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 $\Sigma$ (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

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

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: $a \cdot b \cdot 1^* \cap 1^* \cdot b \cdot a$, where $1$ denotes all letters)
Completeness and Determinism

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 $s \in S$, $a \in \Sigma$. 
Sub-Set Construction

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

\[
S = \mathcal{P}(S_1) \quad (\mathcal{P} = \text{power set}) \\
I = \{I_1\} \\
\Sigma = \Sigma_1 \\
F = \{F' \subseteq S_1 \mid F' \cap F_1 \neq \emptyset\}
\]

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

**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”)

Formal Models  #342.251 SS 2020  Armin Biere   Martina Seidl  JKU Linz
**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:  “abb” | “abba” | “abacus”)

---

Formal Models  #342.251  SS 2020  Armin Biere  Martina Seidl  JKU Linz
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 \( \{0, \ldots, n - 1\} \) is added to \( \Sigma_1 \) instead of \( S_1 \), where \( n \) is the maximum number of successors: \( n = \max_{s \in S, a \in \Sigma} |s \xrightarrow{a}| \).

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

Exercise construct the oracle automaton for \( a \cdot b \cdot 1^* \cap 1^* \cdot b \cdot a \)
implementations of automata have to be deterministic

Definition \textit{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 $T : S \times \Sigma^* \to S$ as follows:

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

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

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

**Definition** the *behavior* $B : \Sigma^* \to \Theta^*$ 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

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

**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 $A' = (\mathcal{P}(S), I, \Sigma, T', \{0, 1\}, O)$ with $T'$ the transition relation of $\mathcal{P}(A)$ and $O(S', a) = 1$ iff $S' \cap F \neq \emptyset$.

**Proposition**  $w \in L(A)$  iff  $B(w \cdot x) \in 1^{\lfloor w \rfloor} \cdot 1$  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 $G$: $N = \varepsilon | aM | bM \quad M = cN | dN$\quad$\text{start symbol } N$

$\Rightarrow \quad \text{language } L(G) = ( (a \mid b)(c \mid d) )^*$ \quad (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

\[ P = a . P \]

equation

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

\[ R. \quad a . P \xrightarrow{a} P \]

‘.’ operator means sequential composition
Choice

graphical representation

\[
P = a.P + b.P
\]

equation

\[
R_1^+ \quad \frac{P \xrightarrow{a} P'}{(P + Q) \xrightarrow{a} P'}
\]

\[
R_2^+ \quad \frac{Q \xrightarrow{a} Q'}{(P + Q) \xrightarrow{a} Q'}
\]

‘+’ operator means non-deterministic choice

operational semantics rule
(here again $P,Q$ are meta variables)
\[ P = 5\text{Euro}.\text{Paid5} + 10\text{Euro}.\text{Paid10} \]
\[ \text{Paid5} = \text{button.\textit{childTicket}}.P + 5\text{Euro}.\text{Paid10} \]
\[ \text{Paid10} = \text{button.\textit{adultTicket}}.P \]
Labelled Transition Systems (LTS)

- 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 $I \subseteq S$
Syntactical Restrictions

- 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 \( x \in \{5, 10\} \):

  \[
  P = \text{euro}(x).\text{Paid}(x) \\
  \text{Paid}(5) = \text{button.print}(.\text{childTicket}).P + \text{euro}(5).\text{Paid}(10) \\
  \text{Paid}(10) = \text{button.print}(.\text{adultTicket}).P
  \]

• it is possible to operate on data as well:

  \[
  \text{Paid}(x) = \text{euro}(y).\text{Paid}(x+y) + \text{button.ticket}(x).P
  \]

  – actually allows modeling of *infinite systems*

  – and turns PA into a real programming language
(and similar rules for if-then alone)

\[
\begin{align*}
    R_{\text{then}} & \quad \frac{P \rightarrow P'}{\text{if } B \text{ then } P \text{ else } Q \rightarrow P'} \quad B \\
    R_{\text{else}} & \quad \frac{Q \rightarrow Q'}{\text{if } B \text{ then } P \text{ else } Q \rightarrow Q'} \quad \neg B
\end{align*}
\]

\[
\begin{align*}
    \text{Paid}(X) &= \text{euro}(Y).\text{Paid}(X + Y) + \text{button}.\text{Print}(X) \\
    \text{Print}(X) &= \text{if } (X = 5) \text{ then } \text{childTicket}.P + \text{if } (X = 10) \text{ then } \text{adultTicket}.P
\end{align*}
\]
Parallel-Operator

synchronization through rendezvous in CSP

\[ \Theta \subseteq \Sigma \]

\[
\begin{align*}
R_{||\Theta} & \quad P \overset{a}{\to} P' \quad Q \overset{a}{\to} Q' \\
\frac{P \parallel \Theta}{} & \quad \frac{Q \parallel \Theta}{} \\
\text{a } & \in \Theta \\
\text{rendezvous}
\end{align*}
\]

\[
\begin{align*}
R_{1||\Theta} & \quad P \overset{a}{\to} P' \\
\frac{P \parallel \Theta}{} & \quad \frac{Q \parallel \Theta}{} \\
\text{a } & \notin \Theta \\
\text{interleaving}
\end{align*}
\]

\[
\begin{align*}
R_{2||\Theta} & \quad Q \overset{a}{\to} Q' \\
\frac{P \parallel \Theta}{} & \quad \frac{Q \parallel \Theta}{} \\
\text{a } & \notin \Theta \\
\text{interleaving}
\end{align*}
\]

rendezvous does not distinguish sender and receiver

\[
\begin{align*}
R_{||} & \quad P \parallel \Theta \overset{a}{\to} P' \parallel \Theta \overset{a}{\to} Q' \\
\frac{P \parallel Q}{} & \quad \frac{Q \parallel Q'}{} \\
\Theta & = \Sigma(P) \cap \Sigma(Q)
\end{align*}
\]

\[ \Sigma(P) \text{ is the subset of actions of } \Sigma \text{ which occur in } P \text{ syntactically} \]
**Proposition** \( \| \) is commutative: \( P \| Q \xrightarrow{a} P' \| Q' \iff Q \| P \xrightarrow{a} Q' \| P' \)

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: \( P \xrightarrow{a} P' \iff Q \xrightarrow{a} Q' \)

8 cases of \( a \in \Sigma(P_i) \) resp. \( a \not\in \Sigma(P_i) \) for each direction

intuition:

1. \( a \in \Sigma(P_i) \Rightarrow P_i \xrightarrow{a} P'_i \)
2. \( P_i \) with \( a \not\in \Sigma(P_i) \) 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:

\[
P \parallel (Q \parallel R) \text{ behaves like } (P \parallel Q) \parallel R \text{ behaves like } P \parallel Q \parallel R
\]

• order is irrelevant:

\[
P \parallel Q \parallel R \text{ behaves like } P \parallel R \parallel Q \text{ behaves like } Q \parallel P \parallel R \text{ etc.}
\]

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

\[
\frac{\forall P_i, a \in \Sigma(P_i) \quad P_i \xrightarrow{a} P'_i \quad \forall P_i, a \notin \Sigma(P_i) \quad P'_i = P_i \quad \exists P_i \quad P_i \xrightarrow{a} P'_i}{P_i \xrightarrow{a} P'_i}
\]
• hiding resp. abstraction of internal, unobservable actions

• abstracted to “silent” action $\tau$
  
  - assumption: $\tau \not\in \Sigma$

  * formally consider only $\Sigma \cup \{\tau\}$ as actions

  * it is not possible to synchronize on $\tau$

  - $\tau$ still needs time

  \[
  \begin{align*}
  R \notin & \quad P \xrightarrow{a} Q \\
  \quad & \quad P \setminus \Theta \xrightarrow{a} Q \setminus \Theta \\
  \quad & \quad a \not\in \Theta
  \end{align*}
  \]

  \[
  \begin{align*}
  R \in & \quad P \xrightarrow{a} Q \\
  \quad & \quad P \setminus \Theta \xrightarrow{\tau} Q \setminus \Theta \\
  \quad & \quad a \in \Theta
  \end{align*}
  \]

• typical usage of internal synchronization

$R = (||_{i=1}^{n} Q_i) \setminus \{x_1, \ldots, x_n\}$
Railroad Crossing

[BradfieldStirling]

\[
\begin{align*}
\text{Road} & = \text{car.up.ccross.down.Road} \\
\text{Rail} & = \text{train.green.tcross.red.Rail} \\
\text{Signal} & = \text{green.red.Signal + up.down.Signal} \\
\text{Crossing} & = (\text{Road} \ | | \ \text{Rail} \ | | \ \text{Signal}) \setminus \{\text{green, red, up, down}\}
\end{align*}
\]
**Linking** as substitution of actions

\[ \begin{array}{c}
\text{R[ ]} \\
\frac{P \xrightarrow{a} Q}{P[b/a] \xrightarrow{b} Q[b/a]}
\end{array} \] 

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

needed to “link” processes or instantiate templates:

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

![Diagram of processes and links](image-url)
\[ P = a.b.c.P \]

\[ \| P[b_i/b] \]

\[ i=1 \]
Milner’s Scheduler

• 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 $P_1, P_2, \ldots$
  – no restriction on the 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 $P_i$, …

• … 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

\[
Q' = a.x.b.y.Q' \\
Q'_1 = Q'[a_1/a, x_1/x, b_1/b, x_n/y] \\
Q'_i = (y.Q')[a_i/a, x_i/x, b_i/b, x_{i-1}/y] \quad i \in \{2, \ldots, n\} \\
R' = \bigparallel_{i=1}^{n} Q'_i
\]
Correct Solution for Milner’s Scheduler

- incorrect solution does **not** accept the legal sequence:
  - ending $P_2$ before $P_1$: $a_1a_2b_2b_1 \ldots$

- decouple ending $(b)$ and accepting control $(y)$
  
  \[
  Q = a.x. (b.y + y.b) . Q \\
  Q_1 = Q[a_1/a, x_1/x, b_1/b, x_n/y] \\
  Q_i = (y.Q)[a_i/a, x_i/x, b_i/b, x_{i-1}/y] \quad i \in \{2, \ldots, n\} \\
  R = \bigparallel_{i=1}^{n} Q_i
  \]

- implemented by non blocking waiting on two different messages
  - in programming languages: try-locking, multiple threads, select (java.nio), \ldots

- slightly sloppy alternative notation $b.y + y.b = b \parallel y$ (we do not have a nil process)
Differences in CCS

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

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

\[
\begin{align*}
R \| | \quad & \quad P \xrightarrow{a} Q \quad \Rightarrow \quad P \| | \Theta \xrightarrow{a} Q \| | \Theta \\
\end{align*}
\]

\[
a \notin \Theta \cup \Theta
\]

- **pairwise explicit synchronization**

\[
\begin{align*}
R \| | | \quad & \quad P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q' \\
\end{align*}
\]

\[
a \in \Sigma \cup \Sigma
\]

\[
\begin{align*}
R_1 \| | | \quad & \quad P \xrightarrow{a} P' \\
\end{align*}
\]

\[
\begin{align*}
R_2 \| | | \quad & \quad Q \xrightarrow{a} Q' \\
\end{align*}
\]

\[
\begin{align*}
P \| | Q \xrightarrow{a} P' \| | Q
\end{align*}
\]
Comparison of CSP and CCS on Train Collision Example

\[
\begin{align*}
\text{Road} &= \text{car.up.ccross.down.Road} \\
\text{Rail} &= \text{train.green.tcross.red.Rail} \\
\text{Signal} &= \text{green.red.Signal} + \text{up.down.Signal} \\
\text{Crossing} &= (\text{Road} || \text{Rail} || \text{Signal}) \setminus \{\text{green, red, up, down}\}
\end{align*}
\]

resp. in CCS

\[
\begin{align*}
\text{Road} &= \text{car.up.ccross.down.Road} \\
\text{Rail} &= \text{train.green.tcross.red.Rail} \\
\text{Signal} &= \overline{\text{green.red.Signal}} + \overline{\text{up.down.Signal}} \\
\text{Crossing} &= (\text{Road} ||| \text{Rail} ||| \text{Signal}) \setminus \{\text{green, red, up, down}\}
\end{align*}
\]
Other Variants

- originally CSP had channels with data
  - inputs: \textit{channel} \ ? \textit{datain},  \ outputs: \textit{channel} \ ! \textit{dataout}

- \(\pi\)-calculus after [MilnerParrowWalker]
  - (references to) channels / connections can be used as data as well
  - example: \(\text{TimeAnnounce} = \text{ring(caller)} \cdot \text{caller(CurrentTime)} \cdot \text{hangup}.\text{TimeAnnounce}\)

- probabilistic behavior
  - transitions have a “transition probability”

- timed process algebra
  - transitions \textit{need} (explicitly specified) time
• beside process algebra the most common modeling language for distributed systems
  – investigated since 60s, 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 $G \subseteq (C \times E) \cup (E \times C)$

- we also use $\rightarrow$ instead of $G$

- can be interpreted as bipartite graph or …

- … hyper graph with multiple source resp. target edges $E$
only one event / transition can fire
two events / transitions can fire
Producer Consumer CEN: Produced Again

produce

receive

deliver

consume

target condition of deliver occupied
Produce Consumer CEN: Consumed

Again choice of two possible events.
Definition  Let $\text{CEN } N = (C, I, E, G)$. 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
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

$n$ conditions:

produce
deliver
receive
consume

buffer capacity $n$

produce
deliver
receive
consume

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 $G \subseteq (P \times T) \cup (T \times P)$, and capacities $C: P \cup G \rightarrow \mathbb{N}_\infty$.

- 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 $C \equiv 1$
given a PTN \( N = (P, I, T, G, C) \)

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

\[
C((p, t)) \leq M(p) \quad \text{for all } p \in G^{-1}(t) \quad \text{and} \quad \\
C((t, q)) + M(q) \leq C(q) \quad \text{for all } q \in G(t).
\]

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

\( t \) can fire in \( M_1 \), and \( M_2 = M_1 - M_- + M_+ \) with

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

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

\[
S = \mathbb{N}^P \quad \Sigma = T \quad \text{and} \quad T_L(M_1, t, M_2) \quad \text{iff} \quad t \text{ leads from } M_1 \text{ to } M_2
\]
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 $\Sigma$ be the alphabet of actions

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

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

e.g. $[a \lor b] f \equiv [\{a, b\}] f$ oder $\langle \neg a \land \neg b \rangle f \equiv \langle \Sigma \backslash \{a, b\} \rangle f$
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. \(\langle a \rangle 1 \land [b]0\) \(a\) has to be possible but not \(b\)

6. \(\langle a \rangle 1 \land [\neg a]0\) \(a\) and only \(a\) should be possible

7. \([a \lor b] \langle a \lor b \rangle 1\) 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.

- $s \models 1$
- $s \not\models 0$
- $s \models [\Theta]g$ iff $\forall a \in \Theta \forall t \in S$: if $s \xrightarrow{a} t$ then $t \models g$
- $s \models \langle \Theta \rangle g$ iff $\exists a \in \Theta \exists t \in S$: $s \xrightarrow{a} t$ 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

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

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 $a_0, a_1, \ldots$ with

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

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

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

\[
\begin{align*}
EXf & \text{ in one (immediate) successor state } f \text{ holds} & \equiv \langle \Sigma \rangle f \\
AXf & \text{ in all successor states } f \text{ holds} & \equiv [\Sigma] f \\
EFf & \text{ in one future } f \text{ holds eventually} & \text{exists finally} \\
AFf & \text{ in all possible orders of events } f \text{ holds eventually} & \text{always finally} \\
EGf & \text{ in one future } f \text{ holds all the time} & \text{exists globally} \\
AGf & f \text{ holds always} & \text{always globally} \\
E[f \ U g] & \text{ potentially } f \text{ holds until finally } g \text{ gilt} & \text{exists until} \\
& \text{(note } g \text{ has to hold on this trace eventually)} \\
A[f \ U g] & f \text{ always holds until finally } g \text{ occurs} & \text{always until} \\
& \text{(note } g \text{ has to hold on all traces eventually)}
\end{align*}
\]
\[ \neg \text{EX} f \equiv \text{AX}\neg f \quad \neg \langle \Theta \rangle f \equiv [\Theta] \neg f \quad \neg \text{EF} f \equiv \text{AG}\neg f \quad \neg \text{EG} f \equiv \text{AF}\neg f \]

(De’Morgan for \( E[\cdot \cup \cdot] \) requires additional temporal path operator)

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

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)

- $s \models \text{EX} f$ iff $\exists \pi[\pi(0) = s \land \pi(1) \models f]$  
- $s \models \text{AX} f$ iff $\forall \pi[\pi(0) = s \Rightarrow \pi(1) \models f]$  
- $s \models \text{EF} f$ iff $\exists \pi[\pi(0) = s \land \exists i [i \leq |\pi| \land \pi(i) \models f]]$  
- $s \models \text{AF} f$ iff $\forall \pi[\pi(0) = s \Rightarrow \exists i [i \leq |\pi| \land \pi(i) \models f]]$  
- $s \models \text{EG} f$ iff $\exists \pi[\pi(0) = s \land \forall i [i \leq |\pi| \Rightarrow \pi(i) \models f]]$  
- $s \models \text{AG} f$ iff $\forall \pi[\pi(0) = s \Rightarrow \forall i [i \leq |\pi| \Rightarrow \pi(i) \models f]]$  
- $s \models E[f \ U g]$ iff $\exists \pi[\pi(0) = s \land \exists i [i \leq |\pi| \land \pi(i) \models g \land \forall j [j < i \Rightarrow \pi(j) \models f]]]$  
- $s \models A[f \ U g]$ iff $\forall \pi[\pi(0) = s \Rightarrow \exists i [i \leq |\pi| \land \pi(i) \models g \land \forall j [j < i \Rightarrow \pi(j) \models f]]$]
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
  – \(<\ )f\) 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 $K = (S, I, T, L)$ consists of the following components:

- set of states $S$.

- initial states $I \subseteq S$ with $I \neq \emptyset$

- a *total* transition relation $T \subseteq S \times S$ (total iff $\forall s [\exists t [T(s, t)]]$)

- labelling/marking/annotation $L : S \rightarrow \mathcal{P}(\mathcal{A})$.

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

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

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

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

similar construction as the oracle automaton

Proposition

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

\[
\text{iff}
\]

\[
(s_0, a_0) \xrightarrow{\mathcal{L}} (s_1, a_1) \cdots \xrightarrow{\mathcal{L}} (s_n, a_n) \quad \text{in } K
\]

Note  often \( S \subseteq \mathbb{B}^n \), \( \Sigma = \{a_1, \ldots, a_n\} \), and \( \mathcal{L}((s_1, \ldots, s_n)) = \{a_i \mid s_i = 1\} \)
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 $\land, \neg, \lor, \rightarrow, \ldots$ and the temporal operators $\text{EX}, \text{AX}, \text{EF}, \text{AF}, \text{EG}, \text{AG}, \text{E}[\cdot \text{U} \cdot]$ and $\text{A}[\cdot \text{U} \cdot]$.

**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**

$\text{AG}(\bar{r} \rightarrow \text{AX}(\bar{a} \land \bar{b}))$

$\text{AG EX}(\bar{a} \land \bar{b})$

$\text{AG EF}(\bar{a} \land \bar{b})$

$\text{AG AF}(\bar{a} \land \bar{b})$

$\text{AG}(\bar{a} \land \bar{b} \land r \rightarrow \text{AX A}[(a \lor b) \text{U} (\bar{a} \land \bar{b})])$

$(\text{AG r}) \rightarrow \text{AF}(a \land b)$

**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
Computation Tree $\mathcal{AG}_p$
Computation Tree $\mathcal{AF}_p$
Linear Temporal Logic (LTL)

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

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

- $\pi \models p$ iff $p \in \mathcal{L}(\pi(0))$
- $\pi \models \neg g$ iff $\pi \not\models g$
- $\pi \models g \land h$ iff $\pi \models g$ and $\pi \models h$
- $\pi \models Xg$ iff $\pi^1 \models g$
- $\pi \models Fg$ iff $\pi^i \models g$ for one $i$
- $\pi \models Gg$ iff $\pi^i \models g$ for all $i$
- $\pi \models g U h$ 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 $\pi$ in $K$ with $\pi(0) \in I$
• LTL only considers one single **linear** order of events

• then \((Gr) \rightarrow F(a \land b)\) suddenly makes sense  (premise is a restriction/assumption)

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

**Proposition**  \(\text{CTL and LTL have different expressibility:} \)

\(AXEX_p\) can not be specified in LTL, \(AFAG_p\) does not have corresponding LTL formula
ACTL Formulas as LTL Formulas

[Clarke and Draghicescu’88]

ACTL is the sub logic of CTL formulas without $E$ path quantifiers in NNF

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

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

**Definition** $f$ and $g$ are equivalent iff $K \models f \iff K \models g$ 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 $f \setminus A$.

**Proof** $K \models f$ assumption $\iff \forall \pi[\pi \models g]$ assumption $\iff \forall \pi[\pi \models f \setminus A]$ Def. $\iff K \models f \setminus A$

(assume $\pi$ 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 $p \in A$.

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

- $p$
- $f \land g$
- $\text{AX} f$
- $\text{AG} f$
- $(\neg p \land f) \lor (p \land g)$
- $A[(\neg p \land f) U (p \land g)]$

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

- $p$
- $f \land g$
- $X f$
- $\text{G} f$
- $(\neg p \land f) \lor (p \land g)$
- $(\neg p \land f) U (p \land g)$

**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**  

- $A[f U p] \equiv A[(\neg p \land f) U (p \land 1)]$
- $\text{AF} p \equiv A[1 U p]$

$\Rightarrow$ 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 \(\pi\) of \(K\) and all \(i = 0 \mod m\).

**Problem**  \(p \land G(p \leftrightarrow \neg X p)\)  denotes  “\textbf{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:  \(\neg \left( (\underbrace{1; \ldots; 1; p}_{m-1})^* ; \underbrace{1; \ldots; 1; \neg p}_{m-1} \right) \)  resp. \( (\underbrace{1; \ldots; 1; p}_{m-1})^\omega \)
• 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: \((\text{GF}f) \rightarrow \text{G}(r \rightarrow \text{Fa})\)

• fair Kripke structures for CTL:
  – additional component \(F\) of fair states
  – path \(\pi\) is fair iff \(|\{i \mid \pi(i) \in F\}| = \infty\)
  – only consider fair paths
• restricted class of quantifiers over sets of states
  – quantified variables \( V = \{X, Y, \ldots\} \)
  – 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

\[
\nu X[p \land [\ ] X] \equiv \text{AG} p \quad \mu X[q \lor (p \land \langle \rangle X)] \equiv \text{E}[p \cup q]
\]

\[
\nu X[p \land [\ ] [\ ] X] \quad \text{corresponds to} \quad \text{“every 2nd step } p \text{ holds”}
\]

\[
\nu X[p \land \langle \rangle \mu Y[(f \land X) \lor (p \land \langle \rangle Y)]] \equiv \nu X[p \land \text{EXE}[p \cup f \land X]] \equiv \text{EG} p \text{ under fairness } f
\]
again over Kripke structures $K = (S, I, T, L)$.

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

**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 $\rho$:

\[
[[p]]_\rho = \{ s \mid p \in L(s) \} \\
[[\neg f]]_\rho = S \setminus [[f]]_\rho \\
[[f \land g]]_\rho = [[f]]_\rho \cap [[g]]_\rho \\
\mu X[f] = \bigcap \{ A \subseteq S \mid [[f]]_\rho[X \mapsto A] = A \} \\
\nu X[f] = \bigcup \{ A \subseteq S \mid [[f]]_\rho[X \mapsto A] = A \}
\]

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