Formale Grundlagen 3

#342215

SS 2007

Johannes Kepler Universität

Linz, Österreich

Univ. Prof. Dr. Armin Biere
Institut für Formale Modelle und Verifikation

http://fmv.jku.at/fg3

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 $F \subseteq S$

Sprache eines EA

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

$$s_0 \stackrel{a_1}{\rightarrow} s_1 \stackrel{a_2}{\rightarrow} s_2 \stackrel{a_3}{\rightarrow} \ldots \stackrel{a_{n-1}}{\rightarrow} s_{n-1} \stackrel{a_n}{\rightarrow} s_n,$$

wobei $n \geq 0$, $s_0 \in I$, $s_n \in F$ und $w = a_1 \cdots a_n \quad (n = 0 \Rightarrow w = \varepsilon)$.

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!

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:

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

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

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

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)

Vollständigkeit und Determinismus

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 $s \in S$, $a \in \Sigma$.

Teilmengenkonstruktion

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

$$S = \mathbb{P}(S_1) \quad (\mathbb{P} = \text{Potenzmenge}) \qquad I = \{I_1\}$$

$$\Sigma = \Sigma_1 \qquad F = \{F' \subseteq S_1 \mid F' \cap F_1 \neq \emptyset\}$$

$$T(S', a, S'') \quad \text{gdw.} \quad S'' = \bigcup_{s \in S'} s \xrightarrow{a}$$

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: “abb” | “abba” | “abacus”)

Komplement eines Automaten

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

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

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

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

Orakel-Konstruktion

Idee: ersetze Nicht-Determinismus durch Orakel

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

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

Zur Orakel-Konstruktion

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

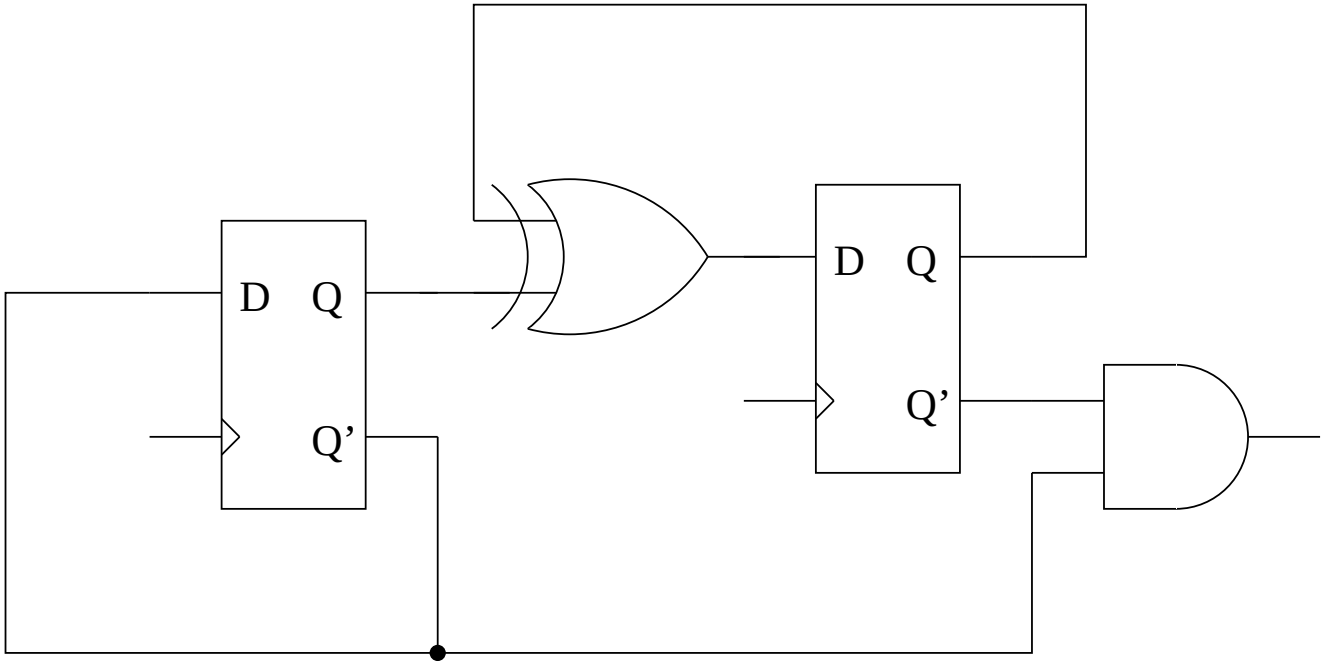
Fakt $Orakel(A_1)$ ist deterministisch gdw. $|I_1| \leq 1$.

Fakt $Orakel(A_1)$ 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 $\{0,\ldots,n-1\}$ zu Σ_1 hinzufügt, wobei n die maximale Anzahl der Nachfolger darstellt: $n = \max_{s\in S, a\in\Sigma}|s \xrightarrow{a}|$.

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

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


D
Q
Q'
D
Q
Q'

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 Übergangs**funktion** $T \subseteq S \times \Sigma \rightarrow S$
- einem Ausgabealphabet Θ, mit
- Ausgabefunktion $O: S \times \Sigma \rightarrow \Theta$ (Moore-Maschine: $O: S \rightarrow \Theta$)

Verhalten eines Ein/Ausgabe-Automaten

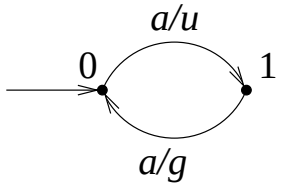
Sei $w \in \Sigma^*$ und $a \in \Sigma$.

Definition Man interpretiere T als *erweiterte* Übergangsfunktion $T \subseteq S \times \Sigma^* \to S$ 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 $O: S \times \Sigma^* \to \Theta^*$: 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 $V: \Sigma^* \to \Theta^*$ eines Ein/Ausgabe-Automaten ist definiert durch $V(w) = O(i, w)$.

Beispiel $S = \{0,1\}$, $\Sigma = \{a\}$, $\Theta = \{g,u\}$,



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

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

Ein/Ausgabe-Automat als EA

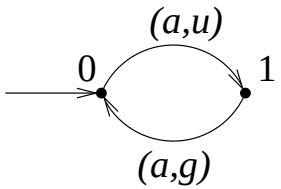
Gegeben ein Ein/Ausgabe-Automat $A = (S, i, \Sigma, T, \Theta, O)$.

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)

EA als Ein/Ausgabe-Automat

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.

Prozess Algebra (PA)

- 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)

PA Gleichungen (PAE)

- Rechtslineare Grammatik = Reguläre Sprache = Chomsky 3 Sprache

  Grammatik G: $N = \varepsilon \mid aM \mid bM$ $M = cN \mid dN$ Startsymbol N

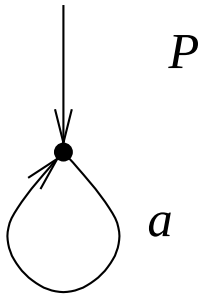
  $\Rightarrow$ Sprache $L(G) = ((a \mid b)(c \mid d))^*$ (als regulärer Ausdruck)

- Syntax bei PA:
  - gleiche Idee: Gleichungen über Nichterminalen = Prozesse
  - Konkatenation nicht durch Hintereinanderschreiben sondern mit '.' Operator
  - Auswahl dargestellt durch '$+$' Operator (nicht durch '$|$')

- Semantik
  - nur interessiert an den möglichen Sequenzen (= Ereignisströme)

Konkatenation

Graphische Darstellung



$P = a.P$

Gleichung

$R_.$ $\quad \dfrac{}{a.P \xrightarrow{a} P}$

Regel Operationale Semantik
(hier ist P eine Metavariable)

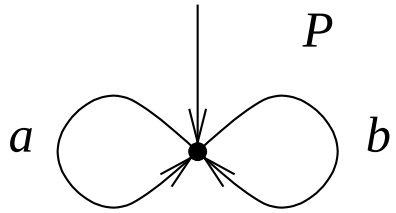
'.' Operator bedeutet sequentielle Konkatentation

Auswahl

Graphische Darstellung

$$R^1_+ \quad \frac{P \xrightarrow{a} P'}{(P+Q) \xrightarrow{a} P'}$$

$$P = a.P + b.P$$



$$R^2_+ \quad \frac{Q \xrightarrow{a} Q'}{(P+Q) \xrightarrow{a} Q'}$$

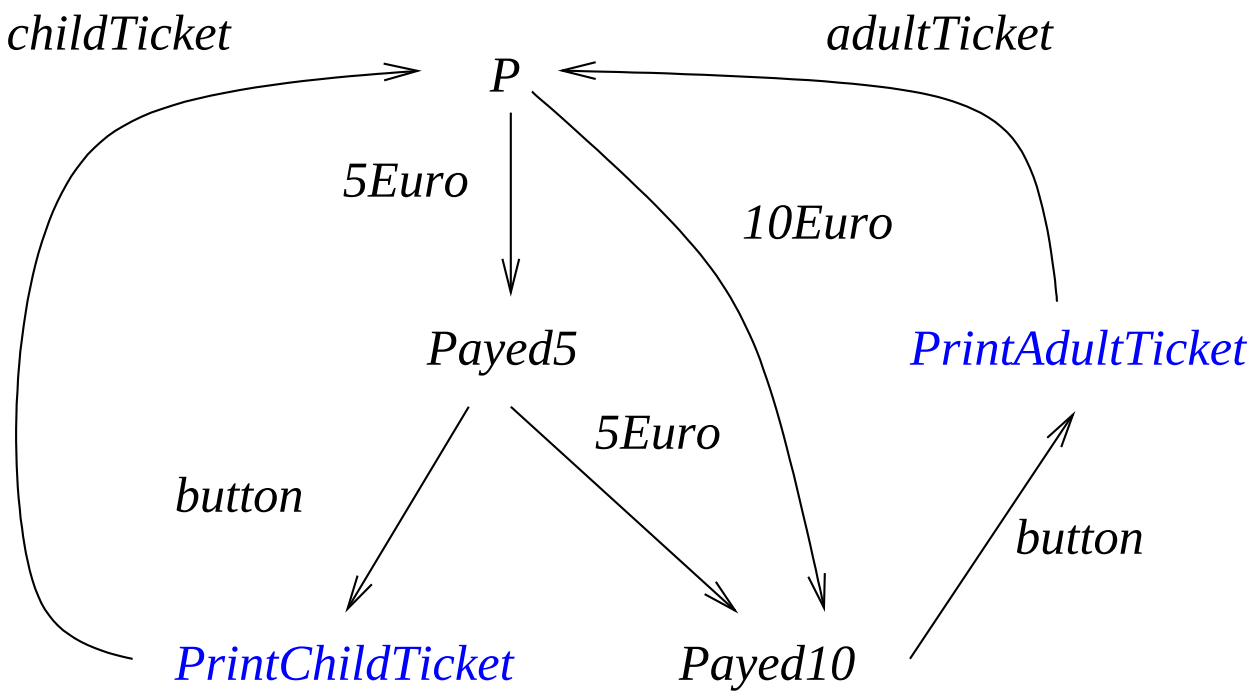
Gleichung

Regeln Operationale Semantik
(hier sind P, Q Metavariablen)

'$+$' Operator bedeutet nicht-deterministische Auswahl

Beispiel Eintrittskarten-Automat

$$
\begin{aligned}
P &= 5Euro.Payed5 + 10Euro.Payed10 \\
Payed5 &= button.childTicket.P + 5Euro.Payed10 \\
Payed10 &= button.adultTicket.P
\end{aligned}
$$



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 $I \subseteq S$

Syntaktische Einschränkungen

- 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

Daten in Prozess Algebra

- Aktionen und Zustände können **parametrisiert** sein
  - somit auch parametrisierte Gleichungen

- voriges Beispiel im neuen Gewand ($x \in \{5, 10\}$):

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

- möglicherweise zusätzliche Operationen auf den Daten erlaubt:

$$Payed(X) = euro(Y).Payed(X+Y) + button.ticket(X).P$$

  - damit insgesamt Beschreibung von *unendlichen Systemen* möglich
  - wird dadurch auch zur echten Programmiersprache

$$R_{\textbf{then}} \qquad \frac{P \xrightarrow{a} P'}{\textbf{if } B \textbf{ then } P \textbf{ else } Q \xrightarrow{a} P'} \quad B$$

$$R_{\textbf{else}} \qquad \frac{Q \xrightarrow{a} Q'}{\textbf{if } B \textbf{ then } P \textbf{ else } Q \xrightarrow{a} Q'} \quad \neg B$$

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

$$\begin{aligned} Payed(X) &= euro(Y).Payed(X+Y) + button.Print(X) \\ Print(X) &= \textbf{if } (X=5) \textbf{ then } childTicket.P + \textbf{if } (X=10) \textbf{ then } adultTicket.P \end{aligned}$$

Parallel-Operator

Synchronisation durch Rendezvous wie in CSP

$$\Theta \subseteq \Sigma$$

$$R_{||_\Theta} \quad \frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{a} Q'}{P \,||_\Theta\, Q \xrightarrow{a} P' \,||_\Theta\, Q'} \quad a \in \Theta \qquad \text{Rendezvous}$$

$$R^1_{||_\Theta} \quad \frac{P \xrightarrow{a} P'}{P \,||_\Theta\, Q \xrightarrow{a} P' \,||_\Theta\, Q} \quad a \notin \Theta \qquad \text{Interleaving}$$

$$R^2_{||_\Theta} \quad \frac{Q \xrightarrow{a} Q'}{P \,||_\Theta\, Q \xrightarrow{a} P \,||_\Theta\, Q'} \quad a \notin \Theta \qquad \text{Interleaving}$$

Beim Rendezvous ist Sender und Empfänger nicht genauer spezifiziert!

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

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

Eigenschaften des Parallel-Operators

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

Beweis folgt unmittelbar aus den Regeln

Fakt $||$ ist assoziativ

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

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

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

Intuition:

1. $a \in \Sigma(P_i) \Rightarrow P_i \xrightarrow{a} P'_i$
2. P_i mit $a \notin \Sigma(P_i)$ ändern sich nicht ($P'_i = P_i$)
3. dasselbe gilt für jede “parallele Zusammenschaltung” von P_i

Folgerungen der Eigenschaften des Parallel-Operators

- Klammerung bei $||$ kann weggelassen werden:

  $P \,||\, (Q \,||\, R)$ verhält sich wie $(P \,||\, Q) \,||\, R$ verhält sich wie $P \,||\, Q \,||\, R$

- weiter kann Anordnung ignoriert werden

  $P \,||\, Q \,||\, R$ verhält sich wie $P \,||\, R \,||\, Q$ verhält sich wie $Q \,||\, P \,||\, R$ etc.

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

$$R_{||} \quad \frac{\forall P_i, a \in \Sigma(P_i) \quad P_i \xrightarrow{a} P_i' \qquad\qquad \forall P_i, a \notin \Sigma(P_i) \quad P_i' = P_i}{||P_i \quad \xrightarrow{a} \quad ||P_i'} \qquad \exists P_i \quad P_i \xrightarrow{a} P_i'$$

Hiding

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

- Abstraktion zur “stillen” Aktion τ
  - Annahme: $\tau \notin \Sigma$
    - formal betracht hat man nun Aktionen $\Sigma \dot{\cup} \{\tau\}$
    - damit kann auch nie auf τ synchronisiert werden
  - τ verbraucht trotzdem einen Zeitschritt

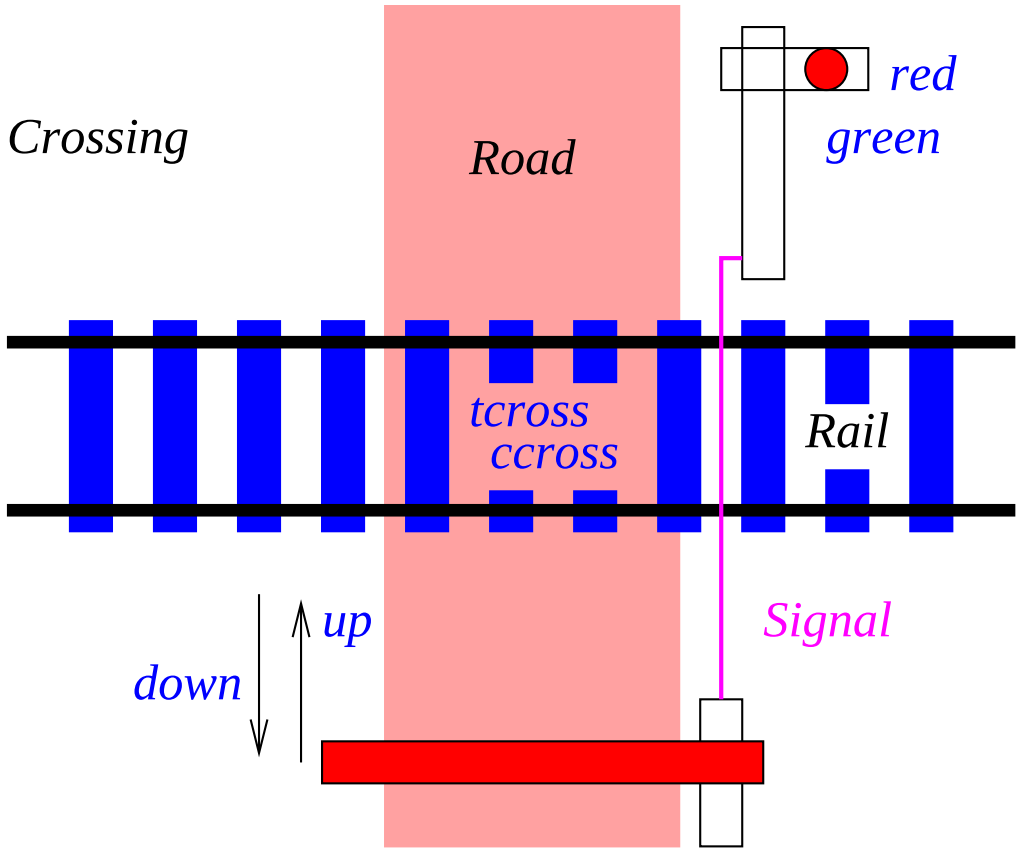
$$R_{\setminus}^{\notin} \quad \frac{P \xrightarrow{a} Q}{P\backslash\Theta \xrightarrow{a} Q\backslash\Theta} \quad a \notin \Theta \qquad\qquad R_{\setminus}^{\in} \quad \frac{P \xrightarrow{a} Q}{P\backslash\Theta \xrightarrow{\tau} Q\backslash\Theta} \quad a \in \Theta$$

- typische Verwendung für interne Synchronisationen $R = (||_{i=1}^{n} Q_i)\backslash\{x_1,\ldots,x_n\}$

Beispiel Bahnübergang

[BradfieldStirling]

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



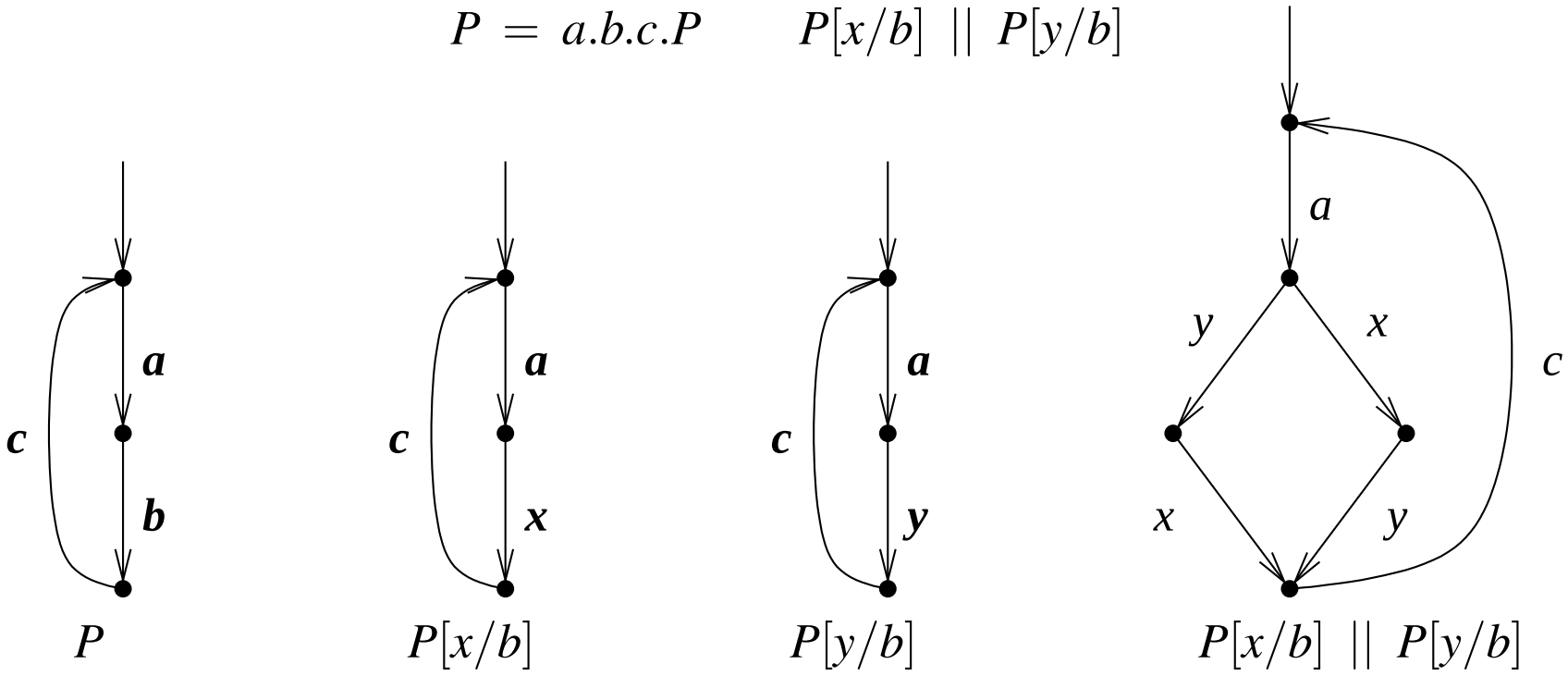
Linking als Substitution von Aktionen

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

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

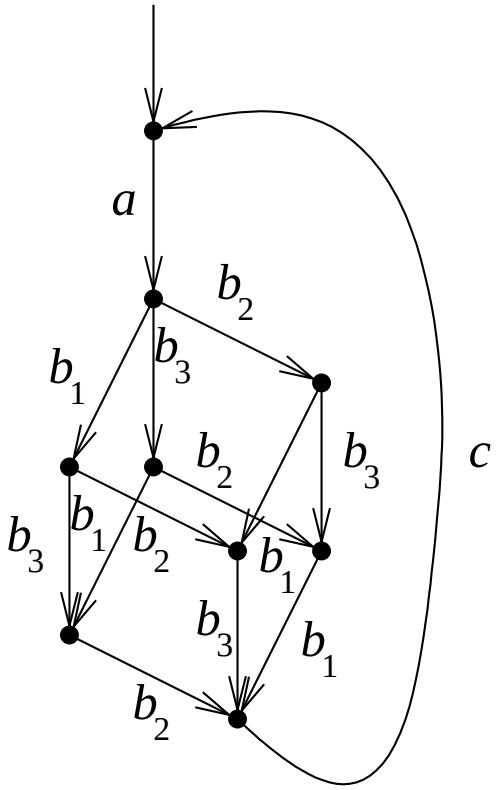
wird benötigt um Prozesse zusammenzubinden oder Templates zu instanziieren:

$$P = a.b.c.P \qquad P[x/b] \;||\; P[y/b]$$



Parametrisierung von Linking

$$P = a.b.c.P \qquad \Big|\Big|_{i=1}^{3} P[b_i/b]$$



Fragestellung Milner's Scheduler

- klassisches Beispiel aus der Prozessalgebra
  - Modellierung eines Round-Robin-Schedulers **möglichst allgemein**
- Scheduling von n Prozessen $||P_i$ 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 $P_1, P_2, \ldots$
  - Prozess kann nicht zweitesmal gestartet werden ohne beendet worden zu sein
  - es wird nichts über die Reihenfolge der b_i gesagt!

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, ...
- ... übergibt Ausführungserlaubnis x_i an nächsten Q'_{i+1} ...
- und wartet dann auf Ausführungserlaubnis x_{i-1} vom vorigen Q'_{i-1} im Ring

$$
\begin{aligned}
Q' &= a.x.b.y.Q' \\
Q'_1 &= Q'[a_1/a,\, x_1/x,\, b_1/b,\, x_n/y] \\
Q'_i &= (y.Q')[a_i/a,\, x_i/x,\, b_i/b,\, x_{i-1}/y] \qquad i \in \{2,\ldots,n\} \\
R' &= \mathop{||}_{i=1}^{n} Q'_i
\end{aligned}
$$

Richtige Lösung Milner's Scheduling

- falsche Lösung akzeptiert folgende legale Sequenz **nicht**:
  - Beenden von P_2 vor P_1:

$$a_1 a_2 b_2 b_1 \ldots$$

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

$$\begin{aligned} Q &= a.x.\,(b.y + y.b)\,.Q \\ Q_1 &= Q[a_1/a,\, x_1/x,\, b_1/b,\, x_n/y] \\ Q_i &= (y.Q)[a_i/a,\, x_i/x,\, b_i/b,\, x_{i-1}/y] \qquad i \in \{2,\ldots,n\} \\ R &= \prod_{i=1}^{n} Q_i \end{aligned}$$

- Implementierung durch Warten auf zwei unterschiedliche Nachrichten

Unterschiede in CCS

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

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

$$R_{\backslash\backslash} \quad \frac{P \xrightarrow{a} Q}{P \backslash\backslash \Theta \xrightarrow{a} Q \backslash\backslash \Theta} \quad a \notin \Theta \cup \overline{\Theta}$$

- paarweise **explizite** Synchronisation

$$R_{|||} \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P \,|||\, Q \xrightarrow{\tau} P' \,|||\, Q'} \quad a \in \Sigma \dot\cup \overline{\Sigma}$$

$$R^1_{|||} \quad \frac{P \xrightarrow{a} P'}{P \,|||\, Q \xrightarrow{a} P' \,|||\, Q} \qquad R^2_{|||} \quad \frac{Q \xrightarrow{a} Q'}{P \,|||\, Q \xrightarrow{a} P \,|||\, Q'}$$

Vergleich CSP und CCS am Beispiel Bahnübergang

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

bzw. in CCS

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

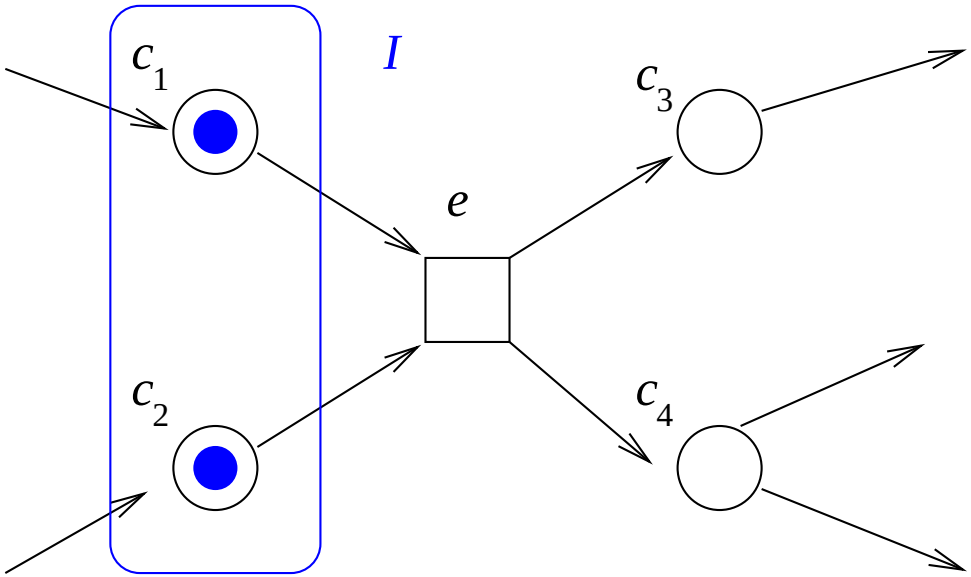
Weitere Varianten

- Orginalversion 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, ...

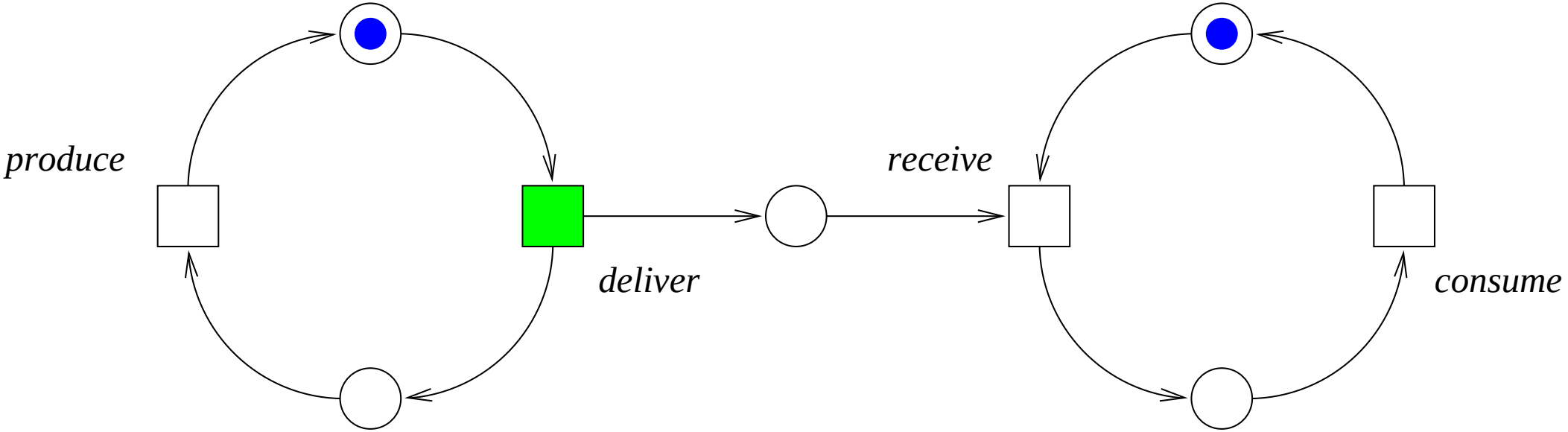
Condition-Event-Netze (CEN)

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



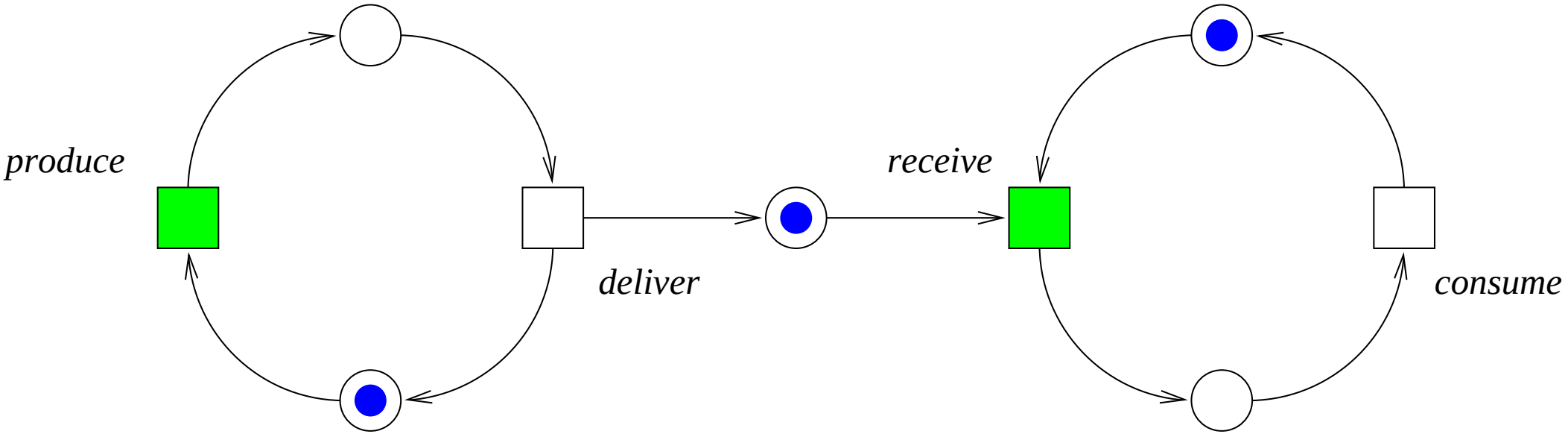
- oftmals $\rightarrow$ statt G
- kann interpretiert werden als *bipartiter* Graph oder ...
- ... Hypergraph mit Multi-Quellen- bzw. Multi-Ziel-Kanten E

Producer Consumer CEN: Anfangsmarkierung



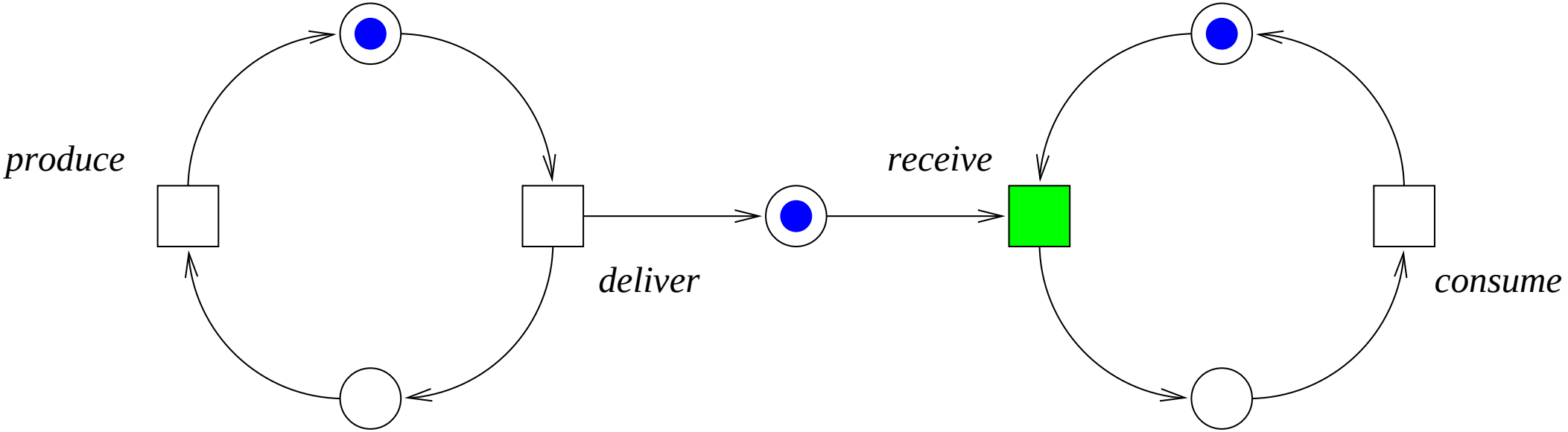
nur ein Event/Transition kann feuern

Producer Consumer CEN: Ausgeliefert



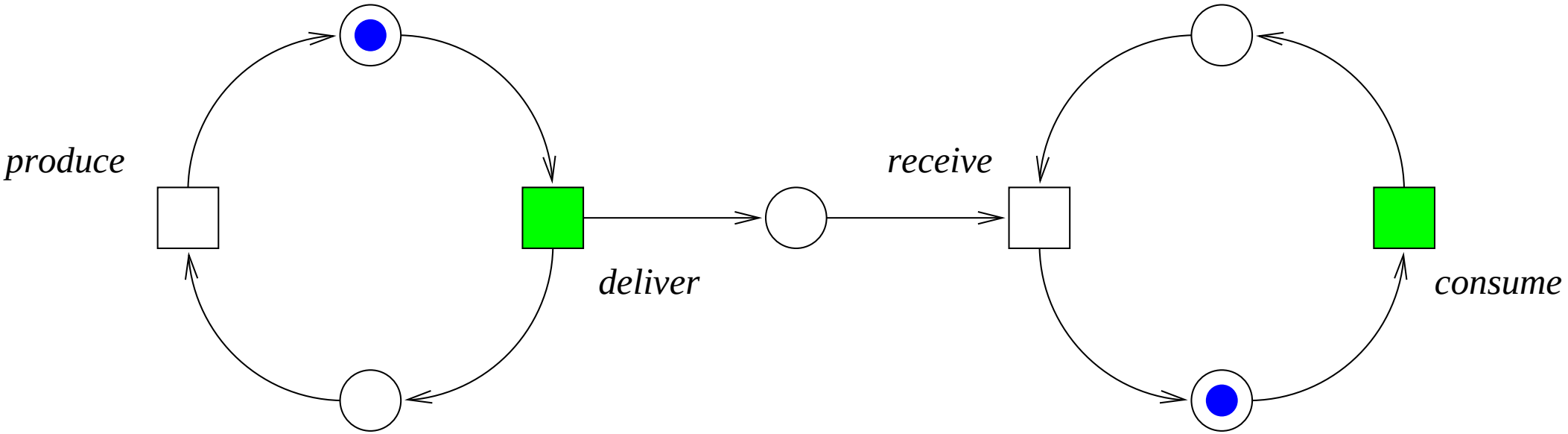
zwei Events/Transitionen können feuern

Producer Consumer CEN: Erneut Produziert



Ziel-Condition von *deliver* besetzt

Producer Consumer CEN: Konsumiert



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

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

$T(C_1, e, C_2)$ gdw.

$$G^{-1}(e) \subseteq C_1 \quad \text{Vorbedingungen erfüllt} \quad (1)$$

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

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

$G(e)$ = Nach bedingungen von Event e (oder auch $e \rightarrow$)

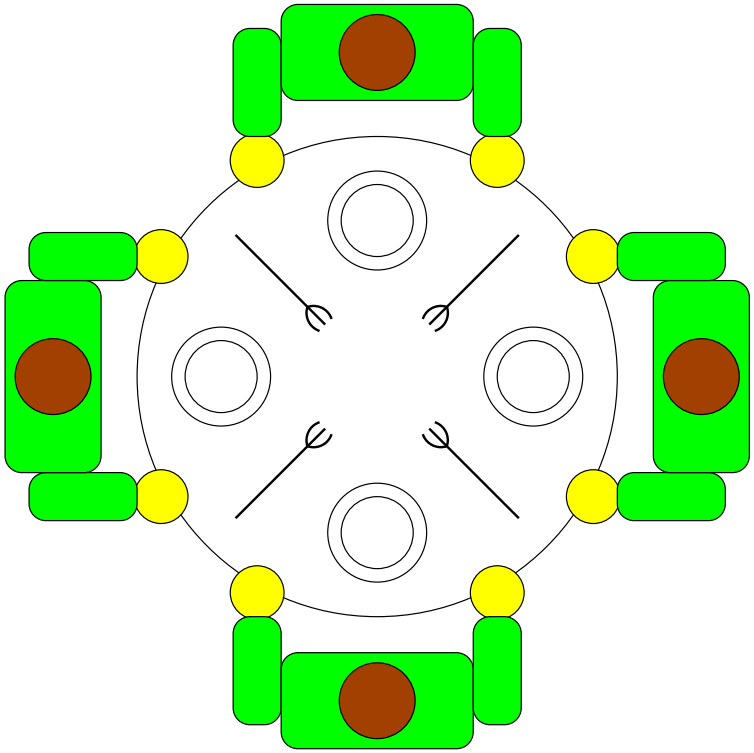
$G^{-1}(e)$ = Vor bedingungen von Event e (oder auch $\rightarrow e$)

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

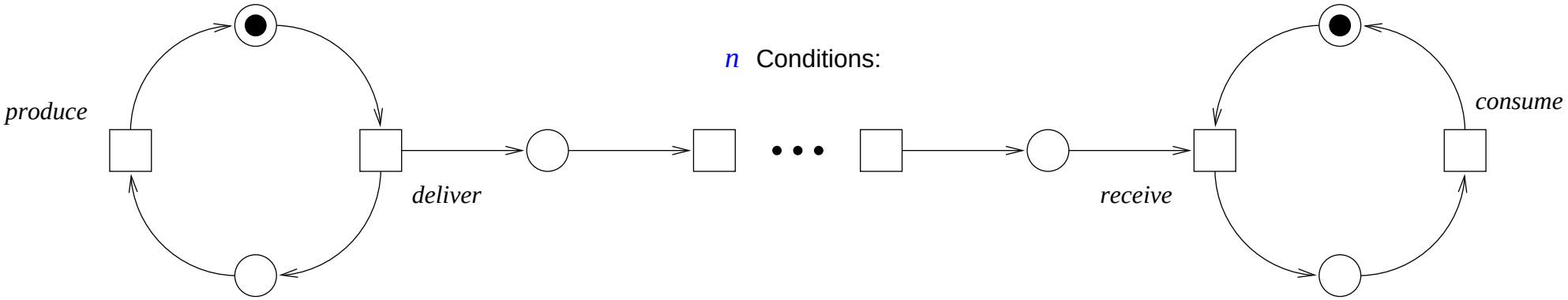
Beispiel Dining Philosophers

n Philosophen, n Gabeln, n Teller

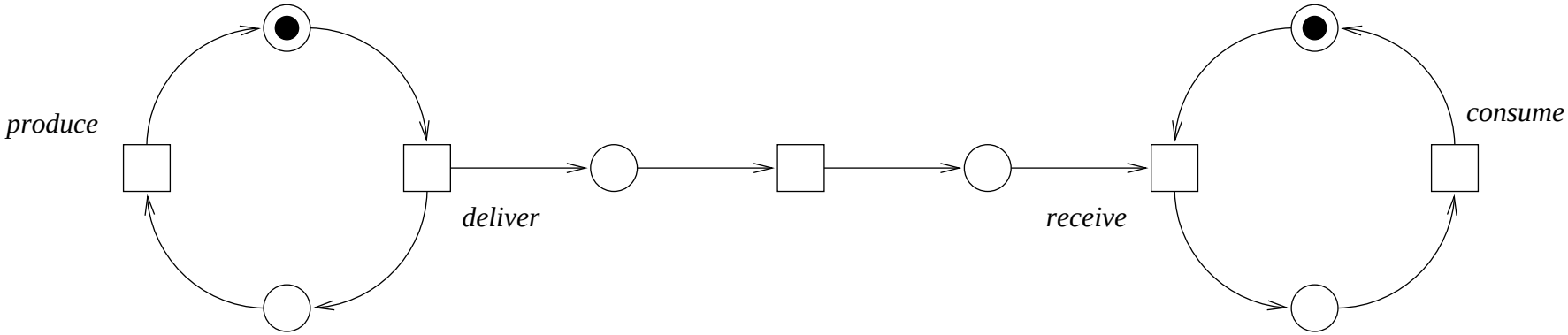


Philosophen Denken und Essen abwechselnd
Essen geht nur mit zwei Gabeln,
welche nicht gleichzeitig vom Tisch genommen werden können

Kapazitäten



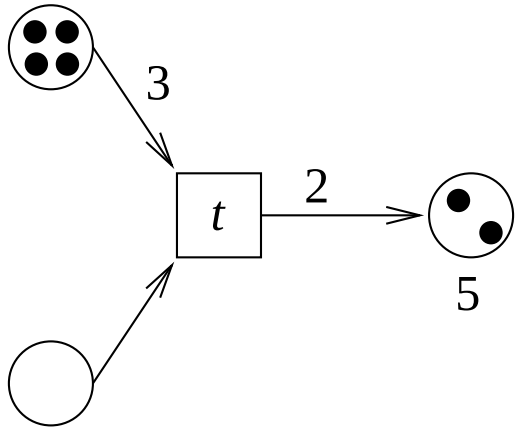
Puffer der Kapazität n



Puffer der Kapazität 2

Place Transition Net (PTN)

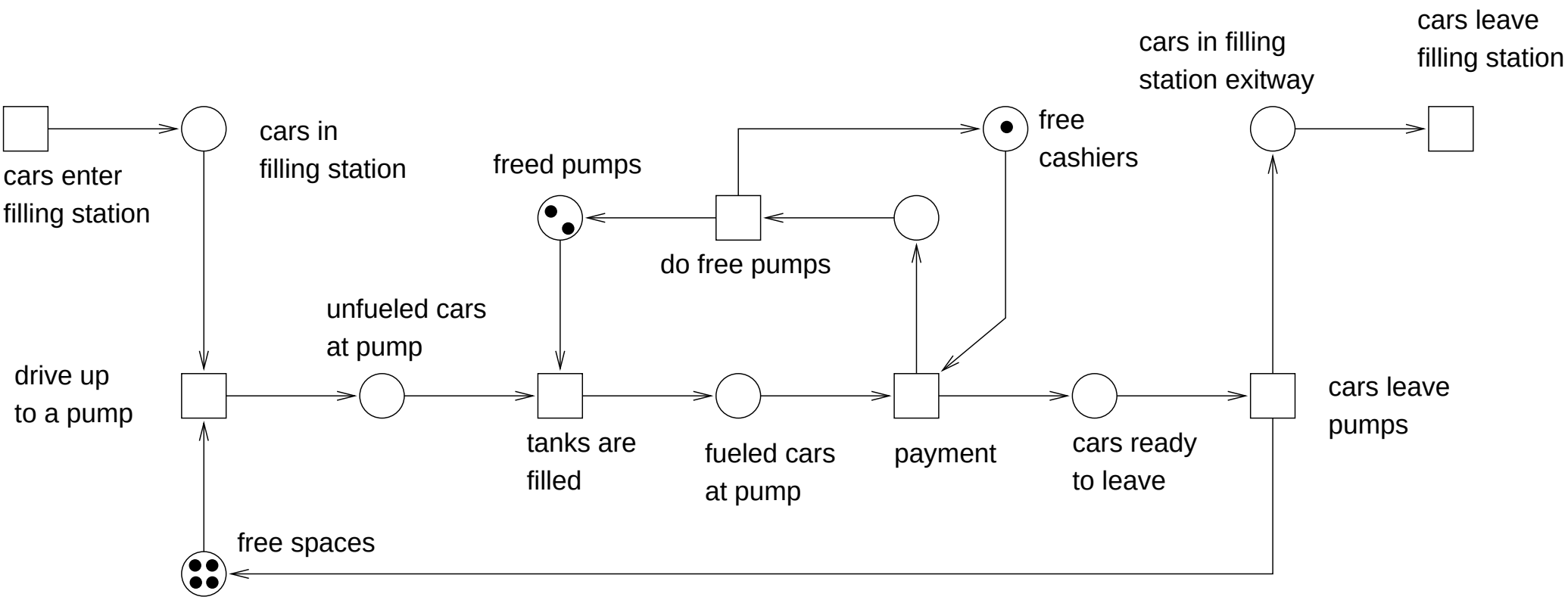
Definition Ein PTN $N = (P, I, T, G, C)$ besteht aus Stellen P, Anfangsmarkierung $I: P \to \mathbb{N}$, Transitionen T, Verbindungsgraph $G \subseteq (P \times T) \dot\cup (T \times P)$, und Kapazität $C: P \dot\cup G \to \mathbb{N}_\infty$.



- 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
- CEN lassen sich als PTN interpretieren mit konstanter Kapazität $C \equiv 1$

Filling Station

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



Semantik eines PTN als LTS

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

Definition Transition $t \in T$ kann in Zustand/Markierung $M : P \to \mathbb{N}$ 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 $M_1 : P \to \mathbb{N}$ nach $M_2 : P \to \mathbb{N}$ über gdw.

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

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

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

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

Temporale Logik

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

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 $\{\wedge,\neg,\rightarrow,\ldots\}$ 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 $\langle\Theta\rangle\,f$ steht für $\bigvee_{a\in\Theta}\langle a\rangle\,f$ bzw. $[\Theta]\,f$ für $\bigwedge_{a\in\Theta}[a]\,f$

Θ kann auch weiterhin als boolescher Ausdruck über Σ geschrieben werden

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

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 keinenfalls zweimal
9.	$[a](\langle a\rangle\,1 \rightarrow [a]\,\langle a\rangle\,1)$	ist nach a wiederum möglich, dann auch ein zweitesmal

Semantik der Vereinfachten Hennessy-Milner Logik

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.

$$s \models 1$$

$$s \not\models 0$$

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

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

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.

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

Endliche und Unendliche Traces

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

Definitionen

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

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

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 $a_0, a_1, \ldots$ mit

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

$|\pi|$ 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 $i \leq |\pi|$

$\pi^i = (s_i, s_{i+1}, \ldots)$ 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 $i \in \mathbb{N}$

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 $\mathbf{X}$, $\mathbf{F}$, $\mathbf{G}$ und ein binärer temporaler Pfad-Operator $\mathbf{U}$.

Pfad-Operatoren müssen einen Pfad-Quantor $\mathbf{E}$ oder $\mathbf{A}$ als Präfix haben.

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

Beispiele zu CTL/HML

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

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

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

CTL/HML erlaubt die Kombination von Zustands- und Aktionsspezifikation

dies ist aber auch notwendig und leider oftmals unelegant

Semantik der CTL/HML Operatoren

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

- 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

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:

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

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

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

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

Ähnliche Konstruktion wie beim Orakel-Automat!

Fakt

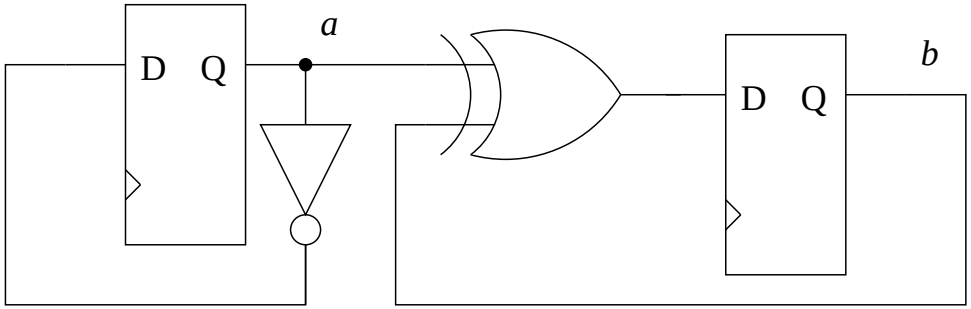
$$s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} \cdots \overset{a_{n-1}}{\to} s_n \quad \text{in } L$$

gdw.

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

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

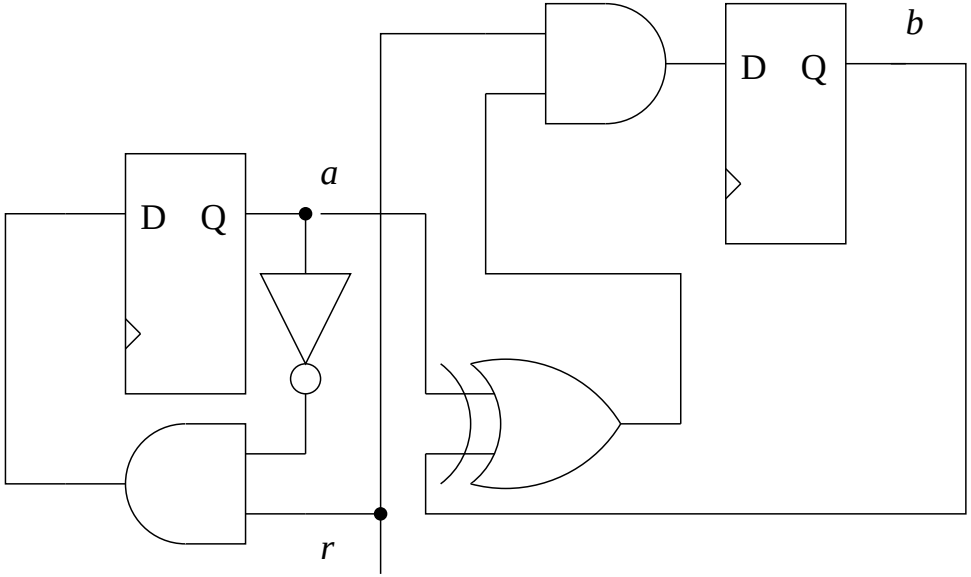
2-Bit Zähler als Kripke-Struktur



$$S = \mathbb{B}^2$$
$$I = \mathbb{B}^2$$
$$T = \{((0,0),(0,1)),$$
$$((0,1),(1,0)),\ldots\}$$

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

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



$$S = \mathbb{B}^3$$
$$I = \mathbb{B}^3$$
$$T = \ldots$$

$a \in L(s)$ 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

Computational Tree Logic (CTL)

klassische Version der CTL für Kripke Strukturen

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

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

$\mathbf{AG}(\overline{r} \rightarrow \mathbf{AX}(\overline{a} \wedge \overline{b}))$

$\mathbf{AG}\ \mathbf{EX}(\overline{a} \wedge \overline{b})$

$\mathbf{AG}\ \mathbf{EF}(\overline{a} \wedge \overline{b})$

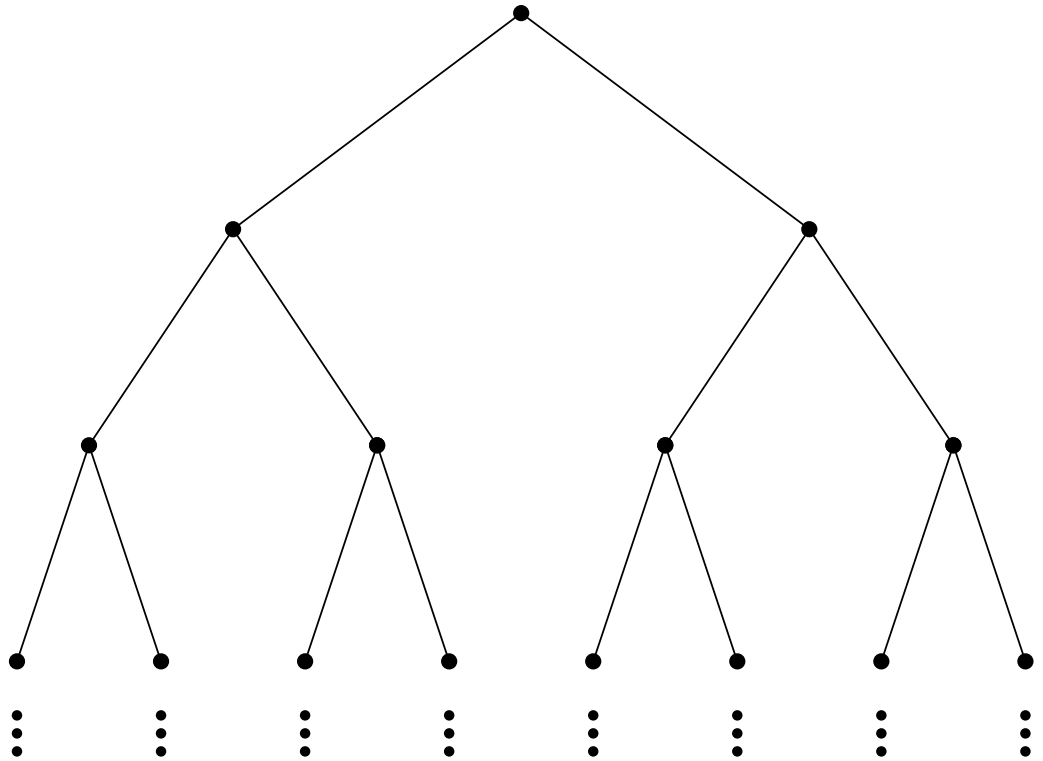
$\mathbf{AG}\ \mathbf{AF}(\overline{a} \wedge \overline{b})$ unendlich oft $\overline{a} \wedge \overline{b}$

$\mathbf{AG}(\overline{a} \wedge \overline{b} \wedge r \rightarrow \mathbf{AX}\ \mathbf{A}[(a \vee b)\ \mathbf{U}\ (\overline{a} \wedge \overline{b})])$

$(\mathbf{AG}\ r) \rightarrow \mathbf{AF}(a \wedge b)$

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

Computation Tree



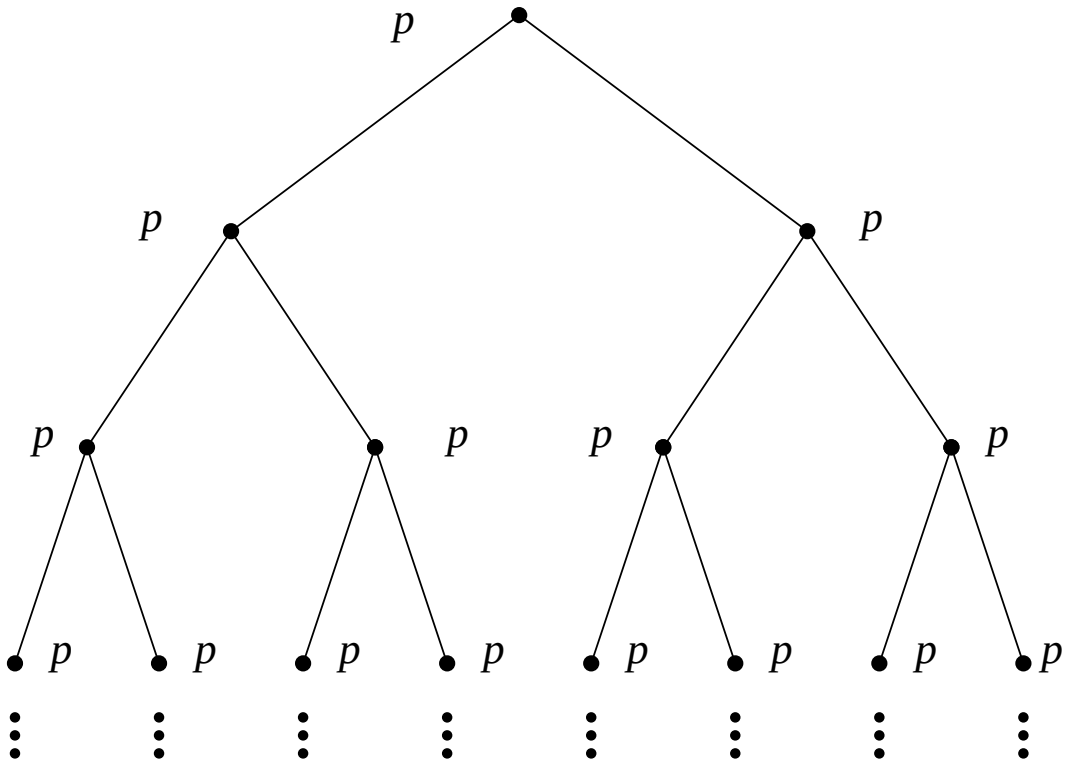
Alle Mögliche Abläufe werden in einem (unendlichen) Baum dargestellt

CTL betrachtet die Verzweigungsstruktur (Branching) des Computation Tree

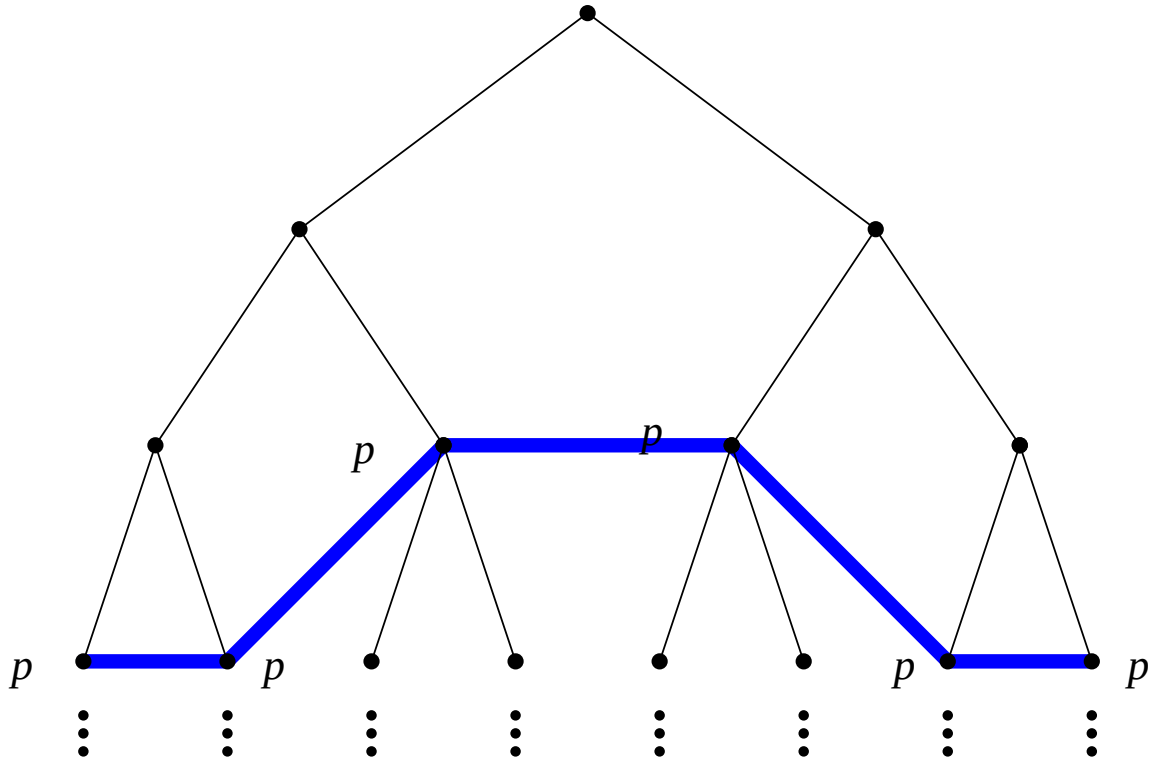
und hat eine lokale Zustandssicht

von jedem betrachteten Zustand verzweigen neue Pfade

Computation Tree $\mathbf{AG}p$



Computation Tree $\mathbf{AF}p$



Linear Temporal Logic (LTL)

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

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

$$
\begin{array}{lll}
\pi \models p & \text{gdw.} & p \in \mathcal{L}(\pi(0)) \\
\pi \models \neg g & \text{gdw.} & \pi \not\models g \\
\pi \models g \wedge h & \text{gdw.} & \pi \models g \text{ und } \pi \models h \\
\pi \models \mathbf{X} g & \text{gdw.} & \pi^1 \models g \\
\pi \models \mathbf{F} g & \text{gdw.} & \pi^i \models g \text{ für ein } i \\
\pi \models \mathbf{G} g & \text{gdw.} & \pi^i \models g \text{ für alle } i \\
\pi \models g \,\mathbf{U}\, h & \text{gdw.} & \text{es gibt ein } i \text{ mit } \pi^i \models h \text{ und } \pi^j \models g \text{ für alle } j < i
\end{array}
$$

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

Vergleich LTL und CTL

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

Fakt CTL und LTL haben unterschiedliche Ausdrucksmächtigkeit:

z.B. lässt sich $\mathbf{AXEX}p$ nicht in LTL ausdrücken, ebenso hat $\mathbf{AFAG}p$ kein LTL Pendant

ACTL Formeln als LTL Formeln

[Clarke and Draghicescu'88]

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

Definition zu einer ACTL Formel f definiere $f\backslash\mathbf{A}$ als die LTL Formel, die aus f durch Wegstreichen aller $\mathbf{A}$ 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 $f\backslash\mathbf{A}$.

Beweis $K \models f \overset{\text{Annahme}}{\Leftrightarrow} \forall\pi[\pi \models g] \underset{+\text{s.}u.}{\overset{\text{Annahme}}{\Leftrightarrow}} \forall\pi[\pi \models f] \overset{!}{\Leftrightarrow} \forall\pi[\pi \models f\backslash\mathbf{A}] \overset{\text{Def.}}{\Leftrightarrow} K \models f\backslash\mathbf{A}$

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

Syntaktisch Charakterisierte Teilmenge von LTL und CTL

[M. Maidl'00]

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

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

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

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

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

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

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

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

$\Rightarrow$ eine nicht-deterministische Spezifikation birgt Gefahren der Falsch-Interpretation

LTL und CTL Können nicht Zählen

[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 $i = 0 \bmod m$.

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

Fairness

- 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

- spezielle Form von Quantoren über Mengen von Zuständen
  - quantifizierte Variablen $V = \{X, Y, \ldots\}$
  - 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

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

$\nu X[p \wedge [\,][\,]X]$ entspricht "jeden 2. Schritt gilt p"

$\nu X[p \wedge \langle\,\rangle \mu Y[(f \wedge X) \vee (p \wedge \langle\,\rangle Y)]] \equiv \nu X[p \wedge \mathbf{EXE}[p\,\mathbf{U}\,f \wedge X]] \equiv \mathbf{EG}p$ 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. $\rho: V \to \mathbb{P}(S)$

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:

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

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.

Prinzip

Schrittweiser Nachweis von Eigenschaften eines Programmes
mit Vor- und Nachbedingungen

Technik

Hoare-Kalkül = Transformationsregeln zwischen Vor- und Nachbedingung

Typische Regeln

$$\frac{\{p\}\ S_1\ \{r\}, \quad \{r\}\ S_2\ \{q\}}{\{p\}\ S_1 ;\ S_2\ \{q\}}$$

$$\{p[v/e]\}\ v := e\ \{p\}$$

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

mit Schleifeninvariante p

Beispiel Hoare Logik: Summe der ersten n Zahlen

$\{n > 0\}$

```
i := 0; s := 0
```

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

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

```
while i < n do
```

Bedingung $B \equiv i < n$

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

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

```
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;
```

Terminierungsfunktion $n - i$

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

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

Halteproblem

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

Schleifen lassen sich automatisch nur begrenzt aufrollen

- 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.