

# Formal Models

#342.251

SS 2014

Johannes Kepler University

Linz, Austria

Prof. Armin Biere

Institute for Formal Models and Verification

<http://fmv.jku.at/fm>

Version 2014.1

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

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

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

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

**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:

$$S = S_1 \times S_2$$

$$I = I_1 \times I_2$$

$$\Sigma = \Sigma_1 = \Sigma_2$$

$$F = F_1 \times F_2$$

$$T((s_1, s_2), a, (s'_1, s'_2)) \quad \text{iff} \quad T_1(s_1, a, s'_1) \quad \text{and} \quad T_2(s_2, a, s'_2)$$

**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)

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

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

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

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

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

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

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

$$I = \{I_1\}$$

$$\Sigma = \Sigma_1$$

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

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

**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”)

**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  $F = S \setminus F_1$ .

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

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

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

**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  $T_1(s, a, t)$
- $F = F_1$



**Proposition**  $\pi_1(L(\text{Oracle}(A_1))) = L(A_1)$  ( $\pi_1$  projection on first component)

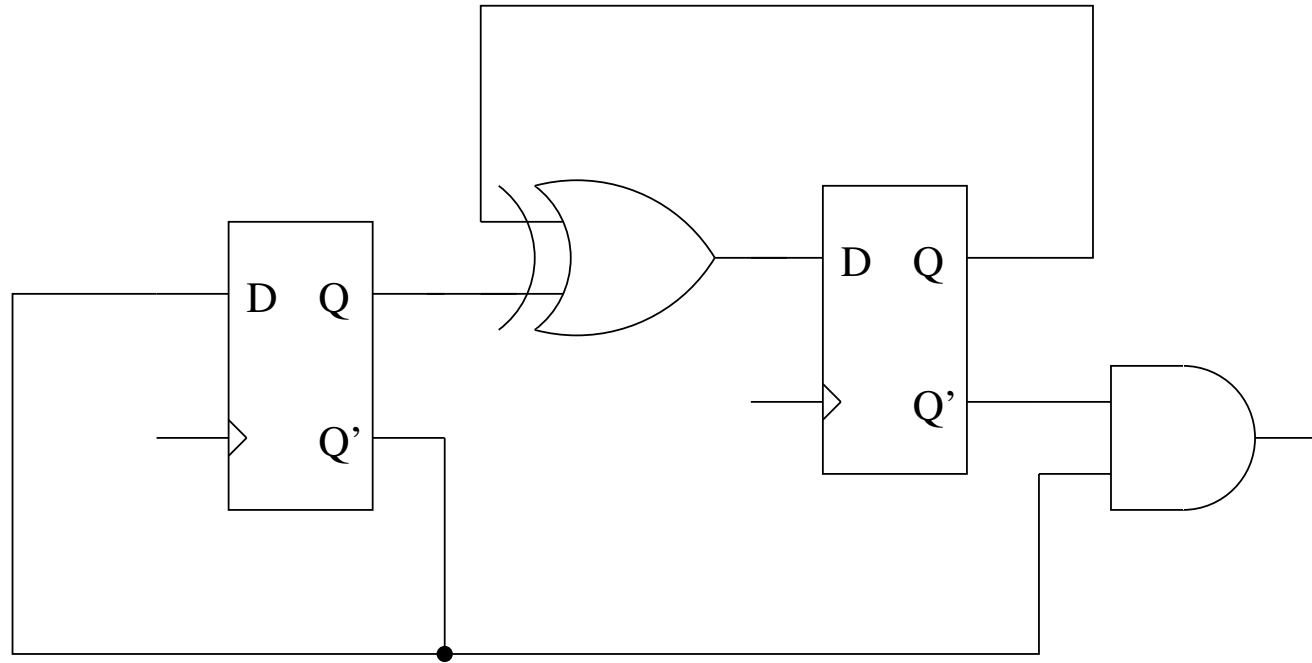
**Proposition**  $\text{Oracle}(A_1)$  is deterministic iff  $|I_1| \leq 1$ .

**Proposition**  $\text{Oracle}(A_1)$  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  $\{0, \dots, n-1\}$  is added to  $\Sigma_1$  instead of  $S_1$ , where  $n$  is the maximum number of successors:  $n = \max_{s \in S, a \in \Sigma} |s \xrightarrow{a}|$ .

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

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



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  $\Sigma$ ,
- a transition **function**  $T: S \times \Sigma \rightarrow S$
- an output alphabet  $\Theta$ , with
- output function  $O: S \times \Sigma \rightarrow \Theta$  (Moore machine:  $O: S \rightarrow \Theta$ )

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

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

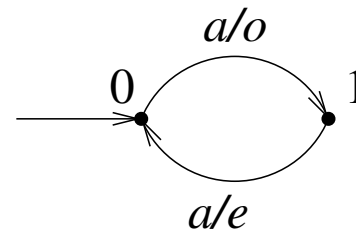
$$s = T(s, \varepsilon) \quad \text{and} \quad s' = T(s, a \cdot w) \Leftrightarrow \exists s'' [s'' = T(s, a) \wedge s' = T(s'', w)].$$

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

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

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

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



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

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

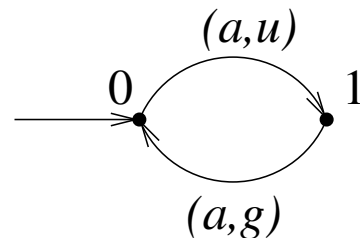
given an I/O-automaton  $A = (S, i, \Sigma, T, \Theta, O)$ .

**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)

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

- 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 ( $\Rightarrow$  model checking course)

- 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 ‘|’)

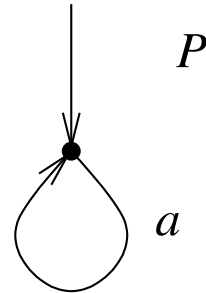
- semantics

- we are only interested in potential sequences = event streams



graphical representation

$$P = a.P$$



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

equation

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

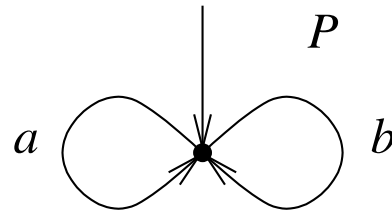
‘.’ operator means sequential composition

graphical representation

 $R_{+}^1$ 

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

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

 $R_{+}^2$ 

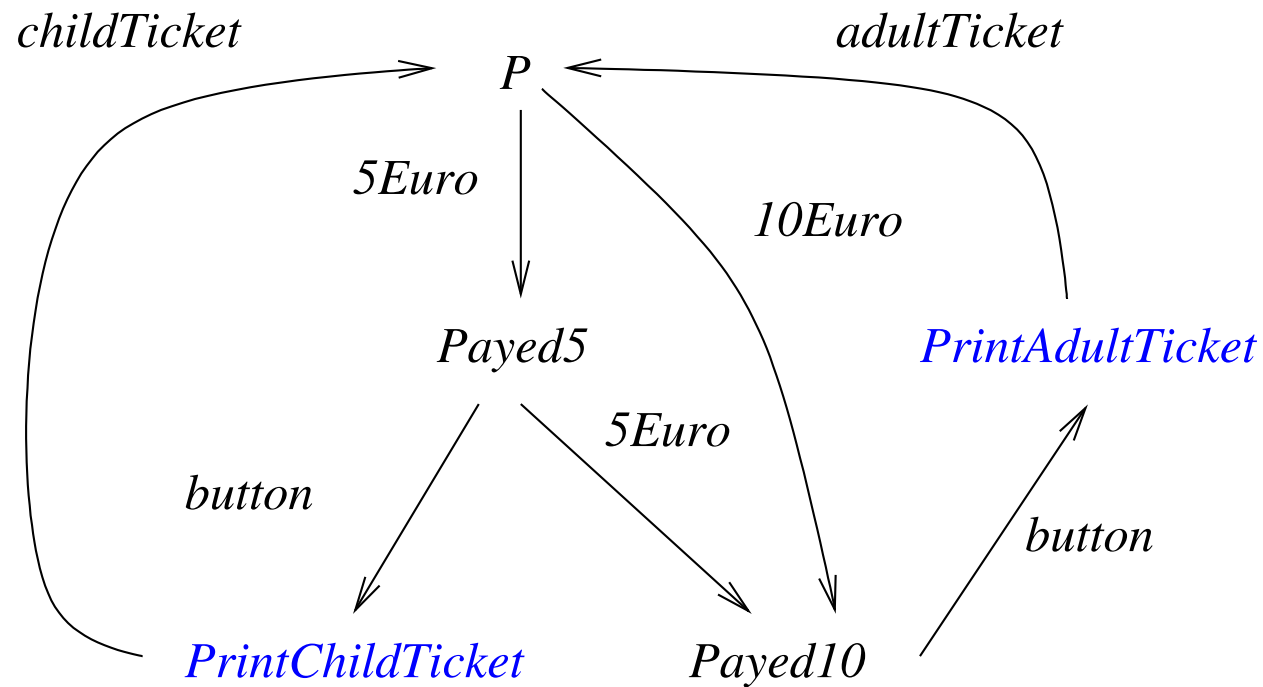
$$\frac{Q \xrightarrow{a} Q'}{(P + Q) \xrightarrow{a} Q'}$$

equation

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

'+' operator means non-deterministic choice

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



- LTS as **operational semantics** of PAE
- almost the same as an automaton, but ...
  - 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  $\Sigma$
  - transition relation  $T \subseteq S \times \Sigma \times S$  defined through operational semantics
  - initial states  $I \subseteq S$

- divergent self-cycles
  - $P = a.P + P$  is an **invalid** PAE
  - there are no  $\varepsilon$ -transitions in contrast to FAs  
(actions “need time”,  $\varepsilon$  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

- actions and states can be **parameterized**
  - which also gives rise to parameterized equations
- previous example with  $x \in \{5, 10\}$ :

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

- it is possible to operate on data as well:

$$\text{Payed}(x) = \text{euro}(y).\text{Payed}(x + y) + \text{button.ticket}(x).P$$

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

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

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

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

$Payed(X) = euro(Y).Payed(X + Y) + button.Print(X)$

$Print(X) = \text{if } (X = 5) \text{ then } childTicket.P + \text{if } (X = 10) \text{ then } adultTicket.P$

## synchronization through rendezvous in CSP

$$\Theta \subseteq \Sigma$$

$$R_{\parallel\Theta} \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \parallel_{\Theta} Q \xrightarrow{a} P' \parallel_{\Theta} Q'} \quad a \in \Theta \quad \text{rendezvous}$$

$$R_{\parallel\Theta}^1 \quad \frac{P \xrightarrow{a} P'}{P \parallel_{\Theta} Q \xrightarrow{a} P' \parallel_{\Theta} Q} \quad a \notin \Theta \quad \text{interleaving}$$

$$R_{\parallel\Theta}^2 \quad \frac{Q \xrightarrow{a} Q'}{P \parallel_{\Theta} Q \xrightarrow{a} P \parallel_{\Theta} Q'} \quad a \notin \Theta \quad \text{interleaving}$$

rendezvous does not distinguish sender and receiver

$$R_{\parallel} \quad \frac{P \parallel_{\Theta} Q \xrightarrow{a} P' \parallel_{\Theta} Q'}{P \parallel Q \xrightarrow{a} P' \parallel Q'} \quad \Theta = \Sigma(P) \cap \Sigma(Q)$$

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



**Proposition**  $\parallel$  is commutative:  $P \parallel Q \xrightarrow{a} P' \parallel Q'$  iff  $Q \parallel P \xrightarrow{a} Q' \parallel P'$

proof follows directly from the rules

**Proposition**  $\parallel$  is associative

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

To show:  $P \xrightarrow{a} P' \Leftrightarrow Q \xrightarrow{a} Q'$

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

intuition:

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

- “parenthesis” around  $\parallel$  can be omitted:

$P \parallel (Q \parallel R)$  verhält sich wie  $(P \parallel Q) \parallel R$  verhält sich wie  $P \parallel Q \parallel R$

- order is irrelevant:

$P \parallel Q \parallel R$  verhält sich wie  $P \parallel R \parallel Q$  verhält sich wie  $Q \parallel P \parallel R$  etc.

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

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

- hiding resp. abstraction of internal, **unobservable** actions
- abstracted to “silent” action  $\tau$ 
  - assumption:  $\tau \notin \Sigma$ 
    - \* formally consider only  $\Sigma \dot{\cup} \{\tau\}$  as actions
    - \* it is not possible to synchronize on  $\tau$
  - $\tau$  still needs time

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

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

- typical usage of internal synchronization  $R = ((\parallel_{i=1}^n Q_i) \setminus \{x_1, \dots, x_n\})$

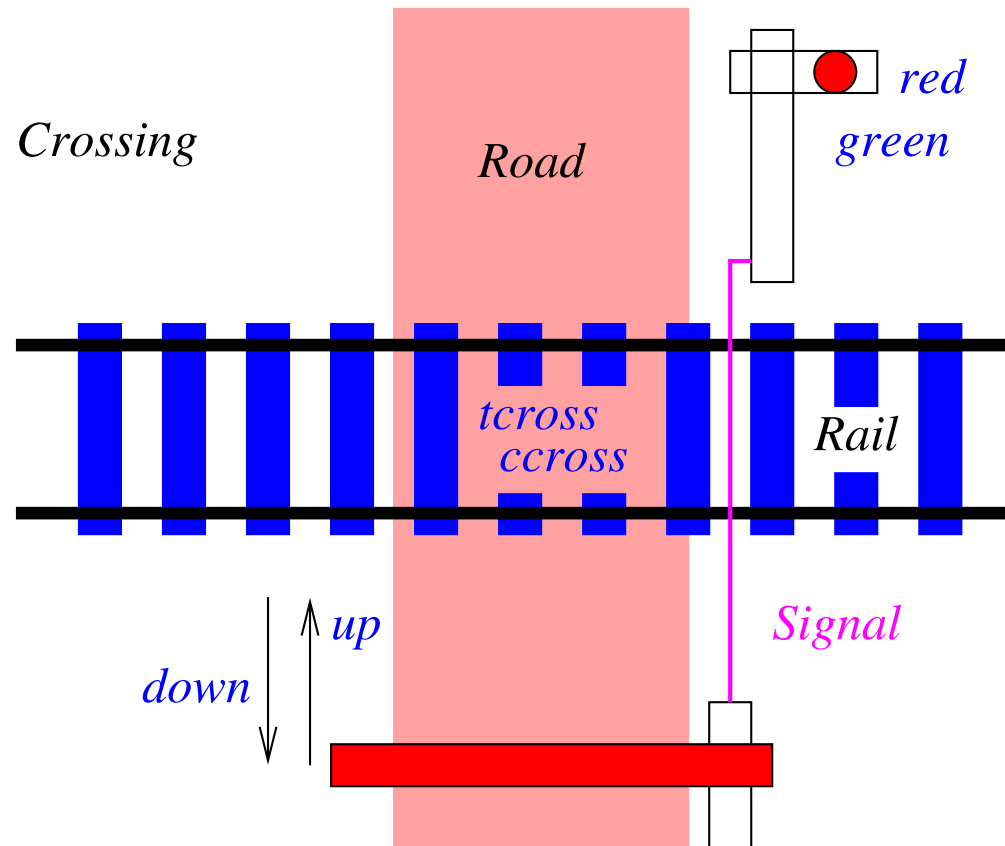
[BradfieldStirling]

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

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

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

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



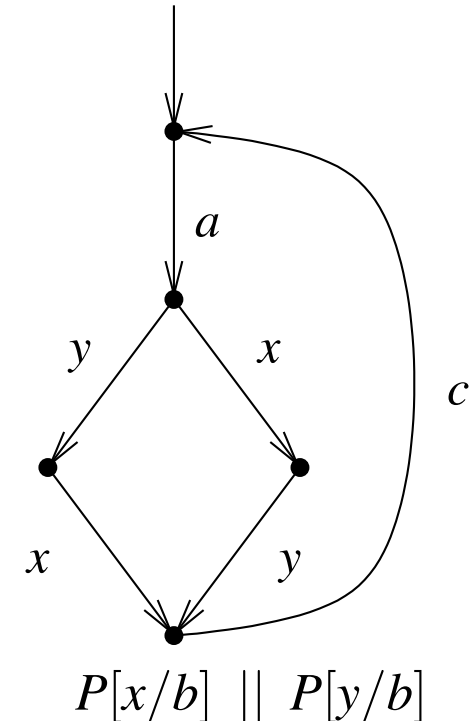
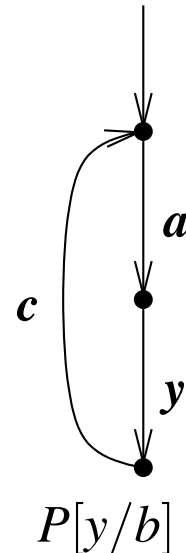
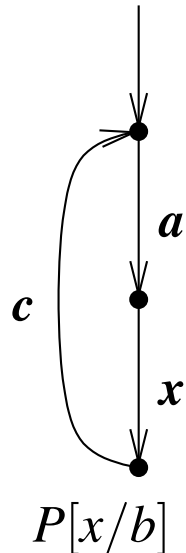
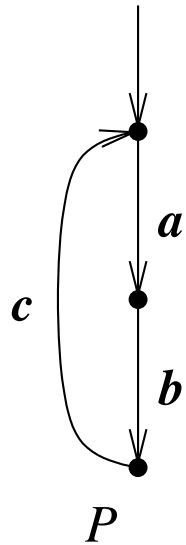
## Linking as substitution of actions

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

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

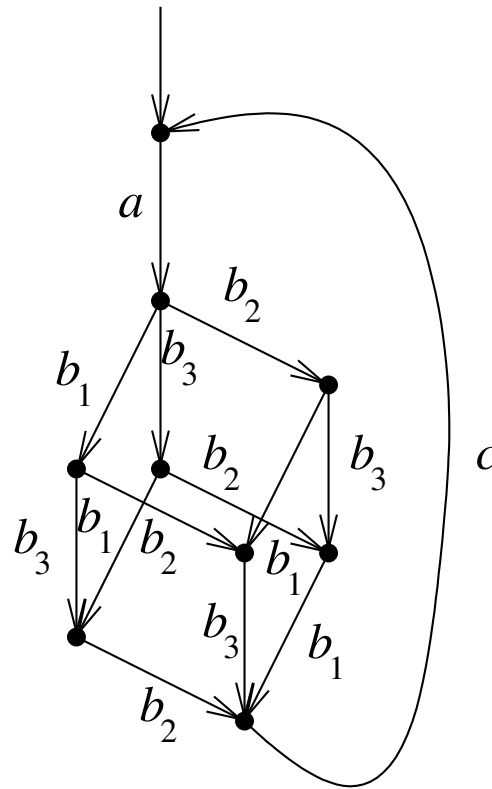
needed to “link” processes or instantiate templates:

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



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

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



- classical example of process algebra
  - modeling of a round robin scheduler
- scheduling of  $n$  processes  $\parallel P_i$  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  $P_1, P_2, \dots$
  - nothing is about execution order of the  $b_i!$

- 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  $P_i$ , ...
- ... hands over  $x_i$  control to next  $Q'_{i+1}$  ...
- and then waits to get control  $x_{i-1}$  from previous  $Q'_{i-1}$  in ring

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

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

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

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



- incorrect solution does **not** accept the legal sequence:

- ending  $P_2$  before  $P_1$ :  $a_1 a_2 b_2 b_1 \dots$

- decouple ending ( $b$ ) and accepting control ( $y$ )

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

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

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

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

- 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)

- actions:  $\Sigma \dot{\cup} \bar{\Sigma} \dot{\cup} \{\tau\}$  overlined actions are outputs, otherwise inputs
- different hiding principle (new syntax: double instead of single backslash)

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

- pairwise **explicit** synchronization

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

$$R_{\parallel\parallel}^1 \frac{P \xrightarrow{a} P'}{P \parallel\parallel Q \xrightarrow{a} P' \parallel\parallel Q}$$

$$R_{\parallel\parallel}^2 \frac{Q \xrightarrow{a} Q'}{P \parallel\parallel Q \xrightarrow{a} P \parallel\parallel Q'}$$

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

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

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

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

resp. in CCS

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

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

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

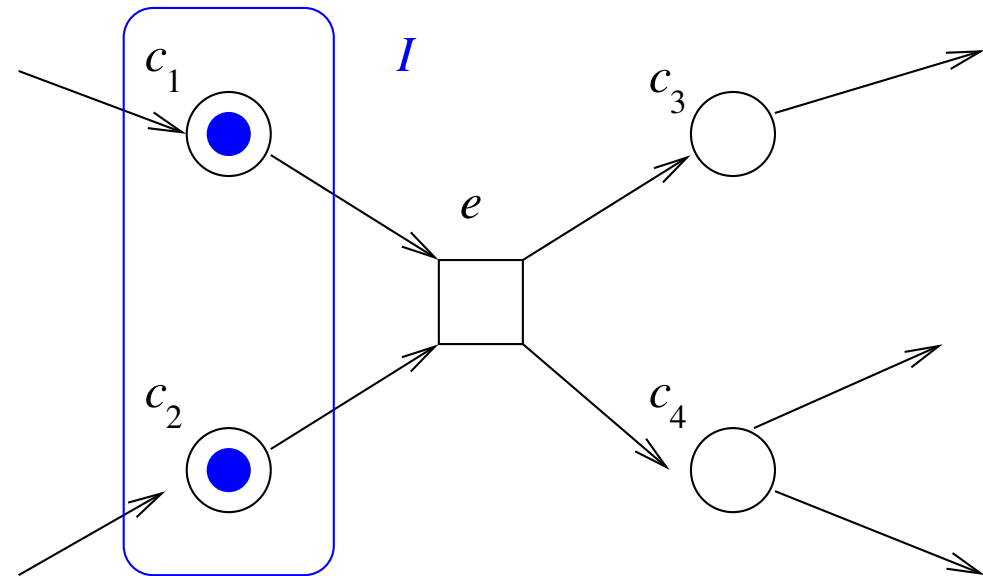
$$Crossing = (Road ||| Rail ||| Signal) \setminus \setminus \{green, red, up, down\}$$

- originally CSP had channels with data
  - inputs:  $channel ? data_{in}$ , outputs:  $channel ! data_{out}$
- $\pi$ -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

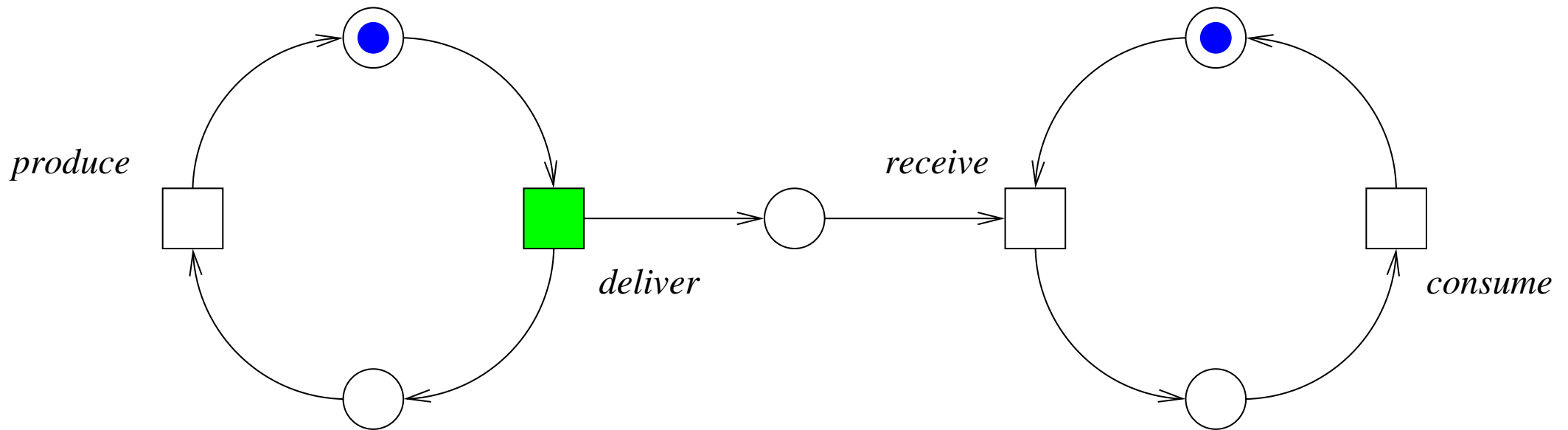
- beside process algebra the most common modeling language for *distributed* systems
  - investigated since 60ies, now also known as **activity diagrams** in UML
  - again: **asynchronously** communicating processes (protocols / SW)
- modeling and verification tools available
- **theory:** many interesting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

## Definition

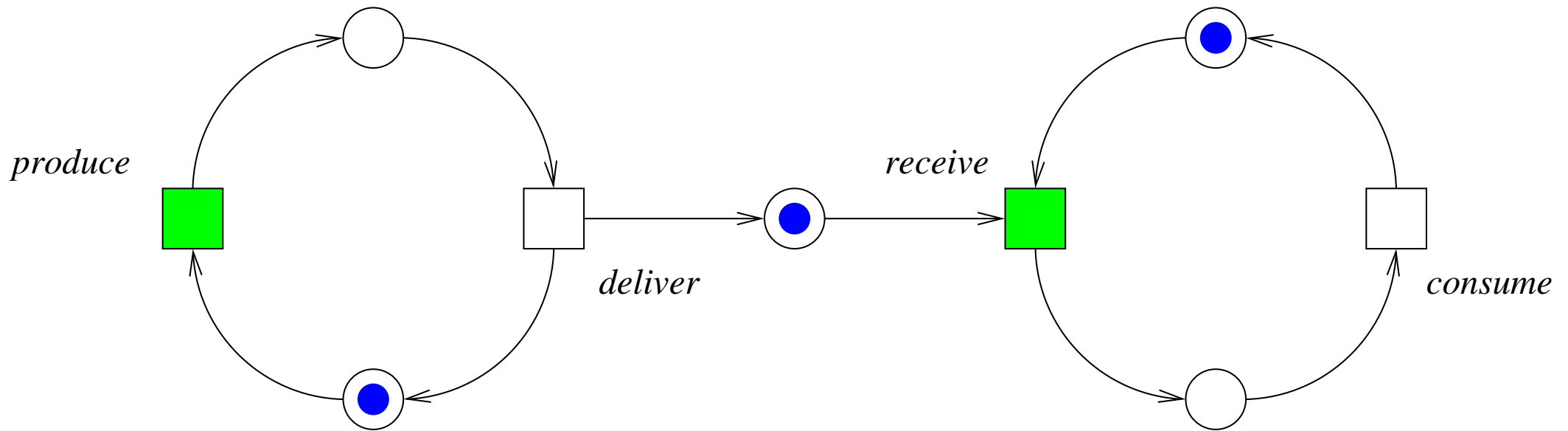
A CEN  $N = (C, I, E, G)$  is made of conditions  $C$ , an initial marking  $I \subseteq C$ , events  $E$  and a dependence graph  $G \subseteq (C \times E) \dot{\cup} (E \times C)$



- we also use  $\rightarrow$  instead of  $G$
- can be interpreted as *bipartite* graph oder ...
- ... hyper graph with multiple source resp. target edges  $E$

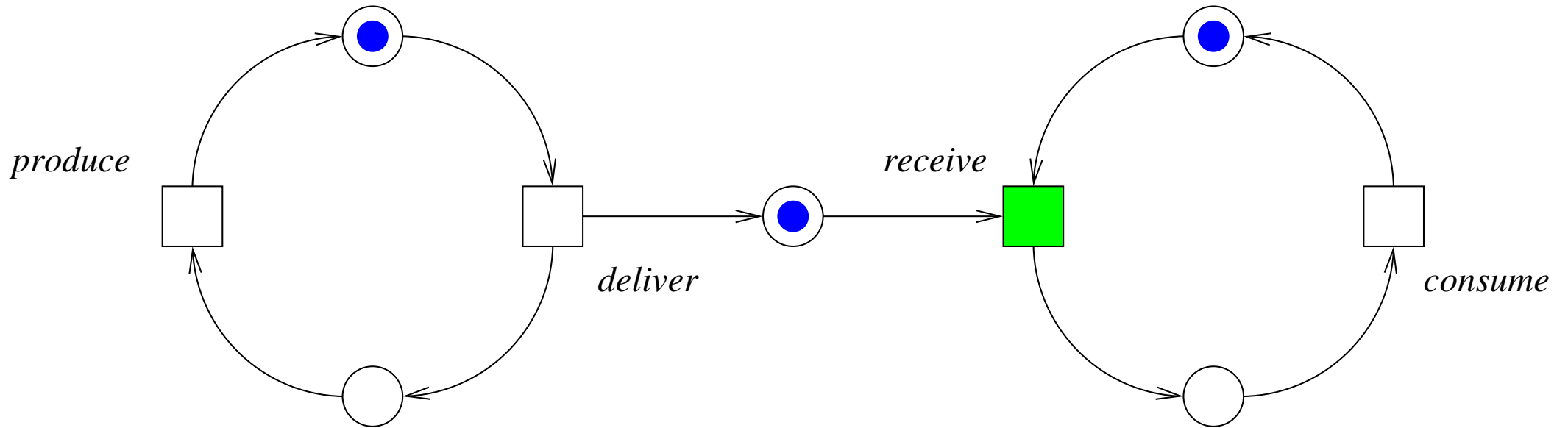


only one event / transition can **fire**

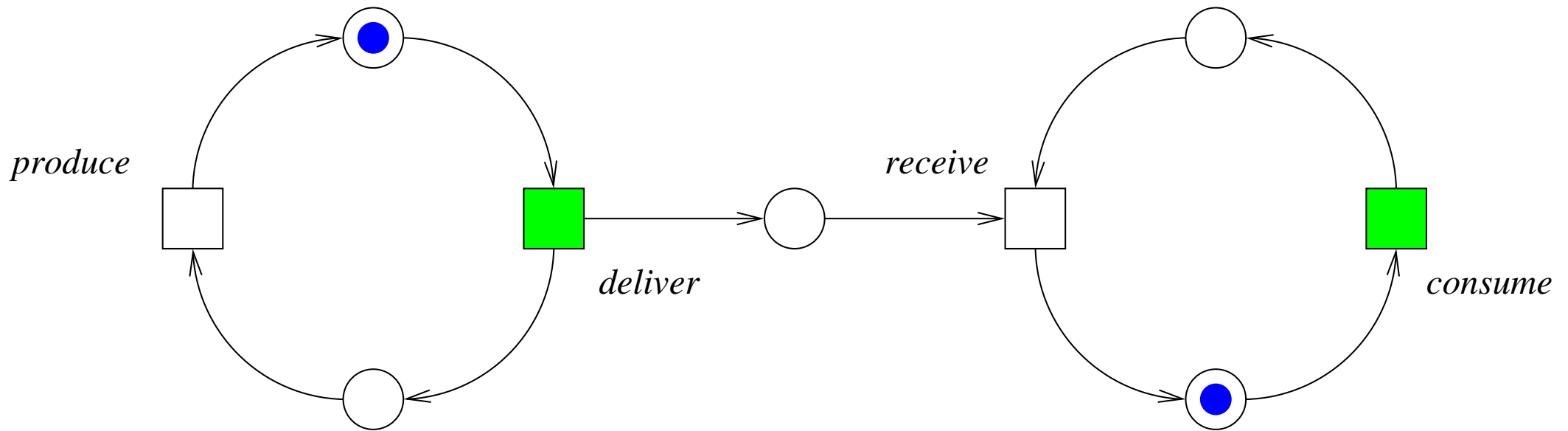


two events / transitions can fire





target condition of *deliver* occupied



again choice of two possible **events**

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

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

$$T(C_1, e, C_2) \text{ iff } G^{-1}(e) \subseteq C_1 \quad \text{pre-conditions satisfied} \quad (1)$$

$$G(e) \cap C_1 = \emptyset \quad \text{post-conditions satisfied} \quad (2)$$

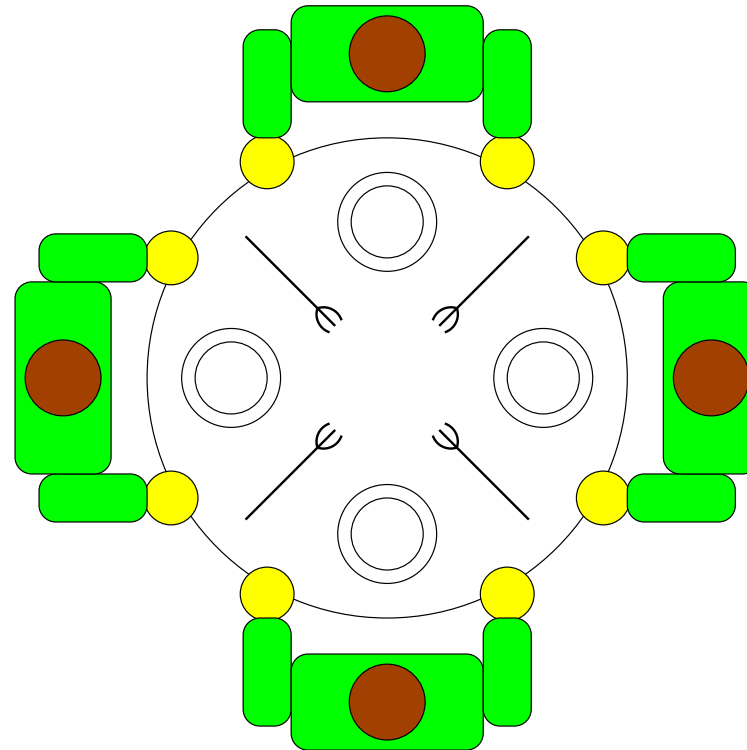
$$C_2 = (C_1 \setminus G^{-1}(e)) \cup G(e) \quad \text{state update}$$

$$G(e) = \text{post-conditions of event } e \quad (\text{or } e \rightarrow)$$

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

- states  $M \in \mathbb{P}(C)$  of the LTS are also called **markings** of the CEN
- event  $e$  is **enabled** in  $M$  iff  $M \xrightarrow{e} \neq \emptyset$
- 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
  - $\forall e \in E[M \xrightarrow{e} = \emptyset]$
- a CEN has a deadlock iff a deadlock is reachable

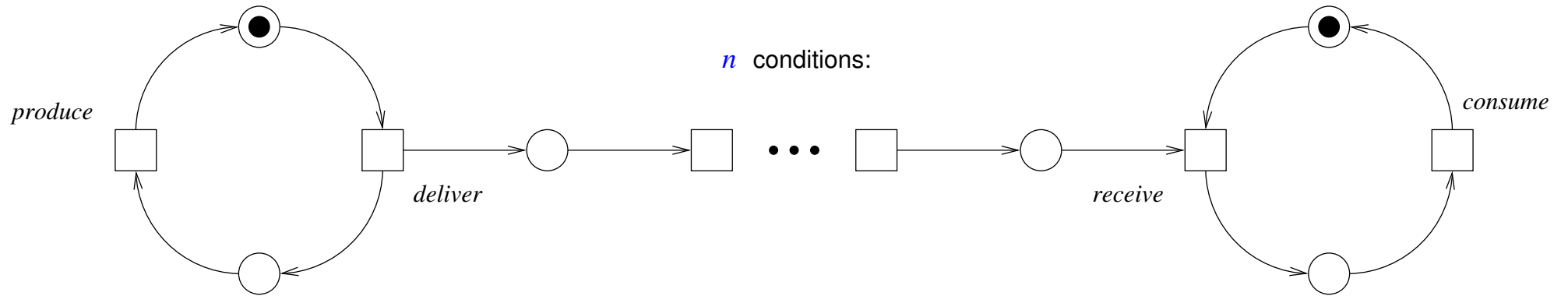
$n$  philosophers,  $n$  forks,  $n$  plates



philosophers alternate in thinking and eating

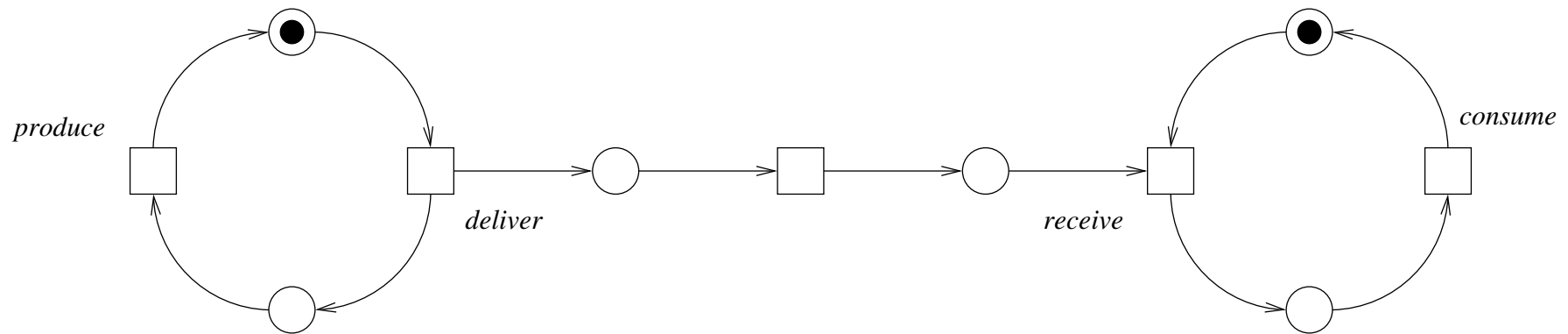
they need to pick up and use two forks to eat

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



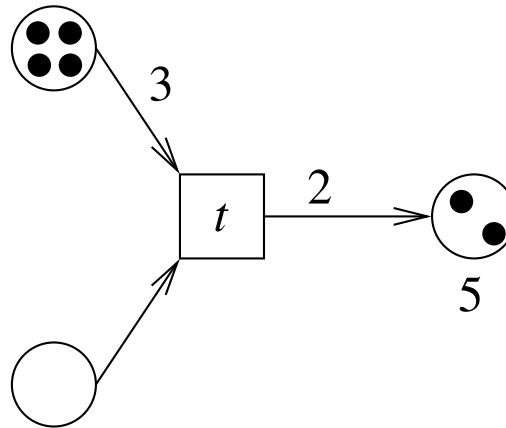
$n$  conditions:

buffer capacity  $n$



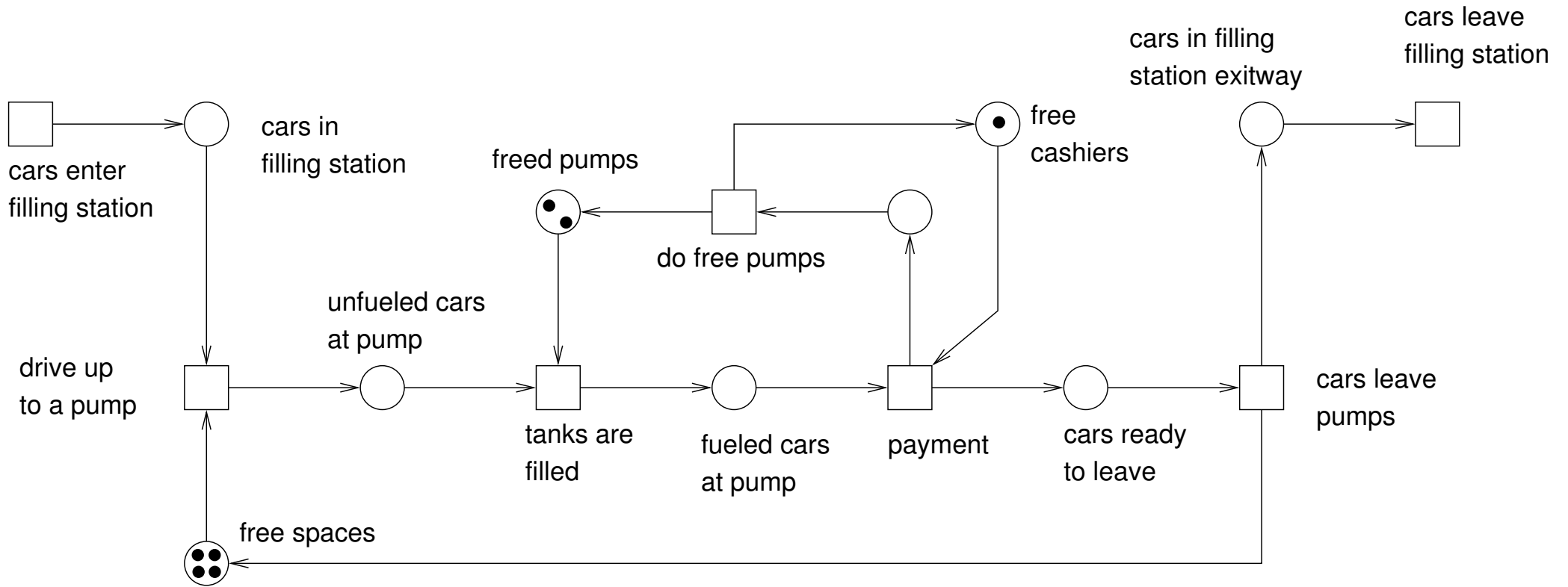
buffer capacity 2

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



- capacity of a *connection* is finite and is one if not specified explicitly
- capacity of a *place* can be  $\infty$  and is  $\infty$  if not specified explicitly
- CEN can be interpreted as PTN with constant capacity  $C \equiv 1$

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





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

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

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

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

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

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

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

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

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

- 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

## HML is an example for temporal logic over LTS

let  $\Sigma$  be the alphabet of actions

**Definition** **syntax** consists of the usual boolean constants  $\{0, 1\}$ , boolean operators  $\{\wedge, \neg, \rightarrow, \dots\}$  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  $\bigwedge_{a \in \Theta} [a]f$

$\Theta$  can also be written as a boolean expression over  $\Sigma$

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

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

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.

$$s \models 1$$

$$s \not\models 0$$

$$s \models [\Theta]g \quad \text{iff} \quad \forall a \in \Theta \forall t \in S: \quad \text{if } s \xrightarrow{a} t \text{ then } t \models g$$

$$s \models \langle \Theta \rangle g \quad \text{iff} \quad \exists a \in \Theta \exists t \in S: \quad s \xrightarrow{a} t \text{ and } t \models g$$

**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.

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

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

**Definitions** A **Trace**  $\pi$  of  $L$  is a finite or infinite sequence of states

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

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

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

$|\pi|$  is the **length** of  $\pi$ , 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  $\pi$  for  $i \leq |\pi|$

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

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

first only in combination with HML

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

**unary** temporal path operators **X**, **F**, **G** and one **binary** temporal path operator **U**.

Path operators have to be prefixed with a path-quantifier **E** or **A**.

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

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

(De'Morgan for  $\mathbf{E}[\cdot \mathbf{U} \cdot]$  requires additional temporal path operator)

$\mathbf{AG} [\neg \text{safe}] 0$

it is never possible to execute unsafe actions

$\mathbf{EF} \langle \neg \text{safe} \rangle 1$

potentially an unsafe action can be executed

$\neg \mathbf{E} [\neg \langle \text{req} \rangle 1 \mathbf{U} \langle \text{ack} \rangle 1]$

there is an order of events in which *ack* becomes possible and *req* was not possible before

$\mathbf{AG} [\text{req}] \mathbf{AF} [\neg \text{ack}] 0$

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



Let  $f$  be a CTL/HML formula,  $L$  an LTS,  $\pi$  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)

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

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

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

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

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

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

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

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

- 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

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/marketing/annotation  $\mathcal{L}: S \rightarrow \mathbb{P}(\mathcal{A})$ .

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

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

**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

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

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

similar construction as the oracle automaton

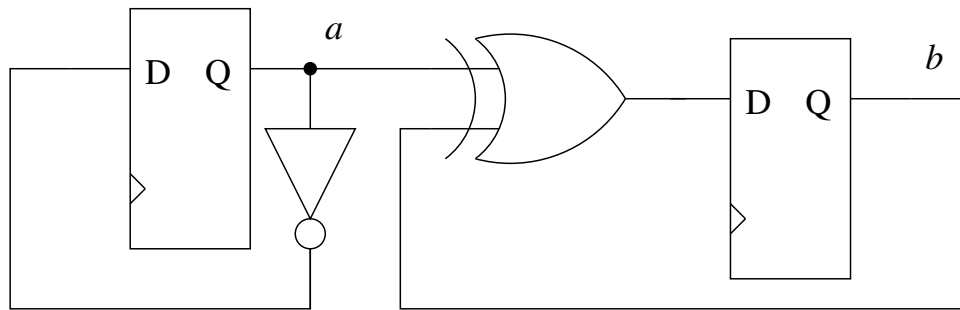
**Proposition**

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

iff

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

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



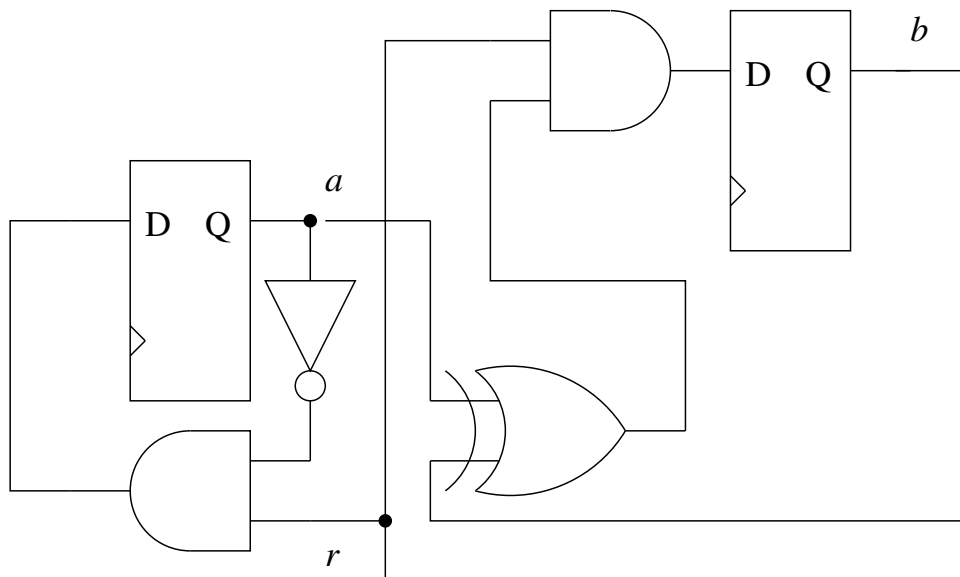
$$S = \mathbb{B}^2$$

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

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

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

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



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

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

$$T = \dots$$

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

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

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

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

## classical version of CTL on Kripke structures

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

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

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

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

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

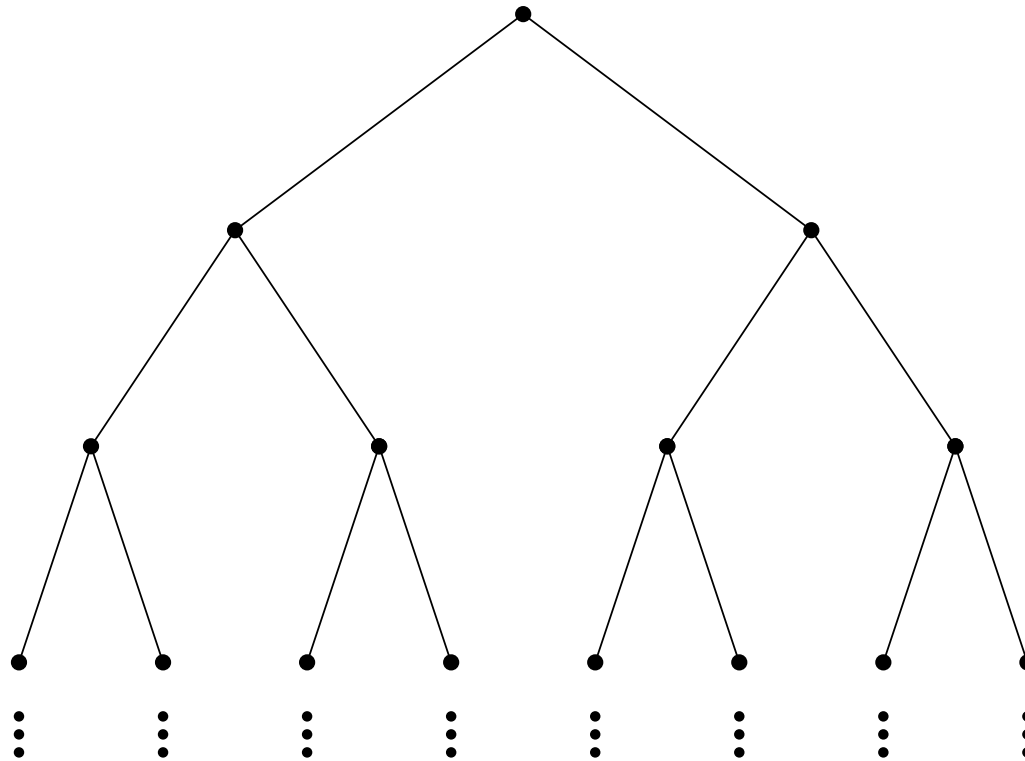
infinitely often  $\bar{a} \wedge \bar{b}$

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

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

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

generic definition

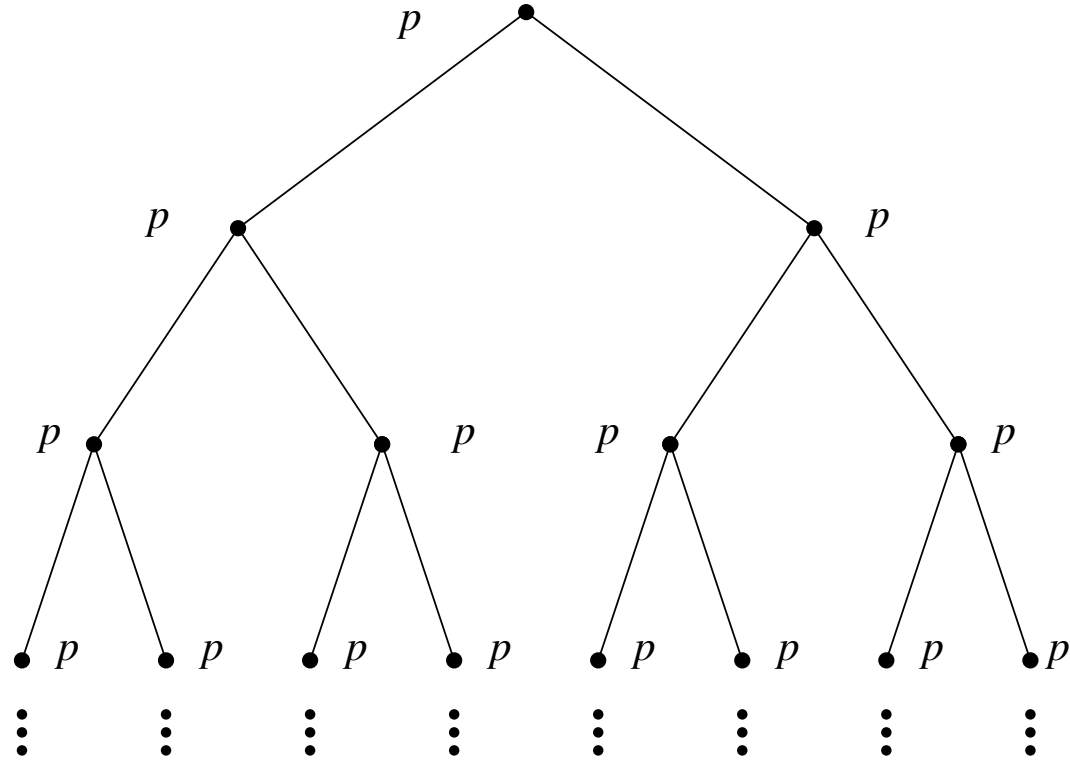


all possible orders of events are represented in one (infinite) computation tree

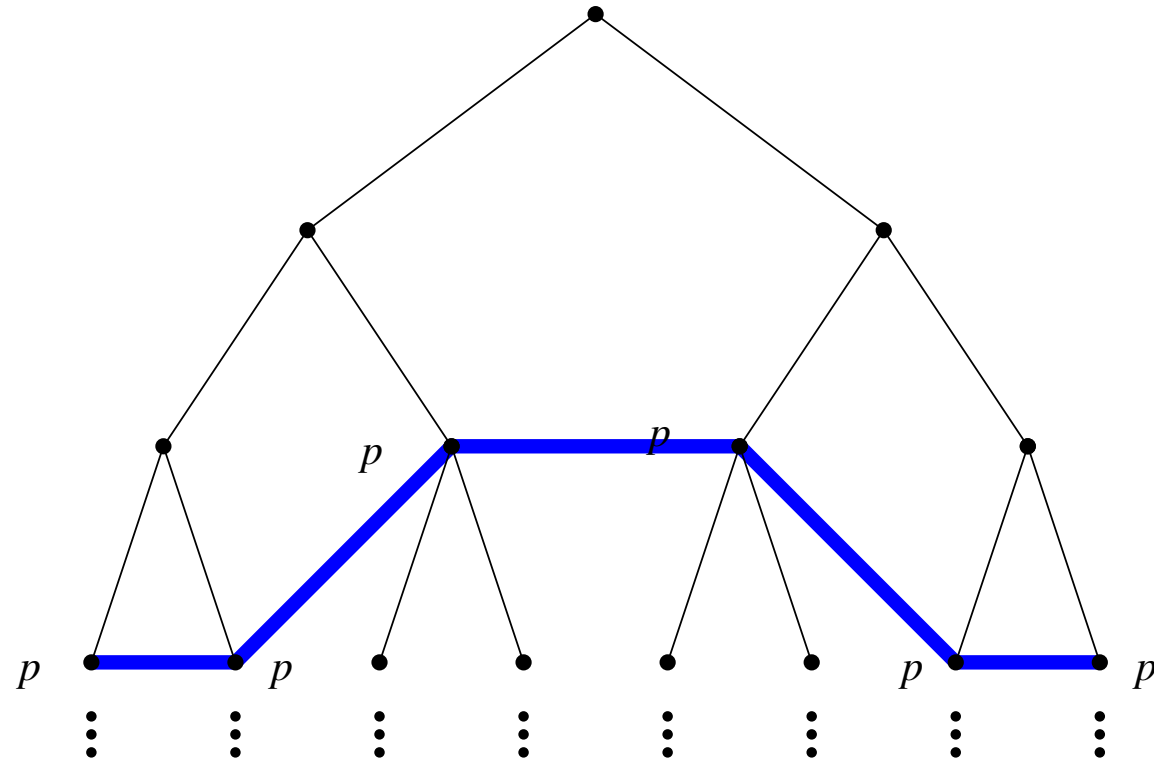
CTL describes the branching behavior of this computation tree

and has a local state view

every state is the starting point of new branching paths







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

**Definition** LTL semantics defined recursively along infinite paths  $\pi$  in  $K$ :

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

$$\pi \models \neg g \quad \text{iff} \quad \pi \not\models g$$

$$\pi \models g \wedge h \quad \text{iff} \quad \pi \models g \text{ and } \pi \models h$$

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

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

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

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

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

- 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):
  - $K_1 \models f_1, K_2 \models f_2 \Rightarrow K_1 \times K_2 \models f_1 \wedge f_2$
  - $K_1 \models f \rightarrow g, K_2 \models f \Rightarrow K_1 \times K_2 \models g$

**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

[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  $f \setminus \mathbf{A}$  as the LTL formula obtained from  $f$  by deleting all path quantifiers, e.g.  $(\mathbf{AGAF}p) \setminus \mathbf{A} = \mathbf{GF}p$ .

**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  $f \setminus \mathbf{A}$ .

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

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

[M. Maidl'00]

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

**Definition** every sub formula of an  $\text{CTL}^{\text{det}}$  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  $\text{LTL}^{\text{det}}$  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  $\text{LTL}^{\text{det}}$  resp.  $\text{CTL}^{\text{det}}$

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

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

⇒ non deterministic specifications can be misinterpreted

[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  $\pi$  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:  $\neg \left( \underbrace{(1; \dots; 1; p)^*}_{m-1}; \underbrace{1; \dots; 1}_{m-1}; \neg p \right)$  resp.  $\underbrace{(1; \dots; 1; p)^\omega}_{m-1}$

- specifications often need additional *fairness* assumptions
  - e.g. abstraction of scheduler: “each process gets its 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  $\pi$  **fair** iff  $|\{i \mid \pi(i) \in F\}| = \infty$
  - only consider fair paths

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

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

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

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



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

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

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

$$[[p]]_\rho = \{s \mid p \in \mathcal{L}(s)\}$$

$$[[X]]_\rho = \rho(X)$$

$$[[\neg f]]_\rho = S \setminus [[f]]_\rho$$

$$[[f \wedge g]]_\rho = [[f]]_\rho \cap [[g]]_\rho$$

$$\mu X[f] = \bigcap \{A \subseteq S \mid [[f]]_{\rho[X \mapsto A]} = A\}$$

$$\nu X[f] = \bigcup \{A \subseteq S \mid [[f]]_{\rho[X \mapsto A]} = A\}$$

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

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

**Proposition**  $\mu$ -calculus subsumes CTL and at least theoretically also LTL.

- 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