Artificial Intelligence for Cyber-Physical Systems

Laura Nenzi

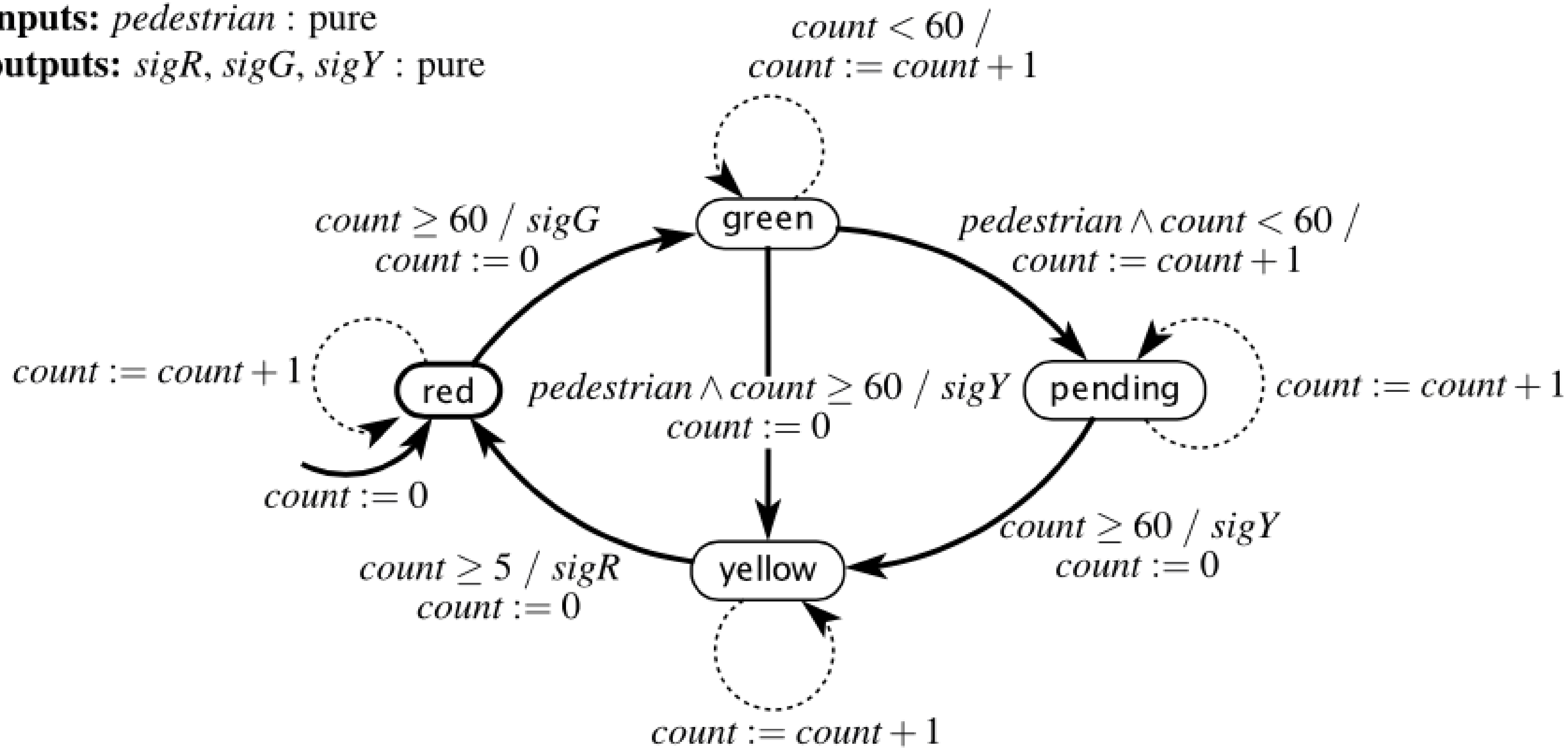
Università degli Studi di Trieste

I Semestre 2024

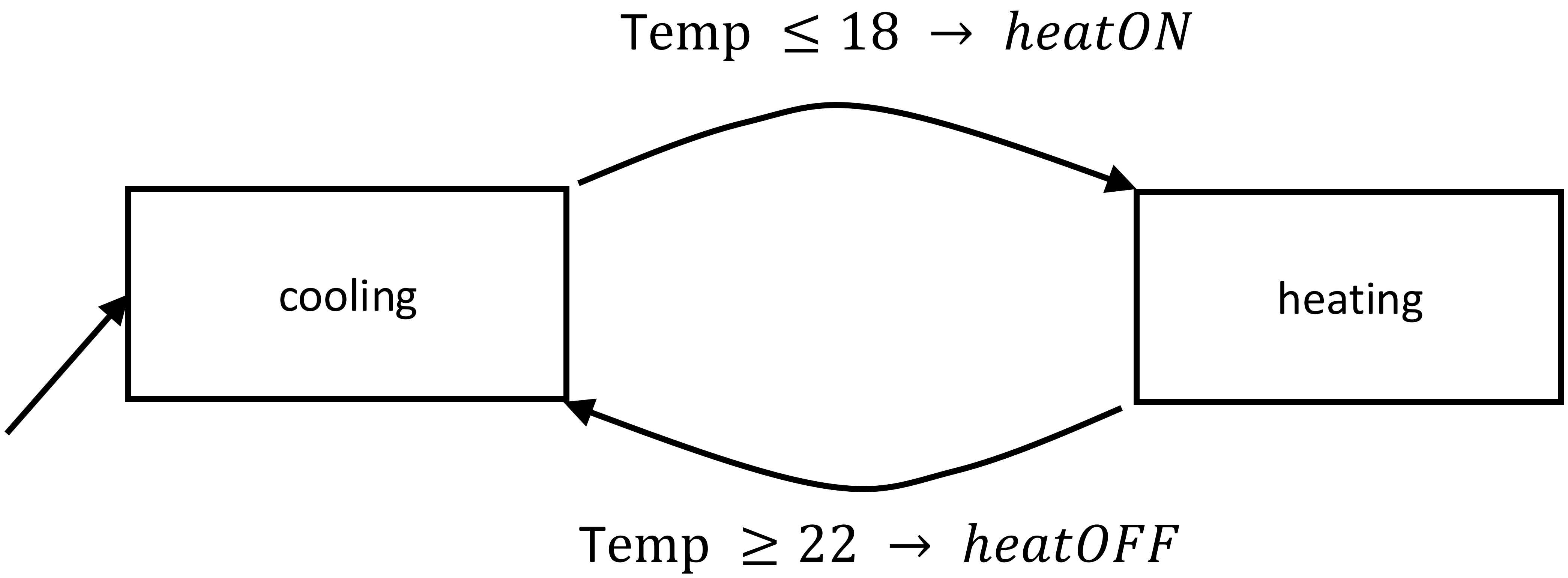
Lecture 4-5: Timed and Hybrid Models

Time Trigger Machine

variable: *count*: $\{0, \cdots, 60\}$
inputs: *pedestrian* : pure
outputs: *sigR*, *sigG*, *sigY* : pure



Thermostat FSM

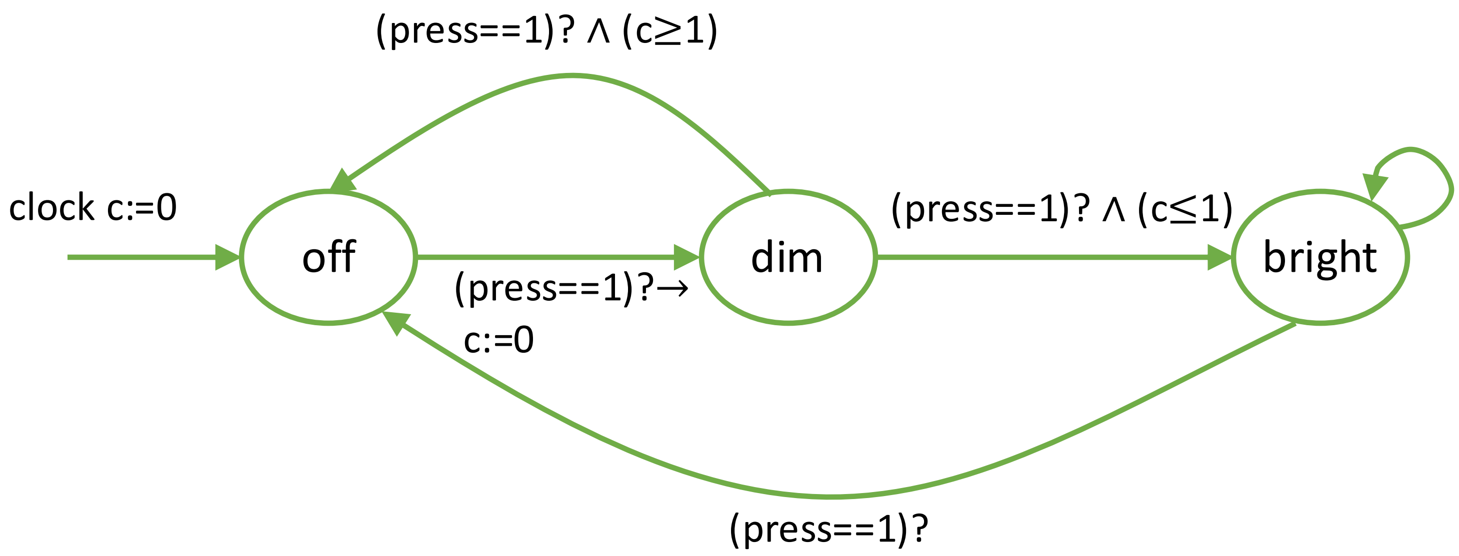


It could be event triggered, like the garage counter, in which case it will react whenever a *temperature* input is provided. Alternatively, it could be time triggered, meaning that it reacts at regular time intervals

Timed Models

- Like Asynchronous models, but with explicit time information
- Can make use of global time for coordination

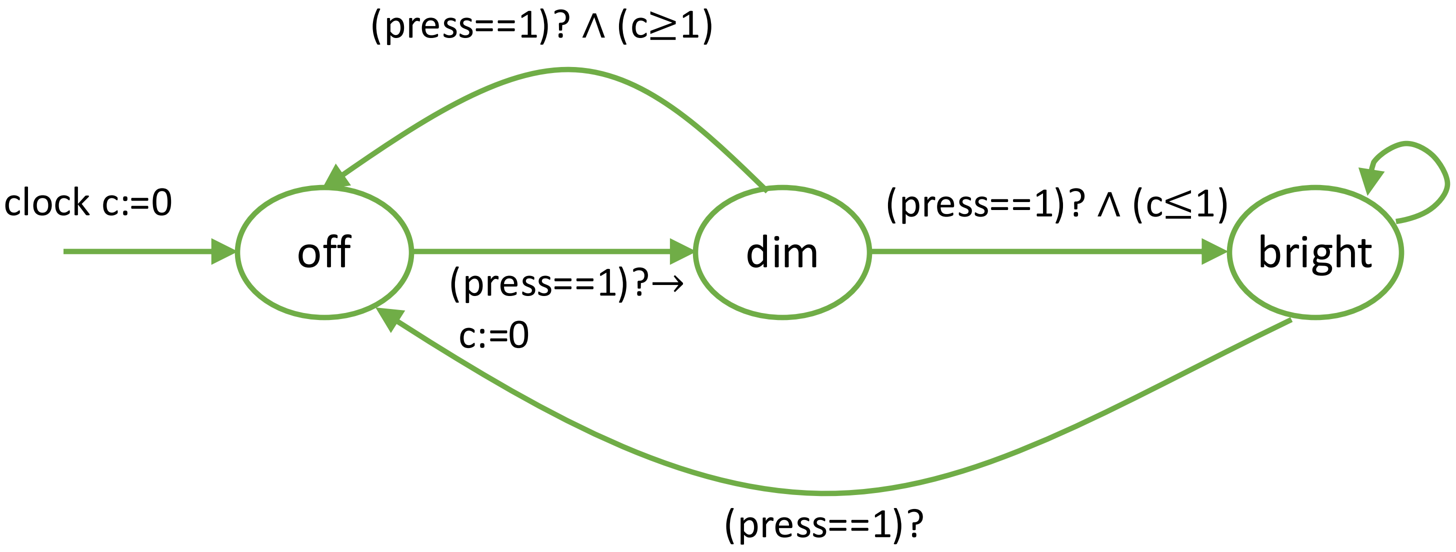
Timed ESMs: a Light Switch

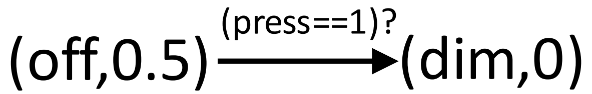


Like asynchronous ESMs, have input, output channels, state variables

- Special type of state variable called "clock"
- Clock variables evolve continuously in time
- ESM can "stay" in a mode with clock increasing monotonically from the start value

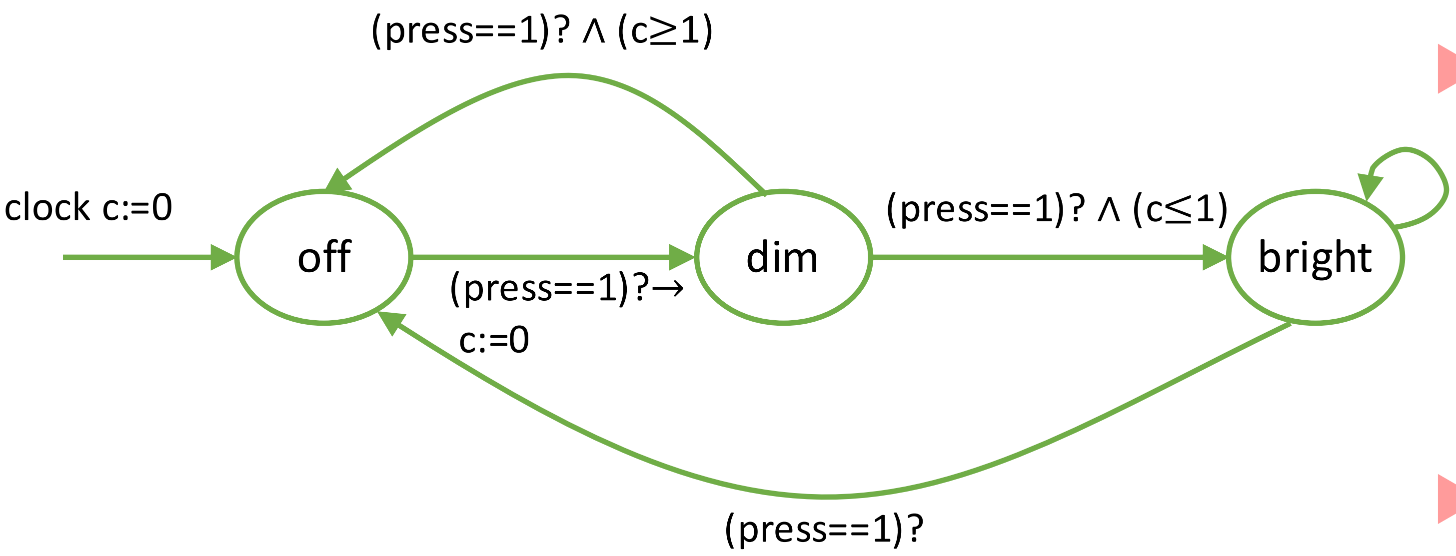
Transitions of a timed ESM





- Mode switch: discrete action
  - machine moves from one mode to another
  - guard on the transition must be true for mode switch to occur
  - update specified by the transition will update/reset clock variables

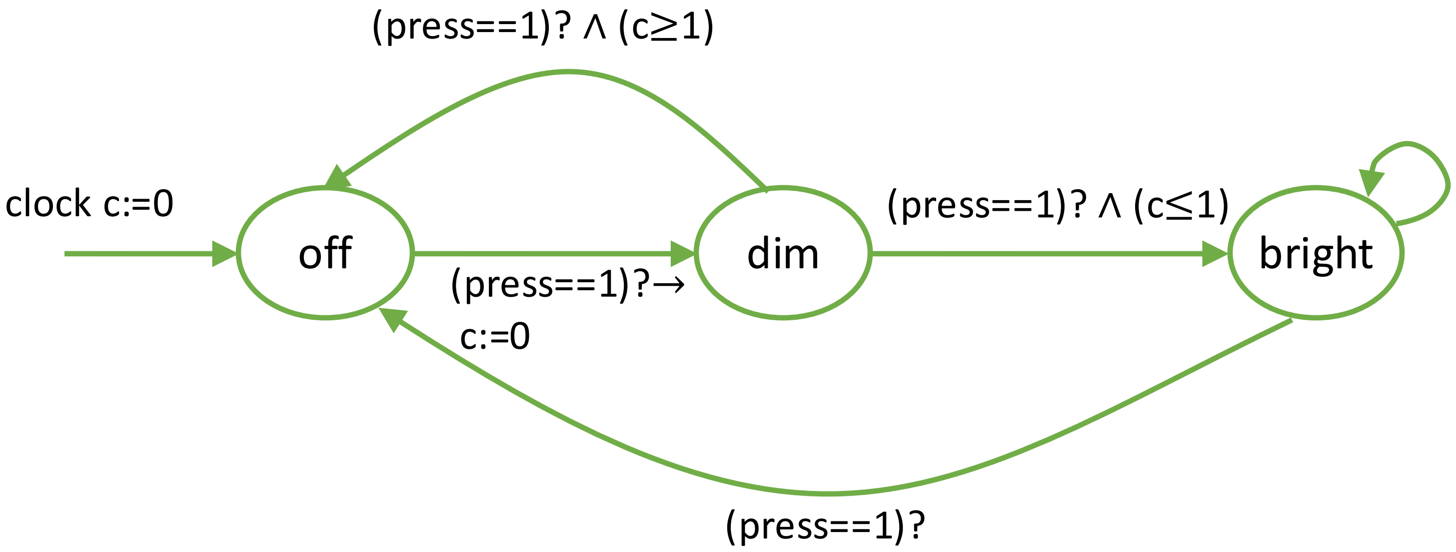
Transitions of a timed ESM



In a mode: Timed action

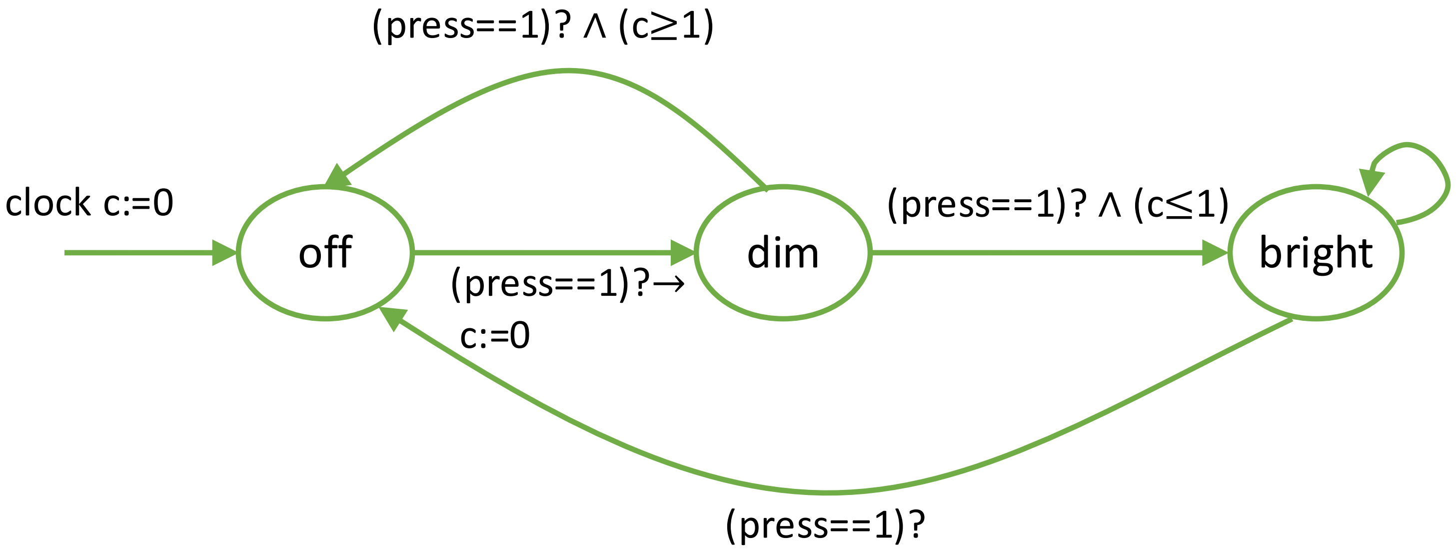
- When machine stays in any given mode for time δ, each clock variable increases by δ and all other state variables remain unchanged
- Captures timing constraints
  - Resetting c to 0 from off→dim and guard c≥1 from dim→off specifies that these mode switches are ≥1

Timed Processes: explicit clock variables

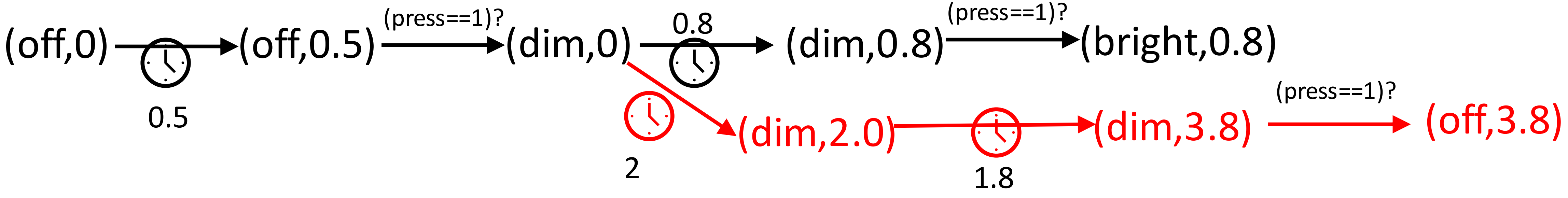


- Clock variables
  - Like other state variables, can be used in guards
  - Can be reset to 0 during mode switches
  - When the machine is in a given mode for duration δ, the clock variable increases by δ

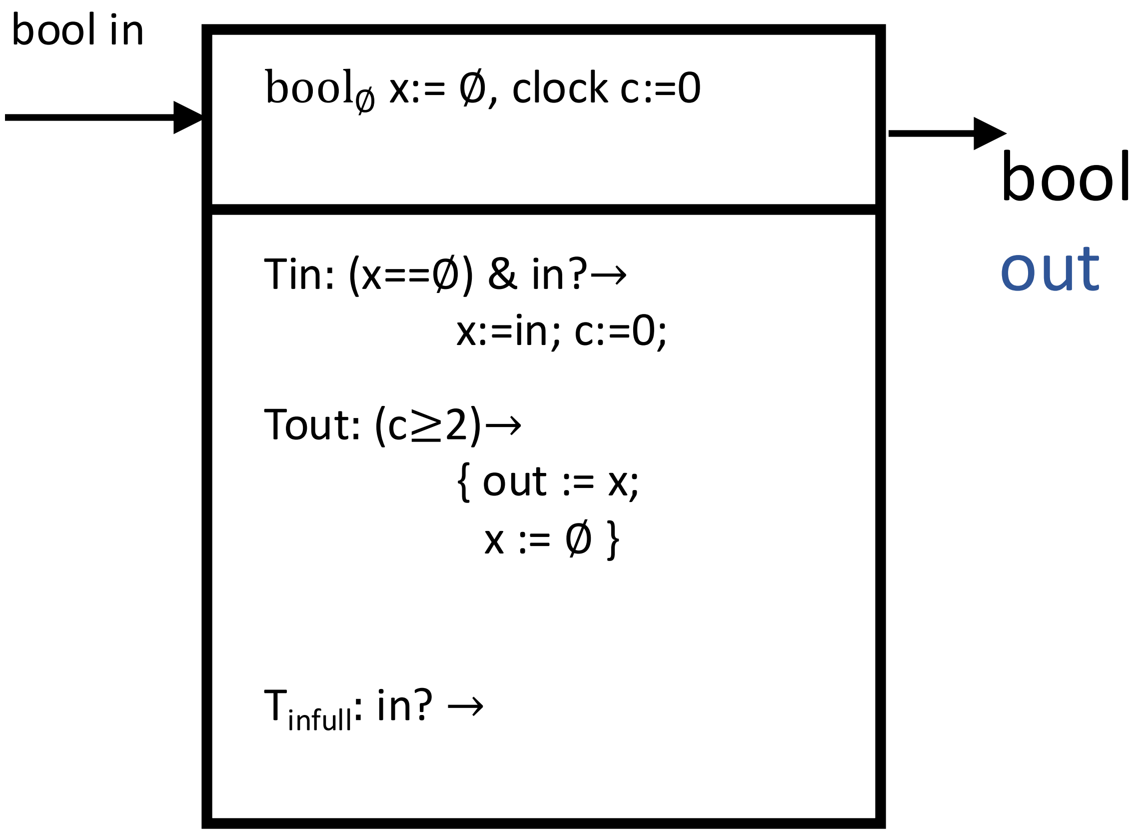
Timed Process Execution



- Machine execution is through alternating timed transitions and mode switches

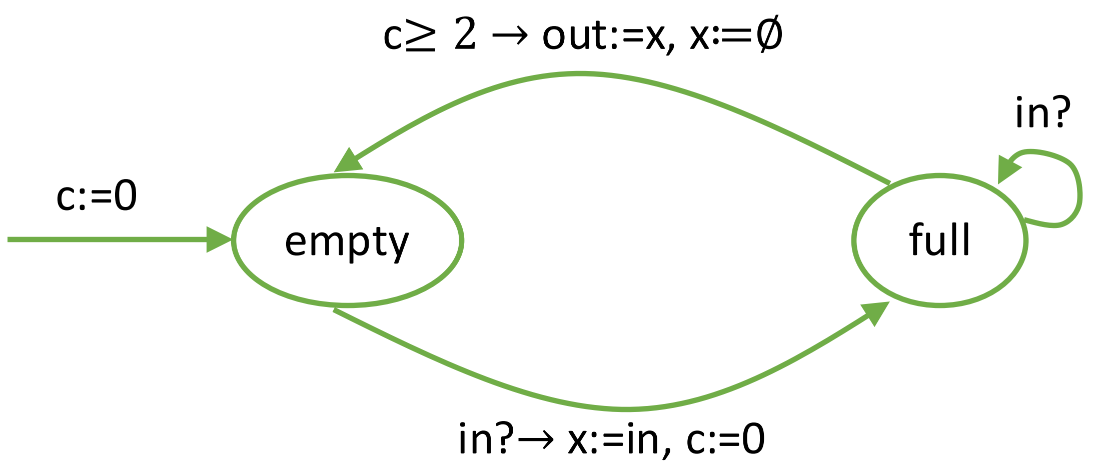


Timed Buffer



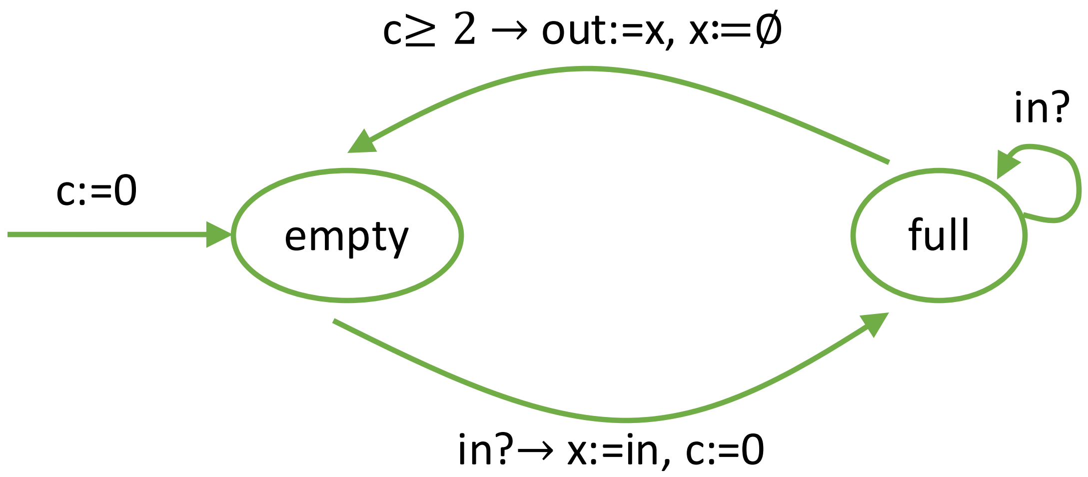
- Input channel in of type bool
- Output channel out of type bool
- State variable x of type bool+∅. The value ∅ indicates empty
- If x is ∅, then read new value into x, and set clock to 0
- If clock value is ≥ 2 seconds, output value of x, and set x to ∅

Timed State Machine representation



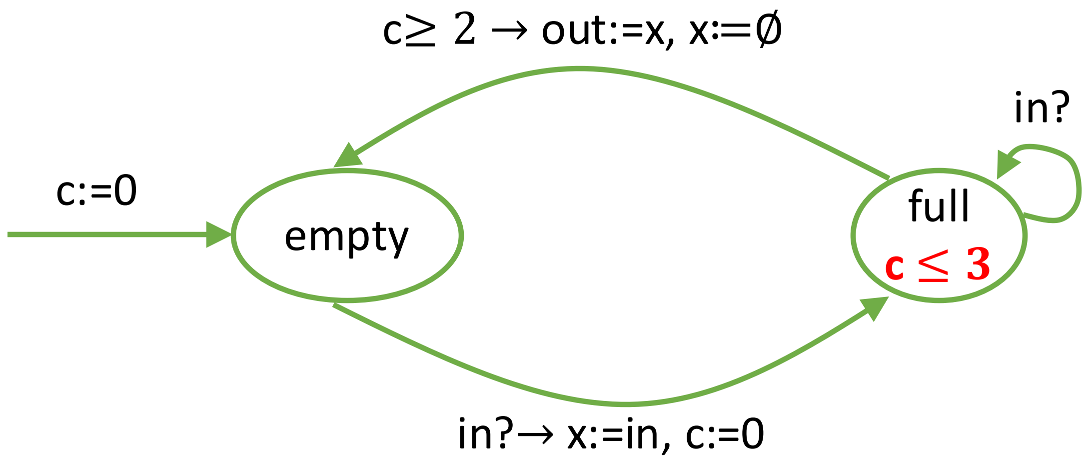
- Mode captures whether x==∅
- Clock variable tracks the time that elapsed since x received a value
- Guard ensures that at least 2 seconds pass before the value of x is output
- Guard ***does not force*** transitions
  - c can keep increasing while process remains in mode full
- How do we make sure that process does not remain in full mode for at most 3 seconds?

Clock invariants



- Attempt 1: we could make the guard $2 \leq c \leq 3$
- Attempt 1 fails because:
  - You could keep getting new input (self-loop executes) till $c \geq 3$
- Larger problem: Guards are non-forcing: nothing requires the guard to be executed
- We can fix this by introducing **clock invariants**
- Clock invariant of any mode: symbolic expression that must evaluate to true at all times, and if not, the process must exit that mode

Clock invariants

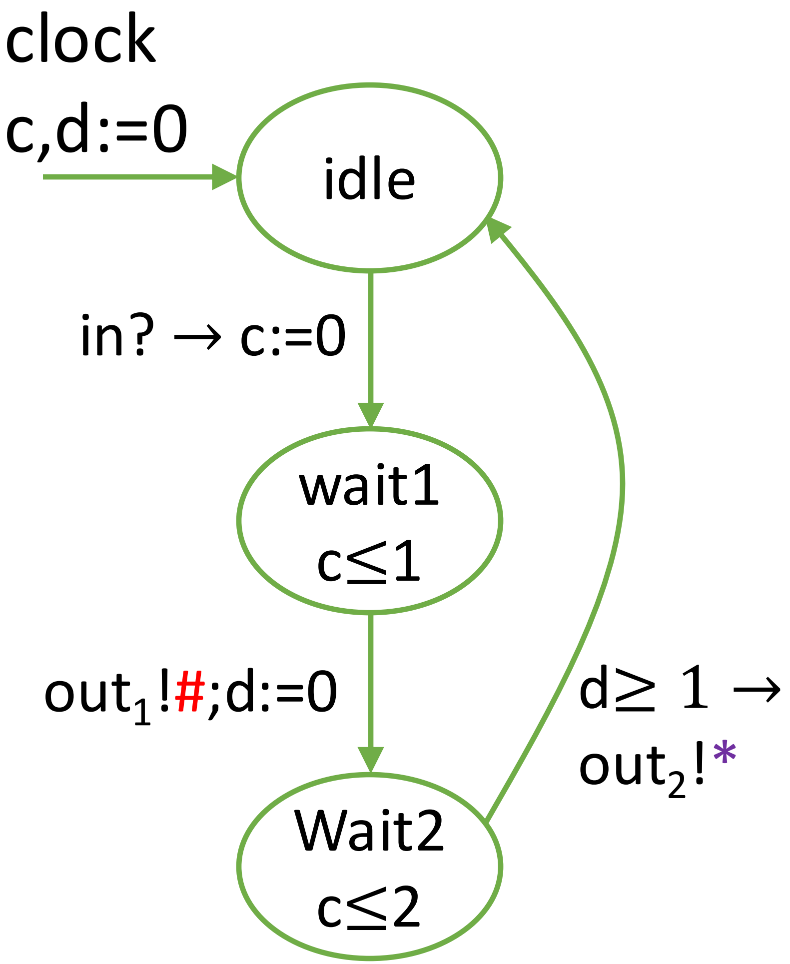


- Add clock invariant:

  $(\text{mode}==\text{full}) \Rightarrow (c \leq 3)$

- Forces process to leave mode full if c becomes greater than 3
- Staying in mode full when c≥ 3 would violate the clock invariant
- Useful construct to limit how long a process stays in a certain mode

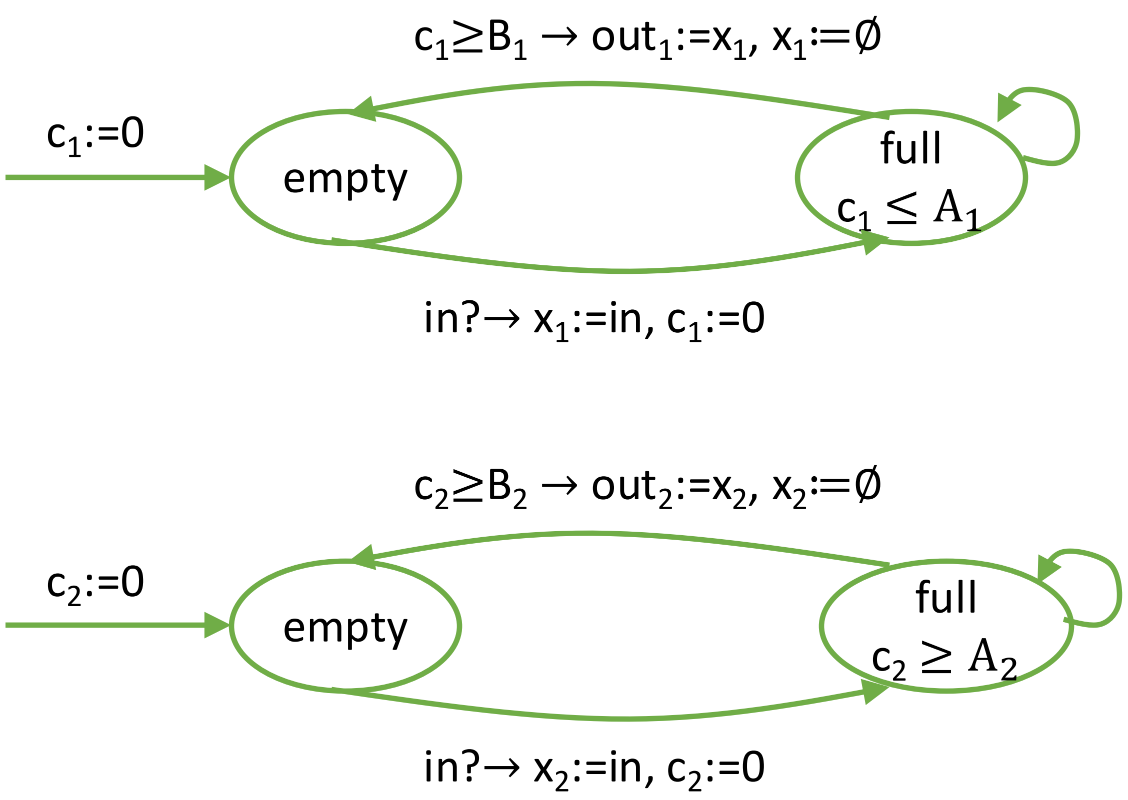
Example with two clocks



- Model with one input channel and two output channels: out_1 and out_2
- Clock c tracks time elapsed since occurrence of the input task execution
- Clock d tracks time elapsed since occurrence of output task for out_1
- Behavior of process: If input event occurs at some time t, then process issues output # on out_1 some time t' ∈ [t,t+1] and then issues output * on out_2 at time t'' ∈ [t'+1, t+2]

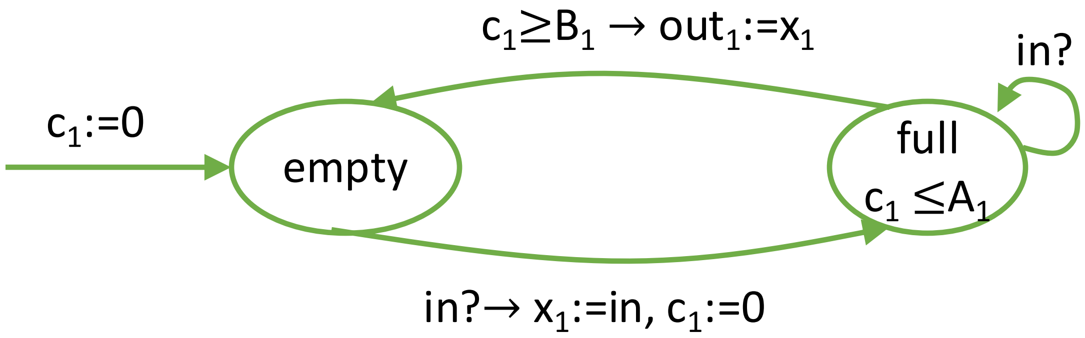
$$(\mathtt{Idle}, 0, 0) \xrightarrow{5.7} , (\mathtt{Idle}, 5.7, 5.7) \xrightarrow{in?} (\mathtt{Wait1}, 0, 5.7) \xrightarrow{0.6} (\mathtt{Wait1}, 0.6, 6.3) \xrightarrow{out_1!}$$
$$(\mathtt{Wait2}, 0.6, 0) \xrightarrow{0.5} (\mathtt{Wait2}, 1.1, 0.5) \xrightarrow{0.8} (\mathtt{Wait2}, 1.9, 1.3) \xrightarrow{out_2!} (\mathtt{Idle}, 1.9, 1.3).$$

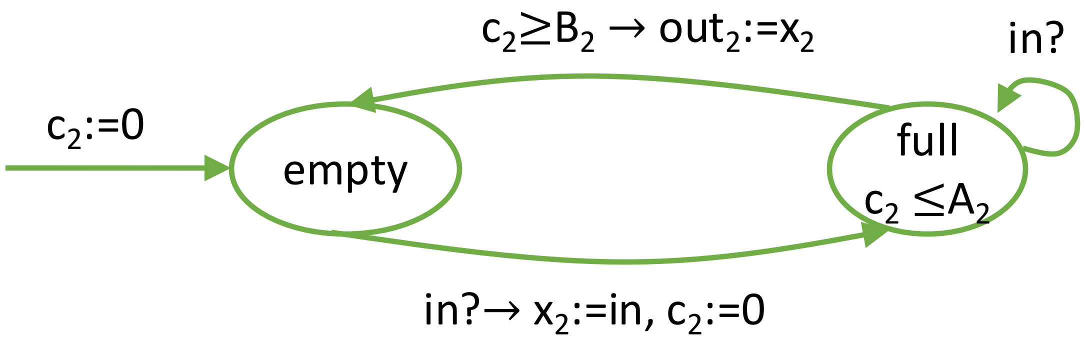
Composing Timed Processes

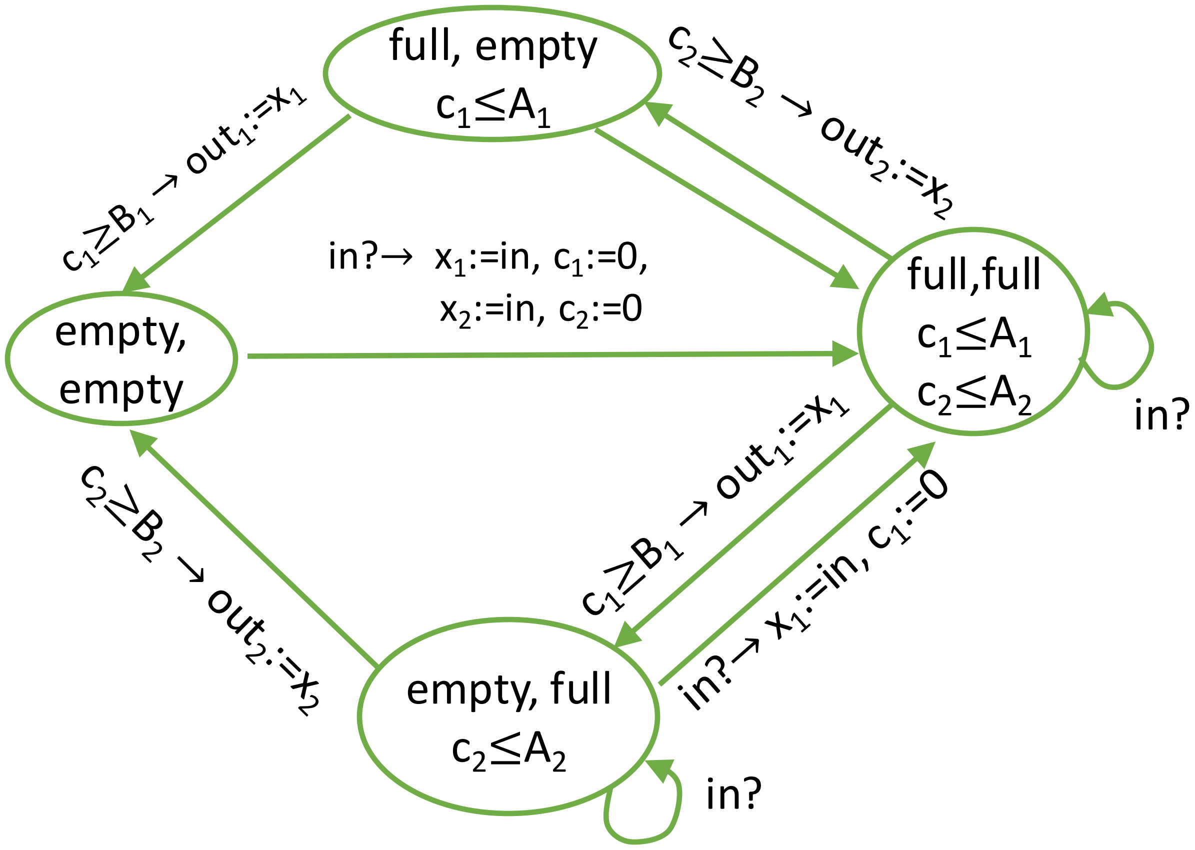


- Each process stays in mode full for $t \in [B_i, A_i]$
- Need to construct a new process with 4 new modes
- Each new mode is a pair consisting of modes from process 1 and 2
- Mode switches in the new machine correspond to mode switches in the old machine
- Interesting timing behavior can arise!

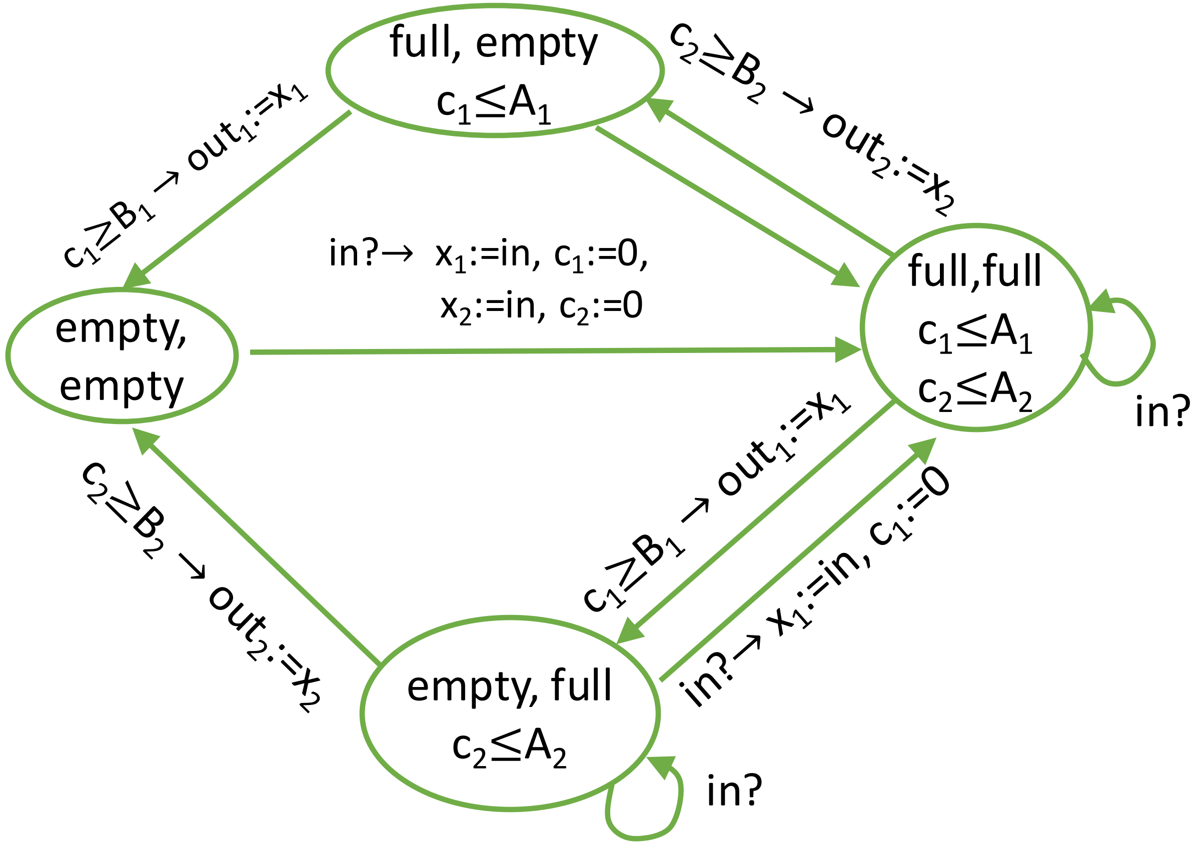
Composing Timed Processes







Semi-synchrony



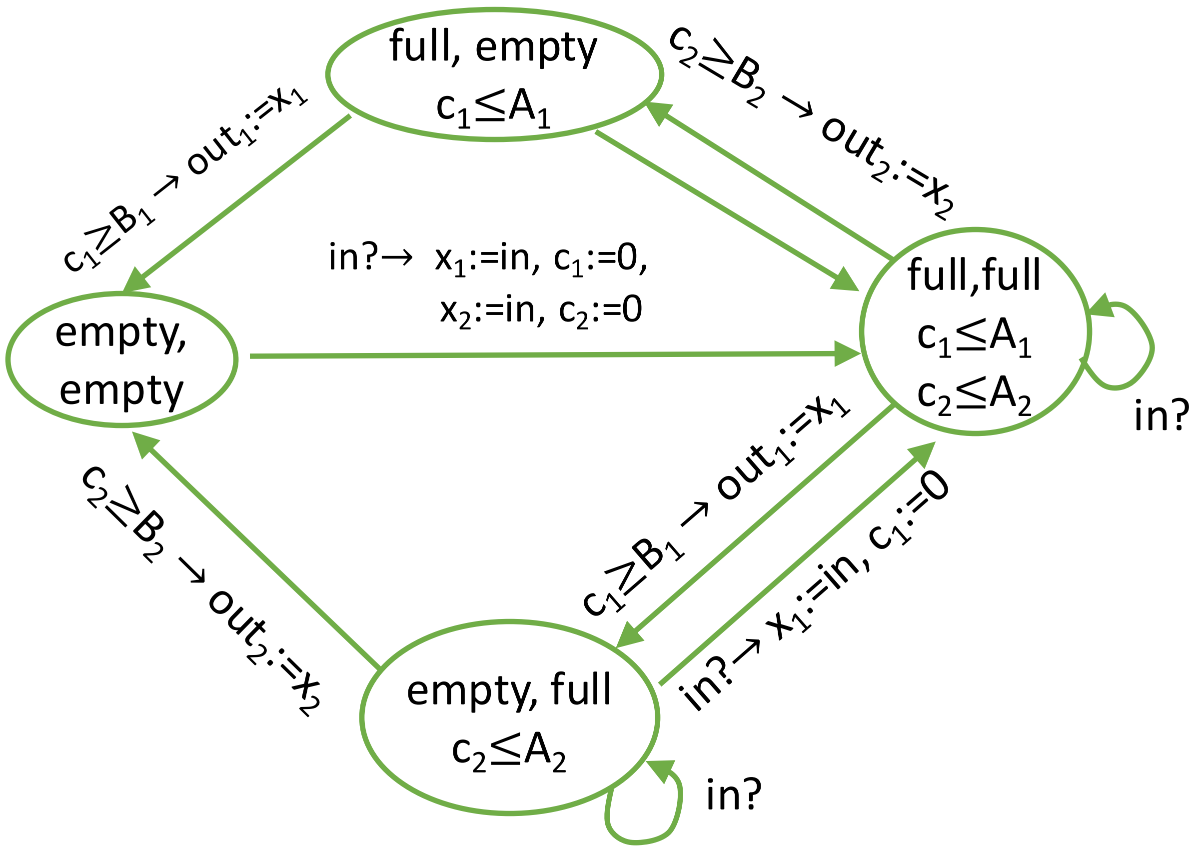
If $B_1 < A_1 < B_2$:

- (full,full) →(full,empty) can never be enabled!

Why?

- c_1 reaches A_1 and the process gets kicked out of state (full,full)
- But c_1 cannot be greater than B_2 so, guard from (full,full) to (full,empty) is not enabled!

Semi-synchrony



- If $B_1 < A_1 < B_2$:
  - (full,full) → (full,empty) cannot happen
- If $B_1 < A_1 < B_2$:
  - (full, full) → (empty,full) will happen eventually
- **out_1 guaranteed to happen before out_2**

- Implicit coordination based on delays
  - Both process clocks increase in tandem
  - Global clock-based synchronization
- Reason why timed models are called semi-synchronous or partially synchronous

Formal recap of a timed process

- Timed process consists of:
  - An asynchronous process, where some of the state variables are of type clock (ranging over non-negative reals)
  - A clock invariant I which is a Boolean expression over the state variables
- Inputs, Outputs, States, Initial states, Actions: Internal, Input and Output: same as for asynchronous processes
- Timed Action: Given a state q and time $\delta > 0$, action q $\xrightarrow{\delta}$q' specifies a transition of duration δ if:
  - q' represents a state where the non-clock variables have the same value as in q, i.e. q'(x) = q(x)
  - q' represents a state where the clock variables in q are incremented by δ, i.e. q'(c) = q(c) + δ, and
  - For all times t ∈ [q(c), q(c)+δ], the clock invariant I is satisfied
  - If clock invariant is *convex*, enough to check clock invariant at q(c) and q(c)+δ

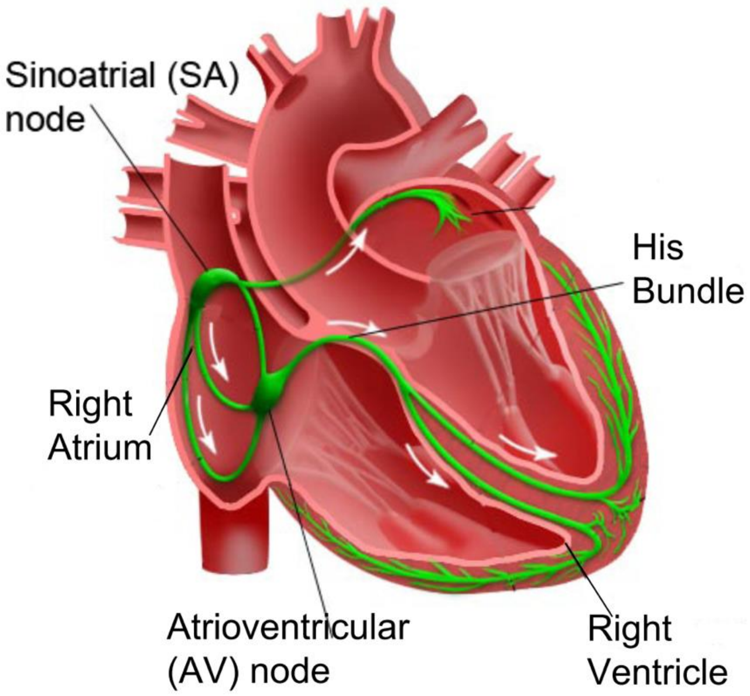
Pacemaker Modeling as a Timed Process

- Most material that follows is from this paper:

Z. Jiang, M. Pajic, S. Moarref, R. Alur, R. Mangharam, *Modeling and Verification of a Dual Chamber Implantable Pacemaker*, In Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2012.

- The textbook has detailed descriptions of some other pacemaker components

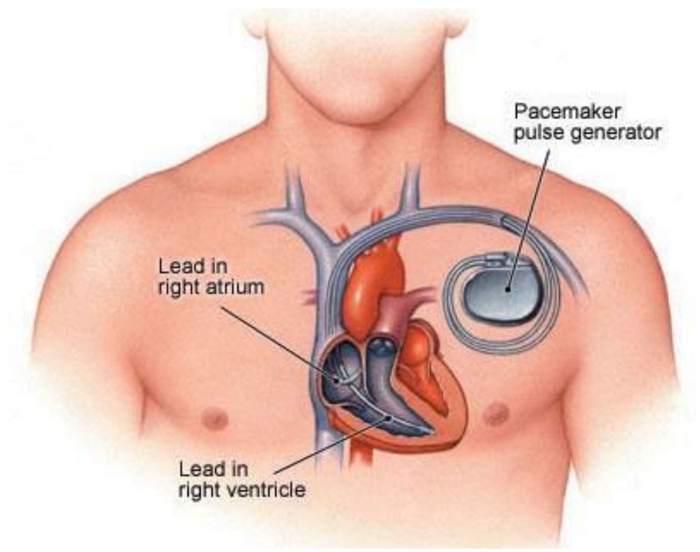
How does a healthy heart work?



Electrical Conduction System of the Heart

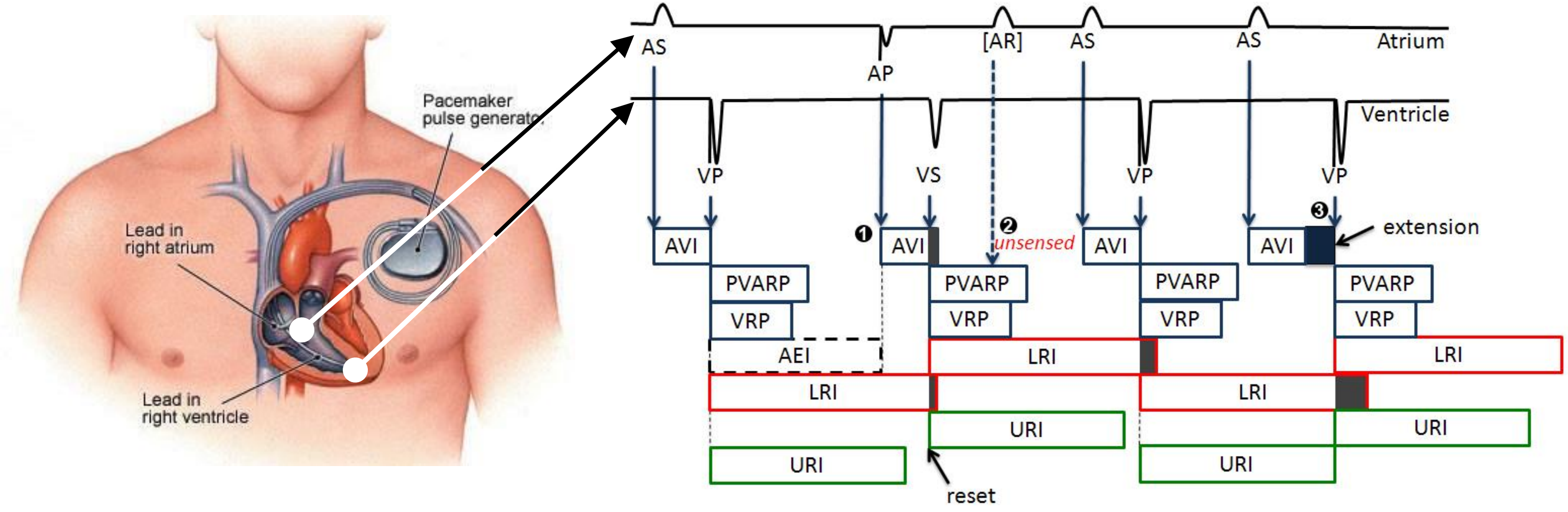
- SA node (controlled by nervous system) periodically generates an electric pulse
- This pulse causes both atria to contract pushing blood into the ventricles
- Conduction is delayed at the AV node allowing ventricles to fill
- Finally the His-Pukinje system spreads electric activation through ventricles causing them both to contract, pumping blood out of the heart

What do pacemakers do?



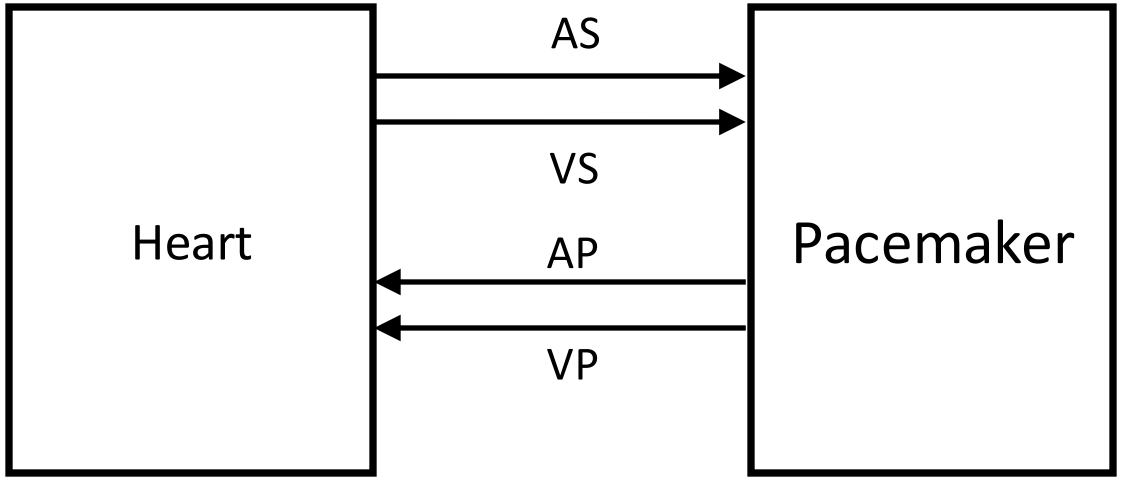
- Aging and/or diseases cause conduction properties of heart tissue to change leading to changes in heart rhythm
- Tachycardia: faster than desirable heart rate impairing hemo-dynamics (blood flow dynamics)
- Bradycardia: slower heart rate leading to insufficient blood supply
- Pacemakers can be used to treat bradycardia by providing pulses when heart rate is low

Implantable Pacemaker modeling



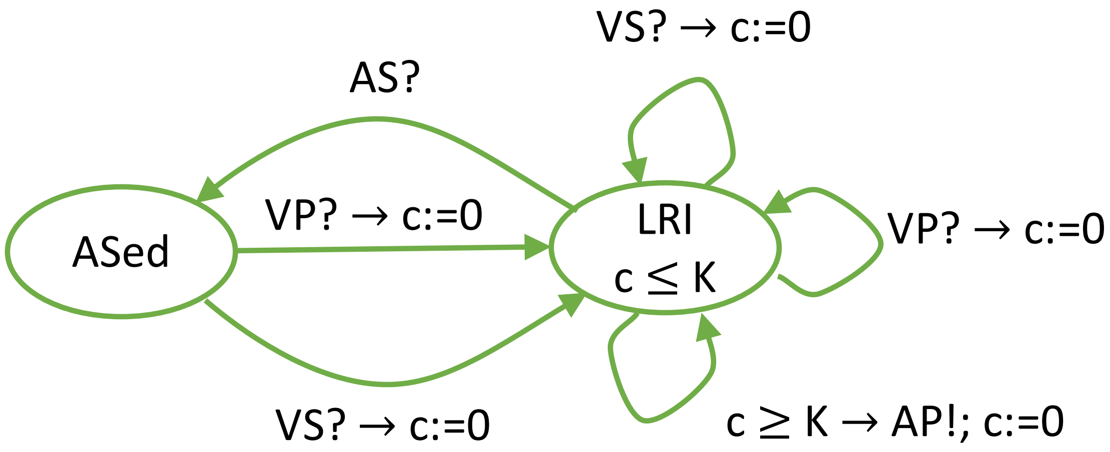
How dual-chamber pacemakers work

- Two fixed leads on wall of right atrium and ventricle respectively
- Activation of local tissue sensed by the leads (giving rise to events Atrial Sense (AS) and Ventricular Sense (VS))
- Atrial Pacing (AP) or Ventricular Pacing (VP) are delivered if no sensed events occur within deadlines



The Lower Rate Interval (LRI) mode

LRI component keeps heart rate above minimum level



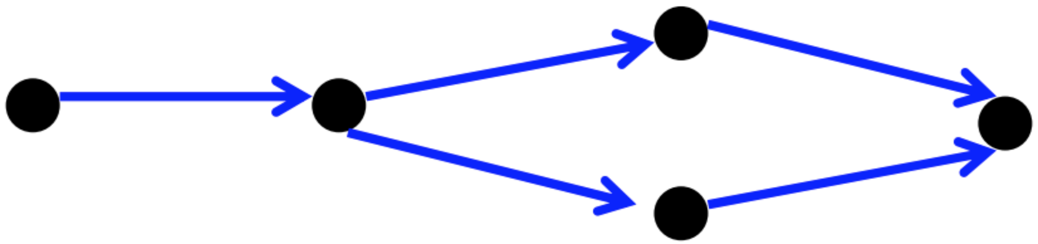
K= 850ms

- LRI = lower rate interval
- LRI component keeps heart rate above minimum level
- One of the pacemaker modes of operation that models the basic timing cycle
- Measures the longest interval between ventricular events
- Clock reset when VS or VP received
- No AS received ⇒ LRI outputs AP after K time units

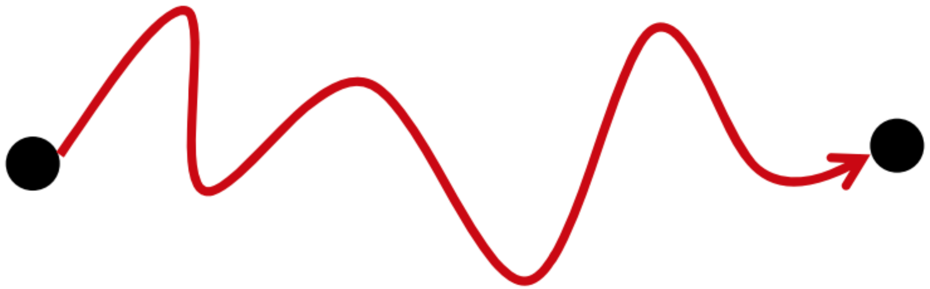
FSM Software Tools

- Statecharts (Harel, 1987), a notation for concurrent composition of hierarchical FSMs, has influenced many of these tools.
- One of the first tools supporting the Statecharts notation is STATEMATE (Harel et al., 1990), which subsequently evolved into Rational Rhapsody, sold by IBM.
- Almost every software engineering tool that provides UML (unified modeling language) capabilities (Booch et al., 1998).
- SyncCharts (André , 1996) is a particularly nice variant in that it borrows the rigorous semantics of Esterel (Berry and Gonthier, 1992) for composition of concurrent FSMs.
- LabVIEW supports a variant of Statecharts that can operate within dataflow diagrams
- Simulink with its Stateflow extension supports a variant that can operate within continuous-time models.
- UPPAAL (Yi, Pettersson, Larseń, mid-1990s) is is a tool for modeling, simulation, and verification of real-time systems. It was jointly developed by Uppsala University in Sweden and Aalborg University in Denmark.

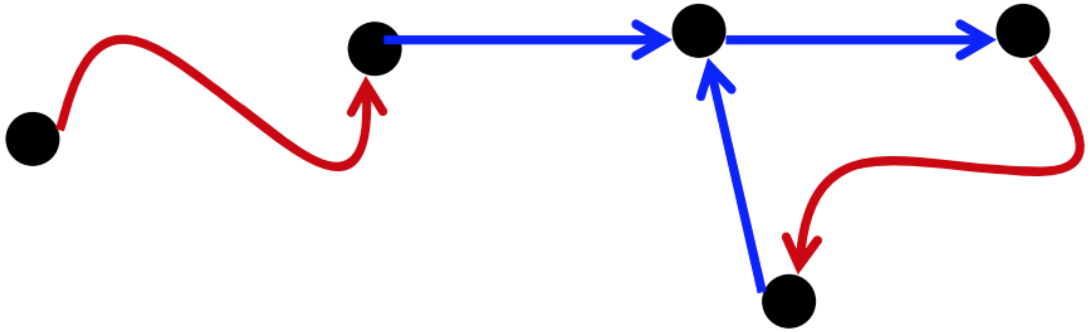
Discrete System (FSM)

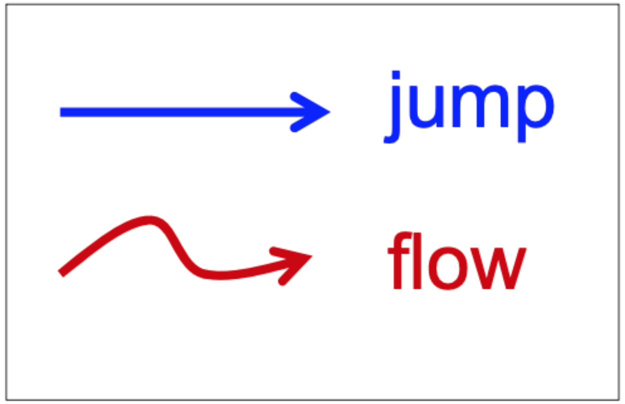


Continuous System



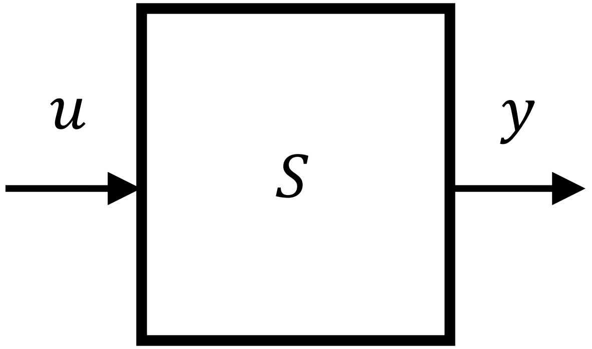
Hybrid System





Actor Models

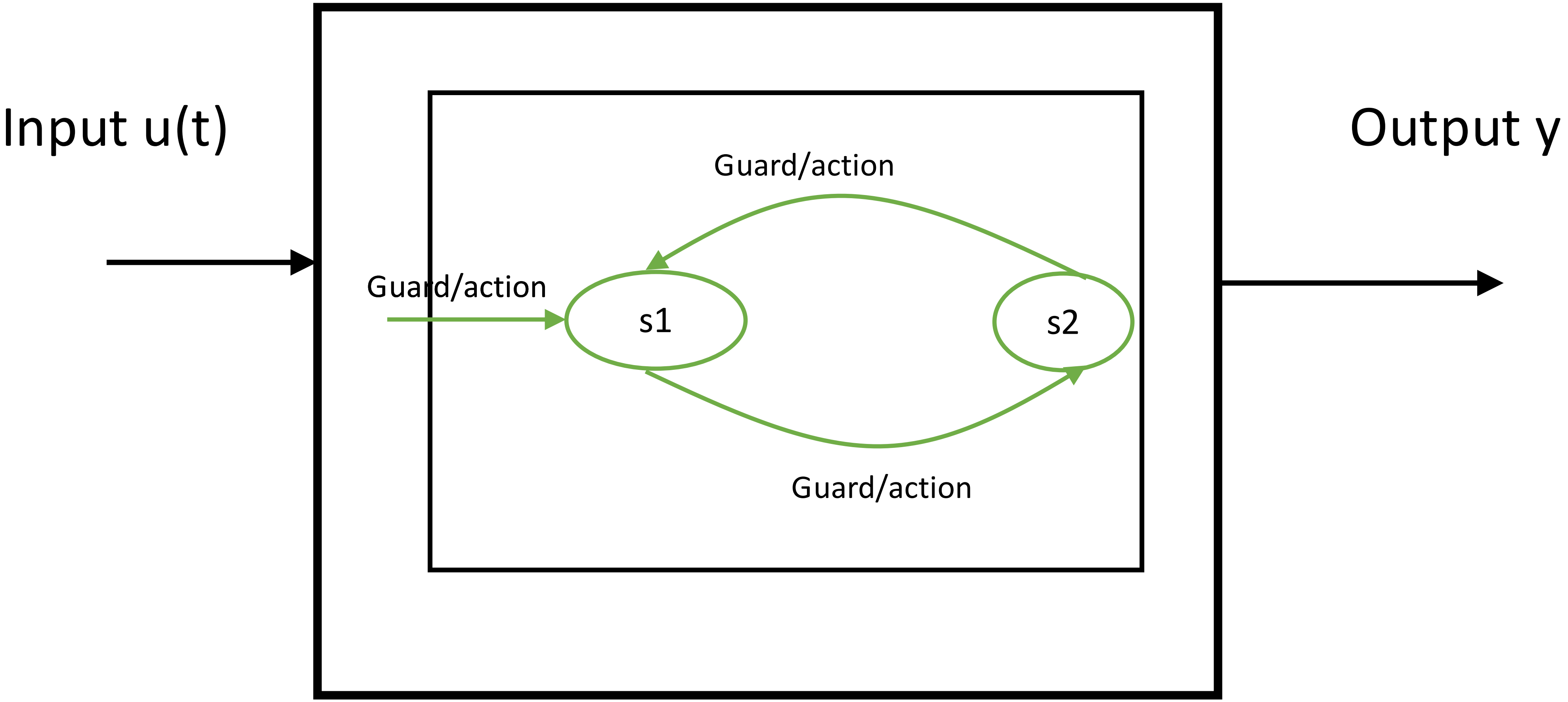
A box, where the inputs and the outputs are functions $S: u \to y$



Actor models are composable. We can form a **cascade composition**

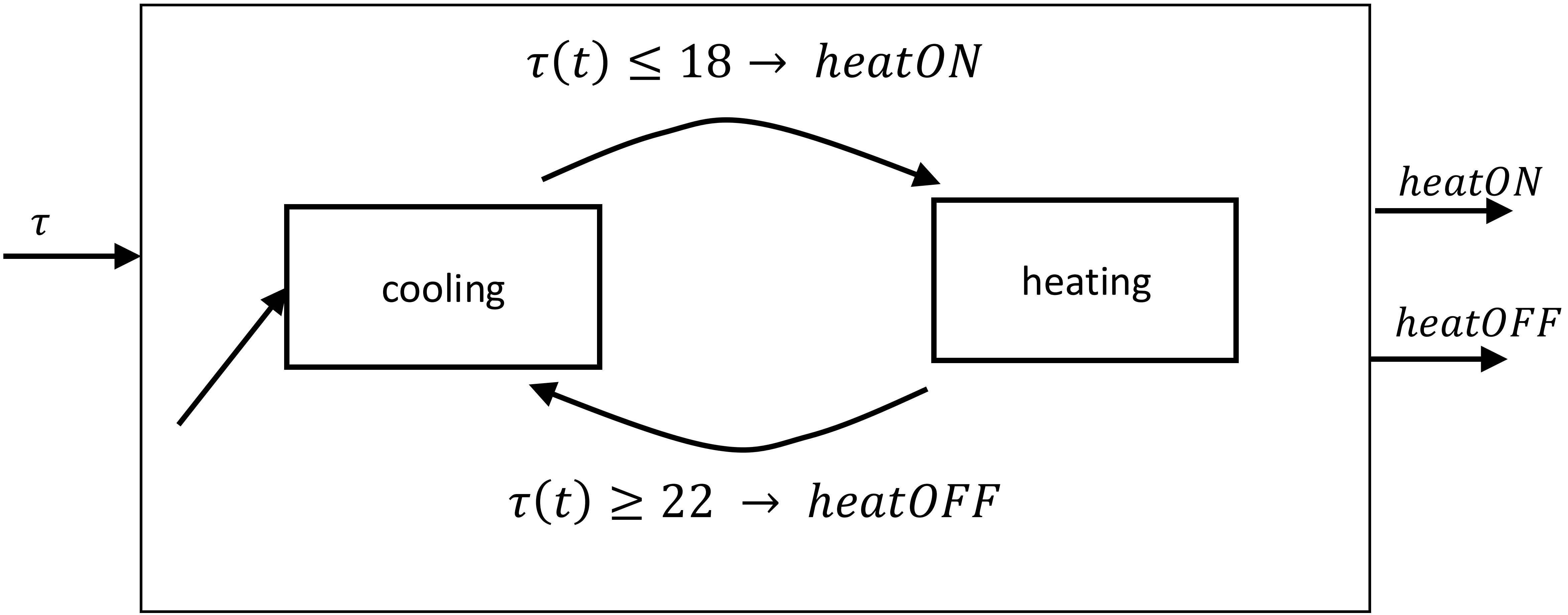
We have so far assumed that state machines operate in a sequence of discrete reactions.
We have assumed that inputs and outputs are absent between reactions.

Having continuous inputs



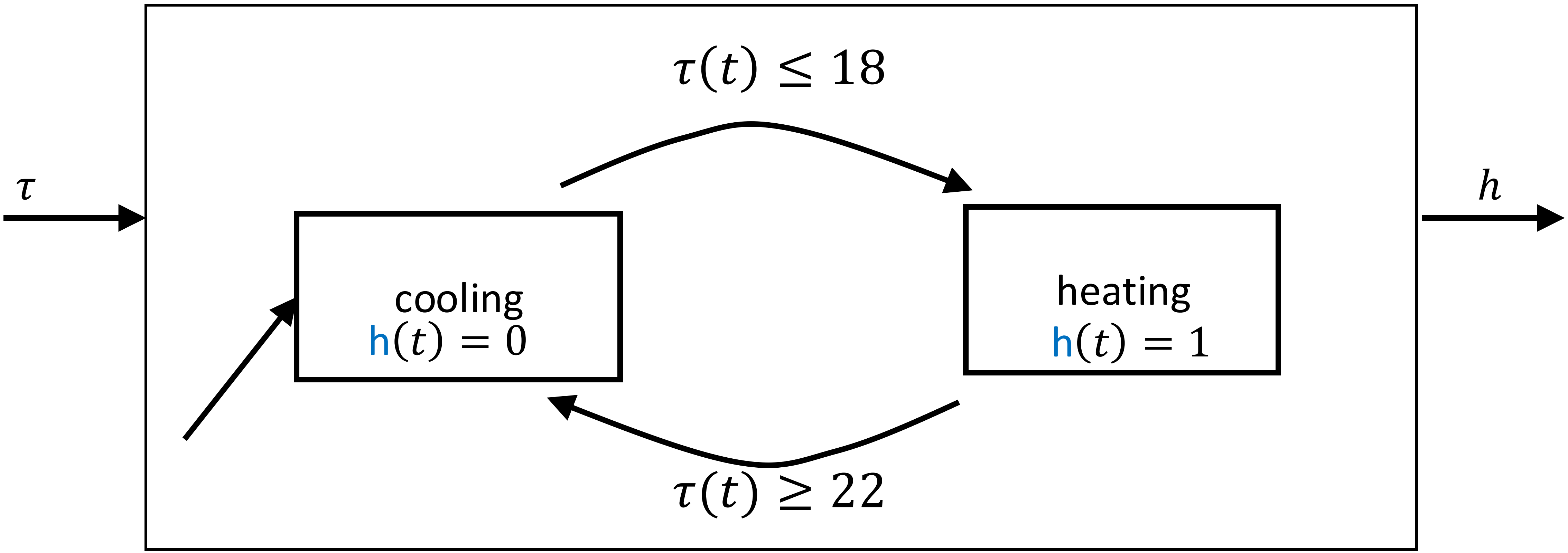
We will define a transition to occur when a guard on an outgoing transition from the current state becomes enabled

Thermostat FSM with a continuous-time input signal



The outputs are present only at the times the transitions are taken

State Refinements



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.

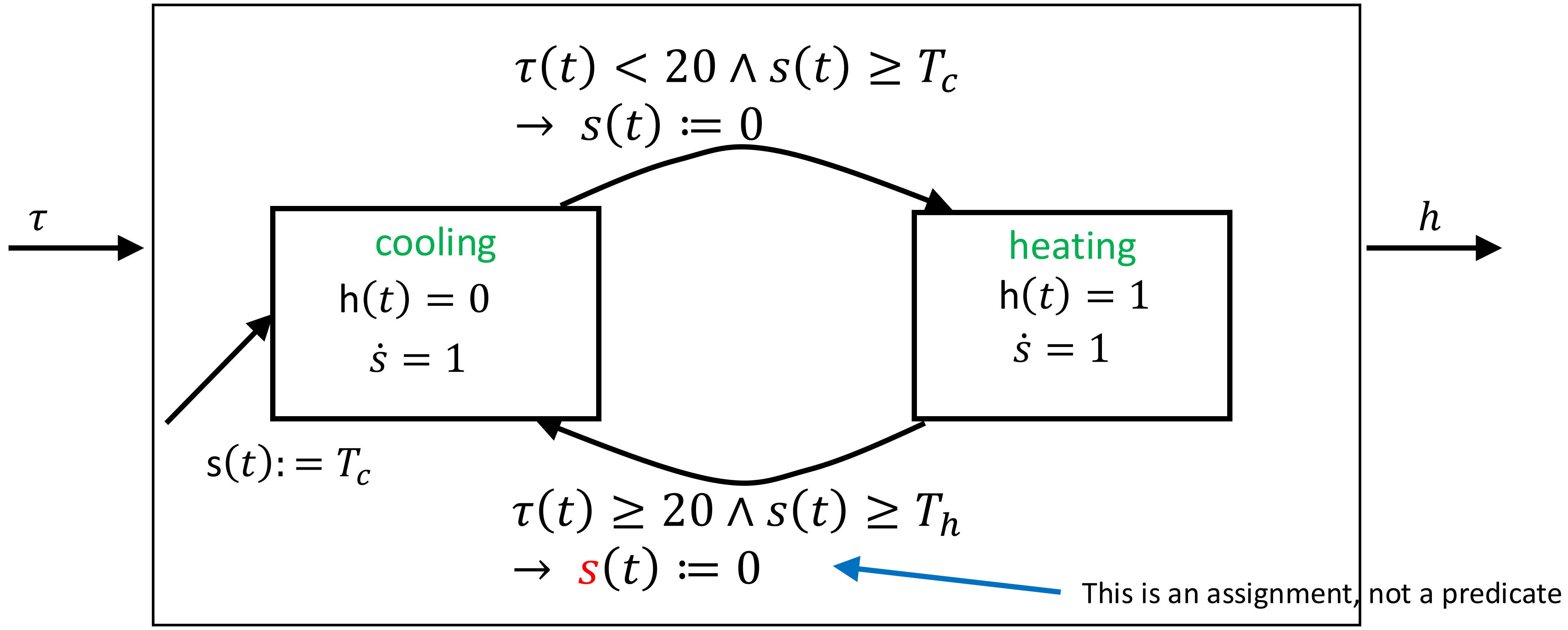
Modal Models

A hybrid system is sometimes called a modal model because it has a finite number of modes, one for each state of the FSM, and when it is in a mode, it has dynamics specified by the state refinement.

Timed Automata

- Introduced by Alur and Dill (A theory of timed Automata, TCS,1994)
- They are the simplest non-trivial hybrid systems
- All they do is measuring the passage of time
- A **clock** $s(t)$ is modeled by a first-ODE: $\dot{s} = a \quad \forall t \in T_m$
  where $s : \mathbb{R} \rightarrow \mathbb{R}$ 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 Automata

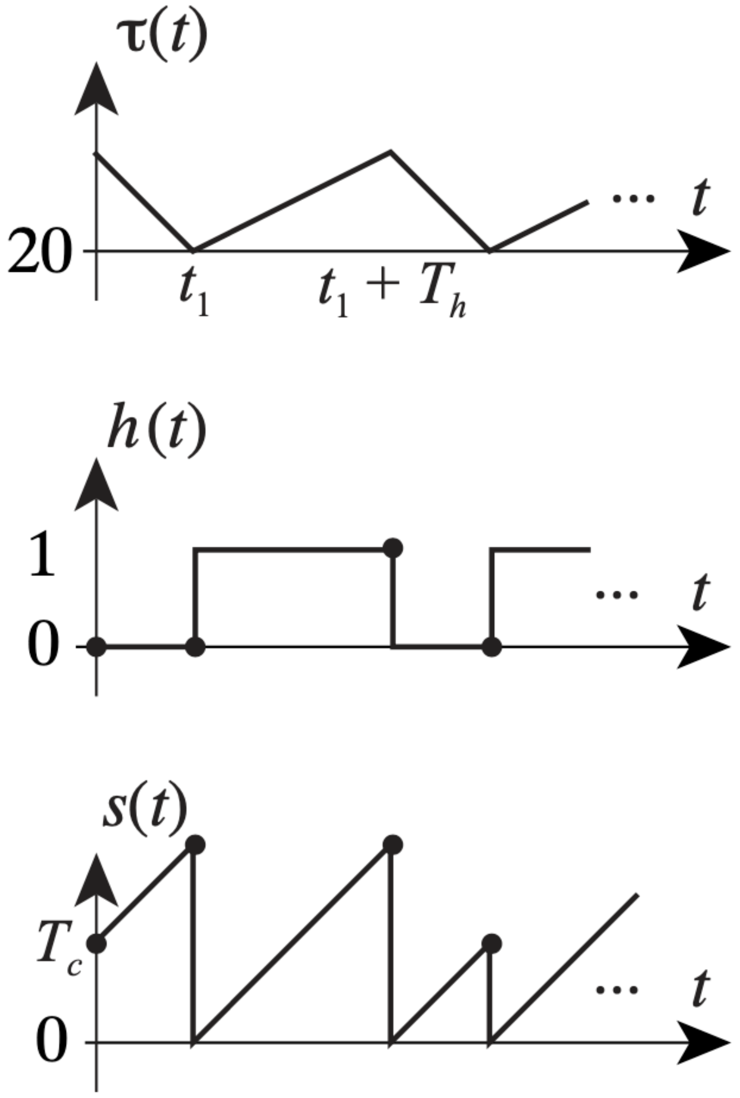


cooling and heating are discrete states, s is a continuous state

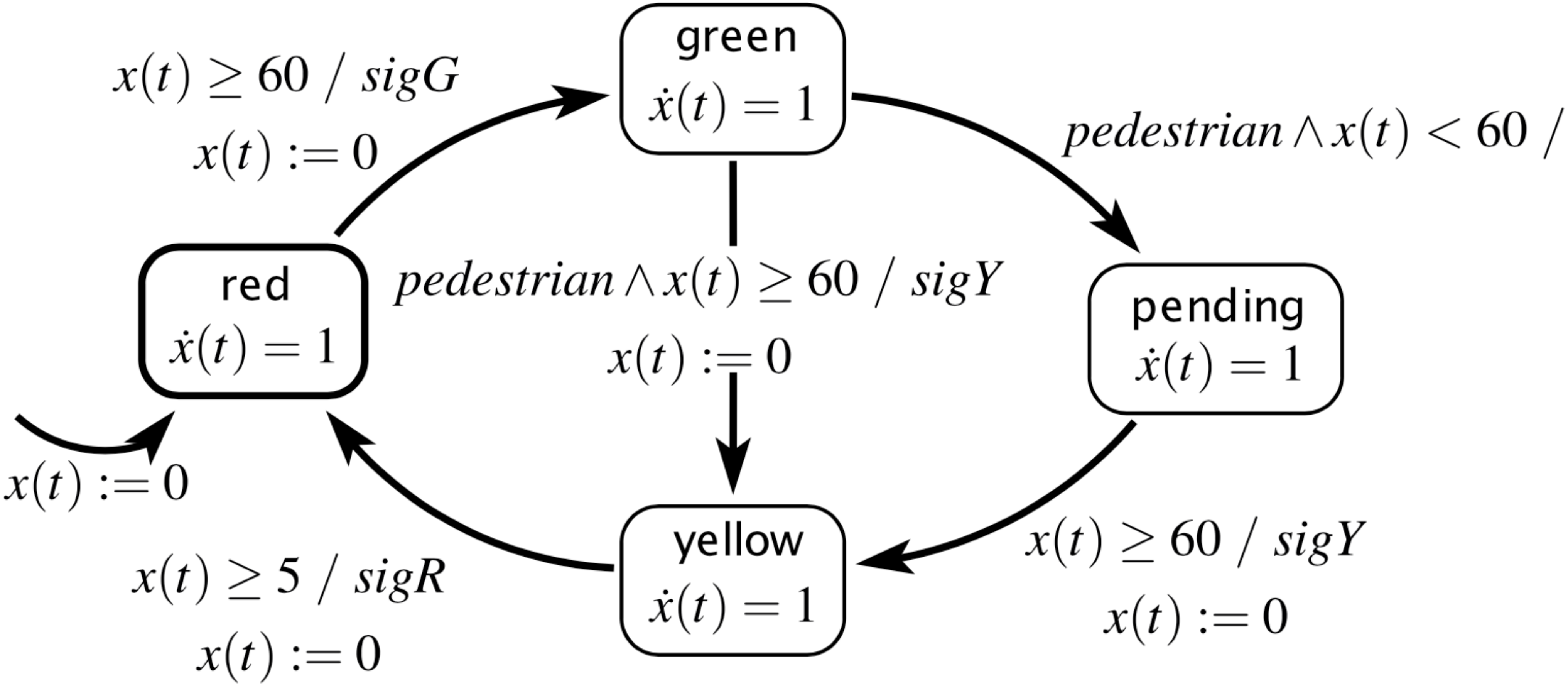
(a) Temperature input $\tau(t)$

(b) The output h

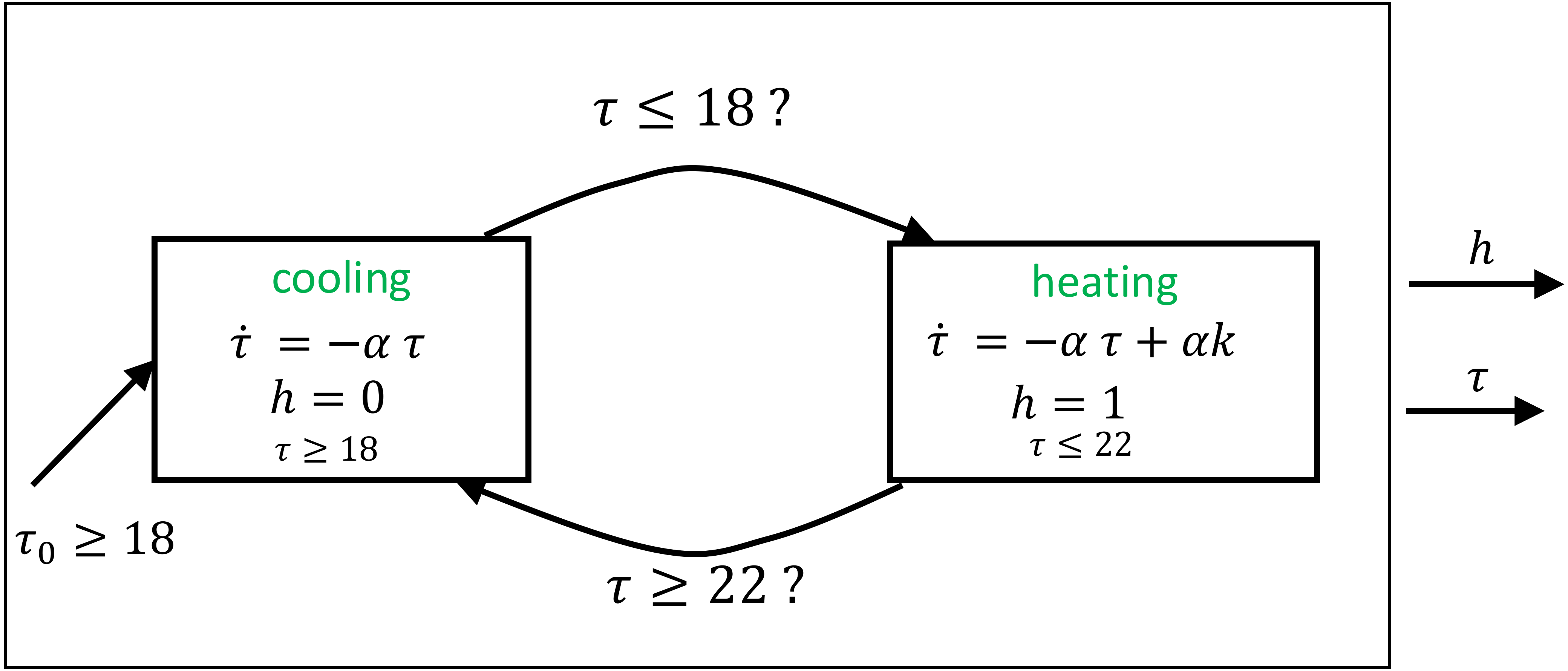
(c) The refinement state s.



continuous variable: $x(t) : \mathbb{R}$
inputs: $pedestrian$: pure
outputs: $sigR, sigG, sigY$: pure



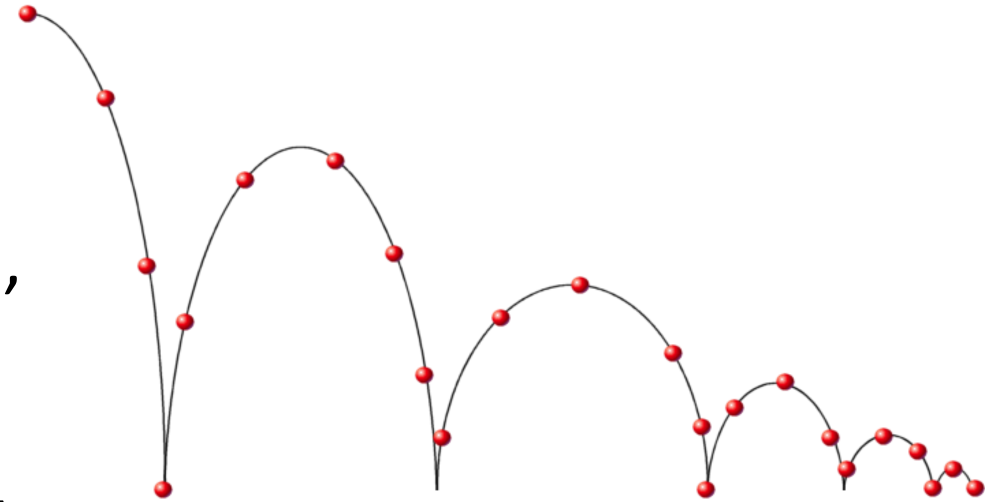
Hybrid Automata



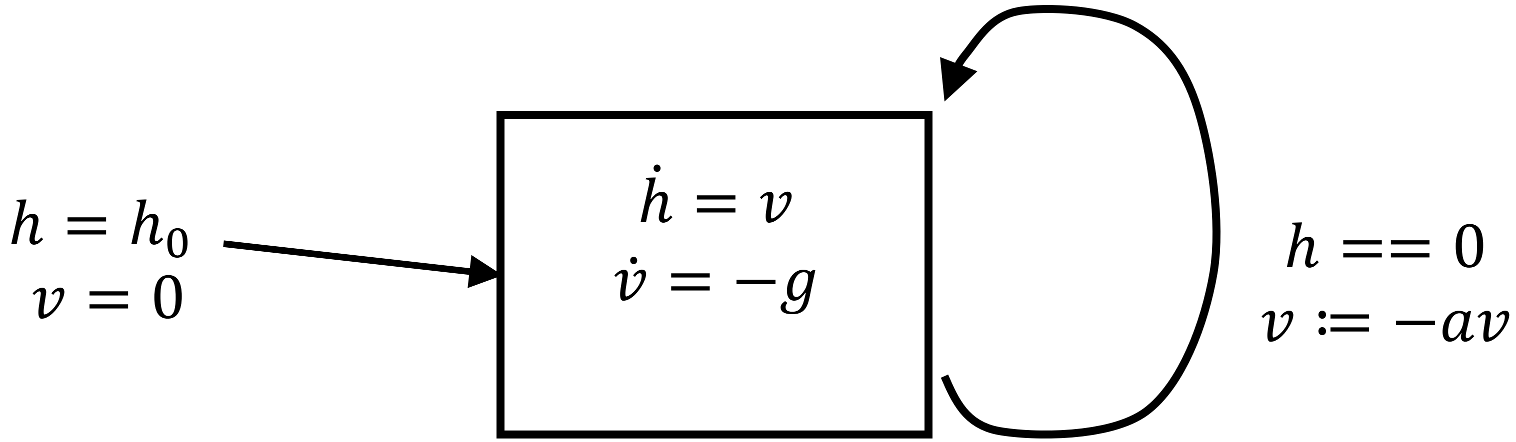
- Generalization of a timed process
- Instead of timed transitions, we can have arbitrary evolution of state/output variables, typically specified using differential equations

Modeling a bouncing ball

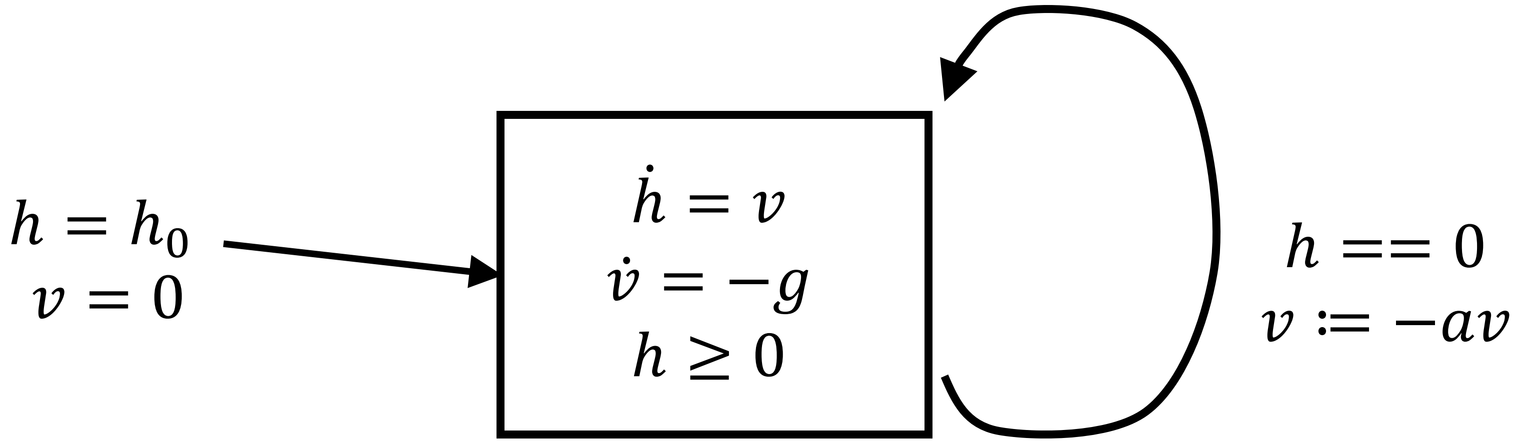
- Ball dropped from an initial height of h_0 with an initial velocity of v_0
- Velocity changes according to $\dot{v} = -g$
- When ball hits the ground, i.e. when $h(t) = 0$, velocity changes discretely from negative (downward) to positive (upward)
  - I.e. $v(t) \coloneqq -av(t)$, where a is a damping constant
- we can model it as a hybrid system!



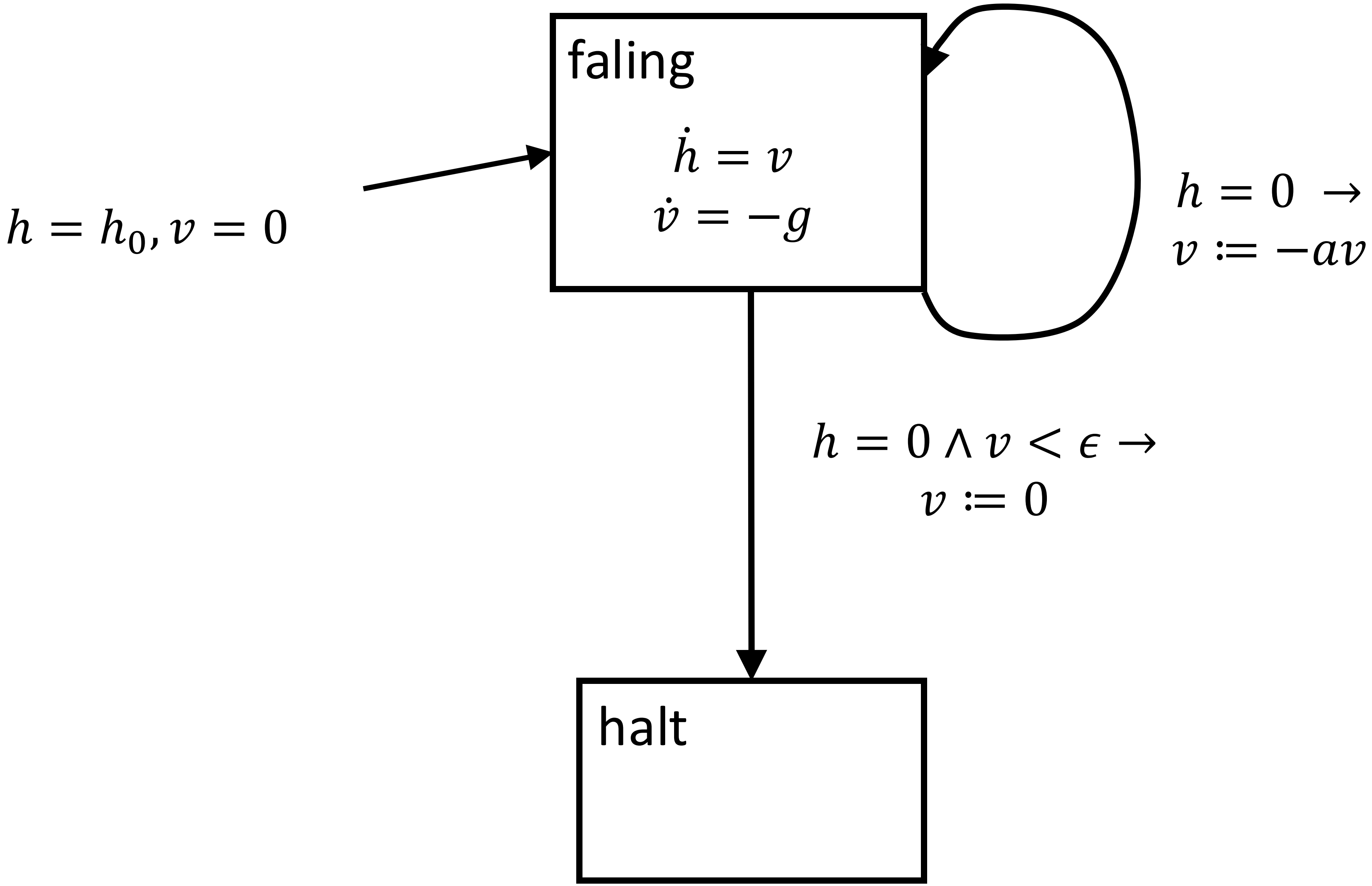
Hybrid Process for Bouncing ball



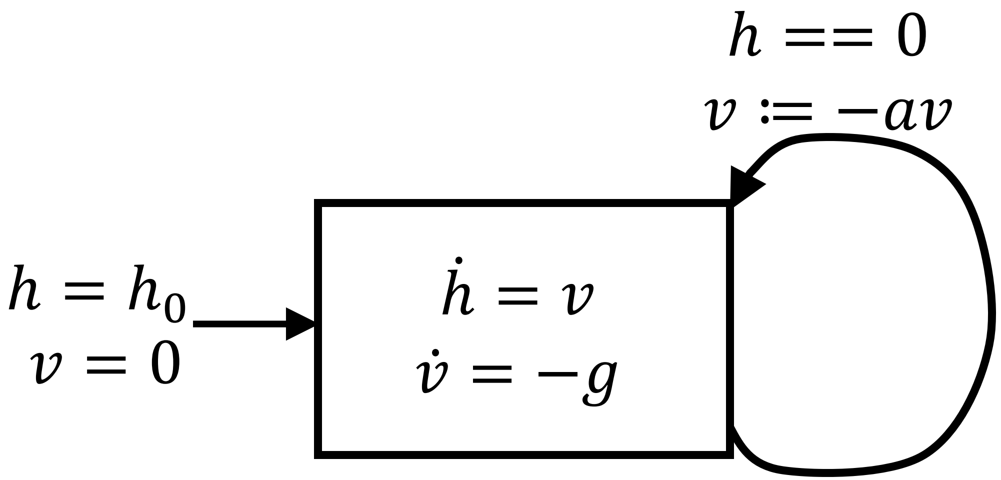
Hybrid Process for Bouncing ball

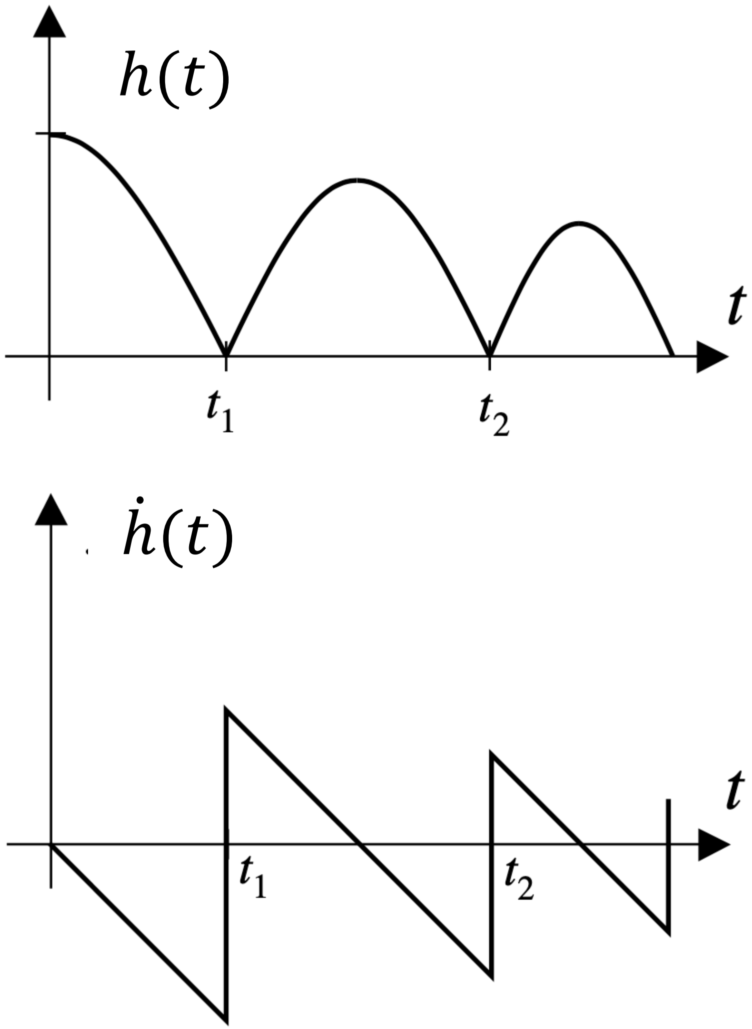


Non-Zeno hybrid process for bouncing ball



Hybrid Process for Bouncing ball





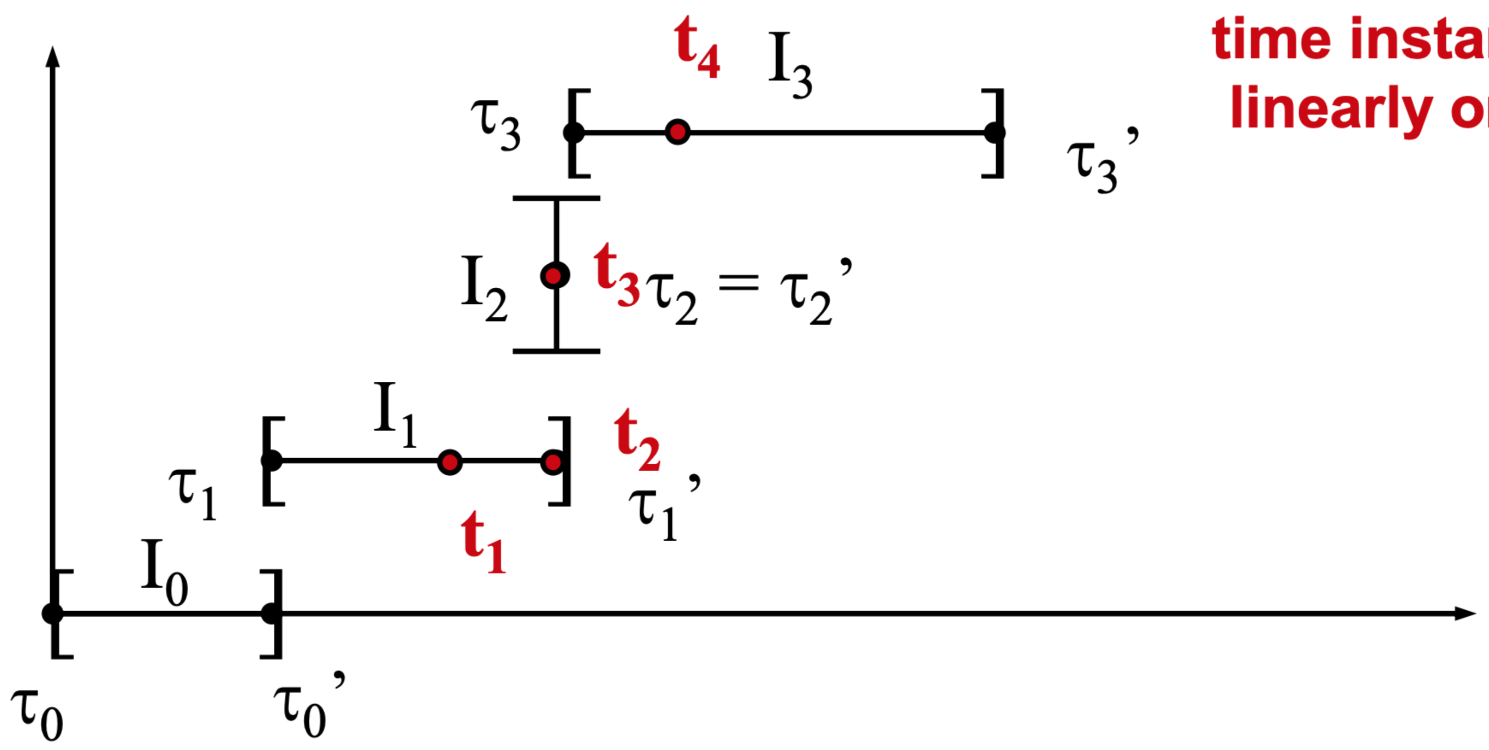
What happens as $h \to 0$?

Hybrid Time Set

A hybrid time set is a finite or infinite sequence of intervals

$\tau = \{ I_i, i = 0, \ldots, M\}$:

- $I_i = [\tau_i, \tau_i'] \; for \; i < M$
- $I_M = [\tau_M, \tau_M']$ or $I_M = [\tau_M, \tau_M')$ if M<∞
- $\tau_i' = \tau_{i+1}$
- $\tau_i \leq \tau_i'$



$t_1 \prec t_2 \prec t_3 \prec t_4$

time instants in τ are linearly ordered

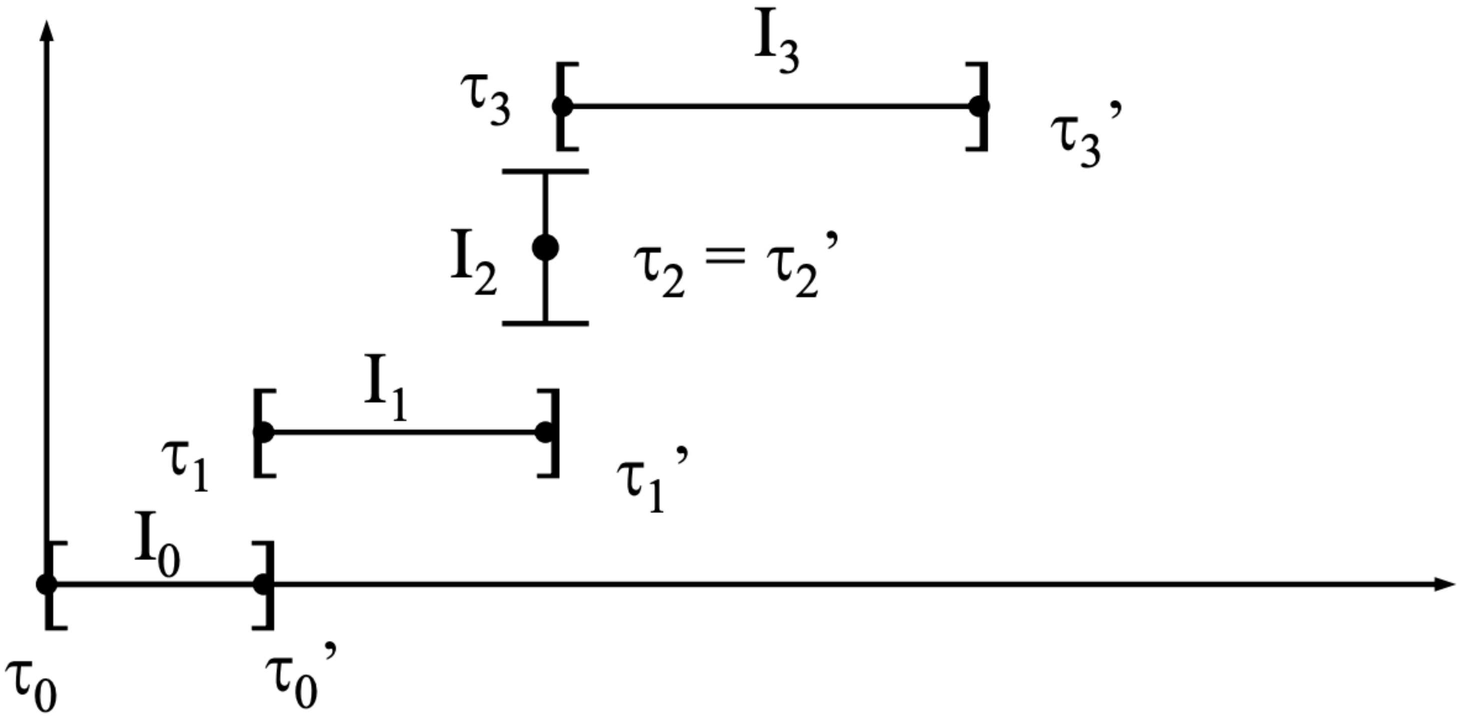
Hybrid Time Set: Length

Two notions of length for a hybrid time set $\tau = \{ I_i, i = 0, \ldots, M\}$:

- Discrete extent: $< \tau > = M + 1$ number of discrete transition
- Continuous extent: $||\tau|| = \sum_{i=0}^{M} |\tau_i' - \tau_i|$ total duration of interval in τ

$<\tau> = 4$

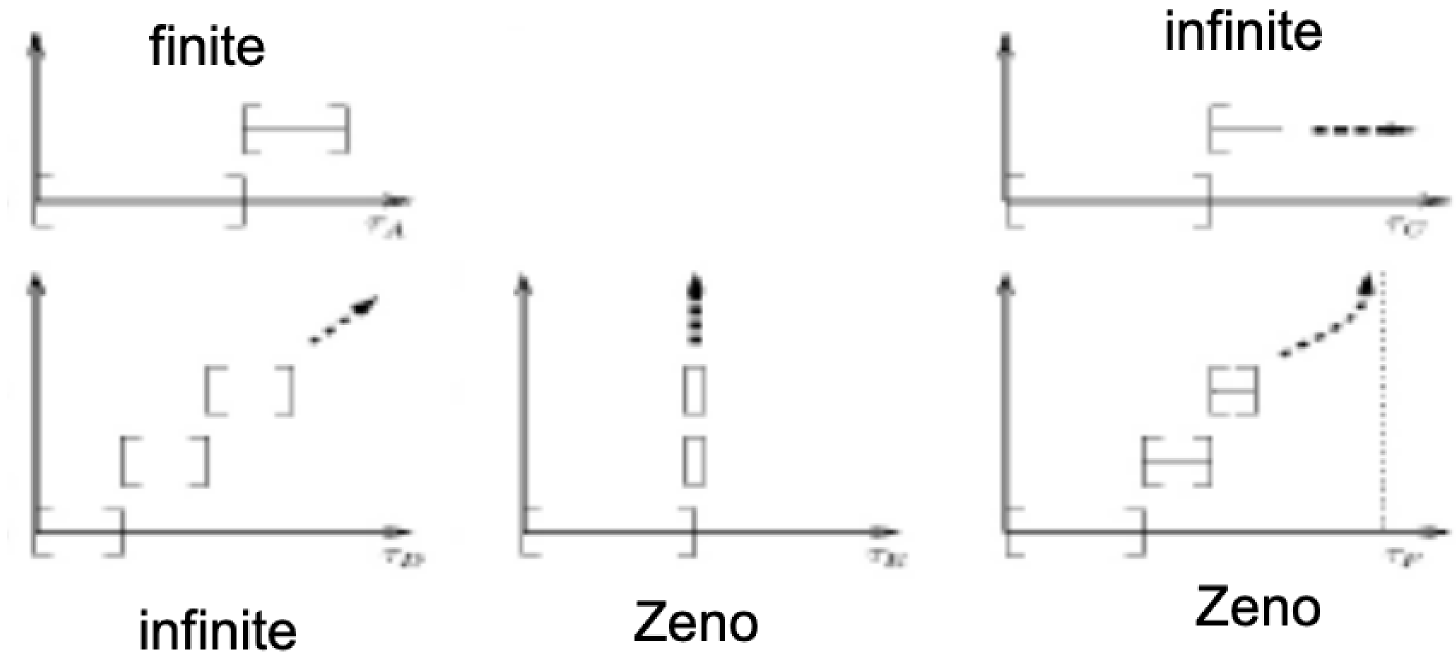
$||\tau|| = \tau_3' - \tau_0$



Hybrid Time Set: Classification

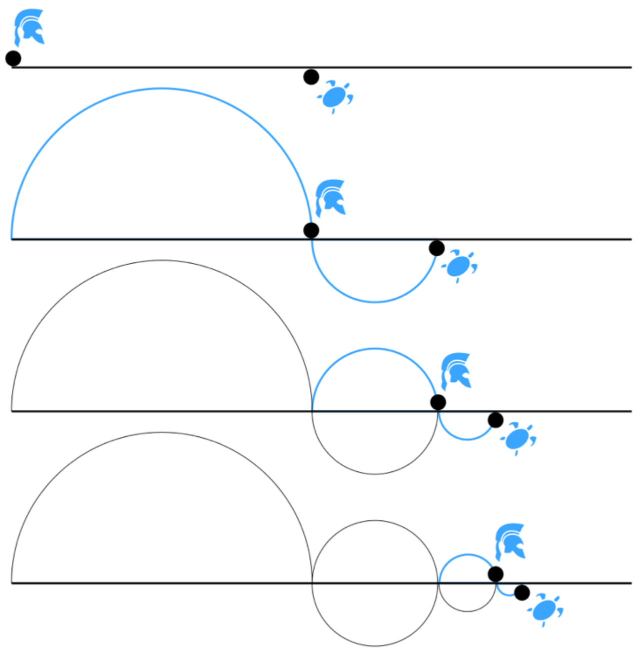
A hybrid set $\tau = \{ I_i, i = 0, \ldots, M\}$ is :

- Finite: if $< \tau >$ is finite and $\mathrm{I}_M = [\tau_M, \tau'_M]$
- Infinite:if $||\tau||$ is infinite
- Zeno: if $< \tau >$ is infinite but $||\tau||$ is finite



Zeno's Paradox

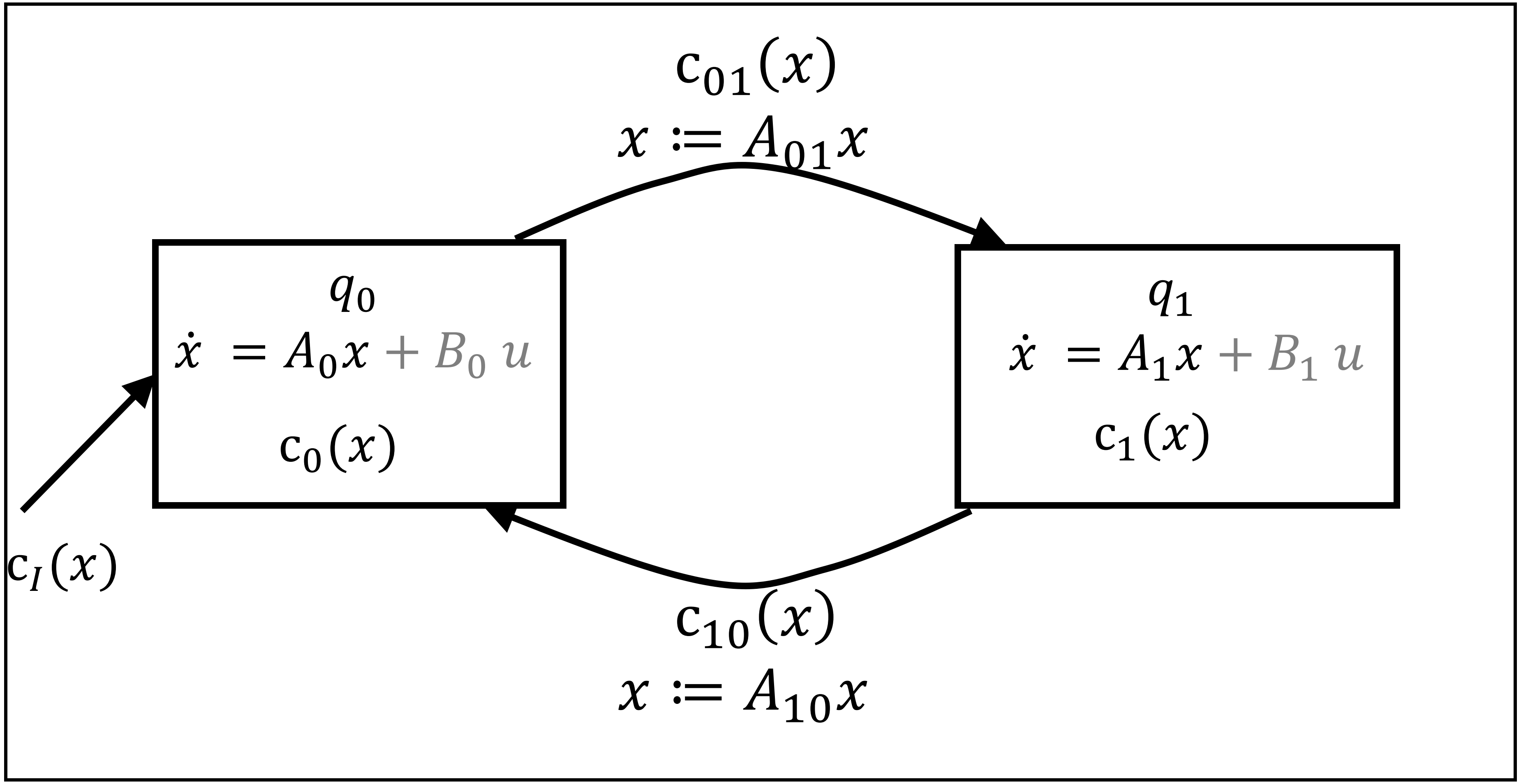
- Greek philosopher Zeno's race between Achilles and a tortoise
  - Tortoise has a head start over Achilles, but is much slower
  - If Achilles is d meters behind at the beginning of a round
    and during the round, suppose Achilles runs d meters
    but by then, tortoise has moved a little bit further
  - At the beginning of the next round, Achilles is still behind, by $a \times d$ meters [0<a<1]
- By induction, if we repeat this for infinitely many rounds, Achilles will never catch up!
- If sum of durations between successive discrete actions converges to constant K, then an execution with infinitely many discrete actions describes behavior only up to time K (and does not tell us the state of the system at time K and beyond)



Zeno behaviors

- An infinite execution is called Zeno if infinite sum of all the durations is bounded by a constant, and non-Zeno if the sum diverges
- Any state in a hybrid process is:
  - Zeno if every execution starting in state is Zeno
  - Non-Zeno if there exists some non-Zeno starting in that state
- Hybrid process is non-Zeno if any state that you can reach from the initial state is non-Zeno
- Thermostat: non-Zeno, Bouncing ball: Zeno
- Dealing with Zeno: remove Zeno-ness through better modeling

(Linear) Hybrid Automata



Hybrid actions/transitions

$$(q, \mathbf{x}_{\tau}) \xrightarrow{\mathbf{u}(t)/\mathbf{y}(t)}_{\delta} \left(q, \mathbf{x}(\mathrm{t}+\delta)\right)$$

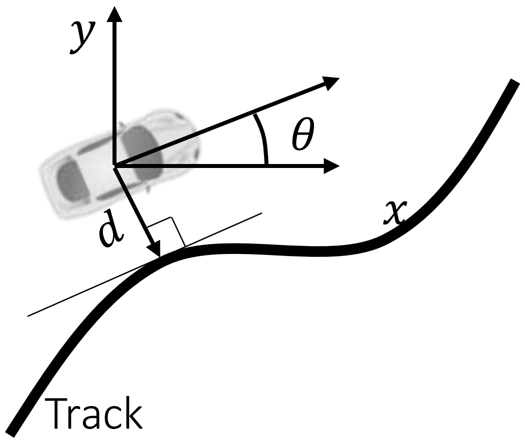
- Continuous action/transition:
  - Discrete mode q does not change
  - $\mathbf{x}_{\boldsymbol{\tau}} = \mathbf{x}(0)$
  - $\frac{d\mathbf{x}(t)}{dt}$ satisfies the given dynamical equation for mode q
  - Output $\mathbf{y}$ satisfies the output equation for mode q: $\mathbf{y}(t) = h_q(\mathbf{x}(t), \mathbf{u}(t))$
  - At all times $t \in [0, \delta]$,the state $\mathbf{x}(t)$ satisfies the invariant for mode m

Hybrid actions/transitions

$$(q, \mathbf{x}_\tau) \xrightarrow{g(\mathbf{x})/\mathbf{x} := r(\mathbf{x})} \left(q', r(\mathbf{x}_\tau)\right)$$

- Discrete action/transition:
  - Happens instantaneously
  - Changes discrete mode q to q'
  - Can execute only if $g(\mathbf{x}_\tau)$ evaluates to true
  - Changes state variable value from $\mathbf{x}_\tau$ to $r(\mathbf{x}_\tau)$
  - $r(\mathbf{x}_\tau)$ should satisfy mode invariant of q′Output will change from $h_q(\mathbf{x}_\tau)$ to $h_{q'}\left(r(\mathbf{x}_\tau)\right)$

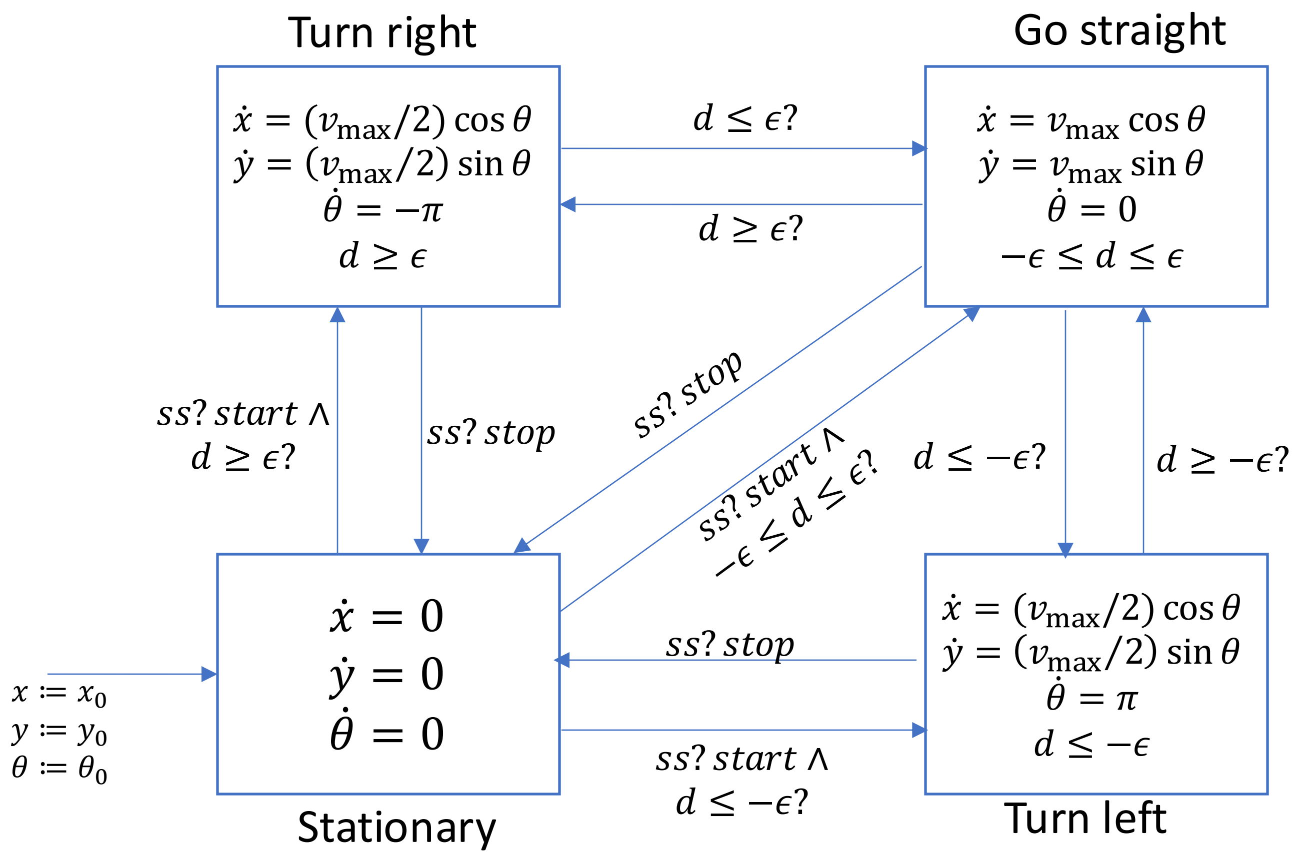
Design Application: Autonomous Guided Vehicle

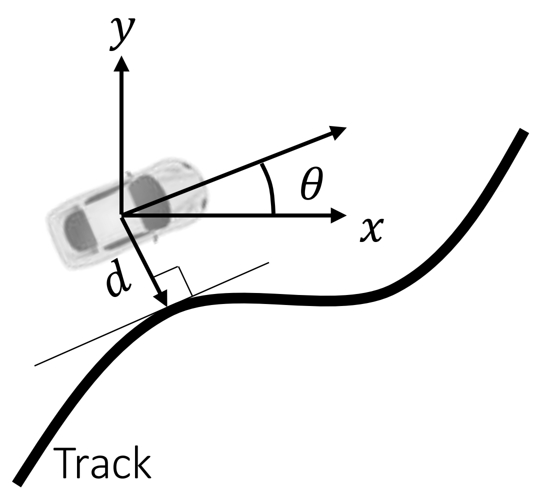


When $d \in [-\epsilon, +\epsilon]$, controller decides that vehicle goes straight, otherwise executes a turn command to bring error back in the interval

- Objective: Steer vehicle to follow a given track
- Control inputs: linear speed (v), angular speed (ω), start/stop
- Constraints on control inputs:
  - $v \in \{v_{\max}, v_{\max}/2, 0\}$
  - $\omega \in \{-\pi, 0, \pi\}$
- Designer choice: $v = v_{\max}$ only if $\omega = 0$, otherwise $v = \frac{v_{\max}}{2}$

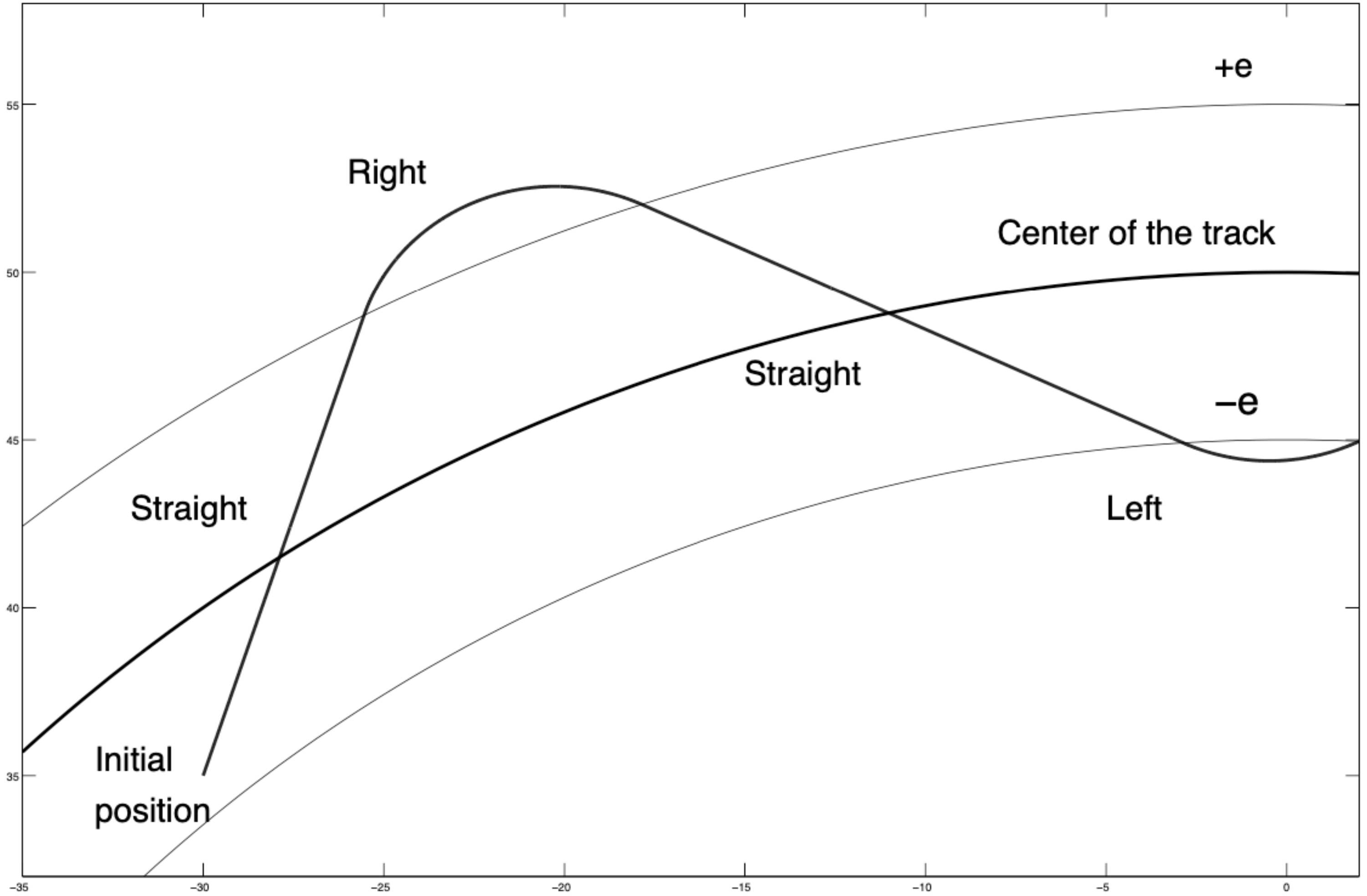
On/Off control for Path following





Inputs: ss $\in \{stop, start\}, d \in \mathbb{R}$

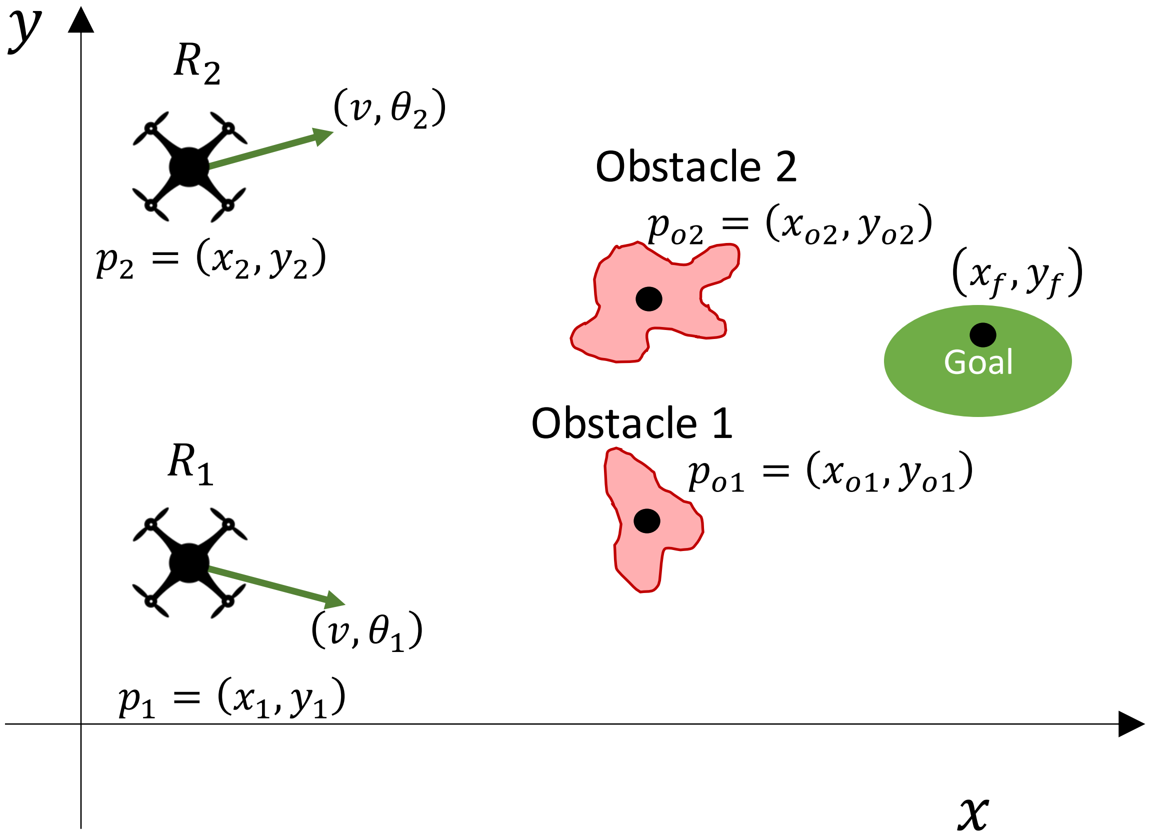
On/Off control for Path following



Design Application: Robot Coordination

- Autonomous mobile robots in a room, goal for each robot:
  - Reach a target at a known location
  - Avoid obstacles (positions not known in advance)
  - Minimize distance travelled
- Design Problems:
  - Cameras/vision systems can provide estimates of obstacle positions
    - When should a robot update its estimate of the obstacle position?
  - Robots can communicate with each other
    - How often and what information can they communicate?
  - High-level motion planning
    - What path in the speed/direction-space should the robots traverse?

Path planning with obstacle avoidance



- Assumptions:
  - Two-dimensional world
  - Robots are just points
  - Each robot travels with a fixed speed
- Dynamics for Robot R_i:
  - $\dot{x}_i = v \cos \theta_i; \dot{y}_i = v \sin \theta_i$
- Design objectives:
  - Eventually reach (x_f, y_f)
  - Always avoid Obstacle1 and Obstacle 2
  - Minimize distance travelled

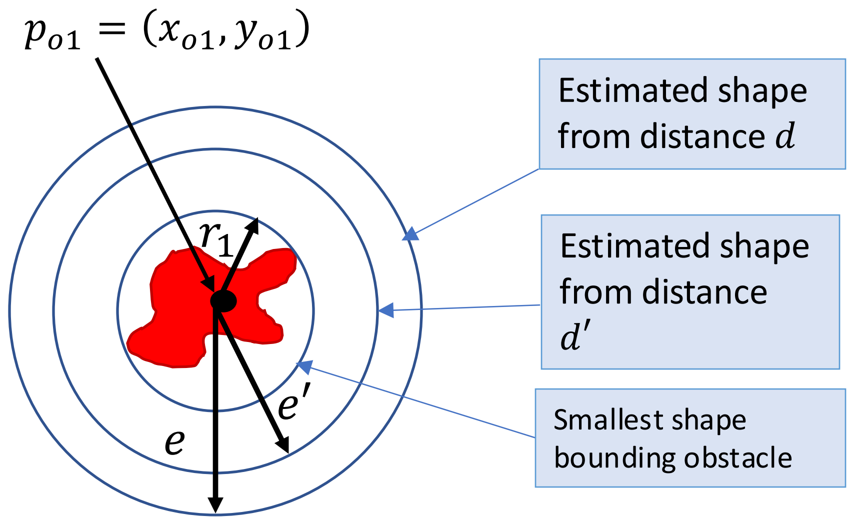
Divide path/motion planning into two parts

1. Computer vision tasks
   - Assume computer vision algorithm identifies obstacles, and labels them with some easy-to-represent geometric shape (such as a bounding boxes)
     - In this example, we will assume a sonar-based sensor, so we will use circles
2. Actual path planning task
   - Assuming the vision algorithm is correct, do path planning based on the estimated shapes of obstacles

Design challenge:

- Estimate of obstacle shape is not the smallest shape containing the obstacle
- Shape estimate varies based on distance from obstacle

Estimation error



Estimated radius (from current distance d)
$e = r + a(d - r)$,
where $a \in [0,1]$ is a constant

- Robot R_1 maintains radii e_1 and e_2 that are estimates of obstacle sizes
- Every τ seconds, R_1 executes following update to get estimates of shapes of each obstacle:

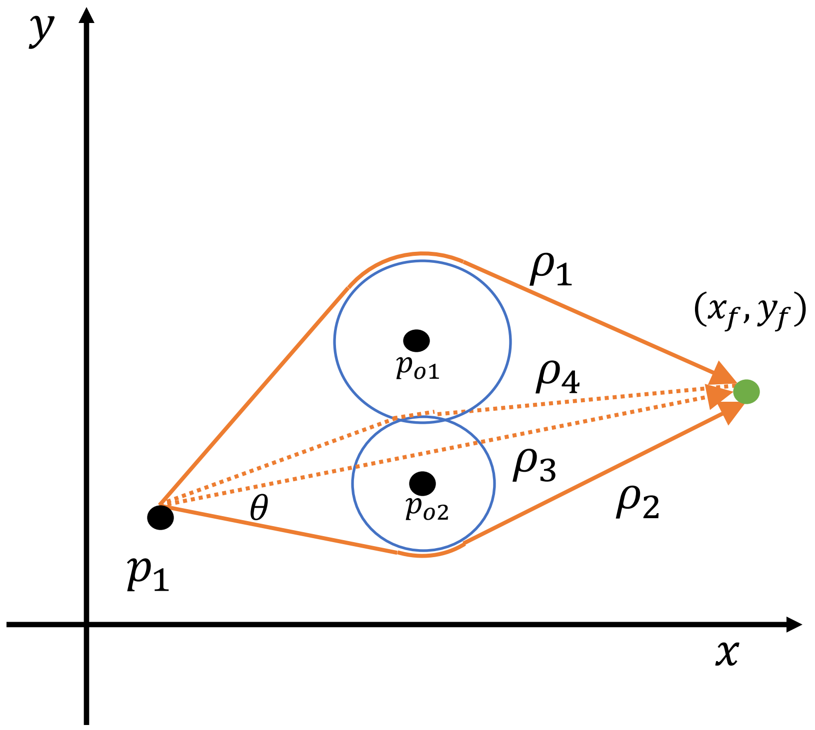
$$e_1 := \min\left(e_1, r_1 + a(\|p_1 - p_{o1}\| - r_1)\right)$$

  - We don't know r_1, but we are guaranteed that we get a radius of an estimated shape of the obstacle that is exactly: $r_1 + a\left(d\left(p_1, p_{o_1}\right) - r_1\right)$
  - p_1 is position of R_1

- Computation of e_2 is symmetric

$$e_2 := \min\left(e_2, r_2 + a(\|p_1 - p_{o2}\| - r_2)\right)$$

Path planning



- Choose shortest path ρ_3 to target (to minimize time)
- If estimate of obstacle 1 intersects ρ_3, calculate two paths that are tangent to obstacle 1 estimate
- If estimate of obstacle 2 intersects ρ_3, or obstacle 1, calculate tangent paths
- Plausible paths: ρ_1 and ρ_2
- Calculate shorter one as the planned path

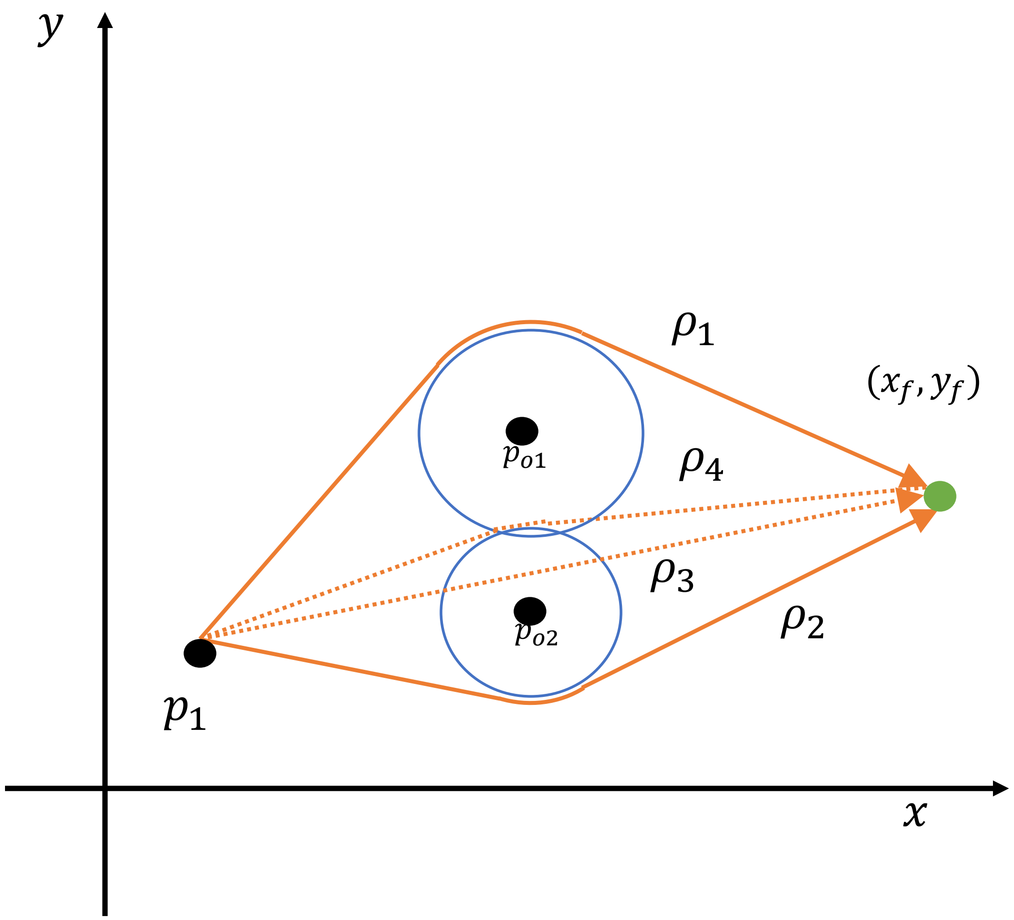
Dynamic path planning

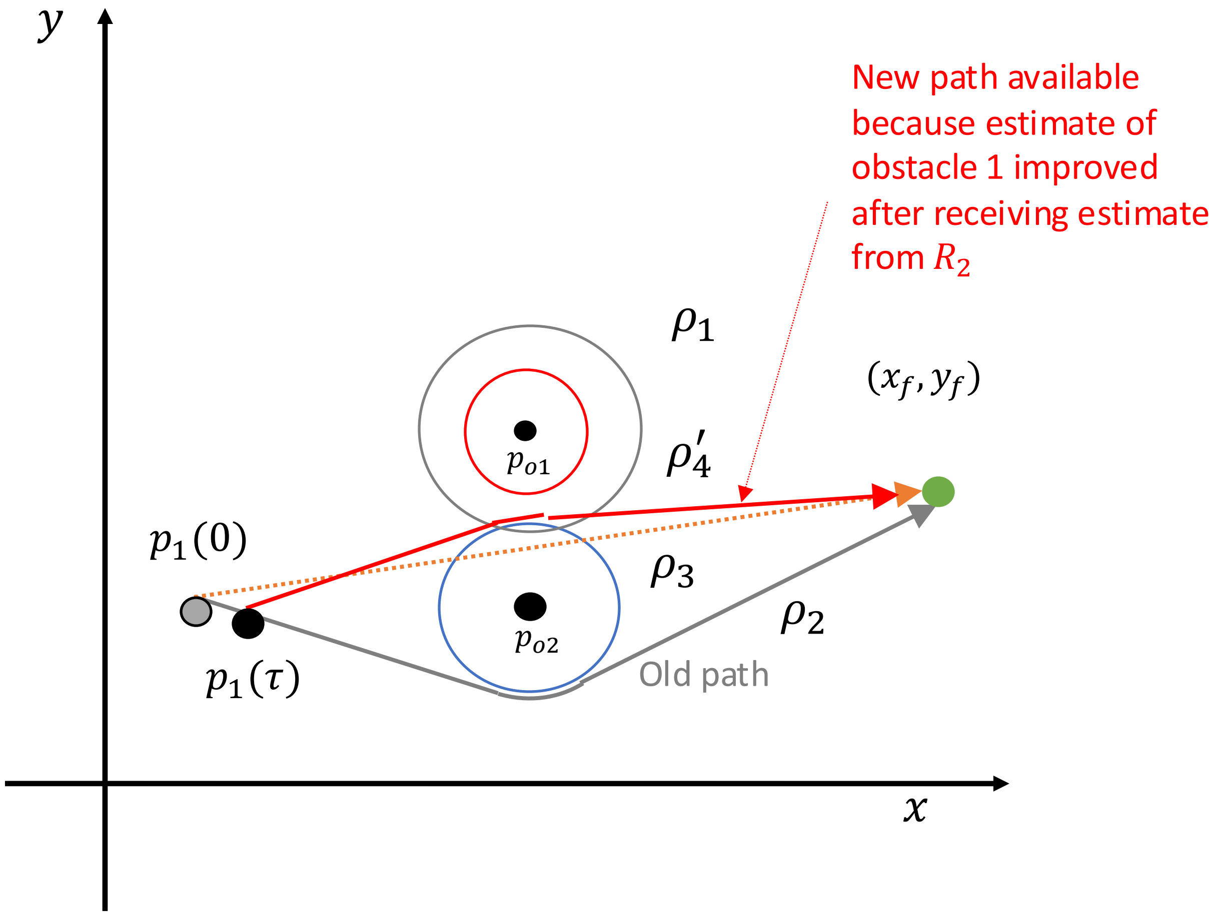
- Path planning inputs:
  - Current position of robot
  - Target position
  - Position of obstacles and estimates
- Output:
  - Direction for motion assuming obstacle estimates are correct
- May be useful to execute planning algorithm again as robot moves!
  - Because estimates will improve closer to the obstacles
  - Invoke planning algorithm every τ seconds

Communication improves planning

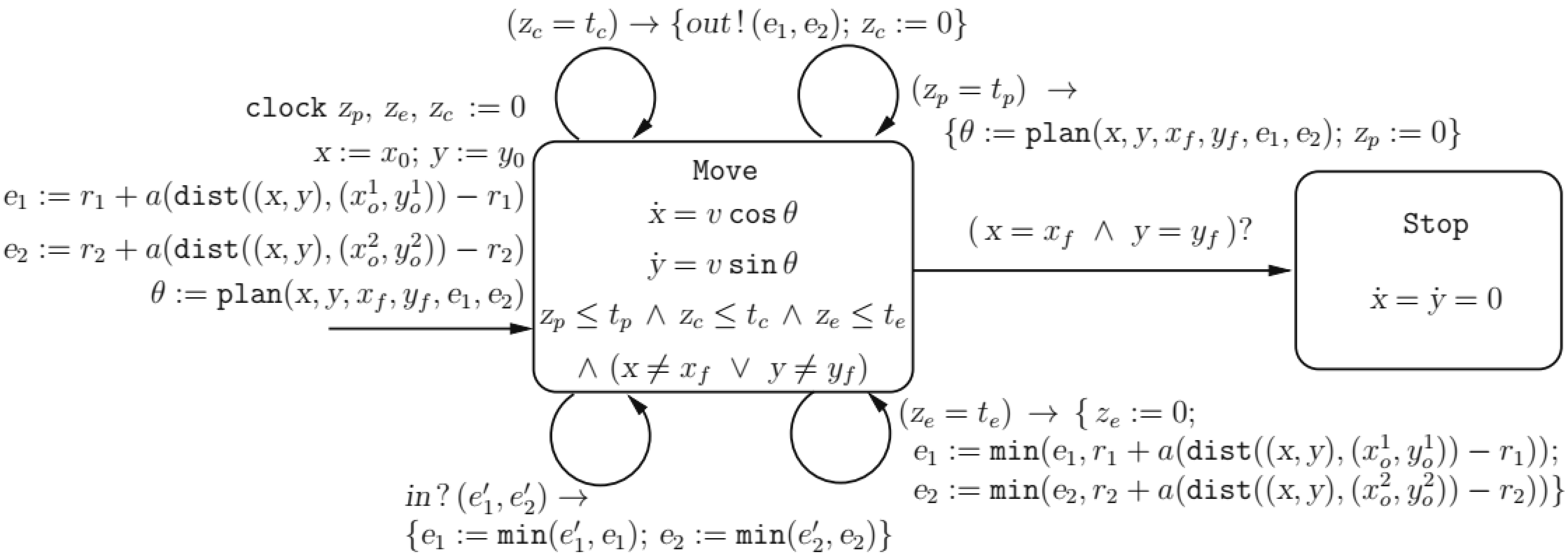
- Every robot has its own estimate of the obstacle
- R_2's estimate of obstacle might be better than R_1's
- Strategy: every τ seconds, send estimates to other robot, and receive estimates
- For estimate e_i, use final estimate = $\min(e_i, e_i^{recv})$
- Re-run path planner

Improved path planning through communication





Hybrid State Machine for Communicating Robot



Advantage of using hybrid processes

- Hybrid models combine computation, communication and control
- Most real-world controllers are digital/discrete-time controllers: hybrid process/automata models describe underlying mathematical model for most CPS applications!
- We can perform design-space exploration through simulations and check safety/correctness through formal techniques such as reachability analysis