xrightarrow{\text{Annahme}} \forall \pi [\pi \models g] \xrightarrow[\text{+s.u.}]{\text{Annahme}} \forall \pi [\pi \models f] \xrightarrow{!} \forall \pi [\pi \models f \setminus A] \xrightarrow{\text{Def.}} K \models f \setminus A$

(π immer initialisiert und in $\pi \models f$ als Kripkestruktur interpretiert)

[M. Maidl'00]

Seien f und g bel. CTL bzw. LTL Formeln und $p \in \mathcal{A}$.

Definition Jede Unterformel einer CTL^{det} Formel hat eine der folgenden Formen:

$$p, f \wedge g, \mathbf{AX}f, \mathbf{AG}f, (\neg p \wedge f) \vee (p \wedge g) \text{ oder } \mathbf{A}[(\neg p \wedge f) \mathbf{U} (p \wedge g)]$$

Definition Jede Unterformel einer LTL^{det} Formel hat eine der folgenden Formen:

$$p, f \wedge g, \mathbf{X}f, \mathbf{G}f, (\neg p \wedge f) \vee (p \wedge g) \text{ oder } (\neg p \wedge f) \mathbf{U} (p \wedge g)$$

Satz Schnittmenge von LTL und ACTL besteht aus LTL^{det} bzw. CTL^{det}

Intuition CTL-Semantik bei CTL^{det} beschränkt sich auf Auswahl genau eines Pfades

Hinweis $\mathbf{A}[f \mathbf{U} p] \equiv \mathbf{A}[(\neg p \wedge f) \mathbf{U} (p \wedge 1)]$ $\mathbf{AF}p \equiv \mathbf{A}[1 \mathbf{U} p]$

⇒ eine nicht-deterministische Spezifikation birgt Gefahren der Falsch-Interpretation

- Spezifikation mach oft nur Sinn unter Fairness-Annahmen
 - z.B. Abstraktion des Schedulers: "jeder Prozess kommt dran"
 - z.B. eine Komponente muss unendlich oft am Zuge sein
 - z.B. der Übertragungskanal produziert unendlich oft keinen Fehler
- kein Problem in LTL: $(\mathbf{GF}f) \rightarrow \mathbf{G}(r \rightarrow \mathbf{F}a)$
- Faire Kripke-Strukturen für CTL:
 - zusätzliche Komponente F von fairen Zuständen
 - ein Pfad π ist **fair** gdw. $|\{i \mid \pi(i) \in F\}| = \infty$
 - betrachte nur noch faire Pfade

[P. Wolper'83]

Spezifikation "jeden m -ten Schritt gilt p " (zumindest)

Fakt für alle $m > 1$ gibt es weder eine CTL noch LTL Formel f , mit

$K \models f$ gdw. $\pi(i) \models p$ für alle initialisierten Pfade π von K und alle $i = 0 \pmod m$.

Problem $p \wedge \mathbf{G}(p \leftrightarrow \neg \mathbf{X}p)$ bedeutet "genau jeden 2. Schritt gilt p "

Lösungen

- modulo m Zähler ins Modell integrieren (Schwierigkeiten mit Kompositionalität)
- Erweiterung der Logik
 - ETL mit zusätzlichen Temporalen Operatoren definiert durch Automaten ...
 - ... bzw. Quantoren über atomaren Variablen (damit Zähler in der Logik)

– reguläre Ausdrücke: $\neg \left(\underbrace{(1; \dots; 1; p)^*}_{m-1}; \underbrace{1; \dots; 1; \neg p}_{m-1} \right)$ bzw. $\underbrace{(1; \dots; 1; p)^{\omega}}_{m-1}$

- spezielle Form von Quantoren über Mengen von Zuständen
 - quantifizierte Variablen $V = \{X, Y, \dots\}$
 - i.Allg. auch für Mengen und damit Logik zweiter Ordnung
- Fixpunkt-Logik: kleinste Fixpunkte spezifiziert durch μ und größte durch ν
- Modaler μ -Kalkül als Erweiterung von HML bzw. CTL

$$\nu X[p \wedge []X] \equiv \mathbf{AG}p \quad \mu X[q \vee (p \wedge \langle \rangle X)] \equiv \mathbf{E}[p \mathbf{U} q]$$

$$\nu X[p \wedge [][]X] \text{ entspricht "jeden 2. Schritt gilt } p"$$

$$\nu X[p \wedge \langle \rangle \mu Y[(f \wedge X) \vee (p \wedge \langle \rangle Y)]] \equiv \nu X[p \wedge \mathbf{EXE}[p \mathbf{U} f \wedge X]] \equiv \mathbf{EG}p \text{ unter Fairness } f$$

Auch wieder über Kripke Struktur $K = (S, I, T, \mathcal{L})$.

Definition eine Belegung ρ über den Variablen V ist eine Abb. $\rho: V \rightarrow \mathbb{P}(S)$

Definition Semantik $[[f]]_\rho$ einer μ -Kalkül Formel f ist rekursiv definiert als Expansion, also als Menge Zustände in denen f für eine geg. Belegung ρ gilt:

$$\begin{aligned} [[p]]_\rho &= \{s \mid p \in \mathcal{L}(s)\} & [[X]]_\rho &= \rho(X) \\ [[\neg f]]_\rho &= S \setminus [[f]]_\rho & [[f \wedge g]]_\rho &= [[f]]_\rho \cap [[g]]_\rho \\ \mu X[f] &= \bigcap \{A \subseteq S \mid [[f]]_{\rho[X \mapsto A]} = A\} & \nu X[f] &= \bigcup \{A \subseteq S \mid [[f]]_{\rho[X \mapsto A]} = A\} \end{aligned}$$

$$\text{mit } \rho[A \mapsto X](Y) = \begin{cases} A & X = Y \\ \rho(Y) & X \neq Y \end{cases}$$

Definition $K \models f$ gdw. $I \subseteq [[f]]_\rho$ für alle Belegungen ρ

Fakt μ -Kalkül subsumiert LTL und CTL.

Beispiel Hoare Logik: Summe der ersten n Zahlen

$\{n > 0\}$

$i := 0; s := 0$

$\{n > 0, i = 0, s = 0\}$

$\{i \leq n, s = \sum_{j=0}^i j\}$

while $i < n$ do

$\{i < n, i \leq n, s = \sum_{j=0}^i j\}$

$\{i < n, s = \sum_{j=0}^i j\}$

$i := i + 1;$

$\{i \leq n, s = \sum_{j=0}^{i-1} j\}$

$s := s + i;$

$\{i \leq n, s = i + \sum_{j=0}^{i-1} j\}$

$\{i \leq n, s = \sum_{j=0}^i j\}$

end;

$\{\neg i < n, i \leq n, s = \sum_{j=0}^i j\}$

$\{s = \sum_{j=0}^n j\}$

Schleifeninvariante p

Bedingung $B \equiv i < n$

Terminierungsfunktion $n - i$

Prinzip

Schrittweiser Nachweis von Eigenschaften eines Programmes mit Vor- und Nachbedingungen

Technik

Hoare-Kalkül = Transformationsregeln zwischen Vor- und Nachbedingung

Typische Regeln

$$\frac{\{p\} S_1 \{r\}, \quad \{r\} S_2 \{q\}}{\{p\} S_1; S_2 \{q\}} \quad \{p[v/e]\} v := e \{p\}$$

$$\{p \wedge B\} S \{p\}$$

mit Schleifeninvariante p

$$\{p\} \text{ while } B \text{ do } S \text{ end } \{p \wedge \neg B\}$$

Hoare Logik Einschränkungen

Halteproblem

Es gibt kein Programm, das die Terminierung von Programmen berechnen kann!

Schleifen lassen sich automatisch nur begrenzt aufröllen

- Schleifeninvarianten sind schwer zu finden (entspricht Induktionsinvarianten)
- Nach- bzw. Vorbedingungen von Schleifen lassen sich nicht automatisch berechnen
- Terminierungsfunktionen sind schwer zu finden
- Pointer und Heap-Speicher problematisch, z.B. Aliasing

Ähnliches gilt für Rekursion statt Schleifen.