Formal Models

#342.251

SS 2018
Johannes Kepler University
Linz, Austria

Armin Biere  Martina Seidl
Institute for Formal Models and Verification

http://fmv.jku.at/fm

Version 2019.2
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 $F \subseteq S$
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
**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:

$$
S = S_1 \times S_2 \\
I = I_1 \times I_2 \\
\Sigma = \Sigma_1 = \Sigma_2 \\
F = F_1 \times F_2 \\
T((s_1, s_2), a, (s'_1, s'_2)) \iff T_1(s_1, a, s'_1) \text{ and } T_2(s_2, a, s'_2)
$$

**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: $a \cdot b \cdot 1^* \cap 1^* \cdot b \cdot a$, where $1$ 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* $A = \mathcal{P}(A_1)$ of an FA $A_1$ consists of the components:

$$S = \mathcal{P}(S_1) \quad (\mathcal{P} = \text{power set}) \quad I = \{I_1\}$$

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

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

**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 2019 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”)


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: \(n = \max_{s \in S_1, a \in \Sigma} |s \xrightarrow{a}|\).

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

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**  
\( I/O \)-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

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

**Proposition** \( B(w) = w' \text{ iff } (w, w') \in L(A') \)

**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 \( B(w \cdot x) \in 1^{|w|} \cdot 1 \) 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  \(\Rightarrow\) model checking course
• right linear grammar = regular language = Chomsky 3 language

\[ \text{grammar } G: \quad N = \varepsilon \mid aM \mid bM \quad M = cN \mid dN \quad \text{start symbol } N \]

\[ \Rightarrow \quad \text{language } L(G) = ((a \mid b)(c \mid d))^* \quad \text{(as regular expression)} \]

• 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)

‘.’ operator means sequential composition
Choice

graphical representation

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

\( \text{equation} \)

\( \text{operational semantics rule} \)

(here again \( P, Q \) are meta variables)

\( \text{‘+’ operator means non-deterministic choice} \)
\[ P = 5\text{Euro}.\text{Paid5} + 10\text{Euro}.\text{Paid10} \]

\[ \text{Paid5} = \text{button.\text{childTicket}.P} + 5\text{Euro}.\text{Paid10} \]

\[ \text{Paid10} = \text{button.\text{adultTicket}.P} \]
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 \( I \subseteq S \)
• 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 $x \in \{5, 10\}$:
  
  $$P = \text{euro}(x)\cdot \text{Paid}(x)$$
  
  $$\text{Paid}(5) = \text{button.print} (\text{childTicket}) \cdot P + \text{euro}(5) \cdot \text{Paid}(10)$$
  
  $$\text{Paid}(10) = \text{button.print} (\text{adultTicket}) \cdot P$$

• it is possible to operate on data as well:
  
  $$\text{Paid}(x) = \text{euro}(y) \cdot \text{Paid}(x + y) + \text{button.ticket}(x) \cdot P$$

  – actually allows modeling of *infinite systems*

  – and turns PA into a real programming language
\( R_{\text{then}} \)

\[
\frac{P \rightarrow P'}{
\begin{array}{c}
\text{if } B \text{ then } P \text{ else } Q \rightarrow P' \end{array}
\]
\[ B \]

\( R_{\text{else}} \)

\[
\frac{Q \rightarrow Q'}{
\begin{array}{c}
\text{if } B \text{ then } P \text{ else } Q \rightarrow Q' \end{array}
\]
\[ \neg B \]

(and similar rules for \textbf{if-then} alone)

\[
\text{Paid}(X) = \text{euro}(Y) \cdot \text{Paid}(X + Y) + \text{button} \cdot \text{Print}(X)
\]

\[
\text{Print}(X) = \text{if } (X = 5) \text{ then } \text{childTicket}.P + \text{if } (X = 10) \text{ then } \text{adultTicket}.P
\]
synchronization through rendezvous in CSP

\[ \Theta \subseteq \Sigma \]

\[
\begin{array}{c}
R_{||\Theta} \\
\frac{P \rightarrow^a P'}{P \parallel_{\Theta} Q \rightarrow^a P' \parallel_{\Theta} Q'} \quad a \in \Theta \quad \text{rendezvous} \\
R^1_{||\Theta} \\
\frac{P \rightarrow^a P'}{P \parallel_{\Theta} Q \rightarrow^a P' \parallel_{\Theta} Q} \quad a \not\in \Theta \quad \text{interleaving} \\
R^2_{||\Theta} \\
\frac{Q \rightarrow^a Q'}{P \parallel_{\Theta} Q \rightarrow^a P \parallel_{\Theta} Q'} \quad a \not\in \Theta \quad \text{interleaving}
\end{array}
\]

Rendezvous does not distinguish sender and receiver

\[
\Theta = \Sigma(P) \cap \Sigma(Q)
\]

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

proof follows directly from the rules

**Proposition** \( \land \) is associative

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

To show: \[ P \overset{a}{\to} P' \iff Q \overset{a}{\to} Q' \]

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 same applies for every “parallel composition” of the \( P_i \)
• “parenthesis” around $||$ can be omitted:

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

• order is irrelevant:

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

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

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

• abstracted to “silent” action \( \tau \)
  
  – assumption: \( \tau \notin \Sigma \)
  
  * formally consider only \( \Sigma \cup \{\tau\} \) as actions

  * it is not possible to synchronize on \( \tau \)

  – \( \tau \) still needs time

\[
\begin{array}{c}
R \notin \\
\Downarrow \quad P \xrightarrow{a} Q \\
\Theta \quad P \xrightarrow{a} Q \Theta \\
\end{array}
\]

\[
\begin{array}{c}
R \in \\
\Downarrow \quad P \xrightarrow{\tau} Q \\
\Theta \quad P \xrightarrow{\tau} Q \Theta \\
\end{array}
\]

• typical usage of internal synchronization

\[
R = (||i=1^n Q_i) \setminus \{x_1, \ldots, x_n\}
\]
Road Crossing

[BradfieldStirling]

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

\[
\begin{align*}
\text{R[\_]} & \quad \frac{P \xrightarrow{a} Q}{P[b/a] \xrightarrow{b} Q[b/a]} \\
\end{align*}
\]

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

needed to “link” processes or instantiate templates:

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

\[
P[x/b] \parallel P[y/b]
\]
Parameterized Linking

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

\[ \parallel \sum_{i=1}^{3} P[b_i/b] \]

Formal Models #342.251 SS 2019 Armin Biere Martina Seidl JKU Linz
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 $P_1, P_2, \ldots$
  – 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

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

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

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

$$R' = \big||_{i=1}^n Q'_i$$
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), …

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

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

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

\[
\begin{align*}
R \parallel & \quad \frac{P \xrightarrow{a} Q}{P \parallel \emptyset \xrightarrow{a} Q \parallel \emptyset} \\
& \quad a \notin \Theta \cup \overline{\Theta}
\end{align*}
\]

- pairwise **explicit** synchronization

\[
\begin{align*}
R \parallel & \parallel & \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \\
& \quad a \in \Sigma \cup \overline{\Sigma}
\end{align*}
\]

\[
\begin{align*}
R^1 \parallel & \parallel & \quad \frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q} \\
R^2 \parallel & \parallel & \quad \frac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'}
\end{align*}
\]
\textbf{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} \parallel \text{Rail} \parallel \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} &= \text{green.red.Signal} + \text{up.down.Signal} \\
\text{Crossing} &= (\text{Road} \parallel \text{Rail} \parallel \text{Signal}) \setminus \{\text{green, red, up, down}\}
\end{align*}
Other Variants

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

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

- probabilistic behavior
  - transitions have a “transition probability”

- timed process algebra
  - transitions \texttt{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 $G \subseteq (C \times E) \cup (E \times C)$.

- 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
Producer Consumer CEN: Consumed

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
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
**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 $C \equiv 1$
Filling Station

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

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

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

**Definition** transition $t \in T$ **leads from** $M_1: P \to \mathbb{N}$ 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 Σ 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 \( ⟨a⟩ \) with \( a \in Σ \).

read \( [a]f \) as for **all** \( a \)-successors of the current state \( f \) holds

read \( ⟨a⟩f \) as for **one** \( a \)-successor of the current state \( f \) holds

abbreviations \( ⟨Θ⟩f \) denotes \( \bigvee_{a ∈ Θ} ⟨a⟩f \) resp. \( [Θ]f \) for \( \bigwedge_{a ∈ Θ} [a]f \)

Θ can also be written as a boolean expression over \( Σ \)

\[
e.g. \quad [a ∨ b]f \equiv [\{a, b\}]f \quad \text{oder} \quad ⟨¬a ∧ ¬b⟩f \equiv ⟨Σ \setminus \{a, b\}⟩f
\]
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.

$[[f]] = \{ s \in S | s \models f \}$
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 $A$.

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

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

\(\text{AG} [\neg \text{safe}] 0\)  it is never possible to execute unsafe actions

\(\text{EF} \langle \neg \text{safe} \rangle 1\)  potentially an unsafe action can be executed

\(-E[\neg \langle \text{req} \rangle 1 \ U \langle \text{ack} \rangle 1]\)  there is an order of events in which \(\text{ack}\) becomes possible and \(\text{req}\) was not possible before

\(\text{AG} [\text{req}] \text{AF} [\neg \text{ack}] 0\)  always after \(\text{req}\) a point is reached, from no other action than \(\text{ack}\) 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)

- $s \models \text{EX} f$ 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 $\forall \pi[\pi(0) = s \Rightarrow \forall i[i \leq |\pi| \Rightarrow \pi(i) \models f]]$
- $s \models \text{E}[f U g]$ iff $\exists \pi[\pi(0) = s \land \exists i[i \leq |\pi| \land \pi(i) \models g \land \forall j[j < i \Rightarrow \pi(j) \models f]]$
- $s \models \text{A}[f U g]$ 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. \(T\)
  - \([\square]\) \(f\) iff for all immediate successor worlds \(f\) holds
  - \([\Diamond]\) \(f\) iff there is an immediate successor world in which \(f\) holds
Let $A$ 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 $T \subseteq S \times S$ (*$T$* total iff $\forall s[\exists t[T(s, t)]]$)
- labelling/marking/annotation $L: S \rightarrow \mathcal{P}(A)$.

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

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

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

similar construction as the oracle automaton

**Proposition**

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

iff

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

**Note**  
often $S \subseteq \mathbb{B}^n$, $\Sigma = \{a_1, \ldots, a_n\}$, and $L((s_1, \ldots, s_n)) = \{a_i \mid s_i = 1\}$
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 \( \mathsf{EX}, \mathsf{AX}, \mathsf{EF}, \mathsf{AF}, \mathsf{EG}, \mathsf{AG}, \mathsf{E}[\cdot \mathsf{U} \cdot] \) and \( \mathsf{A}[\cdot \mathsf{U} \cdot] \).

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

\[
\begin{align*}
\mathsf{AG}(\bar{r} \rightarrow \mathsf{AX}(\bar{a} \land \bar{b})) \\
\mathsf{AG} \; \mathsf{EX}(\bar{a} \land \bar{b}) \\
\mathsf{AG} \; \mathsf{EF}(\bar{a} \land \bar{b}) \\
\mathsf{AG} \; \mathsf{AF}(\bar{a} \land \bar{b}) \\
\mathsf{AG}(\bar{a} \land \bar{b} \land r \rightarrow \mathsf{AX} \; \mathsf{A}[(a \lor b) \; \mathsf{U} \; (\bar{a} \land \bar{b})])
\end{align*}
\]

\( \infinitely \; often \; \bar{a} \land \bar{b} \)

\[
(\mathsf{AG} \; r) \rightarrow \mathsf{AF}(a \land b)
\]

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 $p \in L(\pi(0))$
- $\pi \models \neg g$ 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 $\pi(0) \in I$
Comparison LTL and CTL

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

- then \((G r) \rightarrow F (a \land b)\) 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**  CTL and LTL have different expressibility:

\(AXEX_p\) can not be specified in LTL, \(AFAF_p\) does not have corresponding LTL formula
ACTL Formulas as LTL Formulas
[Clarke and Draghicescu’88]

ACTL is the sub logic of CTL formulas without \( \exists \) path quantifiers in NNF

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

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

**Definition** \( f \) and \( g \) are equivalent iff \( K \models f \iff K \models g \) for all Kripke structures \( K \).

\((f \text{ and } g \text{ can be formulas in different logics})\)

**Theorem** if an ACTL formula \( f \) is equivalent to an LTL formula \( g \), then also to \( f \setminus \mathcal{A} \).

**Proof** \( K \models f \) assumption \( \iff \forall \pi[\pi \models g] \) assumption \( \iff \forall \pi[\pi \models f] \iff \forall \pi[\pi \models f \setminus \mathcal{A}] \iff K \models f \setminus \mathcal{A} \)

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

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

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

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

**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**

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

$\Rightarrow$ 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

\[ K \models f \quad \text{iff} \quad \pi(i) \models p \quad \text{for all initialized paths } \pi \text{ of } K \text{ and all } i = 0 \mod m. \]

**Problem**  $p \wedge G(p \leftrightarrow \neg X p)$ 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:  $\neg \left( (1;\ldots;1;p)^{\ast};1;\ldots;1;\neg p \right)_{m-1}$  resp. $(1;\ldots;1;p)^{\omega}_{m-1}$
• 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 \(\pi\) *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

\[
\begin{align*}
\nu X[p \land [ ] X] & \equiv AGp \\
\mu X[q \lor (p \land \langle \rangle X)] & \equiv E[p U q]
\end{align*}
\]

\( \nu X[p \land [ ] [ ] X] \) corresponds to “every 2nd step \( p \) holds”

\[
\begin{align*}
\nu X[p \land \langle \rangle \mu Y[(f \land X) \lor (p \land \langle \rangle Y)]] & \equiv \nu X[p \land EXE[p U f \land X]] \equiv EGp \text{ under fairness } f
\end{align*}
\]
again over Kripke structures $K = (S, I, T, L)$.

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

**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 $\rho$:

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

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