EL2450 Hybrid and Embedded Control

Lecture 8: Models of Computation

- Discrete-event systems
- Transition systems

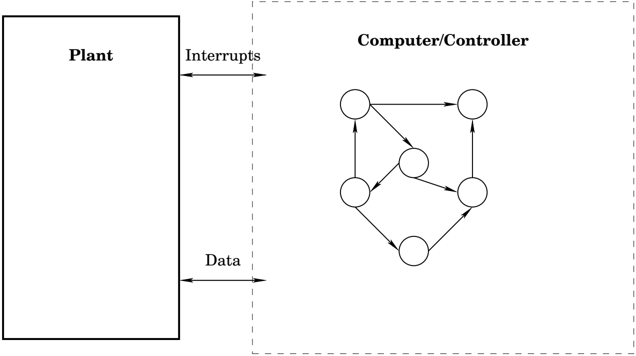
Today's Goal

You should be able to

- model and analyze automata
- deadlock, livelock, blocking, state-space minimization
- model and analyze transition systems
- do reach set computations
- do verification of safety and invariance properties

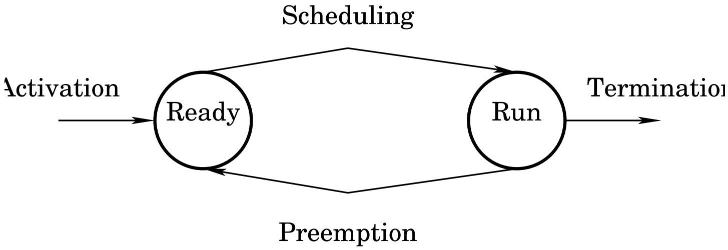
How Model Control Computations?

- Need mathematical models for analysis, design and verification
- Models should capture real-time and discrete-event features



Example: Scheduling

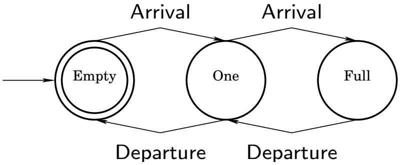
State machine model of task scheduling:



- Nodes represent states
- Edges represent transitions between states

Example: Queue

Model of a small queue with two positions:



- Nodes represent states (number of elements in the queue)
- Edges represent transitions between states
- Transitions are taken at events "Arrival" and "Departure"

Automaton

A deterministic automaton A is a five-tuple

$$A = (Q, E, \delta, q_0, Q_m)$$

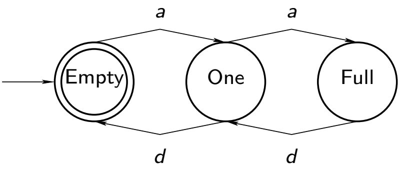
- Q is a finite set of states
- E is a finite set of events
- $\delta : Q \times E \mapsto Q$ is a transition function
- $q_0 \in Q$ is the initial state
- $Q_m \subseteq Q$ is the marked (or final) states

$q' = \delta(q, e)$ means that there is a transition labeled by event e from state q to q'.

Example: Queue Automaton

$A_Q = (Q, E, \delta, q_0, Q_m)$

- $Q = \{\text{Empty}, \text{One}, \text{Full}\}$, $E = \{a, d\}$
- $\delta(\text{Empty}, a) = \text{One}$, $\delta(\text{One}, a) = \text{Full}$, $\delta(\text{Full}, d) = \text{One}$, $\delta(\text{One}, d) = \text{Empty}$
- $q_0 = \text{Empty}$
- $Q_m = \text{Empty}$



Arrow without label indicates initial state and circle indicates marked (final) state

Events

ϵ denotes the empty string (no event): $\delta(q, \epsilon) = q, \forall q \in Q$.
E^* denotes all finite strings of elements of E together with ϵ.
$q = \delta(q_0, s)$, $s \in E^*$, denotes the state reached after executing the string $s = e_1 e_2 \ldots e_k$, $e_i \in E$, i.e.,

$$q = \delta(\delta(\ldots(\delta(q_0, e_1), \ldots, e_k)$$

where all transitions are supposed to be well-defined.

Example: If $E = \{a, d\}$ then $E^* = \{\epsilon, a, d, aa, ad, dd, \ldots\}$ and, for example, $\delta(q_0, ad) = \delta(\delta(q_0, a), d)$

Languages

A **language** L defined over an event set E is a set of finite strings formed from the events in E, so $L \subseteq E^*$.

Example

Let $E = \{a, d\}$. Then $L_1 = \{a\}$, $L_2 = \{a, aad\}$, and $L_3 = \{\epsilon, a, d\}$ are languages.

Operations on Languages

For $L, L_1, L_2 \subseteq E^*$, the **concatenation** of L_1, L_2 is defined as

$$L_1 L_2 = \{s \in E^* : \ (s = s_1 s_2) \text{ and } (s_i \in L_i, i = 1, 2)\}$$

The **Kleene-closure** of L is defined as

$$L^* = \{\epsilon\} \cup L \cup LL \cup LLL \cup \dots$$

Example
Let $E = \{a, d\}$. For $L_1 = \{a\}$, $L_2 = \{a, aad\}$, $L_1 L_2 = \{aa, aaad\}$ and $L_2^* = \{\epsilon, a, aad, aa, aaad, aada, aadaad, \dots\}$.

Generated and Marked Languages

A language **generated** by an automaton A is defined as

$$L(A) = \{s \in E^* : \ \delta(q_0, s) \text{ is defined}\}$$

A language **marked** by an automaton A is defined as

$$L_m(A) = \{s \in L(A) : \ \delta(q_0, s) \in Q_m\}$$

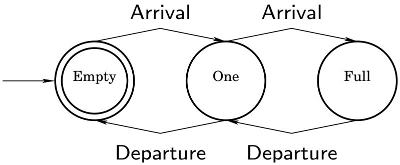
- $L(A)$ represents all directed paths in the state transition diagram starting from the initial state
- $L_m(A)$ represents the subset of these paths that ends in the marked states
- $L_m(A) \subseteq L(A)$

Example

The marked language of the queue automaton A_Q is

$$L_m(A_Q) = \{(a(ad)^* d)^*\}$$

It represents all arrival/departure sequences that start and end with an empty queue.



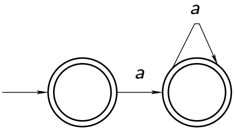
Note that $L(A_Q) \neq E^*$. Why?

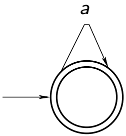
Equivalent Automata

A_1 and A_2 are **equivalent** if $L(A_1) = L(A_2)$ and $L_m(A_1) = L_m(A_2)$.

Example

The following automata are equivalent:



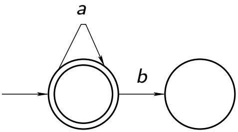


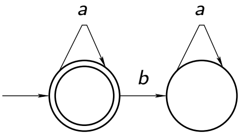
Deadlock and Livelock

A **deadlock** is a situation when an unmarked state can be reached and from which no further event can be executed

A **livelock** is a situation when a subset of unmarked states can be reached, but no transition is going out from the subset

Example





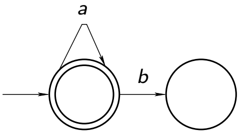
Prefix-Closure

$$\overline{L_m(A)} = \{s \in E^* :\ \exists s' \in E^*, ss' \in L_m(A)\}$$

is the prefix-closure of $L_m(A)$. Includes all prefixes of all strings in $L_m(A)$.

Note that $L_m(A) \subseteq \overline{L_m(A)} \subseteq L(A)$ (why?)

Example



$$L(A) = \{a^*, a^* b\}, \quad L_m(A) = \{a^*\}, \quad \overline{L_m(A)} = L_m(A)$$

Deadlock/Livelock and Prefix-Closure

If **deadlock** happens then $\overline{L_m(A)}$ is a proper subset of $L(A)$. This follows from that any string in $L(A)$ that ends at a deadlock state q cannot be a prefix of a string in $L_m(A)$.

If **livelock** happens then $\overline{L_m(A)}$ is a proper subset of $L(A)$. This follows from that any string in $L(A)$ that reaches the "absorbing" set of unmarked states cannot be a prefix of a string in $L_m(A)$ (since there is no way out of the "absorbing" set).

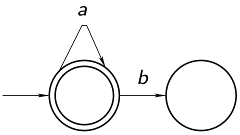
Blocking is used to denote these two cases.

Blocking Automaton

Definition
Automaton A with $\overline{L_m(A)} \subset L(A)$ is called **blocking**.

Definition
Automaton A with $\overline{L_m(A)} = L(A)$ is called **nonblocking**.

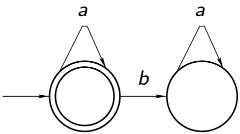
Example: Blocking and Deadlock



$$L(A) = \{a^*, a^*b\}, \quad L_m(A) = \{a^*\}, \quad \overline{L_m(A)} = L_m(A)$$

Since $\overline{L_m(A)} \subset L(A)$, the automaton is blocking. Note the deadlock.

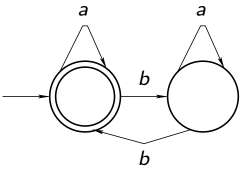
Example: Blocking and Livelock



$$L(A) = \{a^*, a^*ba^*\}, \quad L_m(A) = \{a^*\}, \quad \overline{L_m(A)} = L_m(A)$$

Since $\overline{L_m(A)} \subset L(A)$, the automaton is blocking. Note the livelock.

Example: Nonblocking



$$L(A) = E^*, \quad L_m(A) = \{a^*, (a^*ba^*b)^*\}, \quad \overline{L_m(A)} = E^*$$

Since $\overline{L_m(A)} = L(A)$, the automaton is nonblocking.

Nondeterministic Automaton

A nondeterministic automaton A is a five-tuple

$$A = (Q, E \cup \{\epsilon\}, \delta, q_0, Q_m)$$

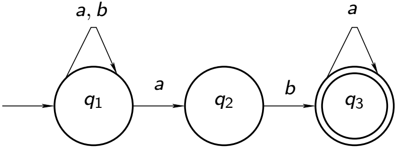
where

- Q is a finite set of states
- $E \cup \{\epsilon\}$ is a finite set of events
- $\delta : Q \times E \cup \{\epsilon\} \mapsto 2^Q$ is a transition function
- $q_0 \subseteq Q$ is the set of initial states
- $Q_m \subseteq Q$ is the marked (or final) states

Note: The differences to a deterministic automaton is that

- δ maps to a set (2^Q denotes the set of all subsets of Q)
- q_0 is a set

Example



What happens when more than one transition is possible?

- The machine "splits" into multiple copies
- Each branch follows one possibility
- Together, branches follow *all* possibilities.

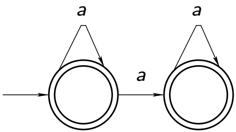
Example

What happens if the automaton receives the sequence *aba* ?

State-Space Minimization

There is no unique way to construct an automaton that marks a given language.
It is sometimes desirable to find the automaton with smallest number of states (lowest cardinality of Q) that marks a language.

Example
Minimize the state-space of the following automaton:

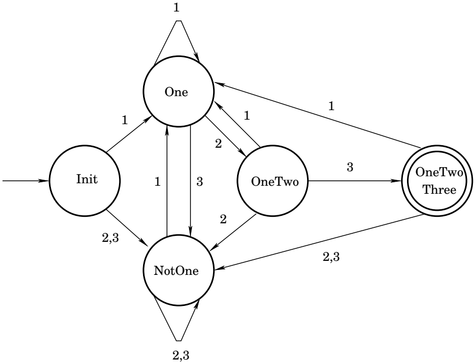


Example: A Digit Sequence Detector

Design an automaton A for a machine that reads digits from $\{1, 2, 3\}$ and detects (i.e., marks) any string that ends with the substring "123".
A should generate E^*, with $E = \{1, 2, 3\}$, since it should accept any input digit at any time. A should mark

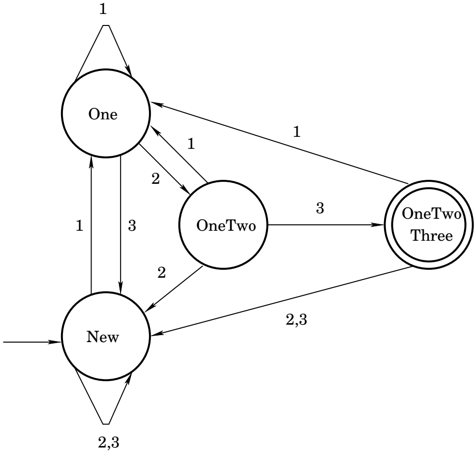
$$L = \{ss' \in E^* : \; s \in E^*, s' = 123\}$$

A non-minimal solution:



Note that $\delta(\mathsf{Init}, e) = \delta(\mathsf{NotOne}, e)$ for all e.

A minimal solution in which Init and NotOne are aggregated:



Summary: Discrete-Event Systems

- An automaton is a mathematical model of a discrete-event system
- Useful to analyze blocking of the system, automata interconnections possible
- Also possible to optimize criteria and check other properties

- Automata are defined only for *finite* sets of states and events
- Transition systems: generalize to possibility of *infinite* sets of states and/or events

Transition Systems

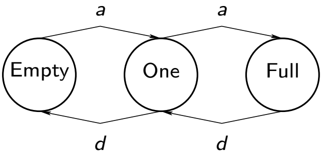
A **transition system** is a tuple $T = (S, \Sigma, \rightarrow)$ where

- S is a set of states
- Σ is a set of generators (or actions)
- $\rightarrow \subset S \times \Sigma \times S$ is a transition relation

- We use the notation $s \stackrel{\sigma}{\rightarrow} s'$ to denote $(s, \sigma, s') \in \rightarrow$

- We may include initial and final states, $S_0 \subseteq S$ and $S_F \subseteq S$, respectively. Then we denote $T = (S, \Sigma, \rightarrow, S_0, S_F)$.

Discrete Event System as a Transition System



$T_{\text{DES}} = (S, \Sigma, \rightarrow)$ where

- $S = \{\text{Empty}, \text{One}, \text{Full}\}$ is a set of states
- $\Sigma = \{a, d\}$ is a set of generators
- $\rightarrow = \{(\text{Empty}, a, \text{One}), (\text{One}, a, \text{Full}), (\text{Full}, d, \text{One}), (\text{One}, d, \text{Empty})\}$ or with the other (more intuitive) notation:

$$\text{Empty} \xrightarrow{a} \text{One}, \quad \text{One} \xrightarrow{a} \text{Full}, \quad \text{Full} \xrightarrow{d} \text{One}, \quad \text{One} \xrightarrow{d} \text{Empty}$$

Differential Equation as a Transition System

Consider the ordinary differential equation $\dot{x} = f(x)$, $x \in \mathbb{R}$, and let $\phi(t)$ denote its solution

The ODE defines the transition system $T_{\text{ODE}} = (S, \Sigma, \rightarrow)$ where

- $S = \mathbb{R}$
- $\Sigma = \text{Time} = \{t : \ t \geq 0\}$
- $x \xrightarrow{t} y$ provided that $x = \phi(0)$ and $y = \phi(t)$

Predecessor and Successors

For transition system $T = (S, \Sigma, \to)$ define the **predecessor** operator for $P \subset S$ as

$$\text{Pre}(P) = \{s \in S :\ \exists \sigma \in \Sigma, \exists p \in P, s \xrightarrow{\sigma} p\}$$

the **successor** operator as

$$\text{Post}(P) = \{s \in S :\ \exists \sigma \in \Sigma, \exists p \in P, p \xrightarrow{\sigma} s\}$$

Recursively, we define

$$\text{Post}^0(P) = P, \qquad \text{Post}^k(P) = \text{Post}(\text{Post}^{k-1}(P))$$

and similarly for Pre^k

Example

For the discrete event system,

$$\mathsf{Pre}(\mathsf{Empty}) = \mathsf{One}, \quad \mathsf{Pre}(\{\mathsf{Empty}, \mathsf{One}\}) = \{\mathsf{Empty}, \mathsf{One}, \mathsf{Full}\}$$

For the ODE $\dot{x} = -x$:

$$\mathsf{Pre}([-\epsilon, \epsilon]) = \begin{cases} \mathbb{R}, & \epsilon > 0 \\ 0, & \epsilon = 0 \end{cases}$$

$$\mathsf{Post}([-\epsilon, \epsilon]) = [-\epsilon, \epsilon]$$

Pre_σ **and** Post_σ

For specific $\sigma \in \Sigma$, the $\mathrm{Pre}_\sigma(P)$ and $\mathrm{Post}_\sigma(P)$ are

$$\mathrm{Pre}_\sigma(P) = \{s \in S : \ \exists p \in P, s \xrightarrow{\sigma} p\}$$

$$\mathrm{Post}_\sigma(P) = \{s \in S : \ \exists p \in P, p \xrightarrow{\sigma} s\}$$

Recursively,

$$\mathrm{Pre}^0_\sigma(P) = P \qquad \mathrm{Pre}^k_\sigma(P) = \mathrm{Pre}_\sigma(\mathrm{Pre}^{k-1}_\sigma(P))$$

and similarly for Post_σ

Reach Set

$\mathrm{Reach}(S_0)$ is the set of states that can be reached from S_0 by a sequence of transitions, i.e.,

$$\mathrm{Reach}(S_0) = \bigcup_{k=0,1,\ldots} \mathrm{Post}^k(S_0)$$

Examples

$$\mathrm{Reach}_{T_{\mathrm{DES}}}(\mathrm{Empty}) = \{\mathrm{Empty}, \mathrm{One}, \mathrm{Full}\}$$

For the differential equation $\dot{x} = ax$, $a > 0$,

$$\mathrm{Reach}_{T_{\mathrm{ODE}}}(x_0) = [x_0, \infty), \mathrm{if}\ x_0 > 0$$

$$\mathrm{Reach}_{T_{\mathrm{ODE}}}(x_0) = (-\infty, x_0], \mathrm{if}\ x_0 < 0$$

$$\mathrm{Reach}_{T_{\mathrm{ODE}}}(x_0) = 0, \mathrm{if}\ x_0 = 0$$

Safety

For a transition system $T = (S, \Sigma, \rightarrow)$ with initial state in S_0, let $B \subset S$ denote a "bad" set, i.e., a set of states that we don't want the system to enter. T is **safe** if

$$\text{Reach}(S_0) \cap B = \emptyset$$

Remark

- B encodes the property to *verify*
- Verification is about verifying that the system fulfills its specification, i.e., $\text{Reach}(S_0) \cap B = \emptyset$

Validating Designs

Embedded system designs can be validated with various degrees of rigour (higher is better):

- By construction: property is inherent
- By verification: property is provable syntactically
- By simulation: check behaviour for all inputs
- By intuition: property is true, I just know it is
- By assertion: property is true, wanna make something of it?
- By intimidation: don't even try to doubt whether it is true

[Pappas,2005]

Verification

- **Prove** that a system fulfill certain specification
- Based on a mathematical model and a computational tool

Decidability

- A verification problem is **decidable** if there exists an algorithm that solves it and terminates in a finite number of steps
- A verification problem is **semi-decidable** if the algorithm may not terminate, but if it does, it produces the correct result
- Verifying **safety** for a finite transition system is decidable
- Verifying safety can be decidable or semi-decidable for some more general transition systems, but is in most cases undecidable
- Verifying **safety** can be solved by computing Reach

Reach Set Computation

Reachability Algorithm to compute $\text{Reach}(S_0)$

$\text{Reach}_{-1} := \emptyset$, $\text{Reach}_0 := S_0$, $i := 0$

`while` $\text{Reach}_i \neq \text{Reach}_{i-1}$ `do`

  $\text{Reach}_{i+1} := \text{Reach}_i \cup \text{Post}(\text{Reach}_i)$

  $i := i + 1$

`end`

- If the algorithm terminates, then $\text{Reach}(S_0) := \text{Reach}_i$
- If the state space is finite, the algorithm terminates in a finite number of steps
- The algorithm does not necessarily terminate for general transition systems
- $s \xrightarrow{e} s'$ is simple, but $s \xrightarrow{t} s'$ in general hard

Invariance

Consider $T = (S, \Sigma, \to)$

- $S_0 \subset S$ is invariant if $\text{Reach}(S_0) = S_0$, ie, all states starting from S_0 remain in S_0
- Reachability algorithm may help in searching for invariant sets
- Finding invariant sets might help in Reach computation

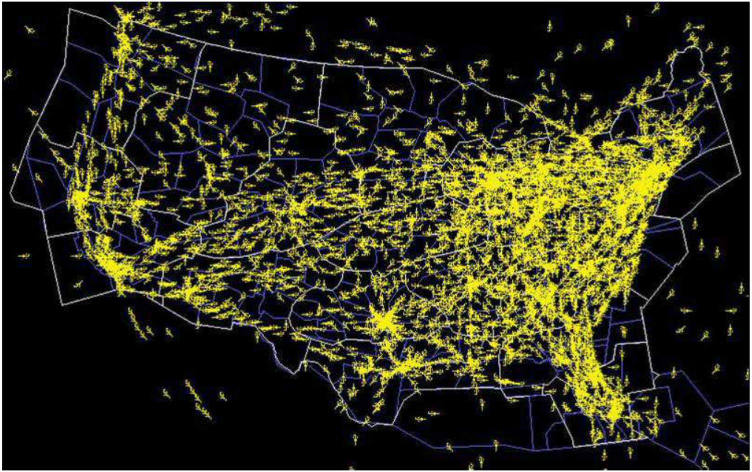
Over-Approximation of Reach Set

- It is often hard to calculate Reach exactly
- Compute an over-approximation $A \supset \text{Reach}$ instead
- Note that $A \cap B = \emptyset$ implies that $\text{Reach} \cap B = \emptyset$, so safety is guaranteed if the algorithm based on over-approximation terminates

Example

Derive an over-approximation of Reach for $\dot{x} = f(x)$. Assume that B is associated to instability.

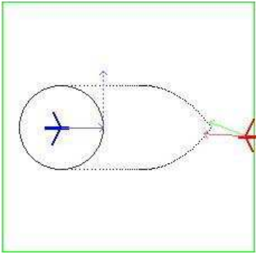
Example: Air Traffic Control [Tomlin]

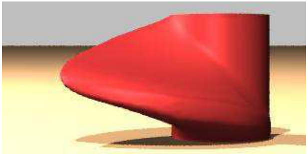


Reachability Analysis

- Autopilots of modern jets are highly automated and have hundreds of flight modes
- Important to verify safe operation of autopilot
- Enables future architectures for free flight

Reach set computations for collision avoidance:





Beyond safety and invariance

- Current trend: define more complicated specifications using formal languages
- Linear Temporal Logic(LTL), Computational Tree Logic(CTL) define specification as a formula ϕ
- Basic verification problem

$$T \models \phi$$

- Approach to solution: Model Checking

Specifications through LTL

- Assign atomic propositions (AP) to states S through map $O : S \to 2^{AP}$
- AP represent things that can happen/observed at each state
- Transition system expanded to $T = (S, \Sigma, \to, AP, O, S_0, S_F)$.
- LTL expresses specifications along sequences of AP using temporal (eg, always, eventually) and boolean (and, or) operators
- Example spec: "eventually go to Region A and always avoid Region B"

LTL Model Checking

Given **finite** transition system T and LTL formula ϕ, determine if T fulfills ϕ (ie, whether $T \models \phi$)

- Model checker returns a counterexample if there exists one
- Computations based on automata-based representation of ϕ (product operator with T)
- Otherwise it returns that the spec is verified
- Off-the-self model checkers exist, eg, SPIN
- Can be used for control synthesis using $\neg\phi$

Next Lecture

Hybrid automata

- Motivate hybrid systems
- Hybrid automata as models of hybrid systems
- Dynamical properties of hybrid automata