Formal Models

#342.251

SS 2016

Johannes Kepler University

Linz, Austria

Prof. Armin Biere
Institute for Formal Models and Verification

http://fmv.jku.at/fm

Version 2016.1

Finite Automaton (FA)

use automata for modeling, specification and verification

Definition a *finite automaton* $A = (S, I, \Sigma, T, F)$ consists of the following components

- set of states S (usually finite)
- set of initial states $I \subseteq S$
- input-alphabet Σ (usually finite as well)
- transition relation $T \subseteq S \times \Sigma \times S$
  written $s \overset{a}{\rightarrow} s'$ iff $(s, a, s') \in T$ iff $T(s, a, s')$ “holds”
- set of final states $F \subseteq S$

Language of an FA

Definition FA A *accepts* a word $w \in \Sigma^*$ iff there exists s_i and a_i with

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

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

Definition the *language* $L(A)$ of A is the set of words accepted by it

- use regular languages for syntax specification (e.g. in a scanner / parser)
- use FA or regular languages to specify event streams

Product Automaton

Definition the product automaton $A = A_1 \times A_2$ of two FA A_1 and A_2 over the same alphabet $\Sigma_1 = \Sigma_2$ has the following components:

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

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

$$T((s_1, s_2), a, (s_1', s_2')) \quad \text{iff} \quad T_1(s_1, a, s_1') \text{ and } T_2(s_2, a, s_2')$$

Theorem let A, A_1, and A_2 as above, then $L(A) = L(A_1) \cap L(A_2)$

Example construct automaton, which accepts words with prefix ab and suffix ba.

(as regular expression: $a \cdot b \cdot \mathbf{1}^* \,\cap\, \mathbf{1}^* \cdot b \cdot a$, where $\mathbf{1}$ denotes all letters)

Completeness and Determinism

Definition for $s \in S$, $a \in \Sigma$ let $s \overset{a}{\to}$ denote the set of successors of s defined as

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

Definition an FA is *complete* iff $|I| > 0$ and $|s \overset{a}{\to}| > 0$ for all $s \in S$ and $a \in \Sigma$.

Definition … *deterministic* iff $|I| \leq 1$ and $|s \overset{a}{\to}| \leq 1$ for all $s \in S$ and $a \in \Sigma$.

Proposition … deterministic and complete iff $|I| = 1$ and $|s \overset{a}{\to}| = 1$ for all $s \in S$, $a \in \Sigma$.

Sub-Set Construction

Definition the *power-automaton* $A = \mathbb{P}(A_1)$ of an FA A_1 consists of the components:

$S = \mathbb{P}(S_1)$ ($\mathbb{P}$ = power set)

$I = \{I_1\}$

$\Sigma = \Sigma_1$

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

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

Theorem let A, A_1 as above, then $L(A) = L(A_1)$ and A is deterministic and complete.

Example: spam-filter based on the white-list “abb”, “abba”, and “abacus”!

(regular expression: “abb” | “abba” | “abacus”)

Complement-Automaton

Definition the *complement-automaton* $A = C(A_1)$ of an FA A_1 has the same components as A_1, except for the set of final states, which is $F = S \backslash F_1$.

Theorem the complement-automaton $A = C(A_1)$ of a deterministic and complete FA A_1 accepts the complement language $L(A) = \overline{L(A_1)} = \Sigma^* \backslash L(A_1)$.

Example: spam-filter based on the black-list “abb”, “abba”, and “abacus”!

(regular expression: $\overline{\text{“abb”} \mid \text{“abba”} \mid \text{“abacus”}}$)

Oracle-Automaton

Idea: replace non-determinism with oracle

Definition the *oracle-automaton* $A = Oracle(A_1)$ of FA A_1 has the following components:

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

Optimized Oracle-Automaton Construction

Proposition $\pi_1(L(Oracle(A_1))) = L(A_1)$ (π_1 projection on first component)

Proposition $Oracle(A_1)$ is deterministic iff $|I_1| \leq 1$.

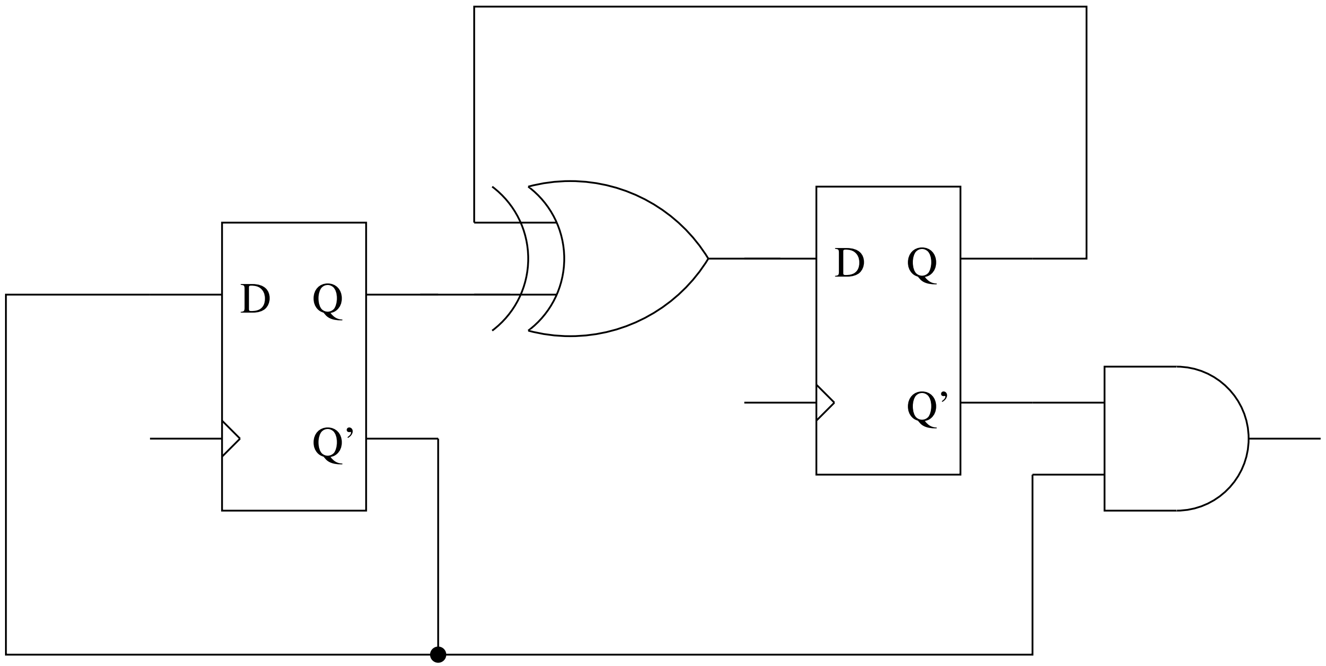
Proposition $Oracle(A_1)$ is almost always incomplete (e.g. $T_1 \neq S_1 \times \Sigma_1 \times S_1$ and $|S_1| > 1$).

Note completeness can be achieved, if A_1 is complete, and if $\{0, \ldots, n-1\}$ is added to Σ_1 instead of S_1, where n is the maximum number of successors: $n = \max_{s \in S, a \in \Sigma} |s \xrightarrow{a}|$.

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

Exercise construct the oracle automaton for $a \cdot b \cdot \mathbf{1}^* \cap \mathbf{1}^* \cdot b \cdot a$

Digital Design



I/O-Automaton

implementations of automata have to be deterministic

Definition *I/O-automaton* $A = (S, i, \Sigma, T, \Theta, O)$ consists of:

- a (finite) set of states S,
- exactly **one** initial state i,
- an input alphabet Σ,
- a transition **function** $T : S \times \Sigma \rightarrow S$
- an output alphabet Θ, with
- output function $O : S \times \Sigma \rightarrow \Theta$ (Moore machine: $O : S \rightarrow \Theta$)

Behavior of an I/O-Automaton

Let $w \in \Sigma^*$ and $a \in \Sigma$.

Definition interpret T as *extended* transition function $T\colon S \times \Sigma^* \rightarrow S$ as follows:

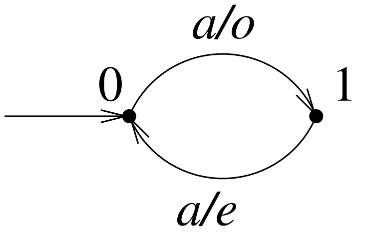
$s = T(s, \varepsilon)$ and $s' = T(s, a \cdot w) \Leftrightarrow \exists s''[s'' = T(s, a) \wedge s' = T(s'', w)]$.

Definition interpret O as *extended* output function $O\colon S \times \Sigma^* \rightarrow \Theta^*$ as follows:

$O(s, \varepsilon) = \varepsilon$ and $O(s, a \cdot w) = b \cdot w'$, with $b = O(s, a)$, $s' = T(s, a)$ and $w' = O(s', w)$.

Definition the *behavior* $B\colon \Sigma^* \rightarrow \Theta^*$ of an I/O-automaton is defined as $B(w) = O(i, w)$.

Example $S = \{0, 1\}$, $\Sigma = \{a\}$, $\Theta = \{e, o\}$,



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

$B(a^{2n}) = (oe)^n$, $B(a^{2n+1}) = (oe)^n o$

I/O-Automaton as FA

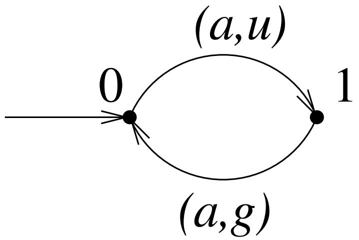
given an I/O-automaton $A = (S, i, \Sigma, T, \Theta, O)$.

Definition the FA for A is defined as $A' = (S, \{i\}, \Sigma \times \Theta, T', S)$ with

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

Proposition $B(w) = w'$ iff $(w, w') \in L(A')$

Example continued:



(graphically almost no difference)

FA as I/O-Automaton

let $A = (S, I, \Sigma, T, F)$ be an FA

Definition the I/O-automaton for A is defined as $A' = (\mathbb{P}(S), I, \Sigma, T', \{0,1\}, O)$ with T' the transition relation of $\mathbb{P}(A)$ and $O(S', a) = 1$ iff $S' \cap F \neq \emptyset$.

Proposition $w \in L(A)$ iff $B(w \cdot x) \in \mathbf{1}^{|w|} \cdot 1$ for one $x \in \Sigma$

Conclusion of the comparison of I/O-automata with FA:

in substance both are the same mathematical structure

we concentrate on the more compact and more elegant FA version

in particular non-determinism is easier to use with FA

Process Algebra (PA)

- modeling of *distributed* systems
  - Calculus of Communicating Systems (CCS) [Milner80]
  - Communicating Sequential Processes (CSP) [Hoare85]
  - more specifically: **asynchronously** communicating processes (protocols / SW)
- synthesis: process algebra (PA) as programming language (e.g. Occam, Lotos)
- verification of (abstract) PA models is simpler
- **theory:** mathematical properties of distributed systems
  - how to compare distributed systems?
  - simulation, bisimulation, observability, divergence (⇒ model checking course)

PA Equations (PAE)

- right linear grammar = regular language = Chomsky 3 language

  grammar G: $N = \varepsilon \mid aM \mid bM$ $M = cN \mid dN$ start symbol N

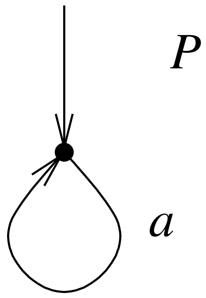
  $\Rightarrow$ language $L(G) = ((a \mid b)(c \mid d))^*$ (as regular expression)

- syntax in PA:
  - same idea: equations of non-terminals = processes
  - concatenation not with juxtaposition but with '.' operator
  - choice represented with '$+$' operator (not with '$|$')

- semantics
  - we are only interested in potential sequences = event streams

Concatenation

graphical representation



$P = a.P$

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

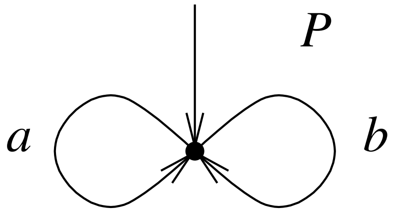
equation

operational semantics rule
(here P is only a meta variable)

'.' operator means sequential composition

Choice

graphical representation



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

$P = a.P + b.P$

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

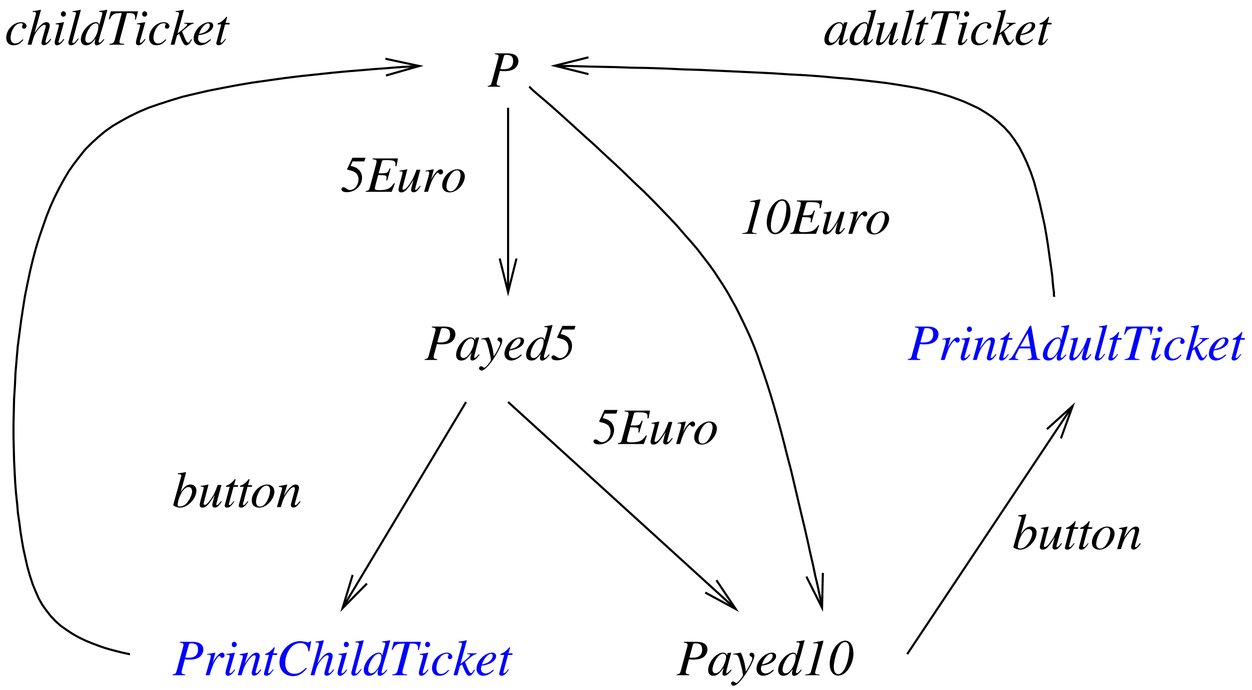
equation

operational semantics rule
(here again P, Q are meta variables)

'$+$' operator means non-deterministic choice

Ticket-Vending-Machine

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



Labelled Transition Systems (LTS)

- LTS as **operational semantics** of PAE
- almost the same as an automaton, but $\ldots$
  - no final states: in some sense all states are final
  - only possible event streams matter
- LTS $A = (S, I, \Sigma, T)$ with
  - state set S
  - actions Σ
  - transition relation $T \subseteq S \times \Sigma \times S$ defined through operational semantics
  - initial states $I \subseteq S$

Syntactical Restrictions

- divergent self-cycles
  - $P = a.P + P$ is an **invalid** PAE
  - there are no ε-transitions in contrast to FAs

    (actions “need time”, ε has connotation of not really taking time)

- avoid self-cycles
  - term T is **guarded** if T only occurs in the form $a.T$

    (where a can be different for all occurrences of T of course)
  - simplest restriction:

    process variables on the right hand side (RHS) of an PAE are all guarded
  - or more complex: each “cycle” contains at least one action

Data in PA

- actions and states can be **parameterized**
  - which also gives rise to parameterized equations

- previous example with $x \in \{5, 10\}$:

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

- it is possible to operate on data as well:

$$Payed(x) \quad = \quad euro(y).Payed(x+y) + button.ticket(x).P$$

  - actually allows modeling of *infinite systems*
  - and turns PA into a real programming language

Conditions

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

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

(and similar rules for **if-then** alone)

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

Parallel-Operator

synchronization through rendezvous in CSP

$$\Theta \subseteq \Sigma$$

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

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

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

rendezvous does not distinguish sender and receiver

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

$\Sigma(P)$ is the subset of actions of Σ which occur in P syntactically

Parallel-Operator Properties

Proposition $||$ is commutative: $P \,||\, Q \overset{a}{\to} P' \,||\, Q'$ iff $Q \,||\, P \overset{a}{\to} Q' \,||\, P'$

proof follows directly from the rules

Proposition $||$ is associative

proof: Let $P = P_1 \,||\, (P_2 \,||\, P_3)$, $P' = P_1' \,||\, (P_2' \,||\, P_3')$, $Q = (P_1 \,||\, P_2) \,||\, P_3$, $Q' = (P_1' \,||\, P_2') \,||\, P_3'$

To show: $P \overset{a}{\to} P' \quad \Leftrightarrow \quad Q \overset{a}{\to} Q'$

8 cases of $a \in \Sigma(P_i)$ resp. $a \notin \Sigma(P_i)$ for each direction

intuition:

1. $a \in \Sigma(P_i) \Rightarrow P_i \overset{a}{\to} P_i'$
2. P_i with $a \notin \Sigma(P_i)$ does not change ($P_i' = P_i$)
3. the sames applies for every “parallel composition” of the P_i

Implications of Properties of the Parallel-Operator

- “parenthesis” around || can be omitted:

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

- order is irrelevant:

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

- parallel composition $\underset{i \in J}{||} P_i$ of arbitrary processes P_i over an index set J:

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

Hiding

- hiding resp. abstraction of internal, **unobservable** actions

- abstracted to “silent” action τ
  - assumption: $\tau \notin \Sigma$
    - formally consider only $\Sigma \,\dot\cup\, \{\tau\}$ as actions
    - it is not possible to synchronize on τ
  - τ still needs time

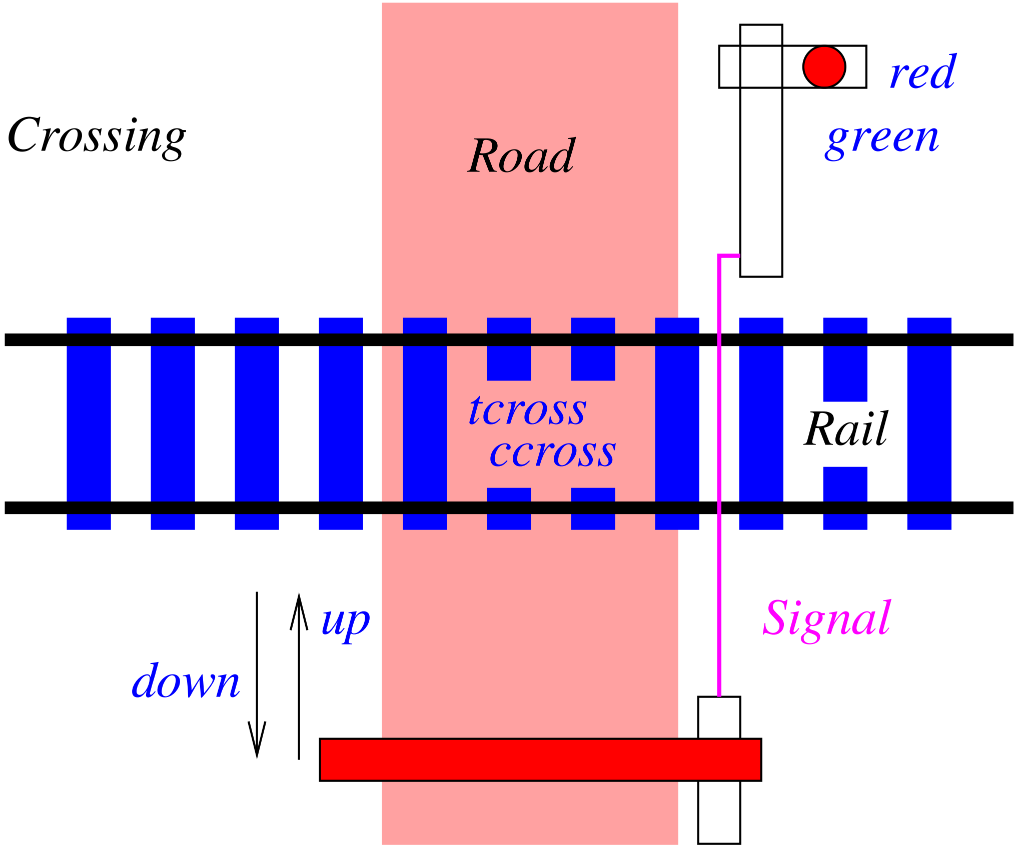
$$R^{\notin}_{\backslash} \quad \frac{P \xrightarrow{a} Q}{P\backslash\Theta \xrightarrow{a} Q\backslash\Theta} \quad a \notin \Theta \qquad\qquad R^{\in}_{\backslash} \quad \frac{P \xrightarrow{a} Q}{P\backslash\Theta \xrightarrow{\tau} Q\backslash\Theta} \quad a \in \Theta$$

- typical usage of internal synchronization $R = (||_{i=1}^{n} Q_i)\backslash\{x_1, \ldots, x_n\}$

Railroad Crossing

[BradfieldStirling]

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

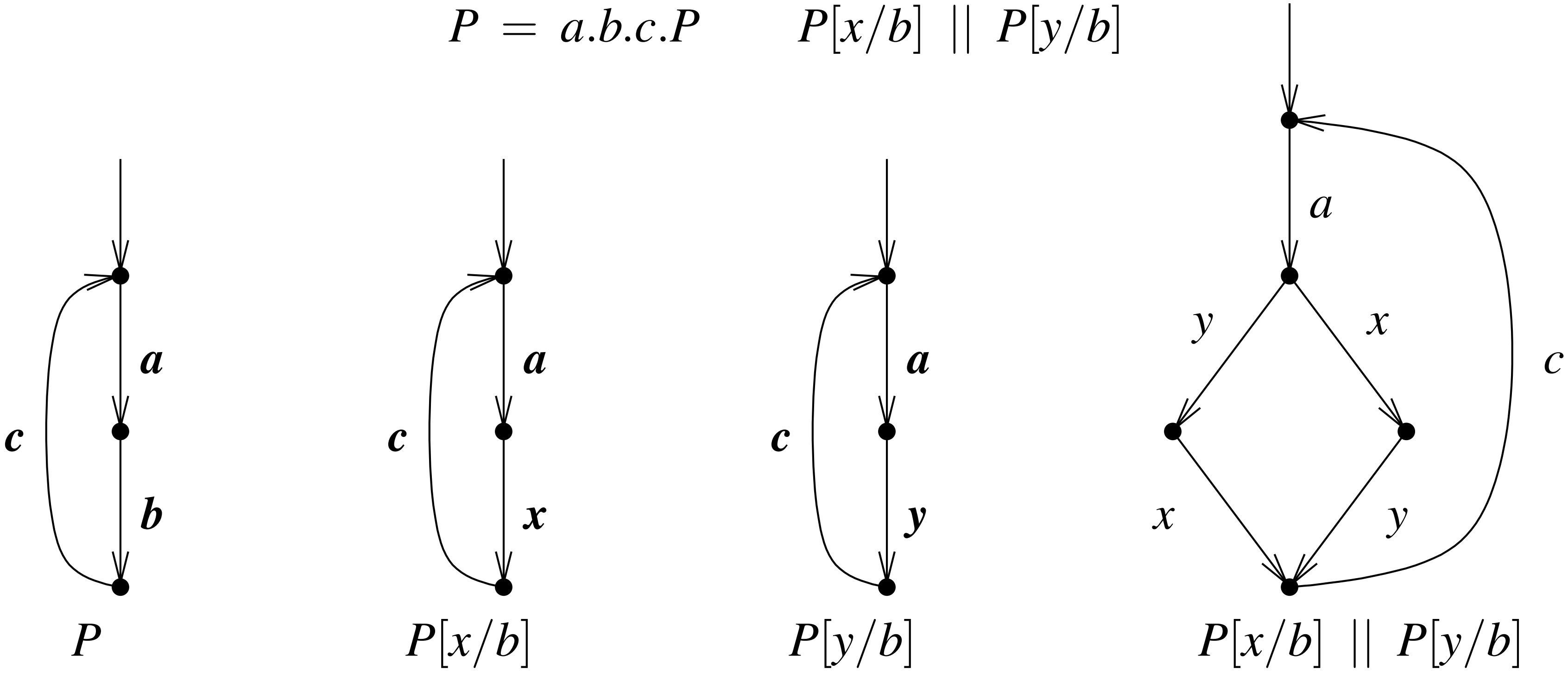


Linking as substitution of actions

$$R_{[\,]}\;\frac{P \xrightarrow{a} Q}{P[b/a] \xrightarrow{b} Q[b/a]} \qquad \text{Example:} \quad (a.P)[b/a] \xrightarrow{b} P[b/a]$$

needed to “link” processes or instantiate templates:

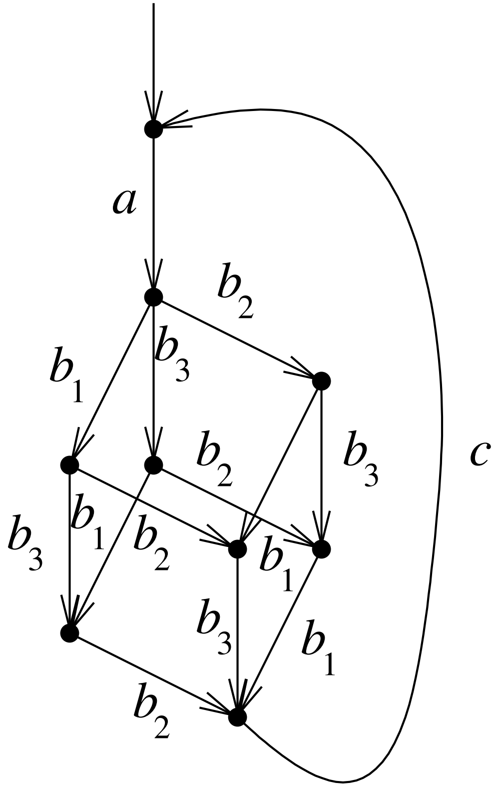
$$P \;=\; a.b.c.P \qquad P[x/b] \;||\; P[y/b]$$



Parameterized Linking

$$P \;=\; a.b.c.P$$

$$\mathop{\Big|\Big|}_{i=1}^{3} \; P[b_i/b]$$



Milner’s Scheduler

- classical example of process algebra
  - modeling of a round robin scheduler
- scheduling of n processes $||P_i$ with $P = a.z.b.P$ and $P_i = P[a_i/a, z_i/z, b_i/b]$
  - a start one run of a process
  - z internal action(s)
  - b end of one run of a process
- **Restrictions**:
  - processes are started round robin in the order $P_1, P_2, \ldots$
  - nothing is about execution order of the b_i!

Incorrect Solution for Milner's Scheduling

- idea: proxy for each process
- divide scheduler R' in token ring of n parallel cyclic processes Q'
- each Q'_i controls start (a_i) and end (b_i) of P_i, $\ldots$
- $\ldots$ hands over x_i control to next Q'_{i+1} $\ldots$
- and then waits to get control x_{i-1} from previous Q'_{i-1} in ring

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

Correct Solution for Milner's Scheduler

- incorrect solution does **not** accept the legal sequence:
  - ending P_2 before P_1: $a_1 a_2 b_2 b_1 \ldots$
- decouple ending (b) and accepting control (y)

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

- implemented by non blocking waiting on two different messages
  - in programming languages: try-locking, multiple threads, select (java.nio), …
- slightly sloppy alternative notation $b.y + y.b = b \parallel y$ (we do not have a nil process)

Differences in CCS

- actions: $\Sigma \,\dot\cup\, \overline{\Sigma} \,\dot\cup\, \{\tau\}$ overlined actions are outputs, otherwise inputs

- different hiding principle (new syntax: double instead of single backslash)

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

- pairwise **explicit** synchronization

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

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

Comparison of CSP and CCS on Train Collision Example

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

resp. in CCS

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

Other Variants

- originally CSP had channels with data
  - inputs: $channel\,?\,datain$, outputs: $channel\,!\,dataout$
- π-calculus after [MilnerParrowWalker]
  - (references to) channels / connections can be used as data as well
  - example: $TimeAnnounce = ring(caller).\overline{caller}(CurrentTime).\overline{hangup}.TimeAnnounce$
- probabilistic behavior
  - transitions have a “transition probability”
- timed process algebra
  - transitions *need* (explicitly specified) time

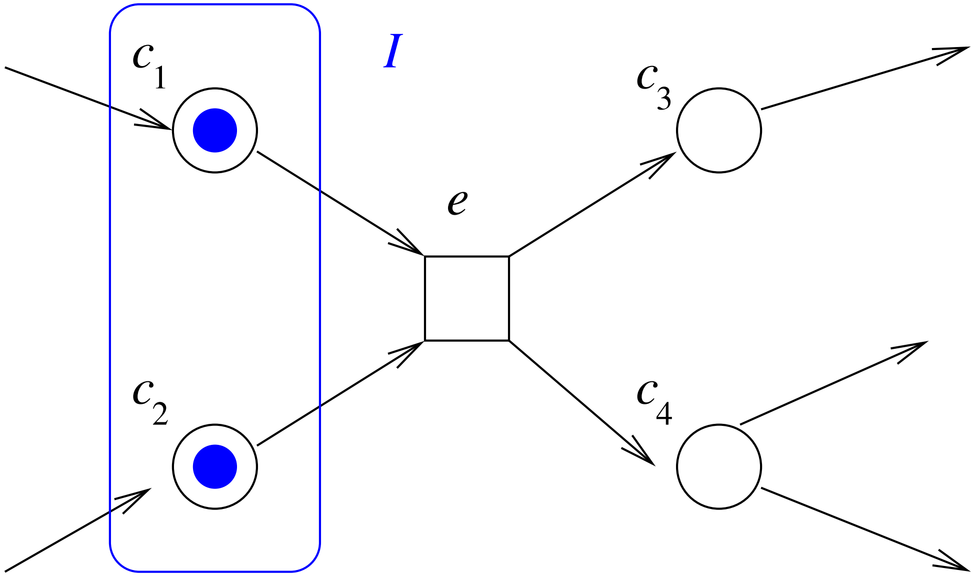
Petri Nets (PN)

- beside process algebra the most common modeling language for *distributed* systems
  - investigated since 60ies, now also known as **activity diagrams** in UML
  - again: **asynchronously** communicating processes (protocols / SW)
- modeling and verification tools available
- **theory:** many interesting results, vast literature
  - finiteness, deadlock, …
- extension motivated by practice
  - data, coloring, hierarchy, and again quantitative aspects etc.

Condition-Event-Net (CEN)

Definition

A CEN $N = (C, I, E, G)$ is made of conditions C, an initial marking $I \subseteq C$, events E and a dependence graph $G \subseteq (C \times E) \dot{\cup} (E \times C)$

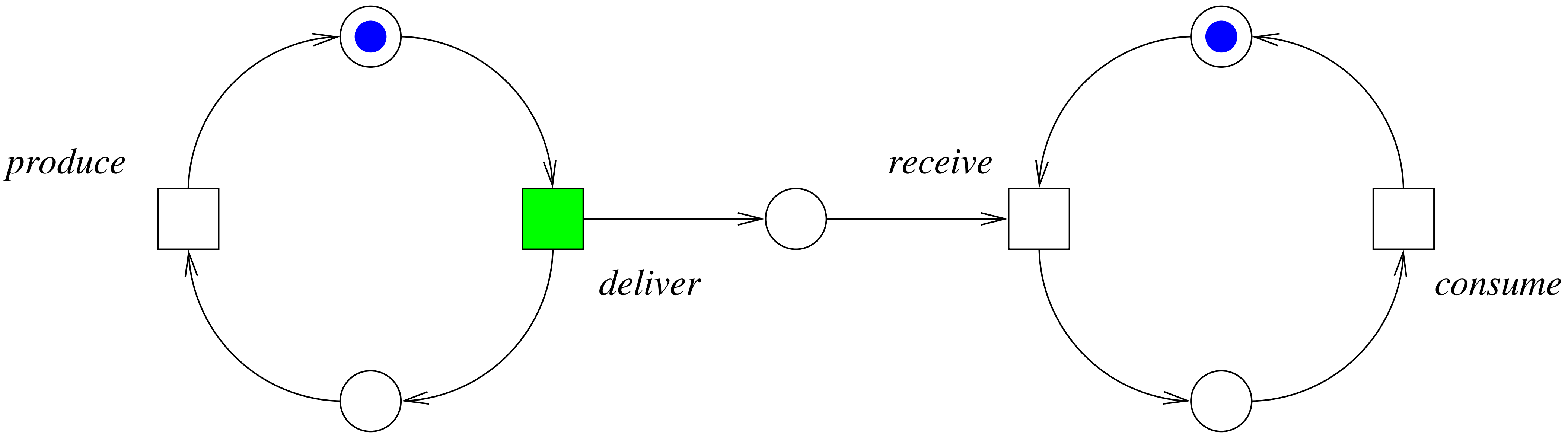


- we also use → instead of G

- can be interpreted as *bipartite* graph oder …

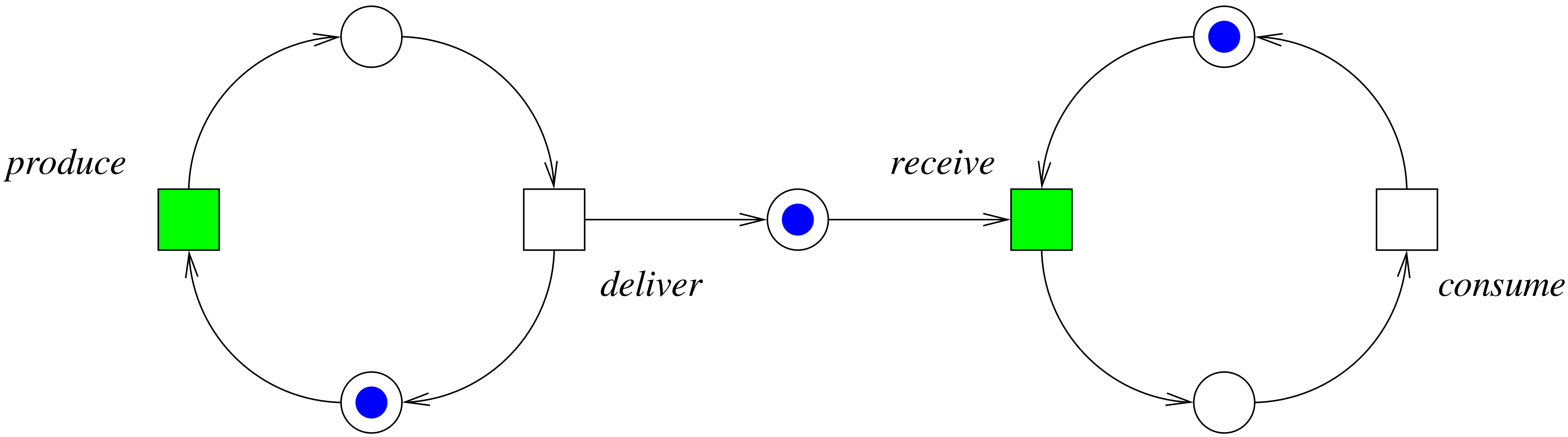
- … hyper graph with multiple source resp. target edges E

Producer Consumer CEN: Initial Marking



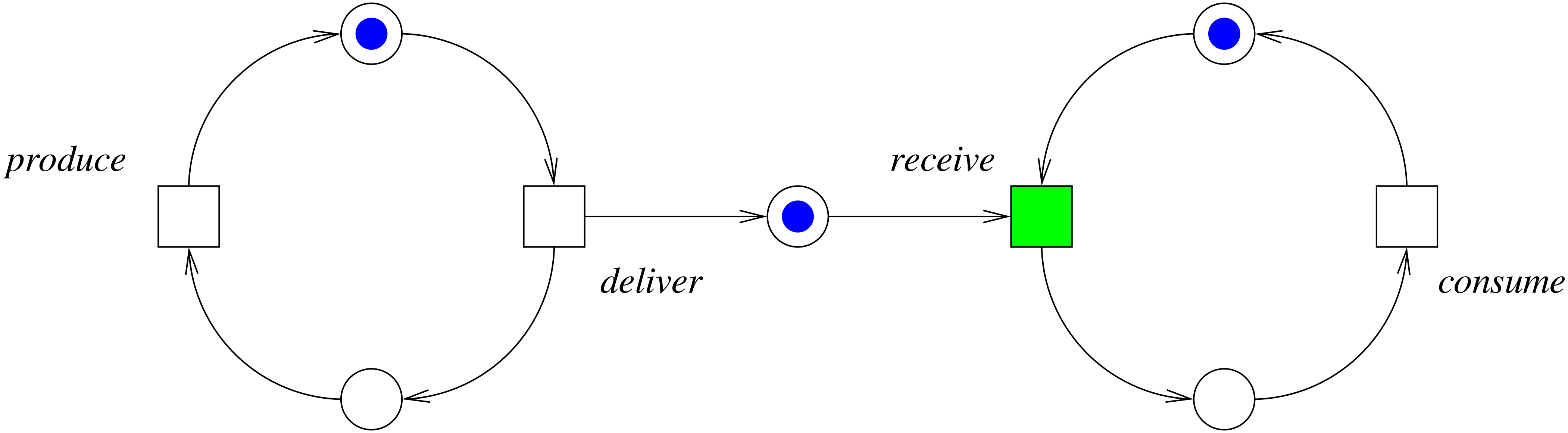
only one event / transition can fire

Producer Consumer CEN: Delivered



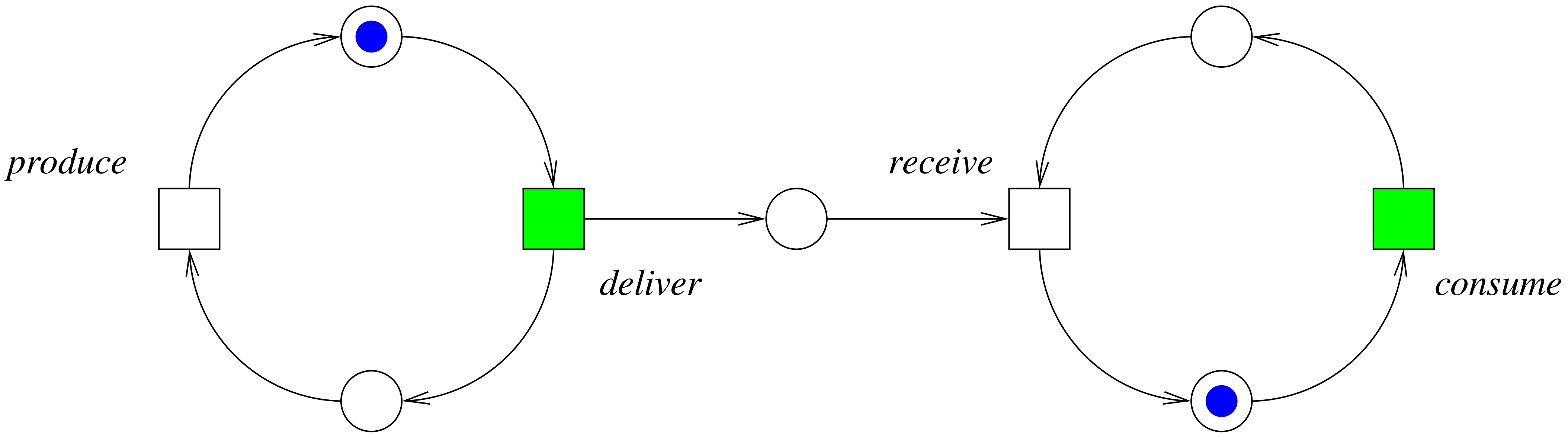
two events / transitions can fire

Producer Consumer CEN: Produced Again



target condition of *deliver* occupied

Producer Consumer CEN: Consumed



again choice of two possible events

CEN Semantics as LTS

Definition Let CEN $N = (C, I, E, G)$. The LTS $L = (S, \{I\}, \Sigma, T)$ for N is defined as

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

$$
\begin{array}{lll}
T(C_1, e, C_2) \quad \text{iff} & G^{-1}(e) \subseteq C_1 & \text{pre-conditions satisfied} \quad (1) \\
& G(e) \cap C_1 = \emptyset & \text{post-conditions satisfied} \quad (2) \\
& C_2 = (C_1 \backslash G^{-1}(e)) \cup G(e) & \text{state update}
\end{array}
$$

$G(e)$ = post -conditions of event e (or $e \rightarrow$)

$G^{-1}(e)$ = pre -conditions of event e (or $\rightarrow e$)

Ad: CEN Semantics

- states $M \in \mathbb{P}(C)$ of the LTS are also called markings of the CEN

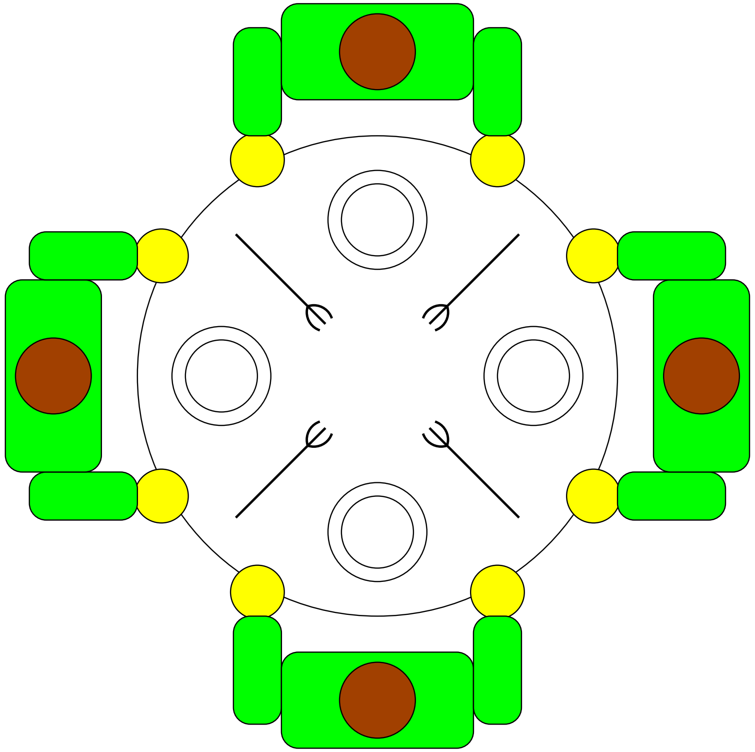
- event e is **enabled** in M iff $M \overset{e}{\to} \neq \emptyset$

- marking $M \in \mathbb{P}(C)$ is a **deadlock** iff
  - M is is “dead end” in the reachability graph of the LTS iff
  - no event in M is enabled iff
  - all events are *disabled* iff
  - $\forall e \in E[M \overset{e}{\to} = \emptyset]$

- a CEN has a deadlock iff a deadlock is reachable

Example Dining Philosophers

n philosophers, n forks, n plates

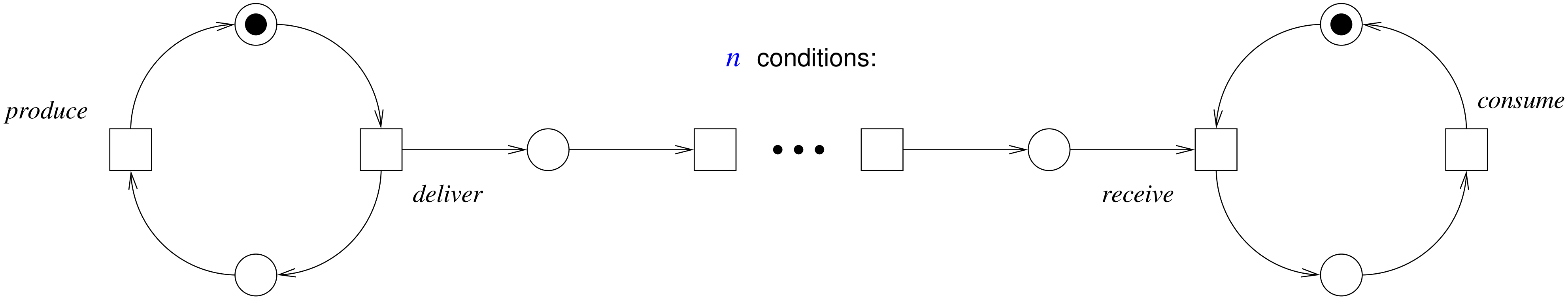


philosophers alternate in thinking and eating

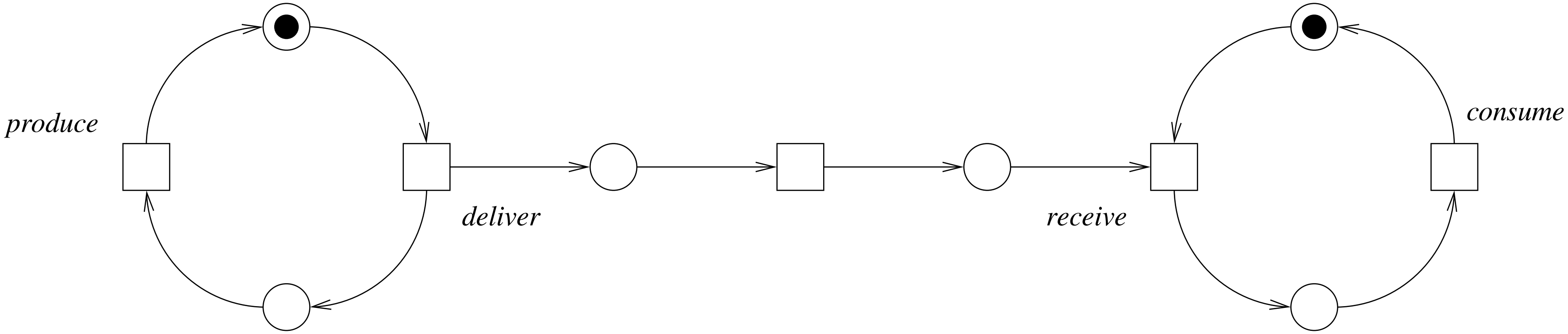
they need to pick up and use two forks to eat

forks can not be picked up at the same time (atomically)

Capacities



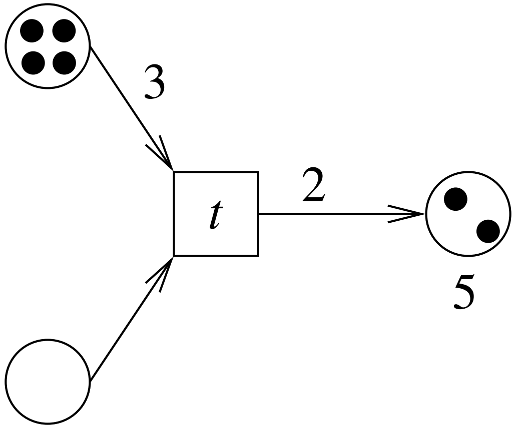
buffer capacity n



buffer capacity 2

Place Transition Net (PTN)

Definition A PTN $N = (P, I, T, G, C)$ consists of places P, initial marking $I\colon P \to \mathbb{N}$, transitions T, connection graph $G \subseteq (P \times T) \dot{\cup} (T \times P)$, and capacities $C\colon P \dot{\cup} G \to \mathbb{N}_\infty$.



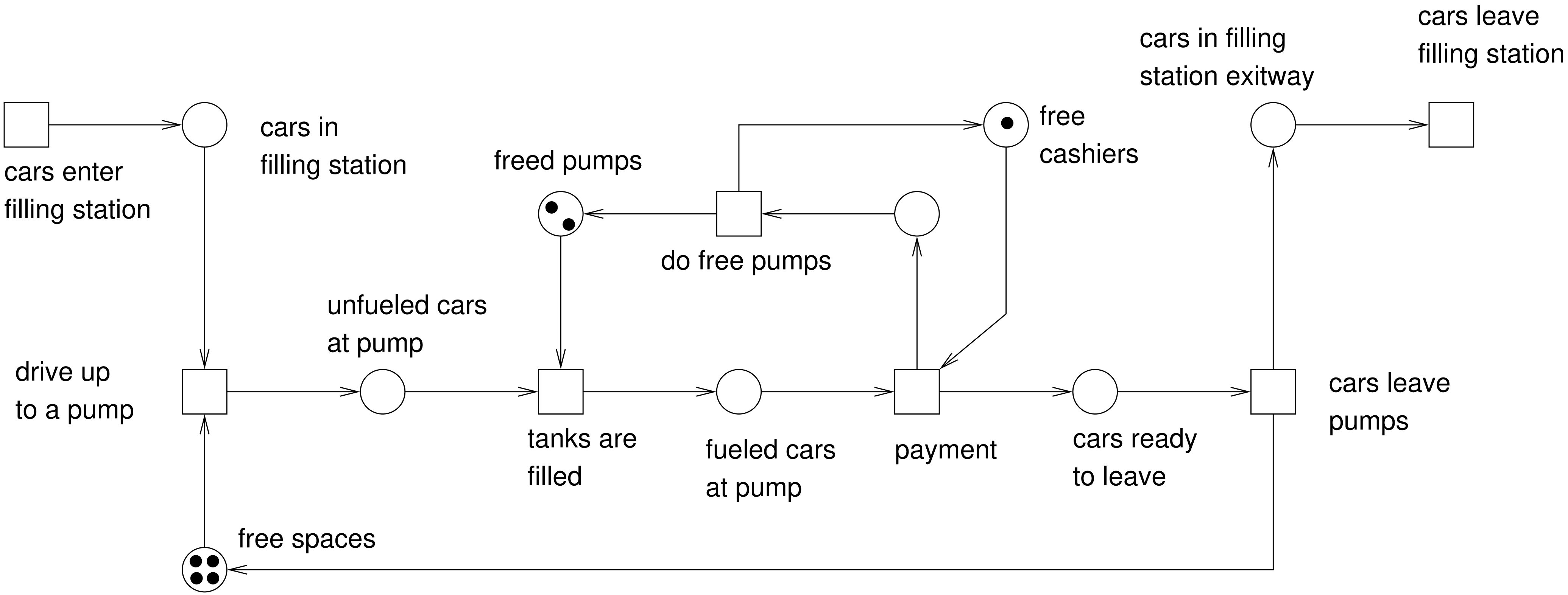
- capacity of a *connection* is finite and is one if not specified explicitly

- capacity of a *place* can be ∞ and is ∞ if not specified explicitly

- CEN can be interpreted as PTN with constant capacity $C \equiv 1$

Filling Station

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



PTN Semantics as LTS

given a PTN $N = (P, I, T, G, C)$

Definition transition $t \in T$ can fire in a state / marking $M\colon P \to \mathbb{N}$ iff

$C((p,t)) \leq M(p)$ for all $p \in G^{-1}(t)$ and

$C((t,q)) + M(q) \leq C(q)$ for all $q \in G(t)$.

Definition transition $t \in T$ leads from $M_1\colon P \to \mathbb{N}$ to $M_2\colon P \to \mathbb{N}$ iff

t can fire in M_1, and $M_2 = M_1 - M_- + M_+$ with

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

Definition the LTS $L = (S, \{I\}, \Sigma, T_L)$ of N is defined through

$S = \mathbb{N}^P$ $\Sigma = T$ and $T_L(M_1, t, M_2)$ iff t leads from M_1 to M_2

Temporal Logic

application in computer science goes back to A. Pnueli

- often used to specify concurrent and reactive systems
- allows to relate properties at different time points
  - “tomorrow the weather is nice”
  - “reactor is not going to overheat”
  - “central locking of a car opens immediately after a crash”
  - “airbag only inflates if a car crash happens”
  - “acknowledge (ack) has to be preceded by a request (req)”
  - “if the elevator is called it will show up eventually”
- granularity of time steps has to be defined

Simplified Hennessy-Milner Logic (HML)

HML is an example for temporal logic over LTS

let Σ be the alphabet of actions

Definition syntax consists of the usual boolean constants $\{0,1\}$, boolean operators $\{\wedge,\neg,\rightarrow,\ldots\}$ and unary **modal operators** $[a]$ and $\langle a\rangle$ with $a\in\Sigma$.

read $[a]\,f$ as for all a-successors of the current state f holds

read $\langle a\rangle\,f$ as for one a-successor of the current state f holds

abbreviations $\langle\Theta\rangle\,f$ denotes $\bigvee_{a\in\Theta}\langle a\rangle\,f$ resp. $[\Theta]\,f$ for $\bigwedge_{a\in\Theta}[a]\,f$

Θ can also be written as a boolean expression over Σ

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

Examples Simplified HML

1.	$[a]\,1$	for all a-successor 1 holds (always true)
2.	$[a]\,0$	for all a-successor 0 holds (a is not possible)
3.	$\langle a\rangle\,1$	for one a-successor 1 holds (a should be possible)
4.	$\langle a\rangle\,0$	for one a-successor 0 holds (always wrong)
5.	$\langle a\rangle\,1 \;\wedge\; [b]\,0$	a has to be possible but not b
6.	$\langle a\rangle\,1 \;\wedge\; [\neg a]\,0$	a and only a should be possible
7.	$[a \vee b]\,\langle a \vee b\rangle\,1$	after a or b again a or b should be possible
8.	$\langle a\rangle\,[b]\,[b]\,0$	a should be possible and afterwards b not twice
9.	$[a](\langle a\rangle\,1 \;\rightarrow\; [a]\,\langle a\rangle\,1)$	if a is possible after a again, then also a second time

Semantics Simplified HML

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

Definition semantics are defined recursively as $s \models f$ (read “f holds in s”), with $s \in S$ and f a simplified HML formula.

$$
\begin{array}{lll}
s & \models & 1 \\
s & \not\models & 0 \\
s & \models & [\Theta]\, g \quad \text{iff} \quad \forall\, a \in \Theta\, \forall\, t \in S\text{:} \quad \text{if } s \xrightarrow{a} t \text{ then } t \models g \\
s & \models & \langle\Theta\rangle\, g \quad \text{iff} \quad \exists\, a \in \Theta\, \exists\, t \in S\text{:} \quad s \xrightarrow{a} t \text{ and } t \models g
\end{array}
$$

Definition $L \models f$ holds (read “f holds in L”) iff $s \models f$ for all $s \in I$

Definition expansion of f is the set of states $[[f]]$ in which f holds.

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

Finite and Infinite Traces

Let $L = (S, I, \Sigma, T)$ be an LTS.

Definitions A Trace π of L is a finite or infinite sequence of states

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

For each pair (s_i, s_{i+1}) in π there is an $a \in \Sigma$ with $s_i \xrightarrow{a} s_{i+1}$. Therefore there exist $a_0, a_1, \ldots$ with

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

$|\pi|$ is the length of π, e.g. $|\pi| = 2$ for $\pi = (s_0, s_1, s_2)$, and $|\pi| = \infty$ for infinite traces.

$\pi(i)$ is the i'th state s_i of π for $i \leq |\pi|$

$\pi^i = (s_i, s_{i+1}, \ldots)$ denotes the suffix of π starting with the i'th state s_i for $i \leq |\pi|$

Note: if $|\pi| = \infty$ then $|\pi^i| = \infty$ for all $i \in \mathbb{N}$

Computation Tree Logic for HML (CTL/HML)

first only in combination with HML

Definition CTL/HML syntax based on the syntax of HML and additionally

unary temporal path operators $\mathbf{X}$, $\mathbf{F}$, $\mathbf{G}$ and one binary temporal path operator $\mathbf{U}$.

Path operators have to be prefixed with a path-quantifier $\mathbf{E}$ or $\mathbf{A}$.

$\mathbf{EX}f$	in one (immediate) successor state f holds	$\equiv \langle\Sigma\rangle f$
$\mathbf{AX}f$	in all successor states f holds	$\equiv [\Sigma] f$
$\mathbf{EF}f$	in one future f holds eventually	*exists finally*
$\mathbf{AF}f$	in all possible orders of events f holds eventually	*always finally*
$\mathbf{EG}f$	in one future f holds all the time	*exists globally*
$\mathbf{AG}f$	f holds always	*always globally*
$\mathbf{E}[f\ \mathbf{U}\ g]$	potentially f holds until finally g gilt (note g has to hold on this trace eventually)	*exists until*
$\mathbf{A}[f\ \mathbf{U}\ g]$	f always holds until finally g occurs (note g has to hold on all traces eventually)	*always until*

Examples 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 for $\mathbf{E}[\cdot\ \mathbf{U}\ \cdot]$ requires additional temporal path operator)

$\mathbf{AG}\,[\neg \mathit{safe}]\,0$	it is never possible to execute unsafe actions
$\mathbf{EF}\,\langle\neg \mathit{safe}\rangle\,1$	potentially an unsafe action can be executed
$\neg\mathbf{E}[\neg\,\langle \mathit{req}\rangle\,1\ \mathbf{U}\ \langle \mathit{ack}\rangle\,1]$	there is an order of events in which ack becomes possible and req was not possible before
$\mathbf{AG}\,[\mathit{req}]\,\mathbf{AF}\,[\neg \mathit{ack}]\,0$	always after req a point is reached, from no other action than ack is possible

CTL/HML allows to combine requirements about states and actions

which is required to express useful facts and unfortunately not very elegant

Semantics of CTL/HML Operators

Let f be a CTL/HML formula, L an LTS, π a trace of L, and $i, j \in \mathbb{N}$.

Definition semantics are defined recursively: $s \models f$ (read "f holds in s")

(only for the new CTL operators here)

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

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

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

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

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

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

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

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

Kripke Structures

- classical semantic model for temporal logic

- only states, no actions
  - LTS with exactly one action $(|\Sigma| = 1)$
  - additionally annotation of states with atomic propositions

- has its roots in modal logics:
  - different “worlds” from S are connected through $\rightarrow$ resp. T
  - $[\,]\,f$ iff for all immediate successor worlds f holds
  - $\langle\,\rangle\,f$ iff there is an immediate successor world in which f holds

Kripke Structures

Let $\mathcal{A}$ be the set of atomic propositions (boolean predicates).

Definition a Kripke structure $K = (S, I, T, \mathcal{L})$ consists of the following components:

- set of states S.
- initial states $I \subseteq S$ with $I \neq \emptyset$
- a *total* transition relation $T \subseteq S \times S$ (T total iff $\forall s[\exists t[T(s,t)]]$)
- labelling/marking/annotation $\mathcal{L}\colon S \to \mathbb{P}(\mathcal{A})$.

Labelling maps a state s on to the set of atomic propositions that hold in s:

$$\mathcal{L}(s) = \{gray, warm, dry\}$$

LTS as Kripke Structure

Definition the Kripke structure $K = (S_K, I_K, T_K, \mathcal{L})$ for a complete LTS $L = (S_L, I_L, \Sigma, T_L)$ is defined with the following components

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

$$T_K((s,a),(s',a')) \quad \text{iff} \quad T_L(s,a,s') \text{ and } a' \text{ arbitrary}$$

similar construction as the oracle automaton

Proposition

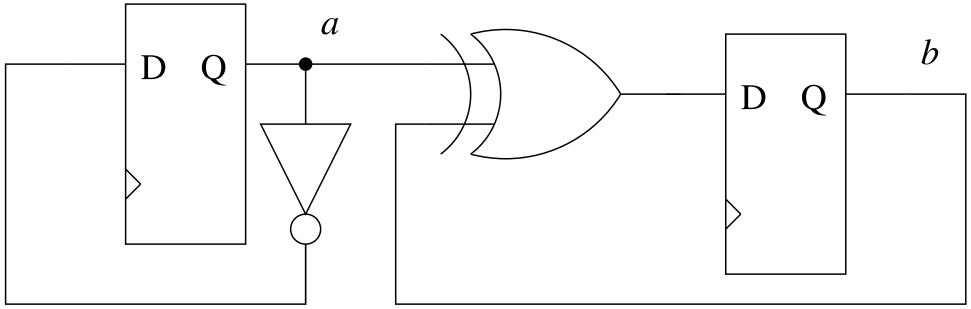
$$s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} \cdots \overset{a_{n-1}}{\to} s_n \quad \text{in } L$$

$$\text{iff}$$

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

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

2-Bit Counter as Kripke Structure



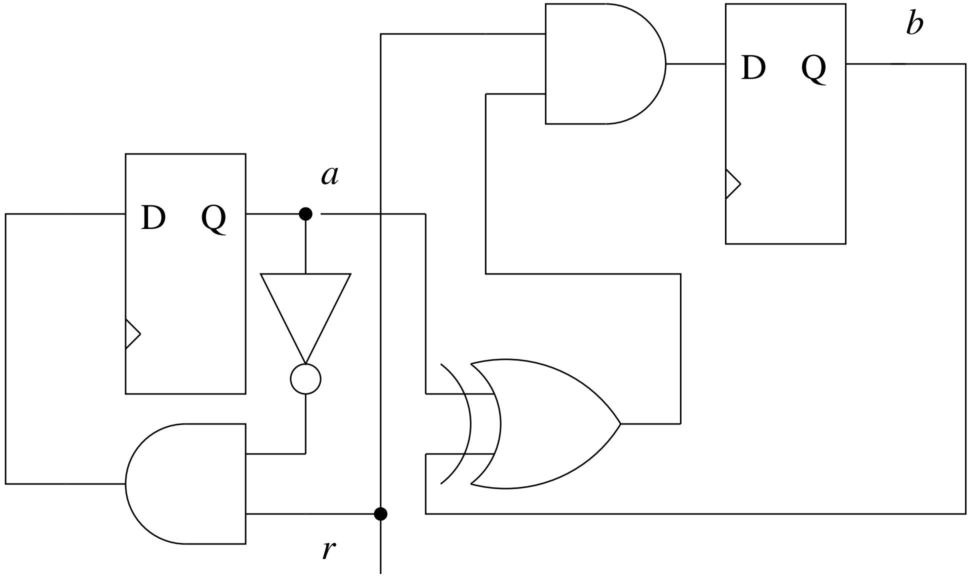
$$S = \mathbb{B}^2$$

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

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

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

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



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

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

$$T = \ldots$$

$a \in L(s)$ iff $s \in \{(-,-,1)\}$

$b \in L(s)$ iff $s \in \{(-,1,-)\}$

$r \in L(s)$ iff $s \in \{(1,-,-)\}$

we assume that circuits abstracted to netlists do not have an initial state

Computational Tree Logic (CTL)

classical version of CTL on Kripke structures

Definition CTL syntax contains all $p \in \mathcal{A}$, all boolean operators $\wedge, \neg, \vee, \rightarrow, \ldots$ and the temporal operators $\mathbf{EX}, \mathbf{AX}, \mathbf{EF}, \mathbf{AF}, \mathbf{EG}, \mathbf{AG}, \mathbf{E}[\cdot\ \mathbf{U}\ \cdot]$ and $\mathbf{A}[\cdot\ \mathbf{U}\ \cdot]$.

Definition CTL semantics over a Kripke structure $K = (S, I, T, \mathcal{L})$ are defined recursively as for CTL/HML, except for the base case in which $s \models p$ iff $p \in \mathcal{L}(s)$.

Examples for 2-Bit counter with reset

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

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

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

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

infinitely often $\overline{a} \wedge \overline{b}$

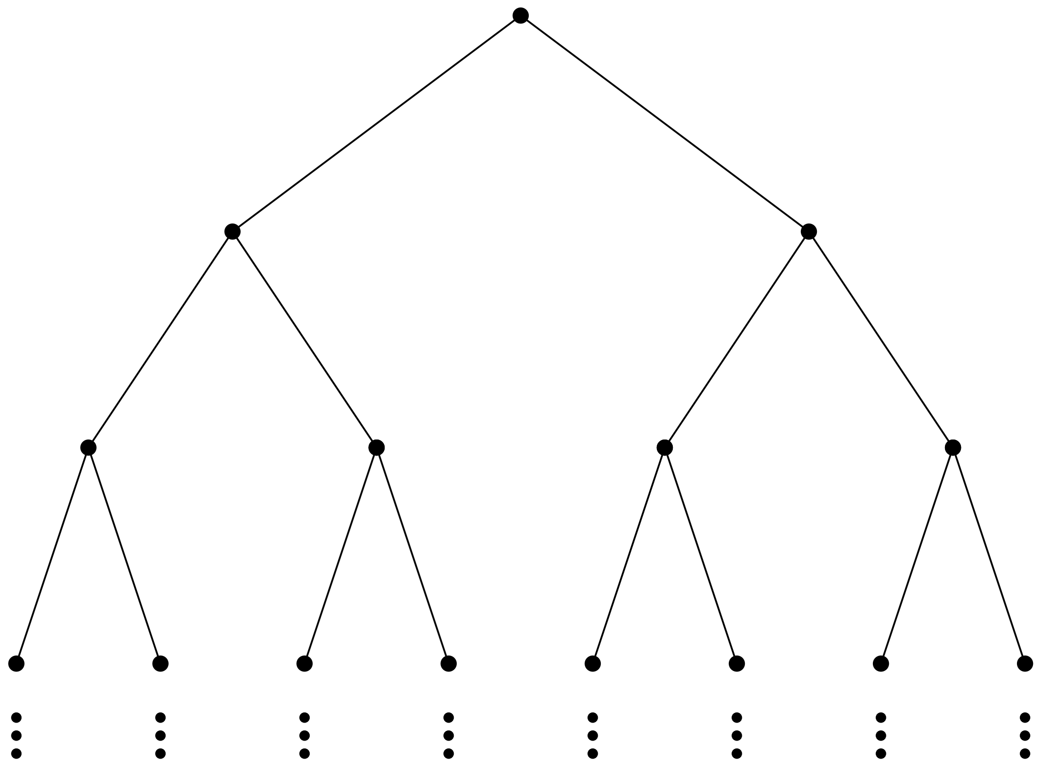
$$\mathbf{AG}(\overline{a} \wedge \overline{b} \wedge r \rightarrow \mathbf{AX}\ \mathbf{A}[(a \vee b)\ \mathbf{U}\ (\overline{a} \wedge \overline{b})])$$

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

Definition f holds in K written $K \models f$ iff $s \models f$ for all $s \in I$

generic definition

Computation Tree



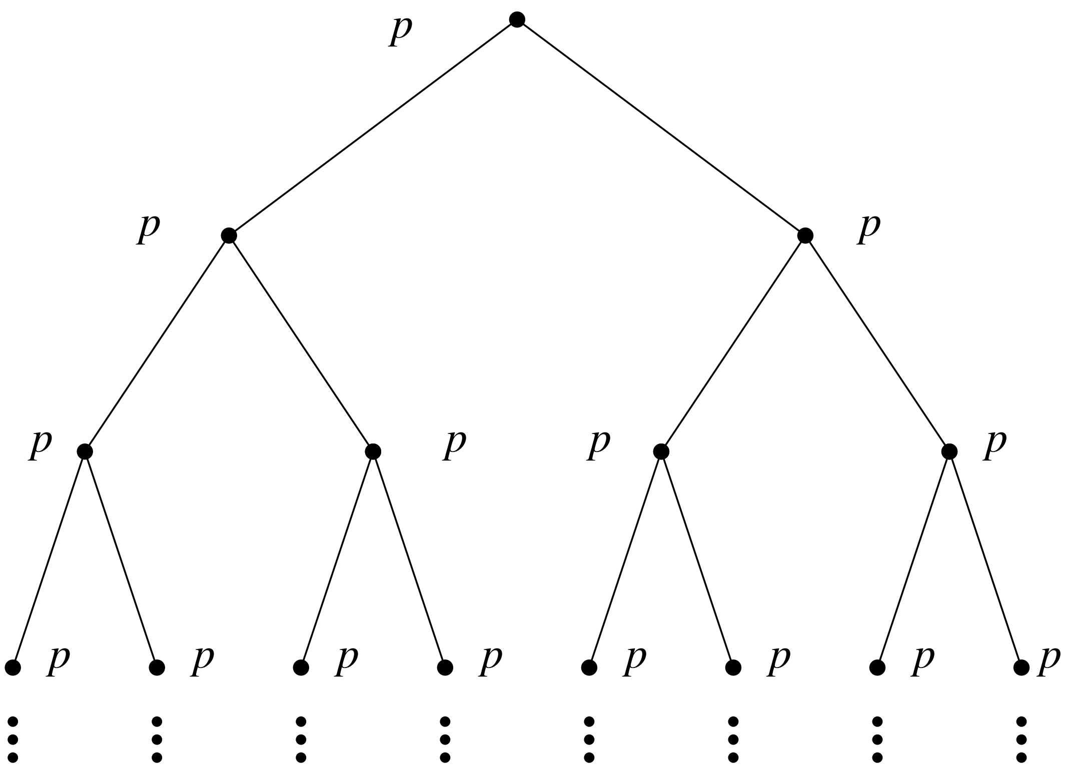
all possible orders of events are represented in one (infinite) computation tree

CTL describes the branching behavior of this computation tree

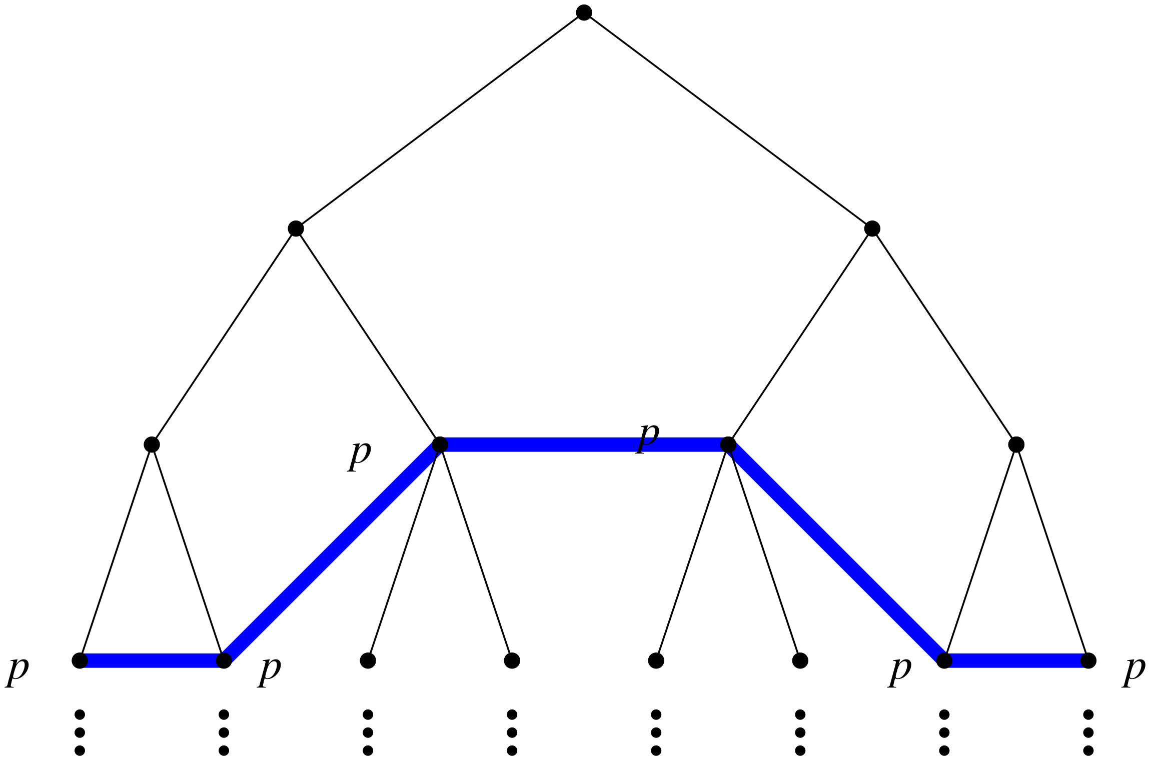
and has a local state view

every state is the starting point of new branching paths

Computation Tree $\mathbf{AG}p$



Computation Tree $\mathbf{AF}p$



Linear Temporal Logic (LTL)

Definition LTL syntax similar to CTL syntax, except that temporal operators do not have path quantifiers: LTL only has $\mathbf{X}, \mathbf{F}, \mathbf{G}$ and $\mathbf{U}$.

Definition LTL semantics defined recursively along infinite paths π in K:

$\pi \models p$ iff $p \in \mathcal{L}(\pi(0))$

$\pi \models \neg g$ iff $\pi \not\models g$

$\pi \models g \wedge h$ iff $\pi \models g$ and $\pi \models h$

$\pi \models \mathbf{X}g$ iff $\pi^1 \models g$

$\pi \models \mathbf{F}g$ iff $\pi^i \models g$ for one i

$\pi \models \mathbf{G}g$ iff $\pi^i \models g$ for all i

$\pi \models g\,\mathbf{U}\,h$ iff exists i with $\pi^i \models h$ and $\pi^j \models g$ for all $j < i$

Definition $K \models f$ iff $\pi \models f$ for all infinite paths π in K with $\pi(0) \in I$

Comparison LTL and CTL

- LTL only considers one single **linear** order of events

- then $(\mathbf{G}r) \rightarrow \mathbf{F}(a \wedge b)$ suddenly makes sense (premise is a restriction/assumption)

- LTL is compositional (w.r.t. sync. product of Kripke structures):
  - $K_1 \models f_1,\ K_2 \models f_2 \quad \Rightarrow \quad K_1 \times K_2 \models f_1 \wedge f_2$
  - $K_1 \models f \rightarrow g,\ K_2 \models f \quad \Rightarrow \quad K_1 \times K_2 \models g$

Proposition CTL and LTL have different expressibility:

$\mathbf{AXEX}p$ can not be specified in LTL, $\mathbf{AFAG}p$ does not have corresponding LTL formula

ACTL Formulas as LTL Formulas

[Clarke and Draghicescu'88]

ACTL is the sub logic of CTL formulas without $\mathbf{E}$ path quantifiers in NNF

NNF: negations only occur in front of atomic propositions $p \in \mathcal{A}$

Definition for an ACTL formula f define $f \backslash \mathbf{A}$ as the LTL formula obtained from f by deleting all path quantifiers, e.g. $(\mathbf{AGAF}p) \backslash \mathbf{A} = \mathbf{GF}p$.

Definition f and g are equivalent iff $K \models f \Leftrightarrow K \models g$ for all Kripke structures K.

(f and g can be formulas in different logics)

Theorem if an ACTL formula f is equivalent to an LTL formula g, then also to $f \backslash \mathbf{A}$.

Proof $K \models f \overset{\text{assumption}}{\Leftrightarrow} \forall\pi[\pi \models g] \underset{\text{+see below}}{\overset{\text{assumption}}{\Leftrightarrow}} \forall\pi[\pi \models f] \overset{!}{\Leftrightarrow} \forall\pi[\pi \models f \backslash \mathbf{A}] \overset{\text{Def.}}{\Leftrightarrow} K \models f \backslash \mathbf{A}$

(assume π to be initialized and in $\pi \models f$ interpreted as Kripke structure)

Syntactically Characterized Intersection of LTL and ACTL

[M. Maidl'00]

Let f and g be CTL resp. LTL formulas and $p \in \mathcal{A}$.

Definition every sub formula of an CTL$^{\text{det}}$ formula is of the following form:

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

Definition every sub formula of an LTL$^{\text{det}}$ formula is of the following form:

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

Theorem the intersection of LTL and ACTL is equivalent to LTL$^{\text{det}}$ resp. CTL$^{\text{det}}$

Intuition CTL semantics for CTL$^{\text{det}}$ are restricted to one path

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

$\Rightarrow$ non deterministic specifications can be misinterpreted

You can not count with LTL and CTL

[P. Wolper'83]

Specification "after m-th step p" holds (at least)

Proposition for all $m > 1$ there is no CTL nor LTL formula f with

$K \models f$ iff $\pi(i) \models p$ for all initialized paths π of K and all $i = 0 \bmod m$.

Problem $p \wedge \mathbf{G}(p \leftrightarrow \neg \mathbf{X} p)$ denotes "exactly every 2nd step p holds"

Solutions

- add modulo m counter to model (problems with compositionality)
- logic extensions
  - ETL with additional temporal operators defined through automata ...
  - ... resp. quantifiers over atomic propositions (embed automata into the logic)
  - regular expressions: $\neg\left((\underbrace{1;\ldots;1}_{m-1};p)^*;\underbrace{1;\ldots;1}_{m-1};\neg p\right)$ resp. $(\underbrace{1;\ldots;1}_{m-1};p)^\omega$

Fairness

- specifications often need additional *fairness* assumptions
  - e.g. abstraction of scheduler: “each process gets it’s turn”
  - e.g. one component must be enabled infinitely often
  - e.g. infinitely often a transmission channel does not produce an error
- no problem in LTL: $(\mathbf{GF}f) \rightarrow \mathbf{G}(r \rightarrow \mathbf{F}a)$
- fair Kripke structures for CTL:
  - additional component F of fair states
  - path π fair iff $|\{i \mid \pi(i) \in F\}| = \infty$
  - only consider fair paths

μ-Calculus

- restricted class of quantifiers over sets of states
  - quantified variables $V = \{X, Y, \ldots\}$
  - in general also over sets and thus gives a second order logic
- fix point logic: least fix points specified with μ and largest with ν
- modal μ-calculus as extension of HML resp. CTL

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

$\nu X[p \wedge [\,][\,]X]$ corresponds to “every 2nd step p holds”

$\nu X[p \wedge \langle\,\rangle \mu Y[(f \wedge X) \vee (p \wedge \langle\,\rangle Y)]] \equiv \nu X[p \wedge \mathbf{EXE}[p\,\mathbf{U}\,f \wedge X]] \equiv \mathbf{EG}p$ under fairness f

μ-Calculus Semantics

again over Kripke structures $K = (S, I, T, \mathcal{L})$.

Definition an assignment ρ of variables V is a mapping $\rho : V \to \mathbb{P}(S)$

Definition semantics $[[f]]_\rho$ of a μ-calculus formula f is defined recursively as expansion, i.e. as set of states in which f holds for a given assignment ρ:

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

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

Definition $K \models f$ iff $I \subseteq [[f]]_\rho$ for all assignments ρ

Proposition μ-calculus subsumes CTL and at least theoretically also LTL.

Practical Temporal Logics for Hardware

- Property Specification Language (PSL)
  - subsumes CTL, LTL and also regular expressions
  - Verilog and VHDL flavor
- System Verilog Assertions (SVA)
  - less general than PSL
  - closer to Hardware
  - part of System Verilog (extension of Verilog)
- verification tools (testing / formal) often come with their own temporal logic