# Robotics
# Finite State Machines

Marco Della Vedova
<marco.dellavedova@unipv.it>

Wednesday 4th November, 2015

15:56

http://robot.unipv.it/toolleeo

## Models of computation and abstract machines

In computer science, automata theory is the study of mathematical objects called *abstract machines* or *automata* and the computational problems that can be solved using them.

> Automata comes from the Greek word
> αὐτόματα = "self-acting"

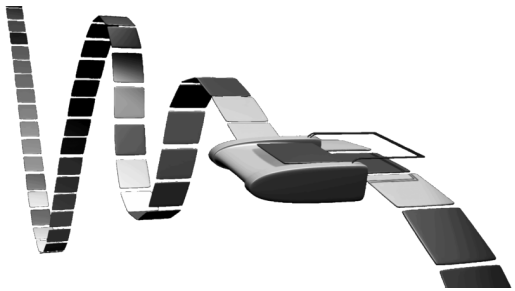
An abstract machine (a.k.a. abstract computer) is a theoretical model of a computer hardware or software systems.

A model of computation is the definition of the set of allowable operations used in computation and their respective costs. It is used for:

- measuring the complexity of an algorithm in execution time and/or memory space
- analyze the computational resources required
- software and hardware design

# Turing Machine

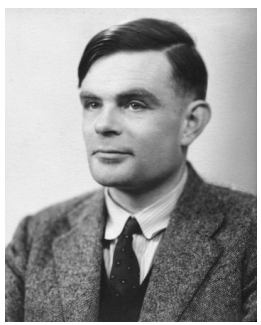
A Turing machine is an abstract device that manipulates symbols on a strip of tape according to a table of rules. It can be adapted to simulate the logic of any computer algorithm.



Artistic representation of a Turing machine (credits: Wikipedia)

It consists of:

1. An unbounded tape divided into cells. Each cell contains a symbol from some finite alphabet.
2. A head that can read and write symbols on the tape and move the tape left and right one (and only one) cell at a time.
3. A state register that stores the state of the machine (within a finite set of states).
4. A finite table of instructions that, given the current state and the symbol it is reading on the tape, tells the machine what to do: write a symbol, move the head, assume a new state.



Alan Turing
(1912-1954)

## Turing Machine in complexity theory

Turing machines are **not intended as a practical computing technology**, but rather as a thought experiment representing a computing machine. It is believed that if a problem can be solved by an algorithm, there exists a Turing machine that solves the problem (**Church–Turing thesis**). Furthermore, it is known that everything that can be computed on other models of computation known to us today, such as a *RAM machine*, *Conway's Game of Life*, *cellular automata* or any programming language can be computed on a Turing machine. Since Turing machines are easy to analyze mathematically, and are believed to be as powerful as any other model of computation, the **Turing machine is the most commonly used model in complexity theory**.

## Finite State Machines (FSMs)

A Finite State Machine (a.k.a. *finite state automaton*) is an abstract device (simpler than a Turing machine), consisting of:

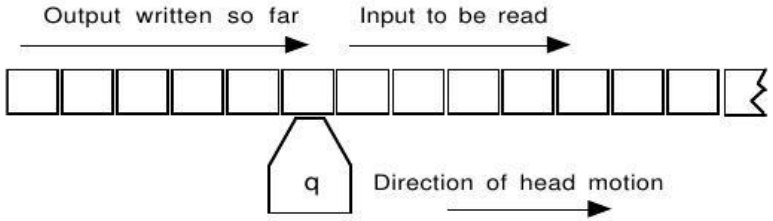
- a set of states (including a start state),
- an alphabet of symbols that serves as a set of possible inputs to the machine,
- and a transition function that maps each state to another state (or to itself) for any given input symbol.

The machine operates by being fed a string of symbols, and moves through a series of states according to the transition function.

Output? Different types of FSM are distinguished depending on if the output is produced and how it is produced: before or after a transition.

# FSMs and Turing machines

One way to view the finite-state machine model as a more restrictive Turing machine is to separate the input and output halves of the tapes: the head can move strictly one-way. However, mathematically we don't need to rely on the tape metaphor; just viewing the input and output as sequences of events occurring in time would be adequate.




Therefore, the computational core of a Turing machine is a FSM.

## Example of FSM: an edge-detector

The function of an edge detector is to detect transitions between two symbols in the input sequence, say $0$ and $1$. It does this by outputting $0$ as long as the most recent input symbol is the same as the previous one. However, when the most recent one differs from the previous one, it outputs a $1$. By convention, the edge detector always outputs $0$ after reading the very first symbol. Thus we have the following input output sequence pairs for the edge-detector, among an infinite number of possible pairs:

$$\text{inputs} \longrightarrow \text{outputs}$$

$$0\ 1\ 1\ 1 \longrightarrow 0\ 1\ 0\ 0$$

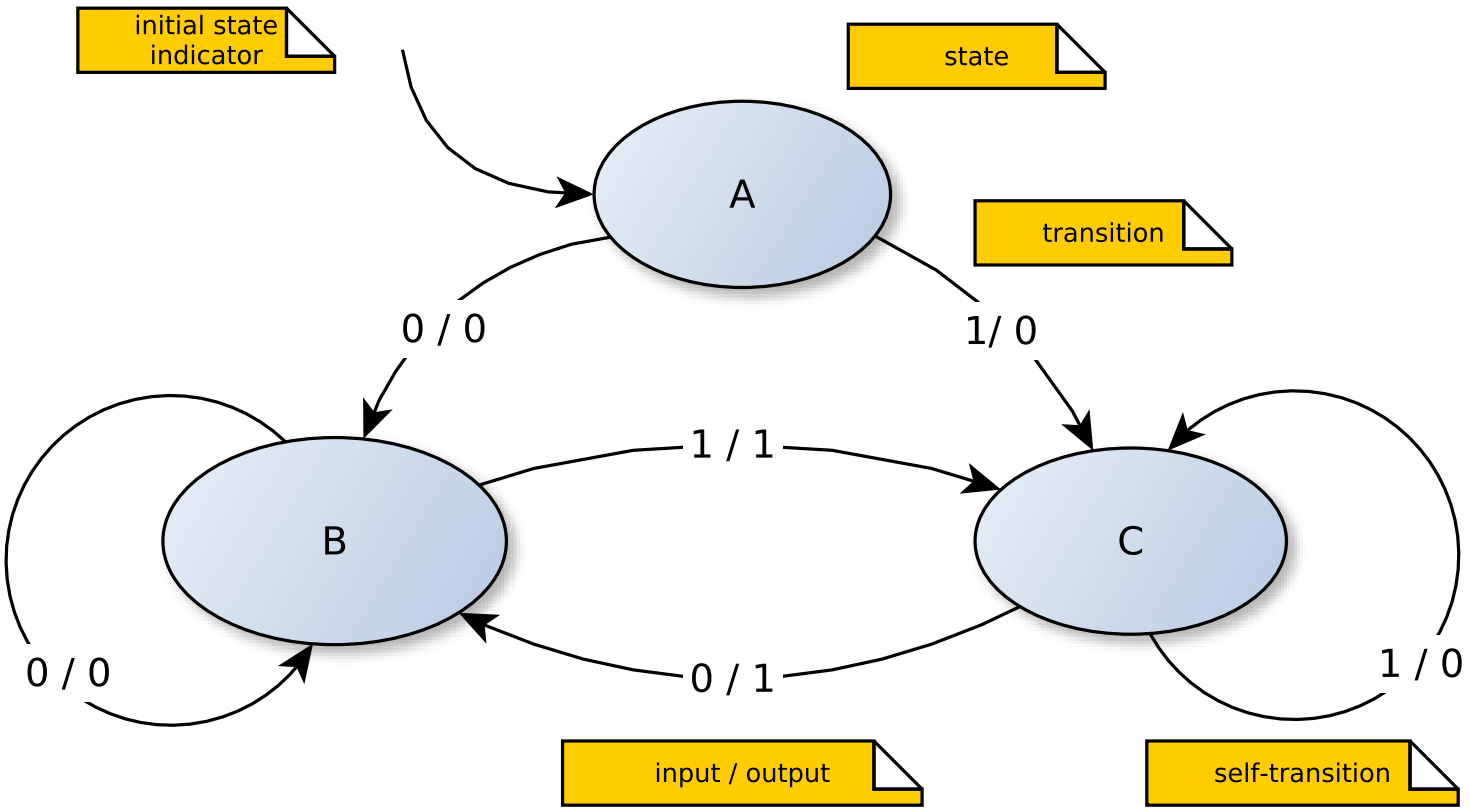
$$0\ 1\ 1\ 1\ 1\ 0 \longrightarrow 0\ 1\ 0\ 0\ 0\ 1$$

$$1\ 0\ 1\ 0\ 1\ 0 \longrightarrow 0\ 1\ 1\ 1\ 1\ 1$$

## Edge-detector example

This graphical representation is knows as state diagram.

A state diagram is a direct graph with a special node representing the initial state.



## Mathematical model of a FSM

A FSM is a five-tuple

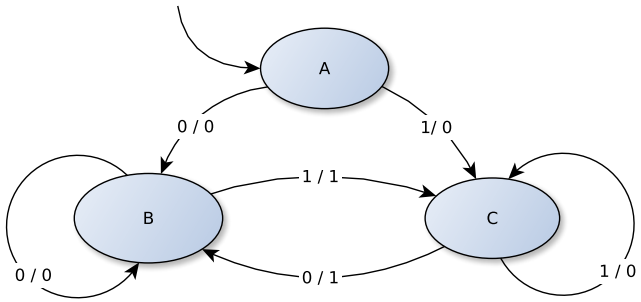
$$(\Sigma, \Gamma, S, s_0, \delta)$$

where:

- $\Sigma$ is the input alphabet (a finite, non-empty set of symbols).
- $\Gamma$ is the output alphabet (a set of symbols).
- $S$ is a finite, non-empty set of states.
- $s_0$ is the initial state, an element of $S$.
- $\delta$ is the transition function: $\delta : S \times \Sigma \rightarrow S \times \Gamma$.

# Exercise: math model of edge detector

What $(\Sigma, \Gamma, S, s_0, \delta)$ are for the edge-detector FSM?

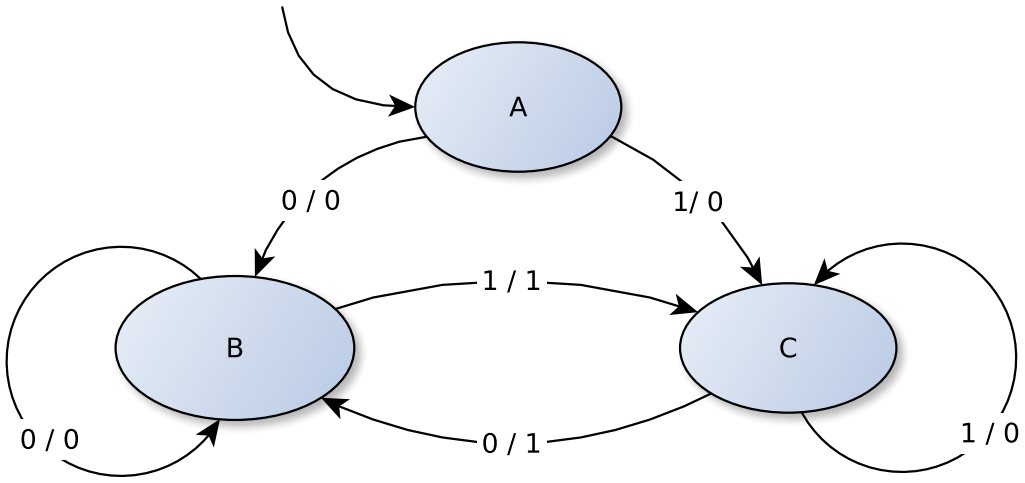



- input alphabet $\Sigma = ?\{0, 1\}$
- output alphabet $\Gamma = ?\{0, 1\}$
- state space $S = ?\{A, B, C\}$
- initial state $s_0 = ?A$
- transition function $\delta =?...$

# Tabular representation of a FSMs' transition function

The transition function $\delta : S \times \Sigma \rightarrow S \times \Gamma$ can be represented by a tabular with states on the rows and inputs on the columns. In each cell there is a tuple $s, \gamma$ indicating the next state and the output.

For example, for the edge-detector FSM, the transition table is:




| | **0** | **1** |
|---|---|---|
| **A** | B,0 | C,0 |
| **B** | B,0 | C,1 |
| **C** | B,1 | C,0 |

## The notion of state

Intuitively, the state of a system is its condition at a particular point in time. In general, the state affects how the system reacts to inputs. Formally, we define the state to be an encoding of everything about the past that has an effect on the system's reaction to current or future inputs.

> The state is a summary of the past.

# State machines as discrete dynamic system

Transitions between states govern the discrete dynamics of the state machine and the mapping of inputs to outputs. The FSM evolves in a sequence of transitions.
We can number these transitions starting from $0$ for the initial state.
Specifically, let $x : \mathbb{N} \to S$ to be a function that gives the state of an FSM at transition $k \in \mathbb{N}$. Let $u : \mathbb{N} \to \Sigma$ and $y : \mathbb{N} \to \Gamma$ denote that input and output at each transition. Hence, $x(0) \in S$ is the first input and $y(0) \in \Gamma$ the first output.

The dynamics of the state machine is given by:

$$\begin{cases} x(0) = s_0 \\ (x(k+1), y(k)) = \delta(x(k), u(k)) \end{cases}$$

The previous system can be rewritten (in accordance to the standard notation for dynamical systems) as:

$$\begin{cases} x(k+1) = \delta'(x(k), u(k)), \quad x(0) = s_0 \\ y(k) = \delta''(x(k), u(k)) \end{cases}$$

## When does a transition occur?

Nothing in the definition of a state machine constrains *when* it reacts.
As a discrete system, we do not need to talk explicitly about the amount of time that passes between transitions, since it is actually irrelevant to the behavior of a FSM.

Still, a FSM could be:

- event triggered, in which case it will react whenever an input is provided, or
- time triggered, meaning that it reacts at regular time intervals.

The definition of the FSM does not change in these two cases. It is up to the environment in which an FSM operates when it should react.

## Mealy FSM and Moore FSM

- By now we talked about Mealy FSM, named after George Mealy, a Bell Labs engineer who published a description of these machines in 1955.
- Mealy FSM are characterized by producing outputs when a transition is taken.

- An alternative, known as a Moore FSM, produces outputs when the machine is in a state, rather than when transition is taken.
- Moore machines are named after Edward Moore, another Bell Labs engineer who described the in a 1956 paper.

## Mealy FSM and Moore FSM

- The distinction between Mealy and Moore machines is subtle but important.
- Both are discrete systems, and hence their operation consists of a sequence of discrete reactions.
- For a Moore machine, at each reaction, the output produced is defined by the current state (at the start of the reaction, not at the end).
- Thus, the output at the time of a reaction does not depend on the input at that same time.
- The input determines which transition is taken, but not what output is produced by the reaction.

With these assumptions, a Moore machine is strictly causal

## Notion of causality

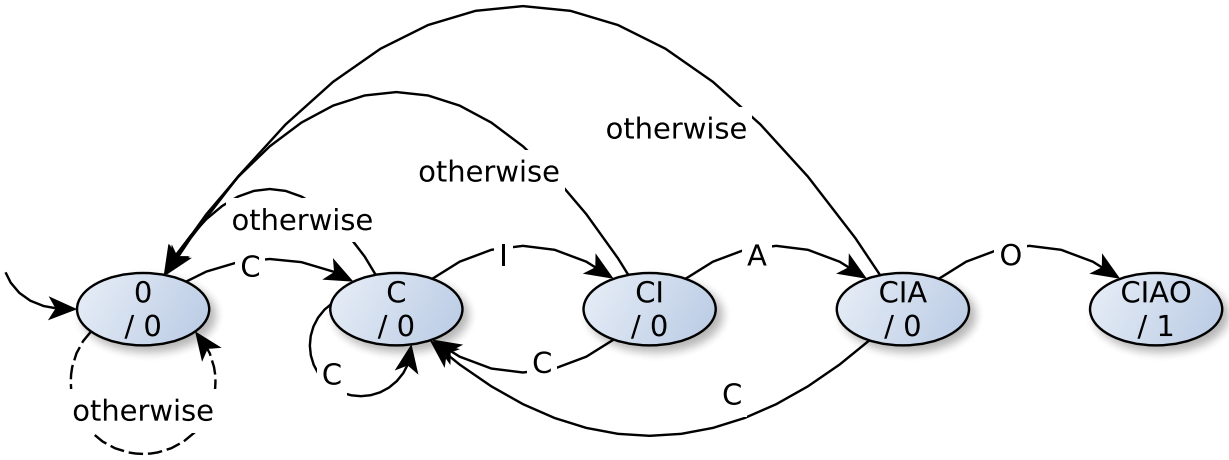
- a system is **causal** if its output depends only on current and past output
- in other words, in casual systems if two possible inputs are identical up to (and including) time $\tau$, the outputs are identical up to (and including) time $\tau$
- in **strictly casual systems** if two possible inputs are identical up to (and not including) time $\tau$, the outputs are identical up to (and not including) time $\tau$

strictly causal systems are useful to build feedback systems

- **non-causal** (**acausal**) systems depends also on future inputs (examples: population growth, weather forecasting, planning)
- **anti-causal** systems depends only on future inputs

# Moore FSM example

**Request:** Design a Moore FSM that takes characters A-Z as input and returns 1 if in the input there is the string "CIAO".
**Note**: since the output depends on the current state only, outputs are shown in the state rather than on the transitions in the state diagram.




**Notes** (valid for Moore and Mealy FSM state diagrams):

- it is often convenient to use the label *otherwise* on transitions
- *otherwise* self-transition are called "default transitions" and can be omitted

# Mealy FSM vs Moore FSM

- any Moore machine can be converted to an equivalent Mealy machine
- a Mealy machine can be converted to an almost equivalent Moore machine
- it differs only in that the output is produced on the next reaction rather than on the current one
- Mealy machines tends to be more compact (requiring fewer states to represent the same functionality), and are able to produce an output that instantaneously responds to the input
- Moore machines are used when output is associated with a state of the machine, hence the output is somehow persistent

## Exercises

- Convert the edge-detector Mealy FSM in an almost-equivalent Moore FSM.
- Convert the CIAO-detector Moore FSM to an equivalent Mealy FSM.

## FSM classification

- Transducers are machines that read strings (sequences of symbols taken from an alphabet) and produce strings containing symbols of another (or even the same) alphabet.
- Acceptors (aka recognizers and sequence detectors) produce a binary output, saying either *yes* or *no* to answer whether the input is accepted by the machine or not. All states of the FSM are said to be either accepting or not accepting. At the time when all input is processed, if the current state is an accepting state, the input is accepted; otherwise it is rejected.
- Classifiers are a generalization that similarly to acceptor produces single output when terminates but has more than two terminal states.
- Generators (aka sequencers) are a subclass of aforementioned types that have a single-letter input alphabet. They produce only one sequence, which can be interpreted as output sequence of transducer or classifier outputs.

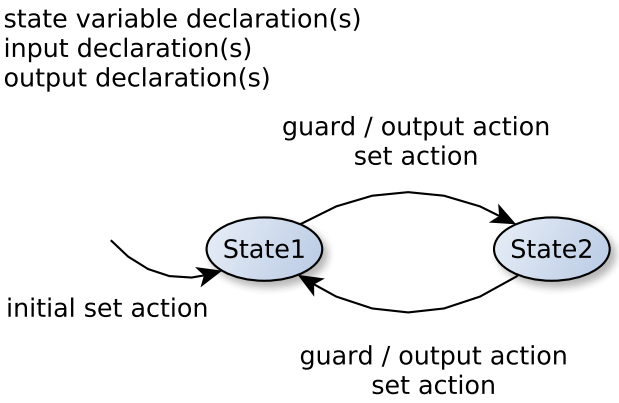
## Extended state machines

The notation for FSMs becomes awkward when the number of states gets large. Moreover, many applications require to read two or more input sources.
Extended state machines address those issues by augmenting the FSM model with:

- internal state variables that may be read and written as part of taking a transition between states;
- input valuations: a valuation of a set of variables is an assignment of value to each variable;
- transitions triggered by guards: a guard is a *predicate* (a boolean-valued expression) that evaluates to *true* when the transition should be taken;
- output actions that may be valuations of output variables or function calls.

# Extended state machines: graphical notation

The general notation for extended state machines is the following:




- set actions specify assignment to variables that are made when the transition is taken
- these assignments are made *after* the guard has been evaluated and the output actions have been fired
- if there are more than one output action or set action, they are made in sequence

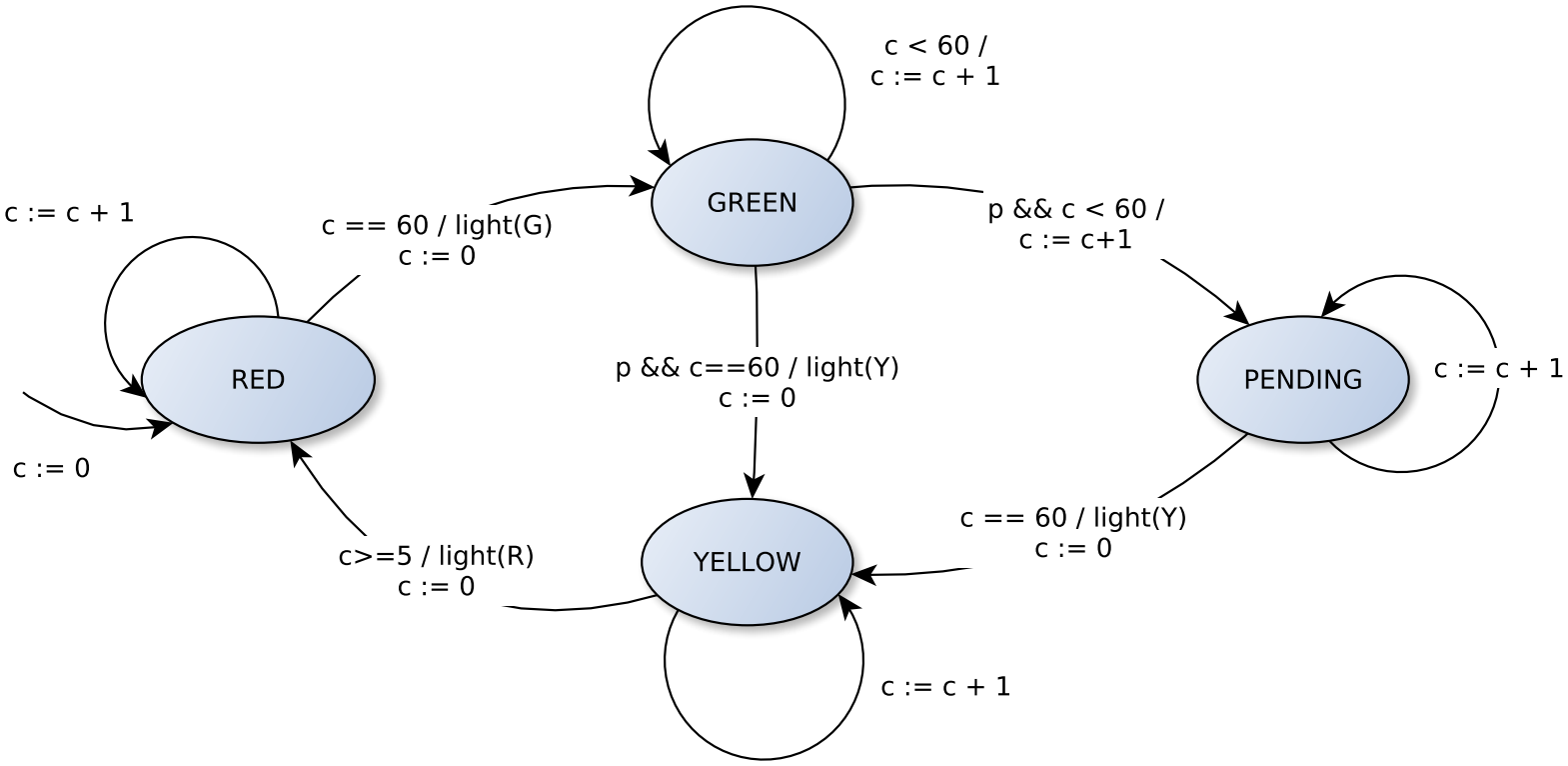
# Extended state machine example: traffic light

**Problem**: model a controller for a traffic light (for cars) at a pedestrian crosswalk.

1. Use a time triggered machine that reacts once per second.
2. It starts in the RED state and counts 60 seconds with the help of the internal variable $c$.
3. It then transitions to GREEN, where it will remain until the input $p$ is true. That input could be generated by a pedestrian pushing a button to request a walk light.
4. When $p$ is true, the machine transitions to YELLOW if it has been in state GREEN for at least 60 seconds.
5. Otherwise, it transitions to pending, where it stays for the remainder of the 60 second interval. This ensures that once the light goes green, it stays green for at least 60 seconds.
6. At the end of 60 seconds, it will transition to YELLOW, where it will remain for 5 seconds before transitioning back to RED.
7. The outputs produced by this machine is a function call to $\mathrm{light}(x)$, where $x \in \{R, G, Y\}$ represents the color light to be turned on.

# Extended state machine example: traffic light

**inputs**: $p : \{true, false\}$
**outputs**: $\text{light}(x),\ x \in \{R, G, Y\}$
**variables**: $c : \{0, \ldots, 60\}$



## Extended state machines: state space

The state of an extended state machine includes not only the information about which discrete state (indicated by a bubble) the machine is in, but also what values any variables have. The number of possible states can therefore be quite large, or even infinite. If there are $n$ discrete states (bubbles) and $m$ variables each of which can have one of $p$ possible values, then the size of the state space of the state machine is

$$|\text{States}| = np^m$$

Extended state machines may or may not be FSMs. In particular, it is not uncommon for $p$ to be infinite. For example, a variable may have values in $\mathbb{N}$, the natural numbers, in which case, the number of states is infinite.

## Reachable states

Some state machines will have states can never be reached, so the set of reachable states – comprising all states that can be reached from the initial state on some input sequence – may be smaller than the set of states.

For example, in the traffic light FSM, the $c$ variable has $61$ possible values and there are $4$ bubbles, so the total number of combination is $61 \times 4 = 244$. The size of the state space is therefore $244$. However, not all of these states are reachable. In particular, while in the YELLOW state, the count variable will have only one of $6$ values in $\{0, \ldots, 5\}$. The number of reachable states, therefore, is $61 \times 3 + 6 = 189$.

## Determinacy

- A state machine is said to be **deterministic** (or **determinate**) if, for each state, there is at most one transition enabled by each input value.
- The formal definition of an FSM given ensures that it is deterministic, since the transition function $\delta$ is a function, not a one-to-many mapping.
- The graphical notation with guards on the transitions, however, has no such constraint.
- Such a state machine will be deterministic only if the guards leaving each state are non-overlapping.

## Receptiveness

- A state machine is said to be **receptive** if, for each state, there is *at least one* transition possible on each input symbol.
- In other words, receptiveness ensures that a state machine is always ready to react to any input, and does not "get stuck" in any state.
- The formal definition of an FSM given above ensures that it is receptive, since $\delta$ is a function, not a partial function.
- It is defined for every possible state and input value.
- Moreover, in our graphical notation, since we have implicit default transitions, we have ensured that all state machines specified in our graphical notation are also receptive.

if a state machine is both deterministic and receptive, for every state, there is exactly one transition possible on each input value

If for any state of a state machine, there are two distinct transitions with guards that can evaluate to true in the same reaction, then the state machine is nondeterminate or nondeterministic.
It is also possible to define machines where there is more than one initial state: such a state machine is also nondeterminate.

## Applications

- modeling unknown aspects of the environment or system
- hiding detail in a specification of the system
- non-deterministic FSMs are more compact than deterministic FSMs
  - a classic result in automata theory shows that a nondeterministic FSM has a related deterministic FSM that is language equivalent
  - but the deterministic machine has, in the worst case, many more states (exponential)

## Behaviors, Traces and Computational Trees

- FSM behavior is a sequence of transitions.
- An execution trace is the record of inputs, states, and outputs in a behavior. A trace looks like:

$$((u_0, x_0, y_0), (u_1, x_1, y_1), (u_2, x_2, y_2), \dots)$$

or

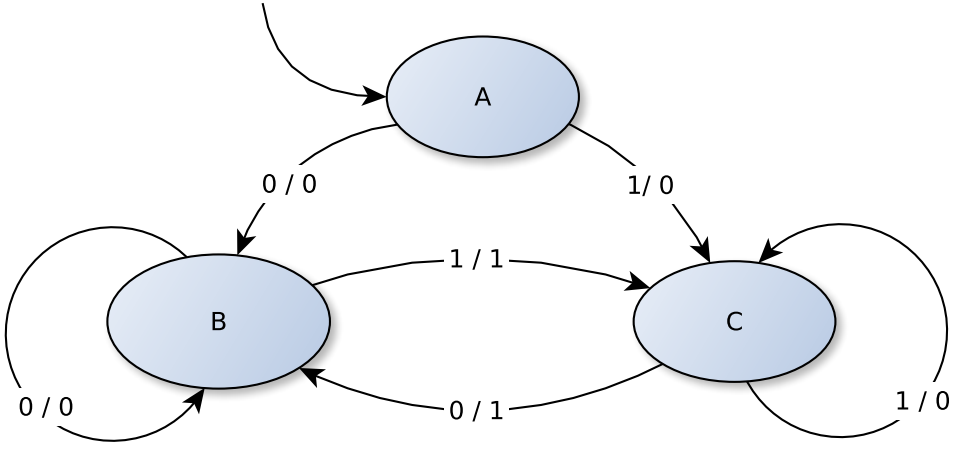
$$x_0 \xrightarrow{u_0/y_0} x_1 \xrightarrow{u_1/y_1} x_2 \xrightarrow{u_2/y_2} \dots$$

  where $u_i, x_i, y_i$ represent valuation of the inputs, current state, and outputs' valuation at transition $i$, respectively.
- A computational tree is a graphical representation of all possible traces

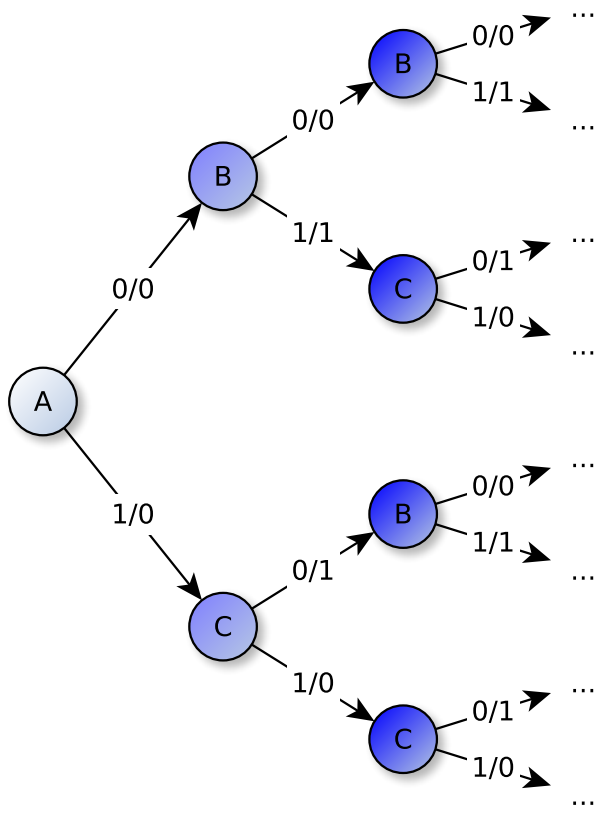
FSMs are suitable for formal analysis. For example, safety analysis might show that some unsafe state is not reachable.

# Computational tree example

Recall the edge-detector FSM:




Computational tree:




Example of trace:

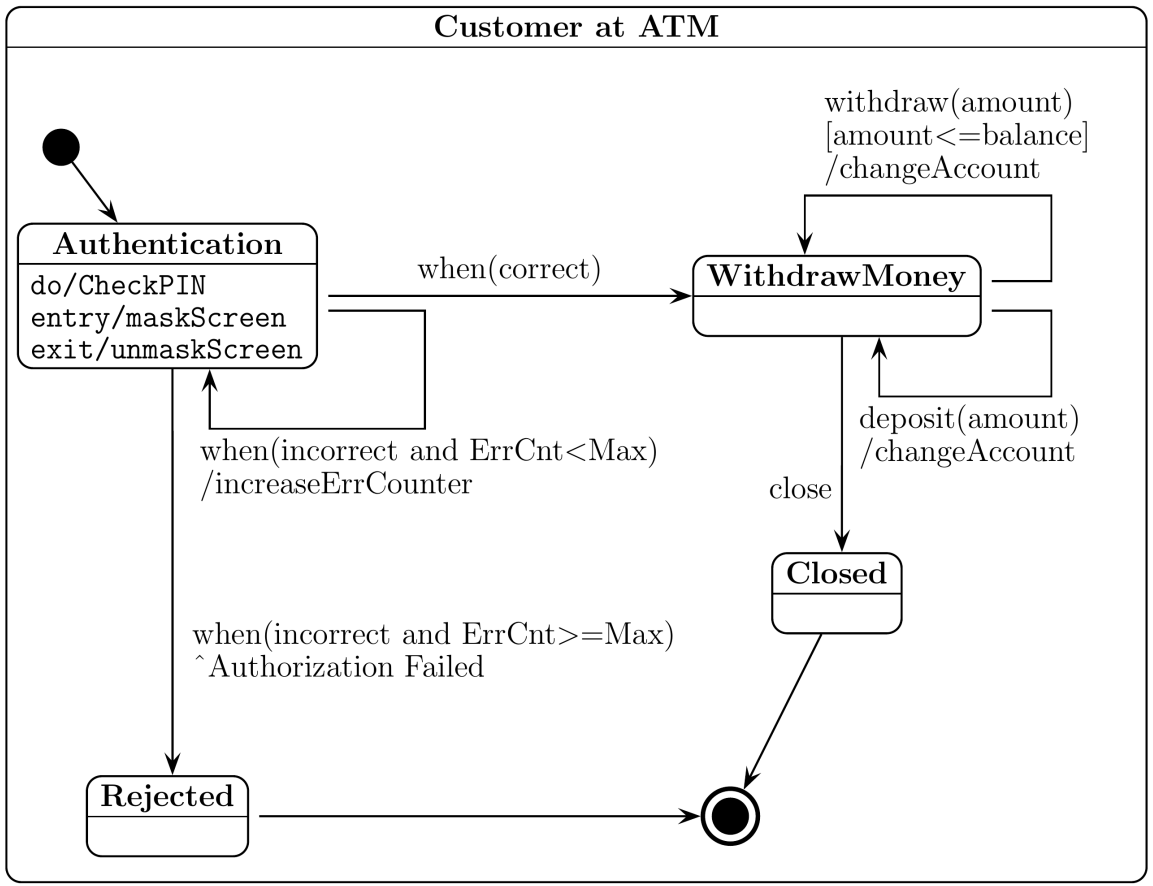
$$((1, A, 0), (1, C, 0), (0, C, 1), \dots) \equiv A \xrightarrow{1/0} C \xrightarrow{1/0} C \xrightarrow{0/1} \dots$$

# Implementation: imperative programming language

```
1  while( (c = read()) != EOF ) {
2      switch( current_state ) {
3      case A: // initial state
4          switch( c ) {
5          case '0': write('0'); next_state = B; break;
6          case '1': write('0'); next_state = C; break;
7          }
8          break;
9      case B: // last input was 0
10         switch( c ) {
11         case '0': write('0'); next_state = B; break;
12         case '1': write('1'); next_state = C; break; // 0 -> 1
13         }
14         break;
15     case C: // last input was 1
16         switch( c ) {
17         case '0': write('1'); next_state = B; break; // 1 -> 0
18         case '1': write('0'); next_state = C; break;
19         }
20         break;
21     }
22     current_state = next_state;
23 }
```

# Implementation: UML State Machine Diagram

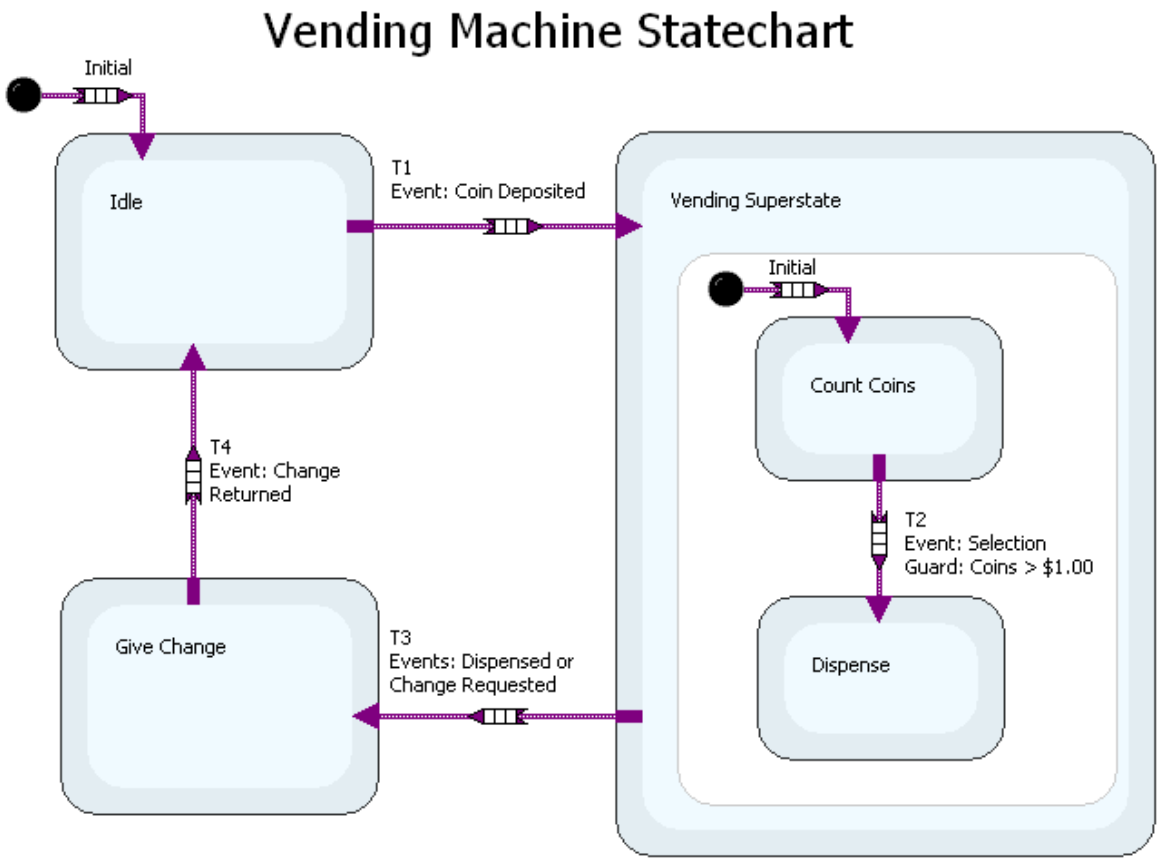
Example: ATM




Reference: http://www.uml-diagrams.org/state-machine-diagrams.html

# Implementation: LabVIEW Statecharts

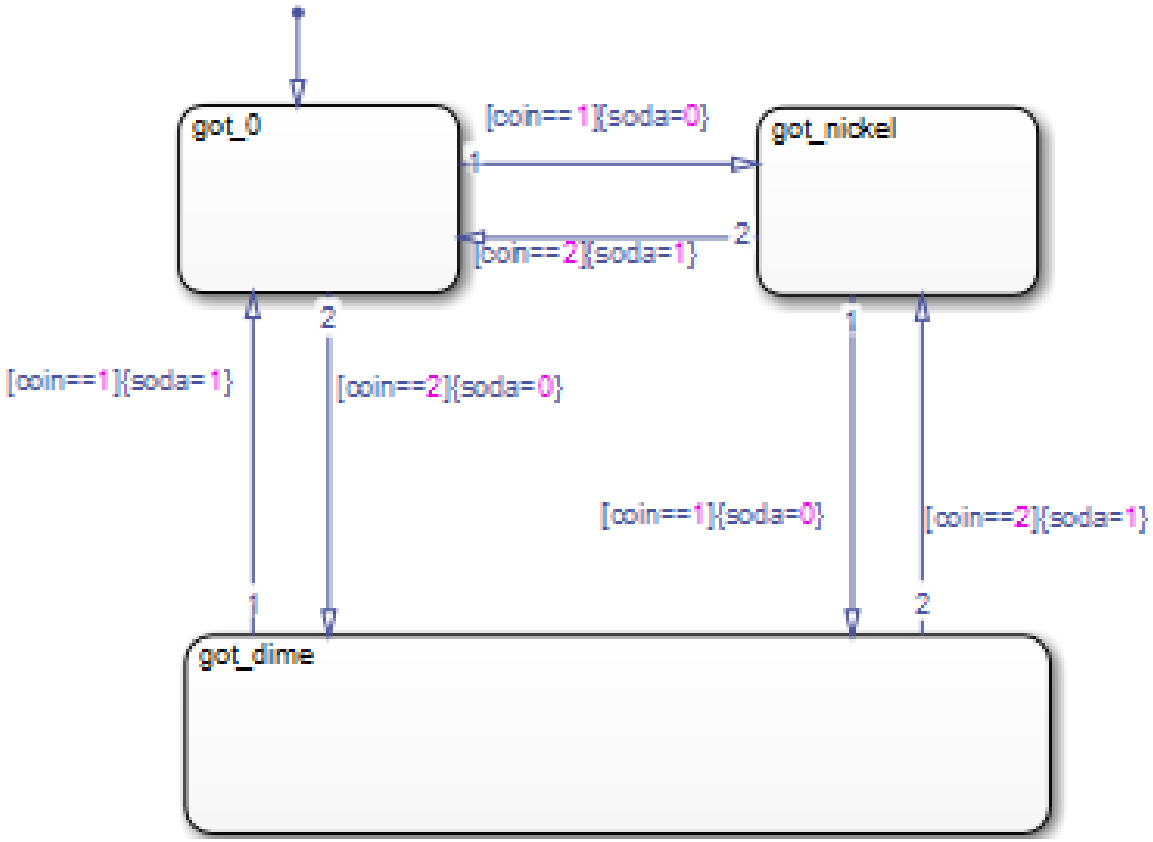
Example: Soda Vending Machine




Statechart Describing a Simple Soda Vending Machine. Source: LabVIEW documentation.

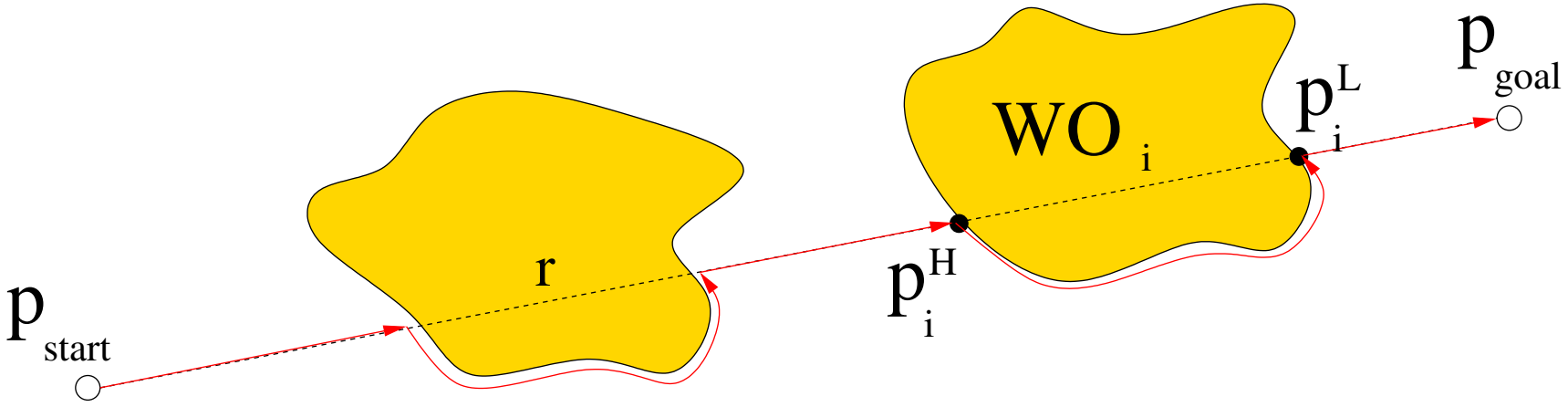
# Implementation: Simulink Stateflow

Example: Soda Vending Machine
$15$ cents required to get a can, nickel (coin $1$) is $5$ cents, dime (coin $2$) is $10$ cents




Source: Matlab-Simulink documentation.

# Exercise: Bug 2 - algorithm overview




## essentials:

- motion-to-goal until an obstacle is encountered
- obstacle circumnavigation until the $r$ straight line is encountered, i.e., the line connecting the starting point and the goal
- at that point, back to motion-to-goal along the $r$ straight line

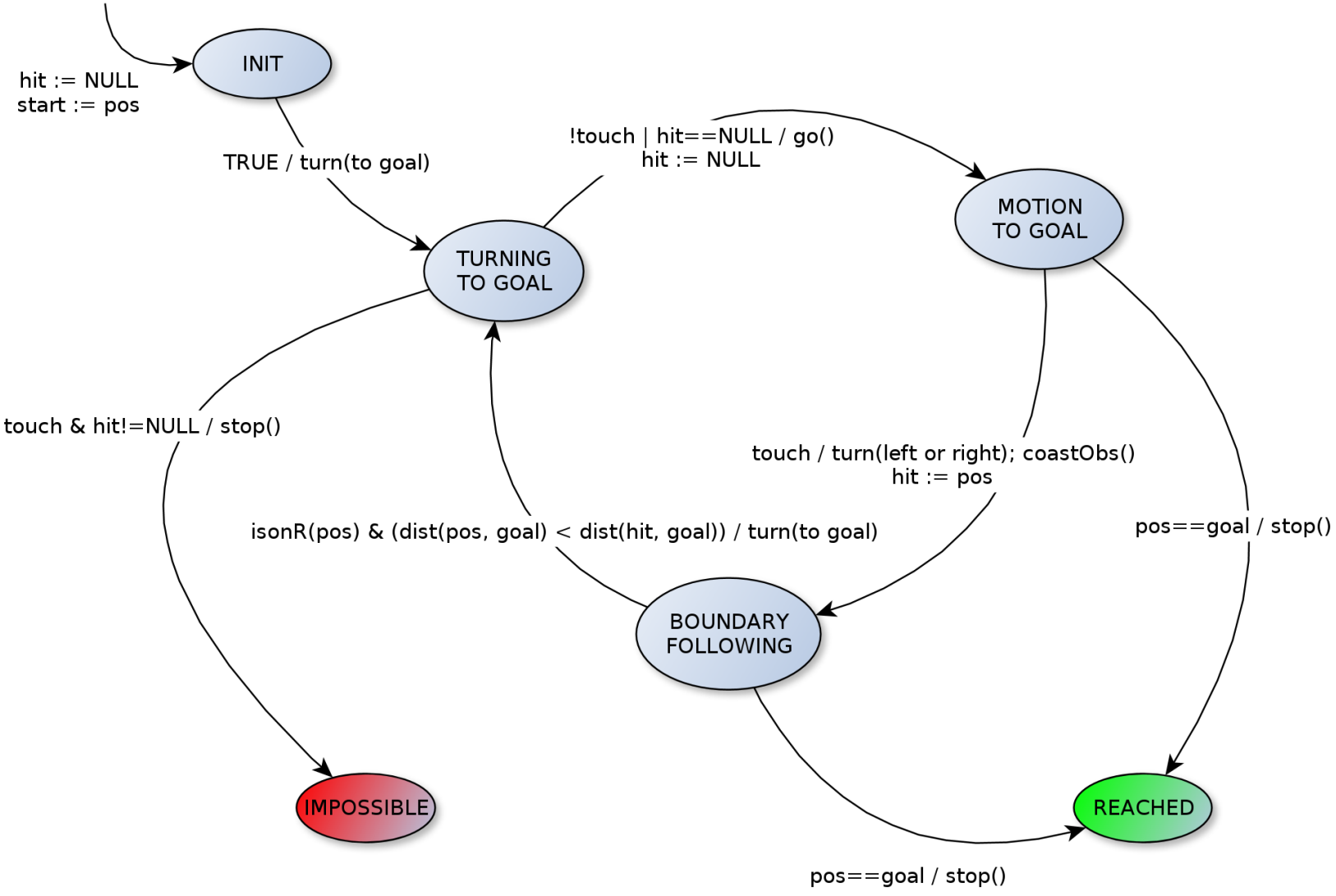
# Exercise: Bug 2 - hypoteses (1/2)

- Hypoteses:
  - discretized workspace : each point belongs to a finite set $W$
  - dist(P1,P2) : is a function that computes the distance between P1 and P2
  - isonR(P1) : is a function that returns true if P1 is on the line $r$
- Input:
  - touch: binary variable set by a proximity sensor in front of the robot
  - pos: variable in $W$, updated by a position sensor
- Output (actions):
  - go() : robot moves along the straight line in front of it
  - turn(...) : robot rotates; the action is instantaneous (simplification)
  - coastObs() : robot proceeds coasting the obstacle
  - stop() : robot stops

# Exercise: Bug 2 - hypoteses (2/2)

- State variables:
  - hit : variable in $W \cup \{NULL\}$, which stores the hit point
  - start : variable in $W$, which stores the starting point. It is necessary for calculating the line start-goal
- Parameter:
  - goal : constant in $W$

# Exercise: Bug 2 - Mealy FSM



# Composition of State Machines

## The problem of complex systems

- State machines provide a convenient way to model behaviors of systems.
- One disadvantage that they have is that for most interesting systems, the number of states is very large, often even infinite.
- Automated tools can handle large state spaces, but humans have more difficulty with any direct representation of a large state space.

A time-honored principle in engineering is that complicated systems should be described as **compositions of simpler systems**

## The problem of complex systems

- there are many different ways to compose state machines
- compositions that look similar on the surface may mean different things to different people
- the rules of notation of a model are called its syntax, and the meaning of the notation is called its semantics
- the same syntax can have many different semantics, which can cause no end of confusion

Beware that in the literature and in softwares there exist many syntaxes, many semantics and even many semantics for the same syntax!

# Different types of composition

We consider:

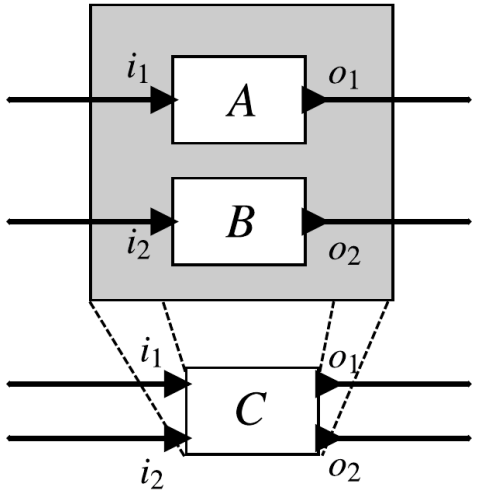
- concurrent composition
  - synchronous
  - asynchronous
- hierarchical composition

## Concurrent composition

Two or more machines react either simultaneously or independently.

- Simultaneous reactions = synchronous model
- Independent reactions = asynchronous model

# Concurrent composition: side-by-side synchronous




- Input and output are disjoint.
- A reaction of $C$ is a **simultaneous** reaction of $A$ and $B$.
- Modular composition $=$ the composition itself can become a component of further compositions.
- $C$ is itself a FSM.
- Determinacy is a compositional property.

## Concurrent composition: side-by-side asynchronous

In an asynchronous composition of FSM, the component machines react **independently**.

Different semantics: *A reaction of $C$ is a reaction of *, where the choice is **.*

| | *$A$ or $B$ | *$A$, $B$ or both |
|---|---|---|
| **nondeterministic | 1 | 2 |
| **made by the environment | 3 | 4 |

- 1, 3 are interleaving semantics ($A$ and $B$ never react at the same time)
- In semantics 1, 2 determinacy is not a compositional property
- In semantics 3, 4 a composition has to provide a scheduling policy
- Inputs may be completely missed

## Concurrent composition: shared variables

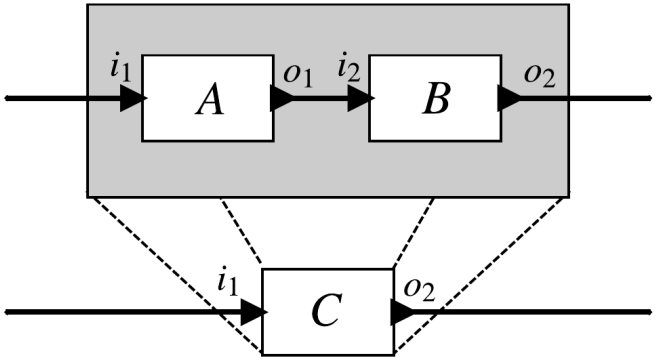
An extended state machine has **local variables** that can be read and written as part of taking transitions. Sometimes it is useful when composing state machines to allow these variables to be **shared** among a group of machines.

Many complications arise.

- What is the meaning when both machines update the shared variables?
- What should happen if in the same reaction one machine reads a shared variable to evaluate a guard and another machine writes to the shared variables?
- What if the transition doing the write to the shared variable also reads the same variable in its guard expression?

Clean solutions require a more sophisticated semantics of concurrent models of computation, like the **synchronous-reactive model**, which gives a synchronous composition semantics that is reasonably compositional.

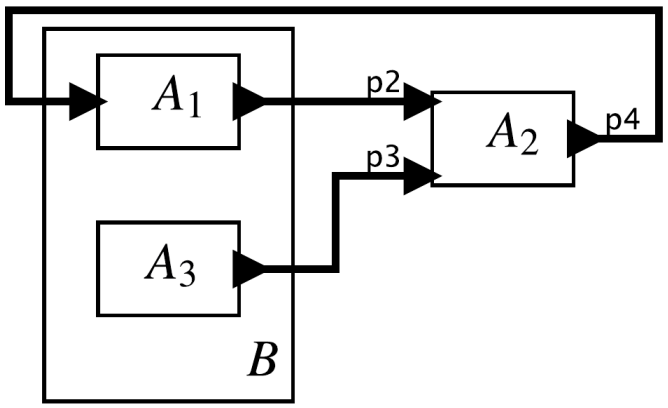
# Concurrent composition: cascade




- **Type check**: any output produced by $A$ must be an acceptable input to $B$.
- Asynchronous:
  - some machinery for **data buffering** from $A$ to $B$
- Synchronous:
  - A reaction of $C$ is a reaction of both $A$ and $B$, which are **simultaneous**, **instantaneous** and **causally related** (outputs of $A$ can affect behavior of $B$).

## Concurrent composition: feedback

Side-by-side (= parallel) and cascade (= series) composition provide the basic building blocks for building more complex composition of machines.

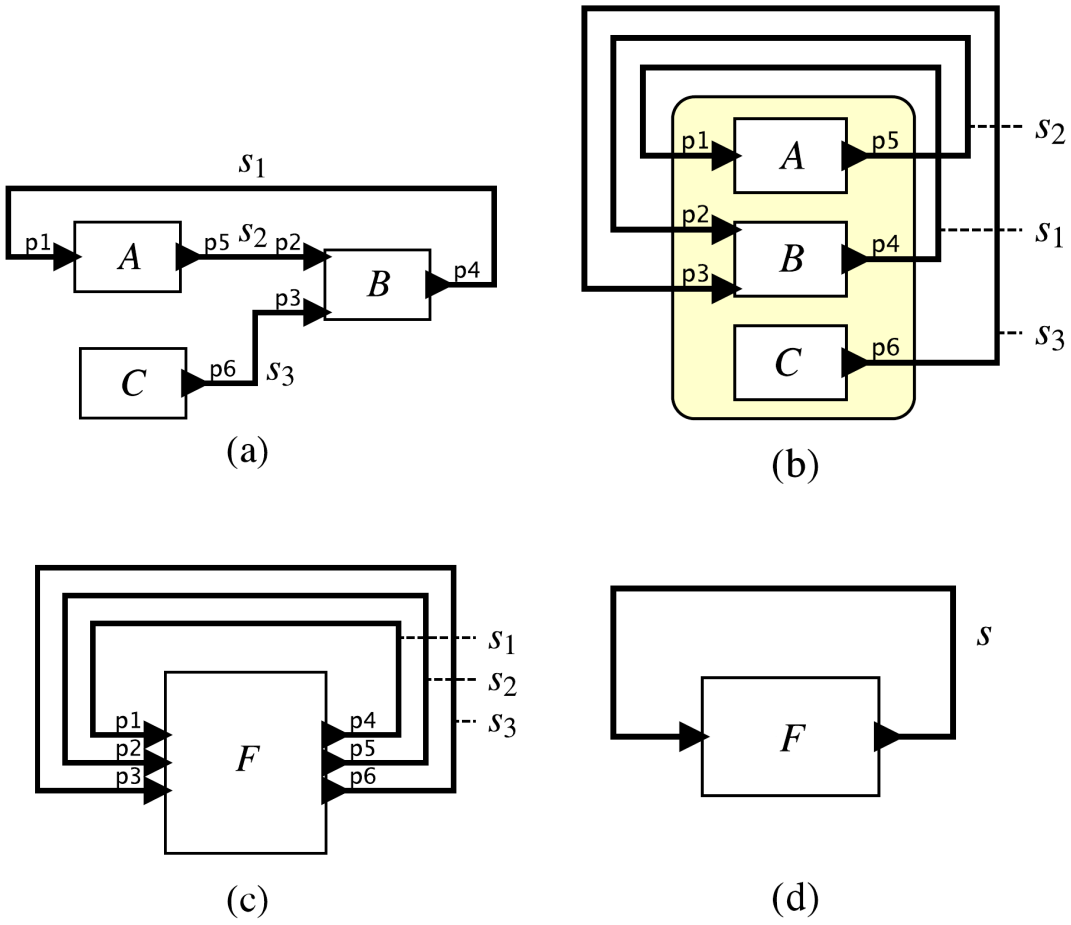



How do we resolve cycles?

Using the **fixed point semantics**.

# Concurrent composition: feedback

Any network of actors can be reduced to a side-by-side composition with feedback.



## Concurrent composition: feedback

If the actors are determinate then each actor is a function that maps input sequences to output sequences (not input symbols to output symbols).
The **semantics of such a feedback model** is a system of equations and the reduced form of Figure (d) becomes

$$s = F(s)$$

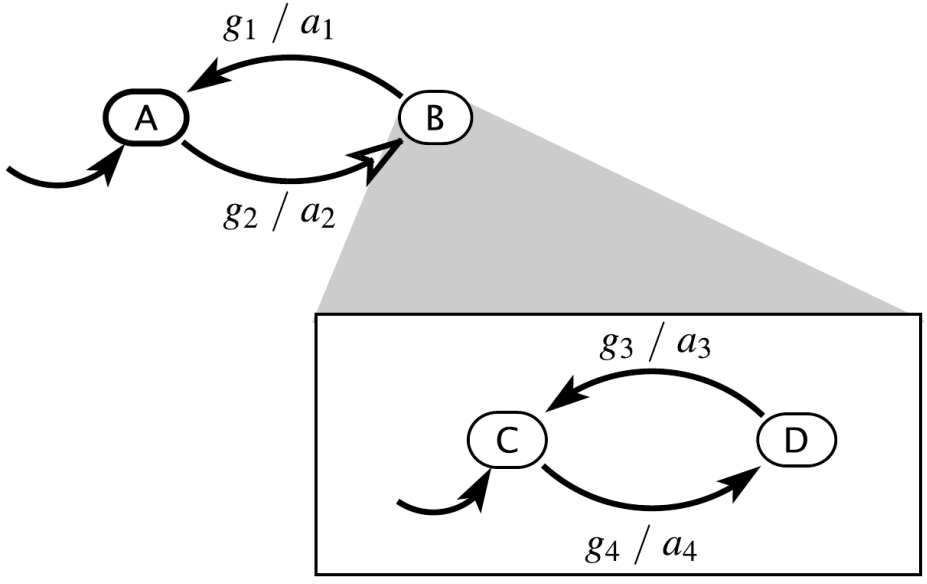
where $s$ is the **fixed point** of the function $F$.

> The semantics of a determinate actor network is a fixed point.

The existence of a fixed point, its uniqueness and methods to find it are very interesting topics, but they are out of the scope of this course. Check `leeseshia.org` for more details.

## Hierarchical composition

The key idea in hierarchical state machines is **state refinement**.

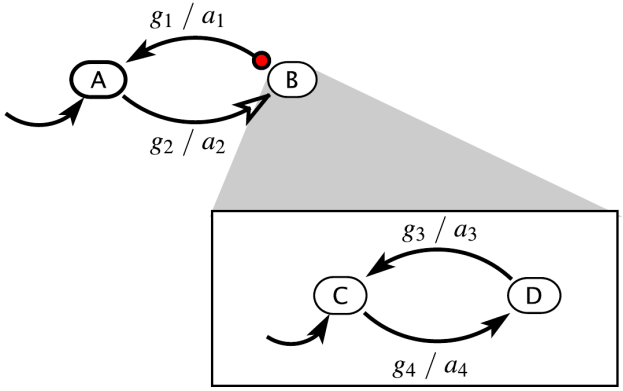


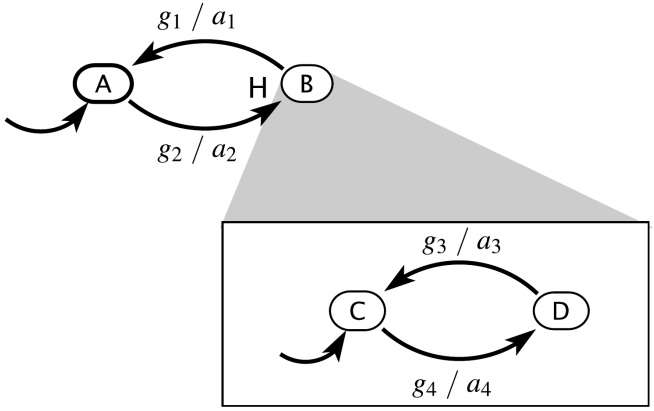

What if the machine is in the state $C$, and $g_1$ and $g_4$ become true at the same time?

Proliferation of different variants

## Hierarchical composition

**Depth-first semantics**: the deepest refinement of the current state react first, then its container state machine, then its container, etc.



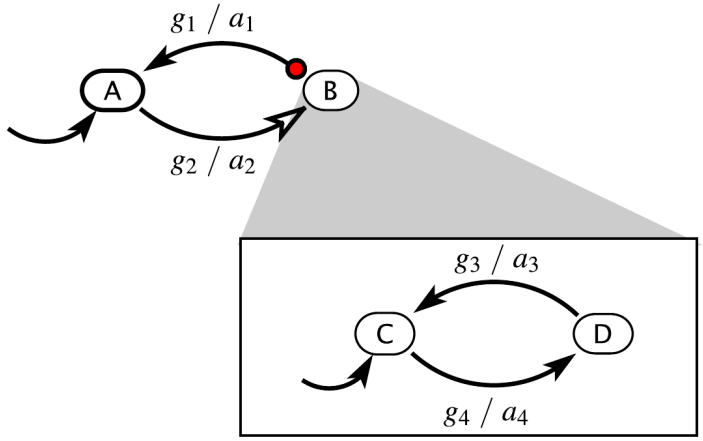




**Preemptive transitions**: its guard is evaluated before the refinement.

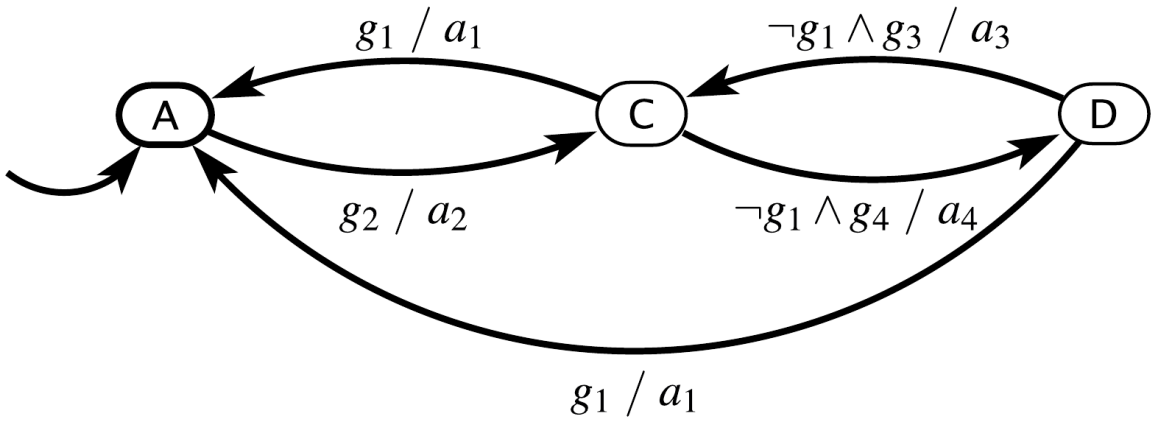
**Reset transitions vs. history transitions**: when a historian transition is taken, the destination refinement resumes in whatever state it was last in.

# Exercise: hierarchical FSM

Considering the following hierarchical state machine




Build an equivalent flat FSM with preemptive transitions semantics.



# Basic aspects of hybrid systems

## Hybrid systems

Hybrid systems combine both **discrete** and **continuous** dynamics.

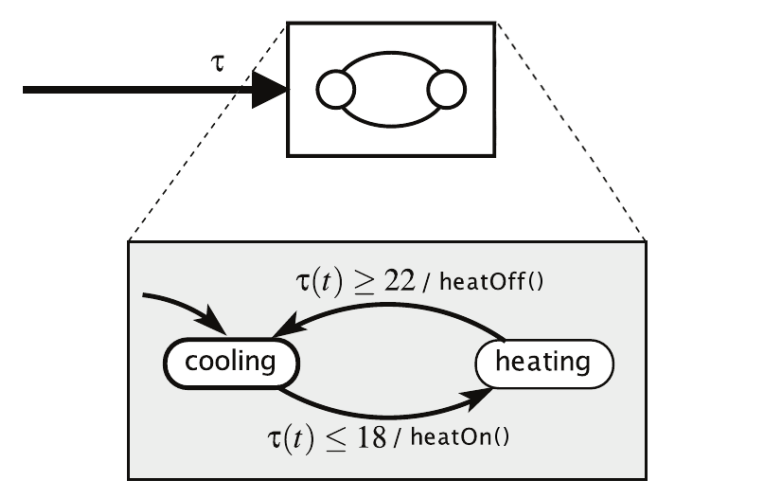
Hybrid system models are often much simpler and more understandable than “brute-force“ models that constrain themselves to only one of the two styles.

Hybrid systems are a powerful tool for understanding and modeling real-world systems.

# FSM with continuous input

We have so far assumed that state machines operates in a sequence of discrete reactions. The extended FSM model with guards on transitions can **coexist with time-based models**. We need to interpret state transitions to occur, instantly, on the same timeline used for the time-based portion of the system.

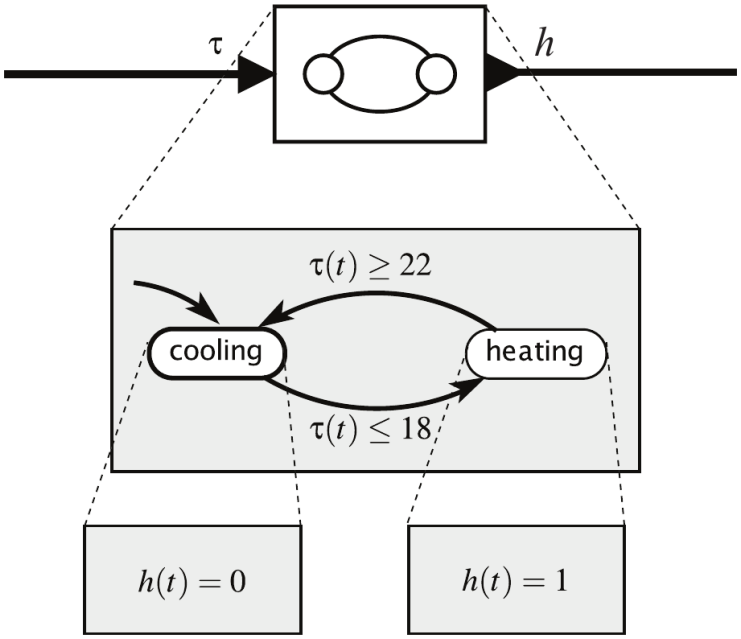
Example Consider a thermostat modeled as a FSM with a continuous time input $\tau : \mathbb{R} \to \mathbb{R}$ where $\tau(t)$ represents the temperature at time $t$.



# FSM with continuous output

In a hybrid system, the current state of the state machine has a **state refinement** that gives the dynamic behavior of the output as a function of the input.

Example Consider the thermostat and suppose to produce a continuous control signal whose value is $1$ when the heat is on and $0$ when the heat is off.



## Modes vs. states

A hybrid system is sometimes called **modal model** because it has finite numbers of **modes**.

The states of the FSM may be referred to as modes rather than states, which help prevent confusion with state variables of the dynamic system.

**Timed automata** (Alur and Dill, 1994) are the simplest non-trivial hybrid systems. They are modal models where the time-based refinements have very simple dynamics; all they do is measure the passage of time.

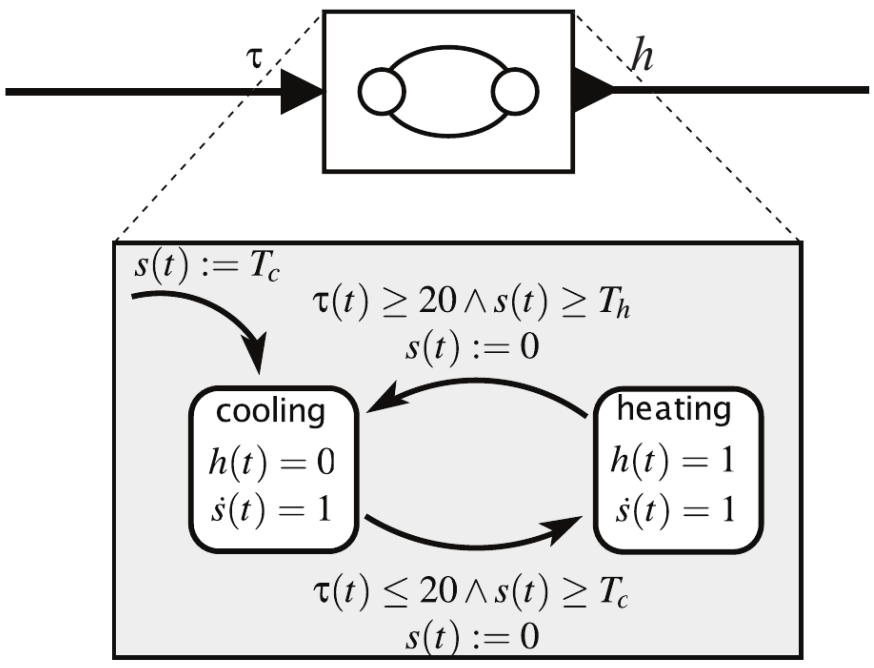
A clock is modeled by a first-order differential equation,

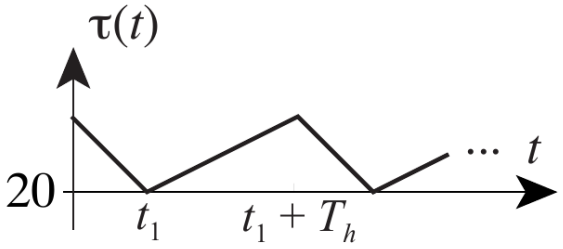
$$\forall t \in T_m, \quad \dot{s}(t) = a,$$

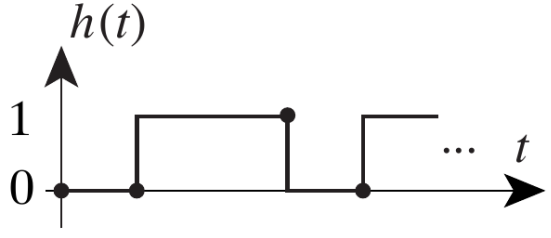
where $s : \mathbb{R} \to \mathbb{R}$ is a continuous-time signal, $s(t)$ is the value of the clock at time $t$, and $T_m \subset \mathbb{R}$ is the subset of time during which the hybrid system is in mode $m$. The rate of the clock, $a$, is a constant while the system is in this mode.

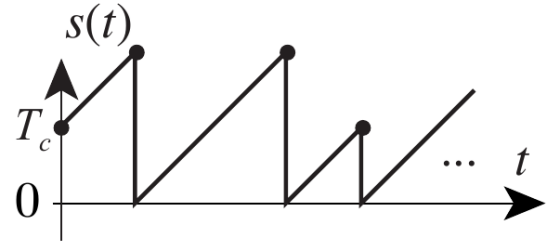
# Timed automaton example

An alternative implementation of a thermostat is to use a **single temperature threshold** and require that the heater remain on or off for at least a **minimum amount of time**, regardless of the temperature.



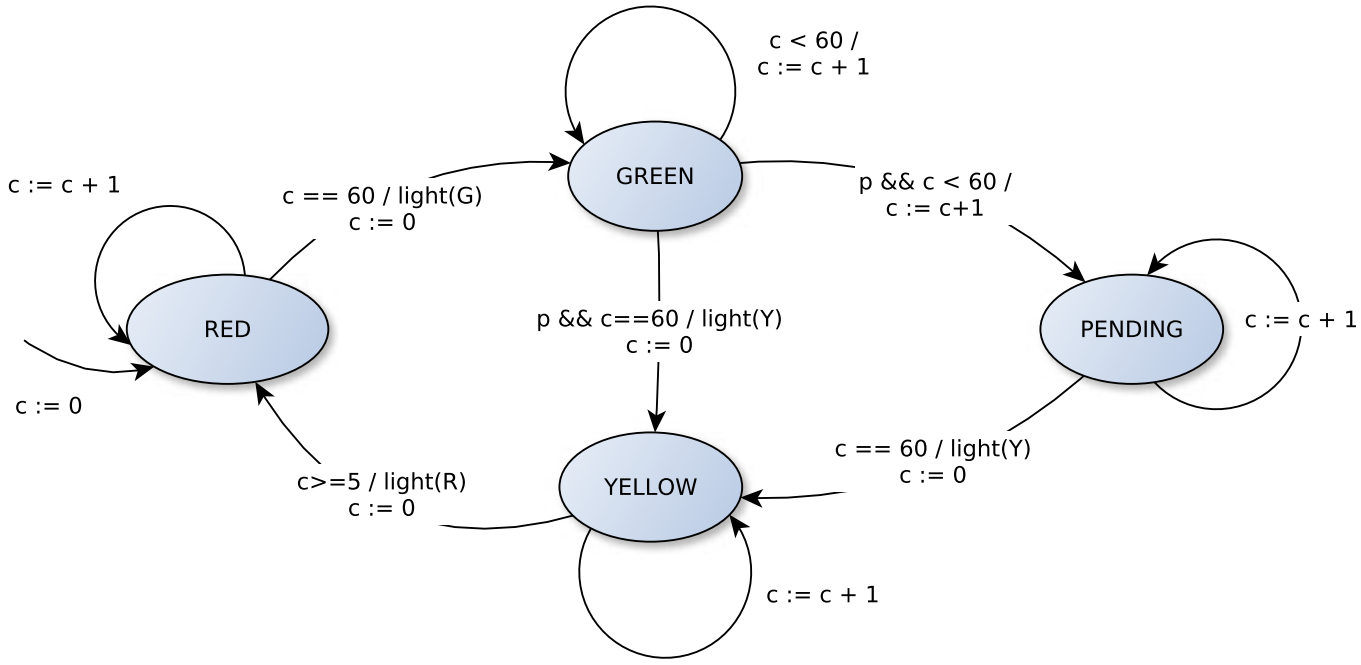



# Exercise: Traffic light controller as timed automaton

Recall the traffic light controller example. We designed a time-triggered FSM that assumes it reacts once each seconds.

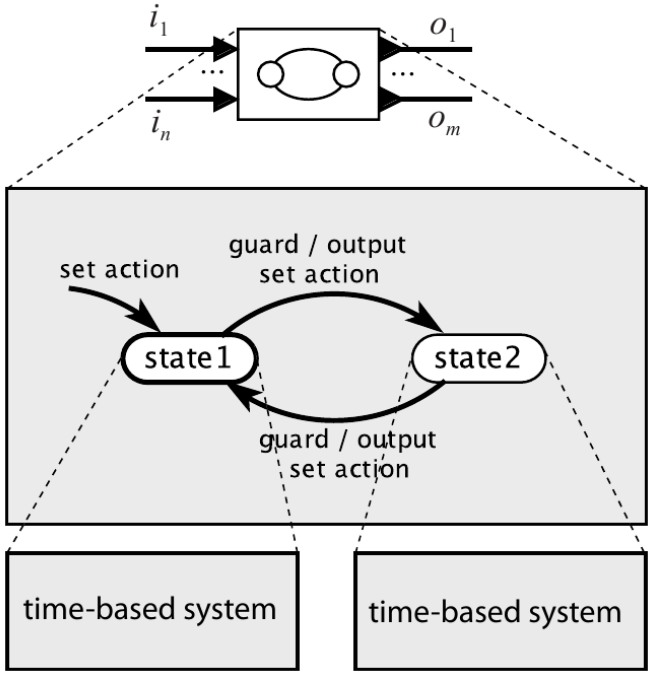



**Exercise** Re-design it as a timed automaton.

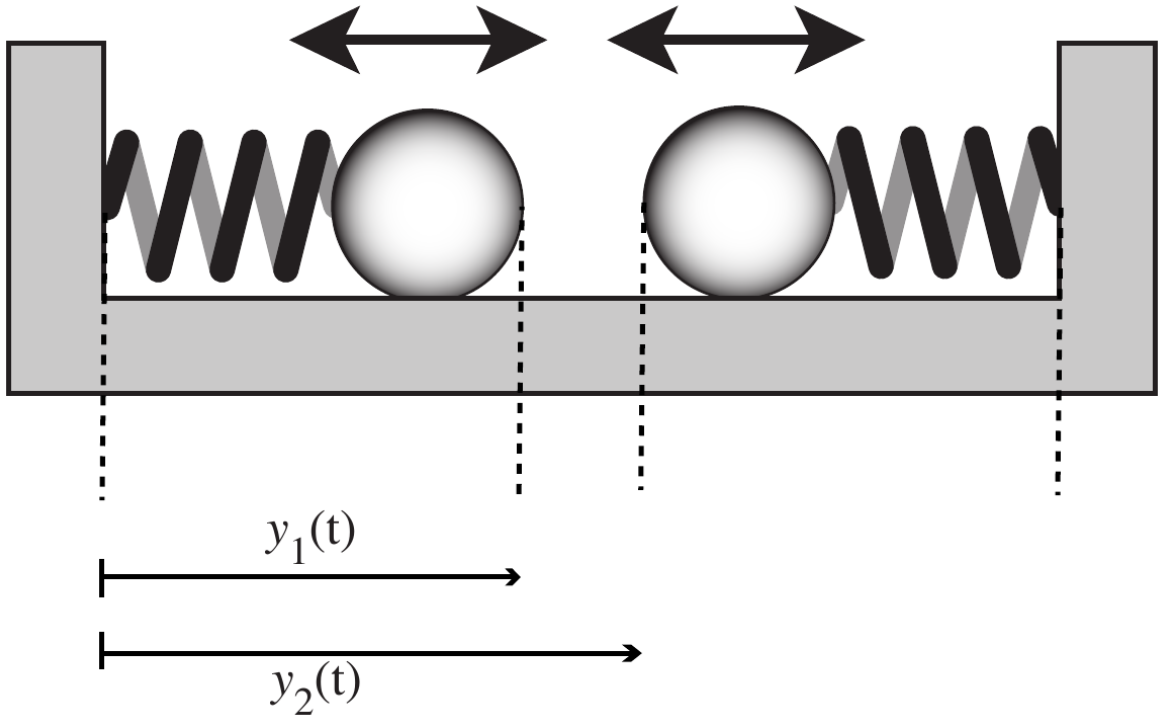
# Higher-order dynamics: hybrid automata

In timed automata, all that happens in the time-based refinement system is that time passes. Hybrid systems, however, are much more interesting when the behavior of the refinements is more complex.

We refer to this systems as **hybrid automata**.

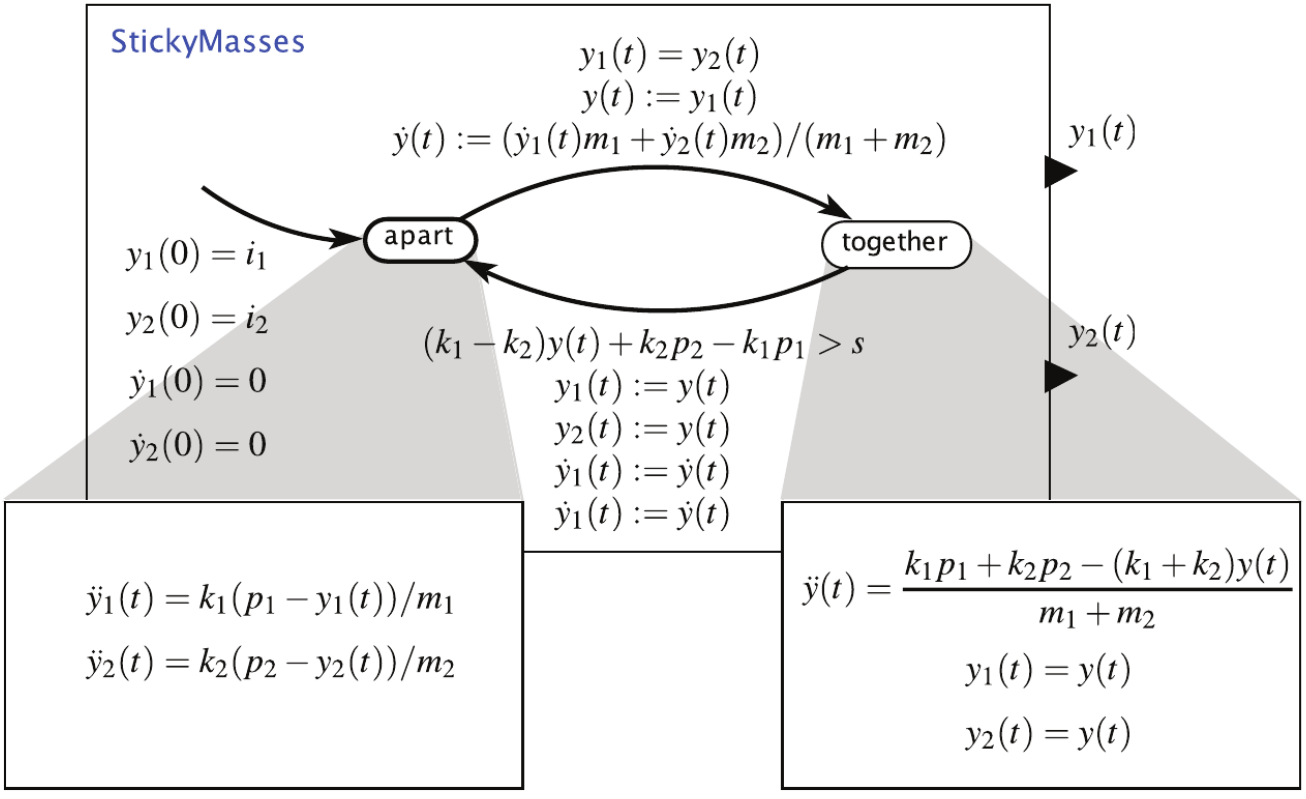


# Example: sticky masses - problem definition



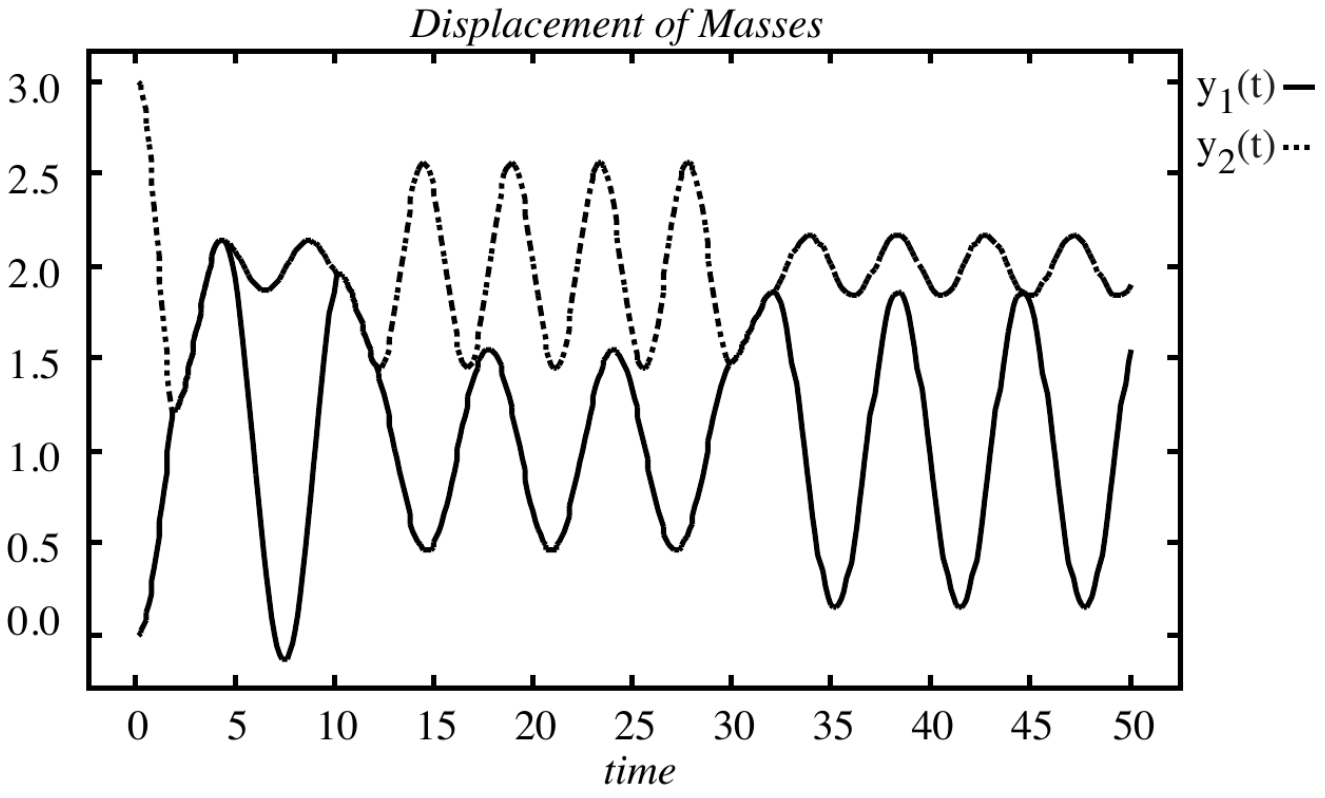

Two stick masses are attached to springs. The masses oscillate on a frictionless table. If they collide, they stick together and oscillate together. After some time, the stickiness decays when the pulling forces exceeds the stickiness force $s$, and masses pull apart again.

# Example: sticky masses - system model



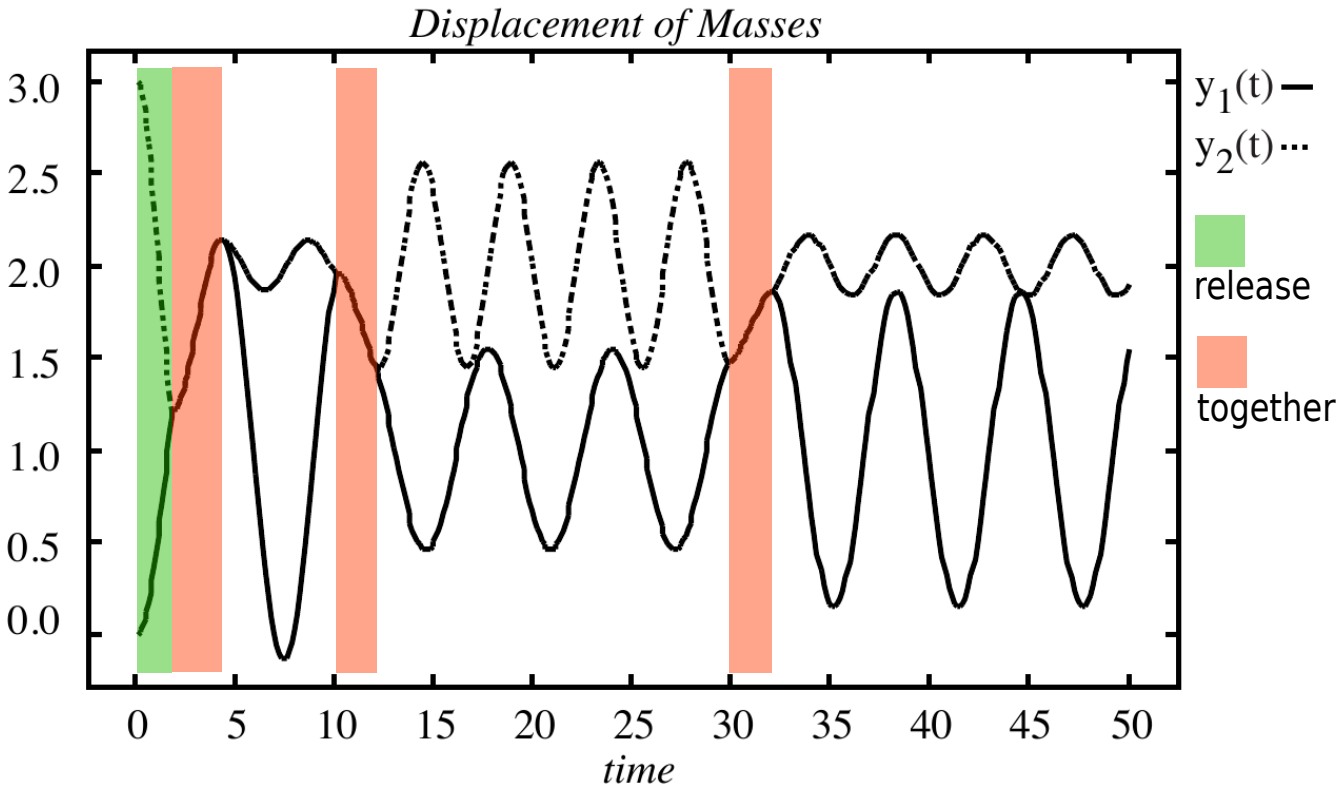

- $p_1$ and $p_2$ denote the neutral position of the two springs, i.e. where the elastic force is zero.

# Example: sticky masses - behavior




- at start, the two springs are completely compressed

# Example: sticky masses - behavior




- at start, the two springs are completely compressed

References:


- E.A. Lee and S.A. Seshia, *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*, LeeSeshia.org, 2011. `http://leeseshia.org`
- J.E. Hopcroft, R. Motwani and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, 2003. `http://infolab.stanford.edu/~ullman/ialc.html`