

# Cyber-Physical Systems (CSC.T431)

Hybrid Model

Instructor: Takuo Watanabe (Department of Computer Science)

# Agenda

- Hybrid Systems

## Course Support & Material

- Slides: OCW-i
- Course Web: <https://titech-cps.github.io>
- Course Slack: [titech-cps.slack.com](https://titech-cps.slack.com)

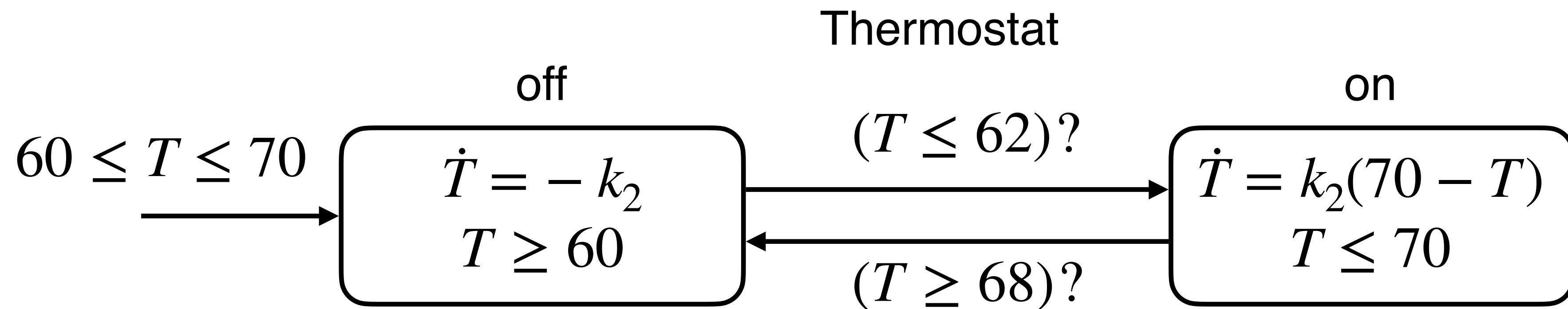
# Models of Reactive Computation

- Continuous-Time Model for Dynamical Systems
  - Components run synchronously under continuous time.
  - The execution of a component is described using a system of differential equations.
- Timed Models
  - Processes run asynchronously, with discrete actions
  - Clocks evolve continuously, and constraints on clocks allow synchronous / global coordination
- Hybrid Systems
  - Generalization of Timed Processes
  - During timed transitions, evolution of state/output variables specified differential equations

# Hybrid Process

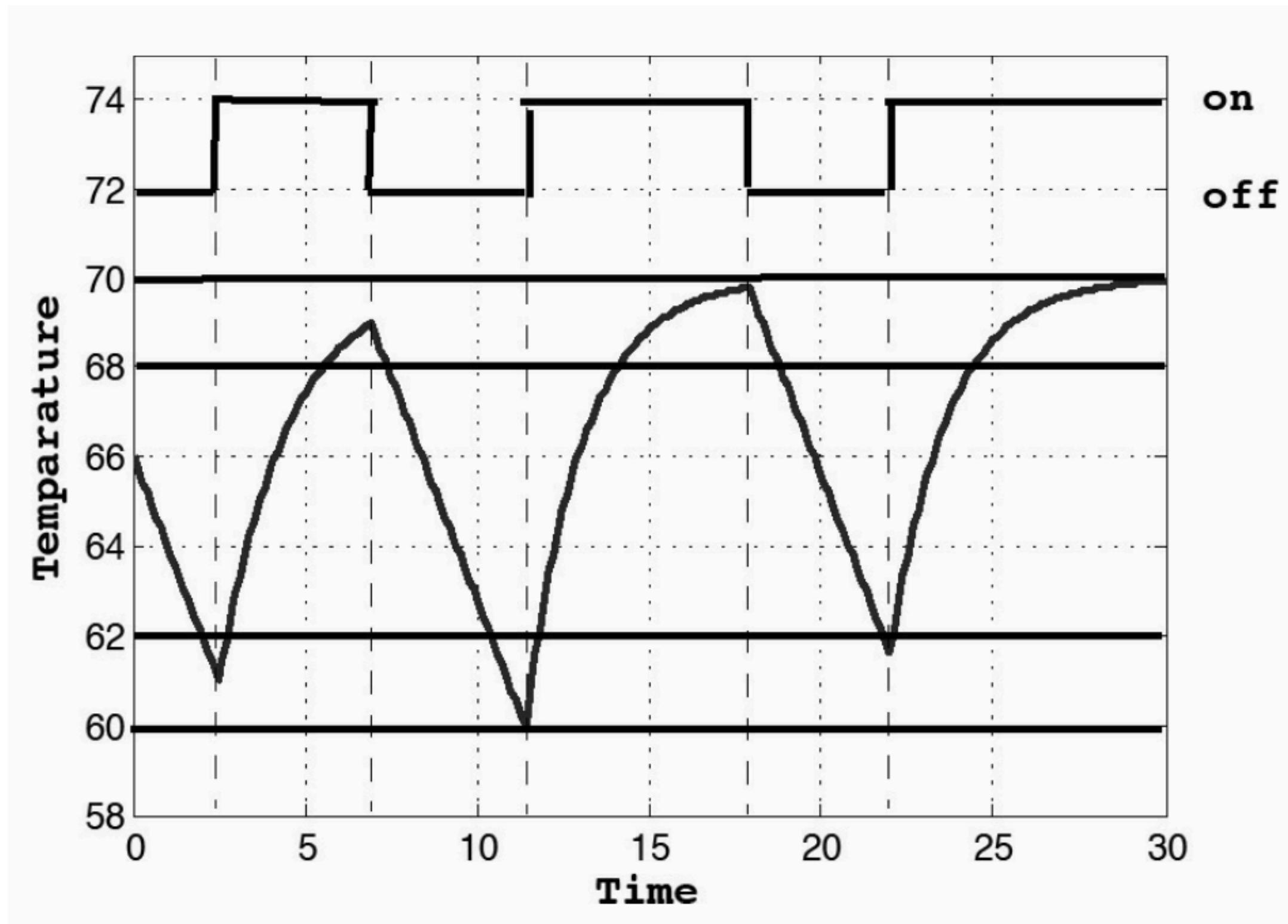
- Similar to timed process
  - Each process has input / output channels and state variables
- **cont** : type of continuously evolving values
  - A variable of type cont takes values from the set of real numbers (or an interval of real numbers) and is updated continuously as time progresses while a process stays in a mode.
  - Specified by differential / algebraic equations associated with modes.

# Ex. Switching Thermostat



- $T$  : continuous-time variable (of type cont) expressing the temperature
- The dynamics of the system is given by differential equations
- The constraint associated with each mode specifies that the process can stay in the mode only as long as the condition holds.
- A mode switch may happen any time while the guard associated with it is satisfied.

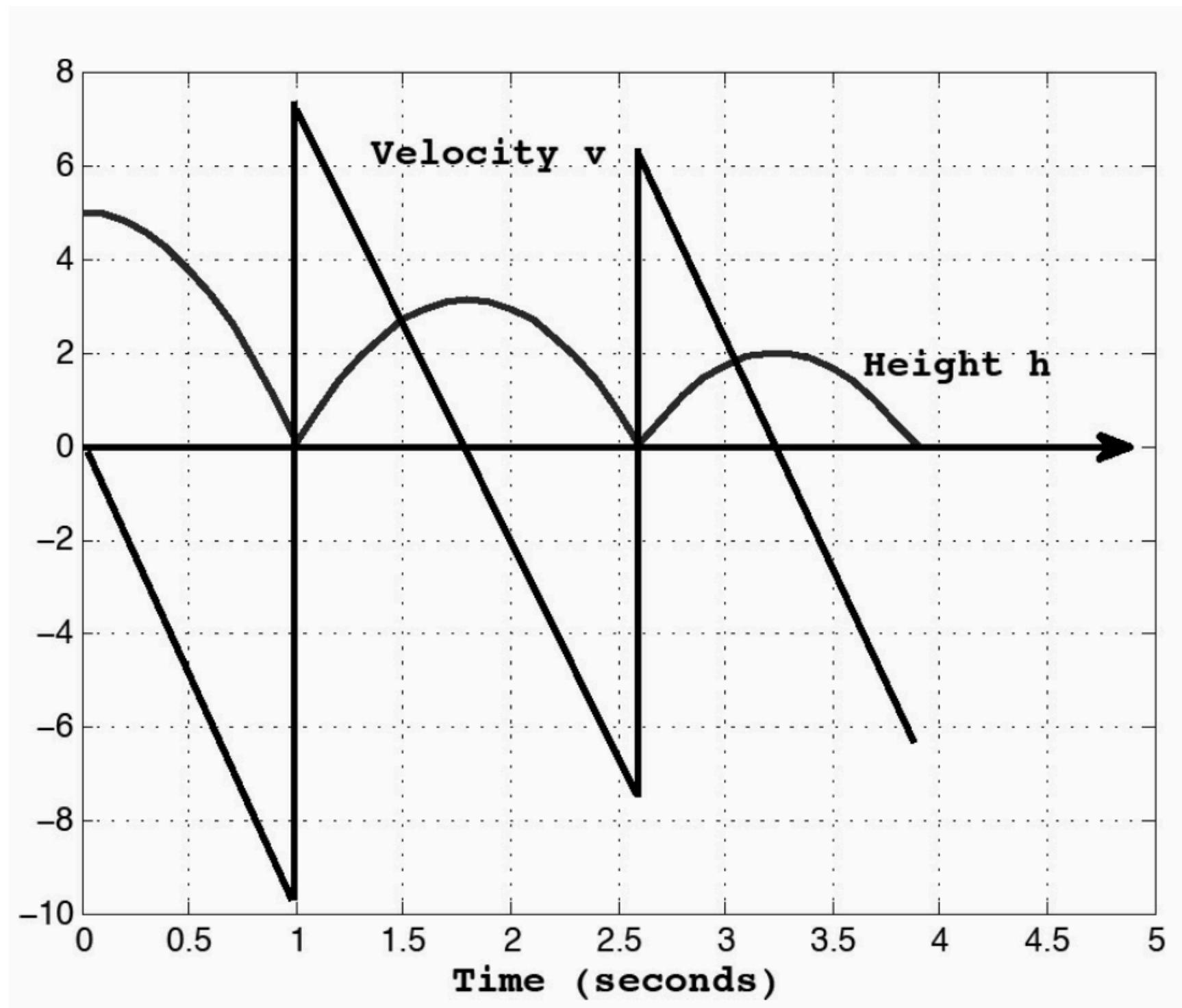
# Ex. Switching Thermostat



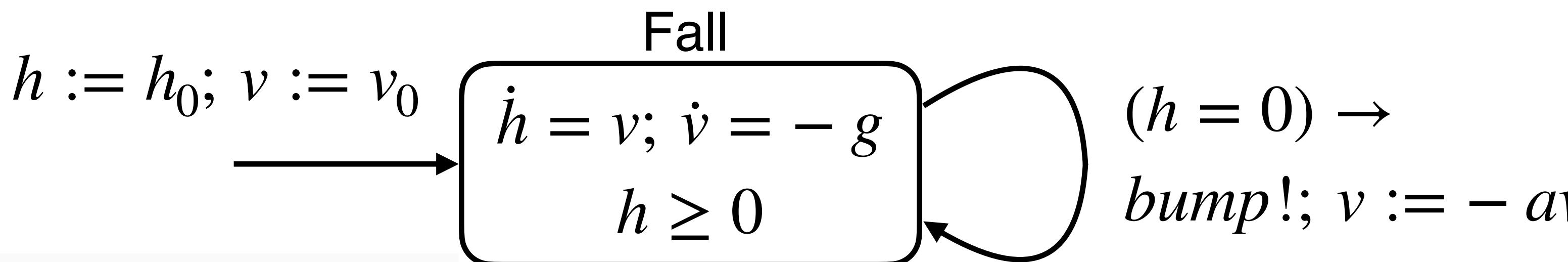
$$T_0 = 66, k_1 = 0.6, k_2 = 2$$

- If the process switches to off at time  $t^*$  with temperature  $T^*$ , then until the next mode switch,  $T = T^* - k_2(t - t^*)$ .
- If the process switches to on at time  $t^*$  with temperature  $T^*$ , then until the next mode siwtch,  
$$T = 70 - (70 - T^*)e^{-k_1(t-t^*)}.$$

# Ex. Bouncing Ball



BouncingBall



- Consider a ball dropped from an initial height  $h_0$  with an initial velocity  $v_0$ .
- Differential equations  $\dot{h} = v$  and  $\dot{v} = -g$  give the dynamics of the falling ball.
- When the ball hits the ground, *i.e.*, when  $h = 0$ , the velocity discontinuously changes to  $-av$  where  $a$  is a constant ( $0 < a < 1$ ).

# Hybrid Process

## Definition (1/3)

A hybrid process  $HP$  consists of:

- (1) an asynchronous process  $P$  where some of the input, output, and state variables are of type cont,
- (2) a continuous-time invariant  $CI$ , which is a Boolean expression over the state variables  $S$ ,
- (3) for every output variable  $y$  of type cont, a real-valued expression  $h_y$  over the state and input variables of type cont, and
- (4) for every state variable  $x$  of type cont, a real-valued expression  $f_x$  over the state and input variables of type cont.

# Hybrid Process

## Definition (2/3)

- Inputs, outputs, states, initial states, internal actions, input actions, and output actions of  $HP$  are the same as those of the asynchronous process  $P$ .
- Given a state  $s$ , a real-valued time  $\delta > 0$ , and an input signal  $\bar{u}$  for every input variable  $u$  of type cont over the interval  $[0, \delta]$ , the corresponding timed action of  $HP$  is the differentiable state signal  $\bar{S}$  over the state variables, and the signal  $\bar{y}$  for every output variable  $y$  of type cont over  $[0, \delta]$  such that:
  - (1) for every state variable  $x$ ,  $\bar{x}(0) = s(x)$ ,
  - (2) for every discrete state variable  $x$  and time  $t \in [0, \delta]$ ,  $\bar{x}(t) = s(x)$ ,

# Hybrid Process

## Definition (2/3)

- (3) for every output variable  $y$  of type cont and time  $t \in [0, \delta]$ ,  $\bar{y}(t)$  equals the value of  $h_y$  evaluated using the values  $\bar{u}(t)$  and  $\bar{S}(t)$ ,
- (4) for every state variable  $x$  of type cont and time  $t \in [0, \delta]$ , the time derivative  $(d/dt)\bar{x}(t)$  equals the value of  $f_x$  evaluated using the values  $\bar{u}(t)$  and  $\bar{S}(t)$ ,
- (5) for all  $t \in [0, \delta]$ , the continuous-time invariant  $CI$  is satisfied by the values  $\bar{S}(t)$  of the state variables at time  $t$ .

# Ex. Thermostat

## Formal Definition (1/2)

- No input variables. A single output variable  $T$  of type cont.
- A discrete state variable  $mode$  of type {off, on}, and a state variable  $T$  of type cont.
  - The initial value of  $mode$  is off and the initial value of  $T$  is nondeterministically chosen from [60, 70].
- No output tasks. This means that the value of  $T$  is not transmitted during the discrete actions
- Two internal tasks corresponding to the mode-switches:
  - $(mode = \text{off} \wedge T \leq 62) \rightarrow mode := \text{on}$
  - $(mode = \text{on} \wedge T \geq 68) \rightarrow mode := \text{off}$

# Ex. Thermostat

## Formal Definition (2/2)

- The expression defining the value of the output variable  $T$  equals to the state variable  $T$
- The expression defining the derivative of the state variable  $T$  is given by the conditional expression:  
$$\text{if } (\textit{mode} = \text{off}) \text{ then } -k_2 \text{ else } k_2(70 - T).$$
- The continuous-time invariant  $CI$  is given by the expression:  
$$[(\textit{mode} = \text{off}) \rightarrow T \geq 60] \wedge [(\textit{mode} = \text{on}) \rightarrow T \leq 70].$$

# Execution

- At each step, either an internal, input, output, or a timed action is executed.
- Ex. Thermostat
  - (off, 66)  $\xrightarrow{2.5}$  (off, 61)  $\xrightarrow{\varepsilon}$  (on, 61)  $\xrightarrow{3.7}$  (on, 69.02)  $\xrightarrow{\varepsilon}$  (off, 69.02)  $\xrightarrow{4.4}$  (off, 60.22)  $\xrightarrow{\varepsilon}$ 
    - (on, 60.22)  $\xrightarrow{7.6}$  (on, 69.9)  $\xrightarrow{\varepsilon}$  (off, 69.9)  $\xrightarrow{4.1}$  (off, 61.7)  $\xrightarrow{\varepsilon}$  (on, 61.7)  $\xrightarrow{7.7}$  (on, 69.92)
- The notion of reachability and invariant can be defined in the same way as asynchronous processes.

# Process Composition

- Two hybrid processes are compatible and can be composed if
  - their state variables are disjoint,
  - their output variables are disjoint, and
  - there are no cyclic dependencies among the continuously updated common input / output variables.
- The continuous-time invariant of the composite process is the conjunction of the continuous-time invariants of the component processes.

# Execution of the Bouncing Ball

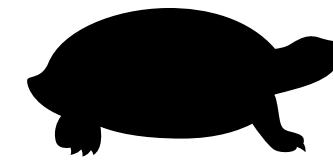
- Assumption:  $v_0 = 0$
- Before the 1st bump:  $\bar{h}(t) = h_0 - gt^2/2$ ,  $\bar{v}(t) = -gt$
- Time at which the 1st bump occurs:  $\delta_1 = \sqrt{2h_0/g}$
- The velocity just before the 1st bump:  $v_1 = -\sqrt{2gh_0}$
- The velocity just after the 1st bump:  $v_2 = -av_1 = a\sqrt{2gh_0}$
- Between the 1st and 2nd bumps:  $\bar{h}(t') = v_2 t' - gt'^2/2$ ,  $\bar{v}(t') = v_2 - gt'$
- Time between 1st and 2nd bumps:  $\delta_2 = 2v_2/g$
- The velocity just before the 2nd bump:  $-v_2$
- The velocity just after the 2nd bump:  $v_3 = -av_2 = a^2 v_1$

$$t' = t - \delta_1$$

# Execution of the Bouncing Ball

- Execution of the bouncing ball
  - $(h_0, 0) \xrightarrow{\delta_1} (0, -v_1) \xrightarrow{\text{bump!}} (0, v_2) \xrightarrow{\delta_2} (0, -v_2) \xrightarrow{\text{bump!}} (0, v_3) \xrightarrow{\delta_2} \dots$
- For each  $i$ ,  $v_{i+1} = av_i = a^i v_1$  and  $\delta_{i+1} = 2v_{i+1}/g = 2a^i v_1/g$ .
- Since  $0 < a < 1$ ,  $\lim_{i \rightarrow \infty} v_i = 0$  and  $\lim_{i \rightarrow \infty} \delta_i = 0$ .
- We have  $\sum_{i=0}^{\infty} \delta_i = v_1(1 + a)/(1 - a)$
- Zeno behavior: Infinitely many bumps in finite time!

# Zeno's Paradox



- Described by Greek philosopher Zeno in context of a race between Achilles and a tortoise
- The tortoise has a head start over Achilles. Suppose the tortoise is ahead of Achilles by  $d_1$  meters at the beginning of the 1st round.
- By the time Achilles has covered this distance  $d_1$ , the tortoise has moved a little bit ahead, say  $d_2$  meters, with  $d_2 < d_1$ .
- In the second round, Achilles has to cover another  $d_2$  meters, but the tortoise has moved farther by  $d_3$  meters, with  $d_3 < d_2$ .
- By inductive reasoning, for every  $n$ , after  $n$  rounds, the tortoise is ahead of Achilles by a non-zero distance, and thus Achilles will never be able to catch up with the tortoise!

# Zeno Execution and Zeno State

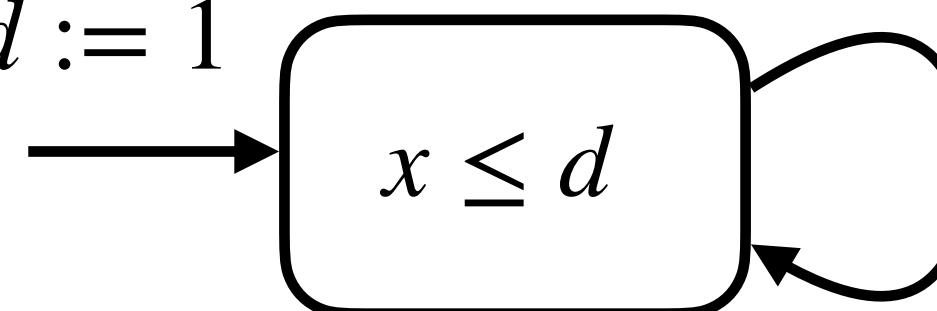
- Zeno Execution: An infinite execution of a hybrid process  $HP$  is said to be a *Zeno execution* if the sum of the durations of all the timed actions in the execution is bounded by a constant.
- Non-Zeno execution: An execution in which the sum diverges.
- Zeno State: A state  $s$  of  $HP$  is said to be (1) Zeno if every execution from  $s$  is Zeno, or (2) Non-Zeno if there exists some non-Zeno executions from  $s$ .
- $HP$  is said to be Zeno process if some reachable state is Zeno.
- Ex. Thermostat: non-Zeno, BouncingBall : Zeno

# Zeno Process

Zeno

Every possible execution is Zeno

clock  $x := 0$ ; real  $d := 1$

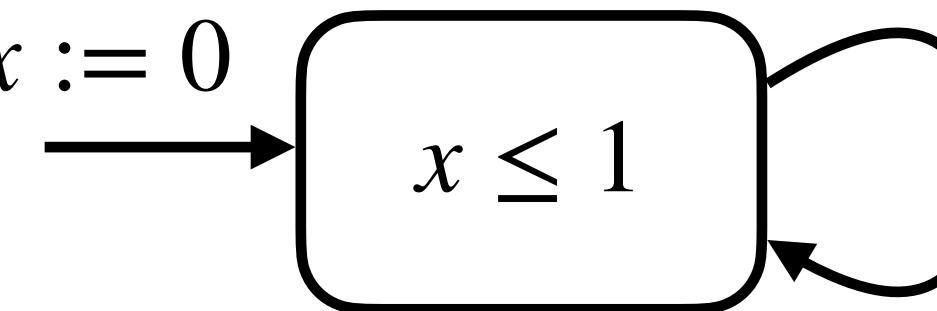


$(x = d) \rightarrow a!; d := d/2; x := 0$

Non-Zeno

Some executions are Zeno  
and some are non-Zeno

clock  $x := 0$

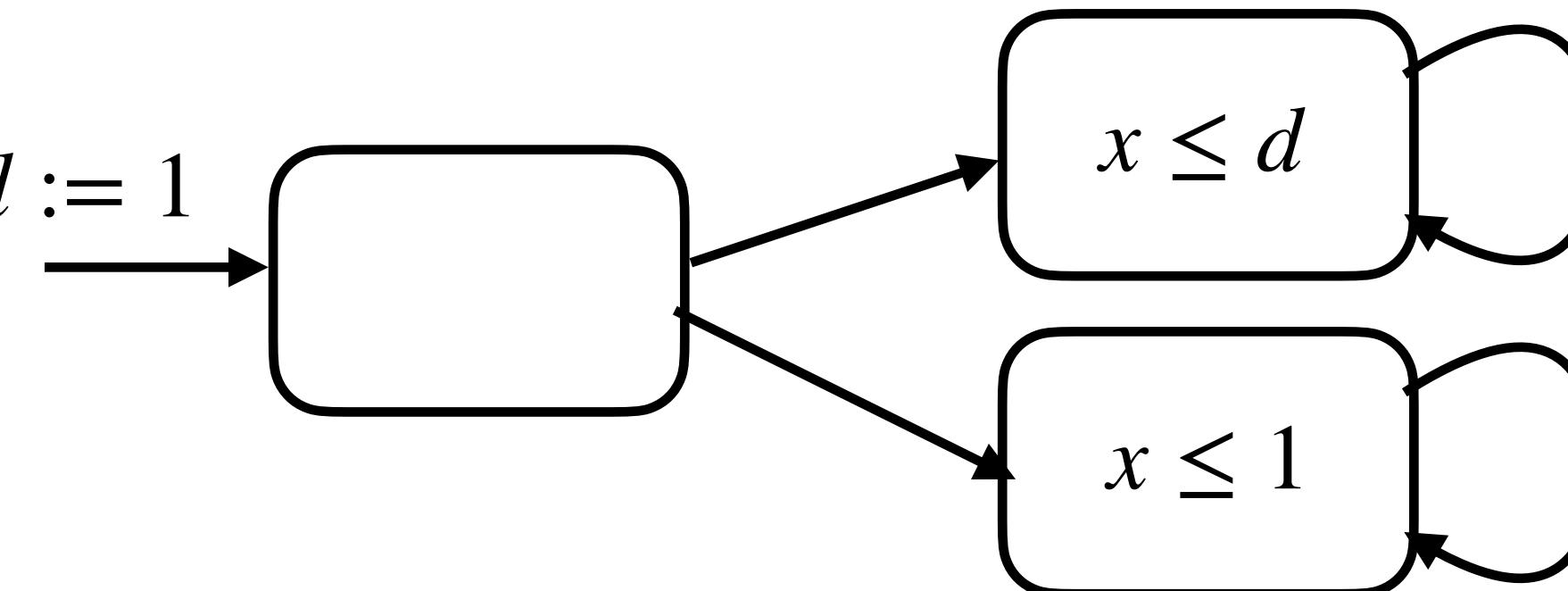


$(x > 0) \rightarrow a!; x := 0$

Zeno

clock  $x := 0$ ; real  $d := 1$

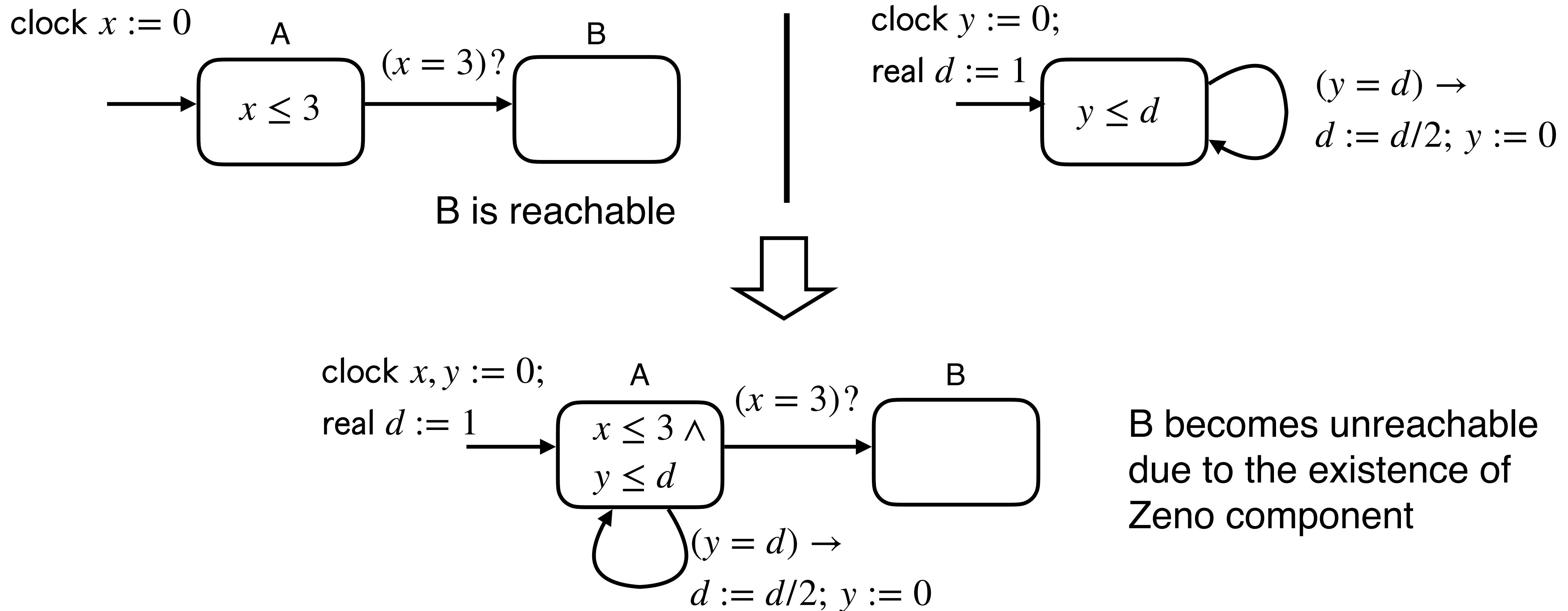
Zeno state is reachable



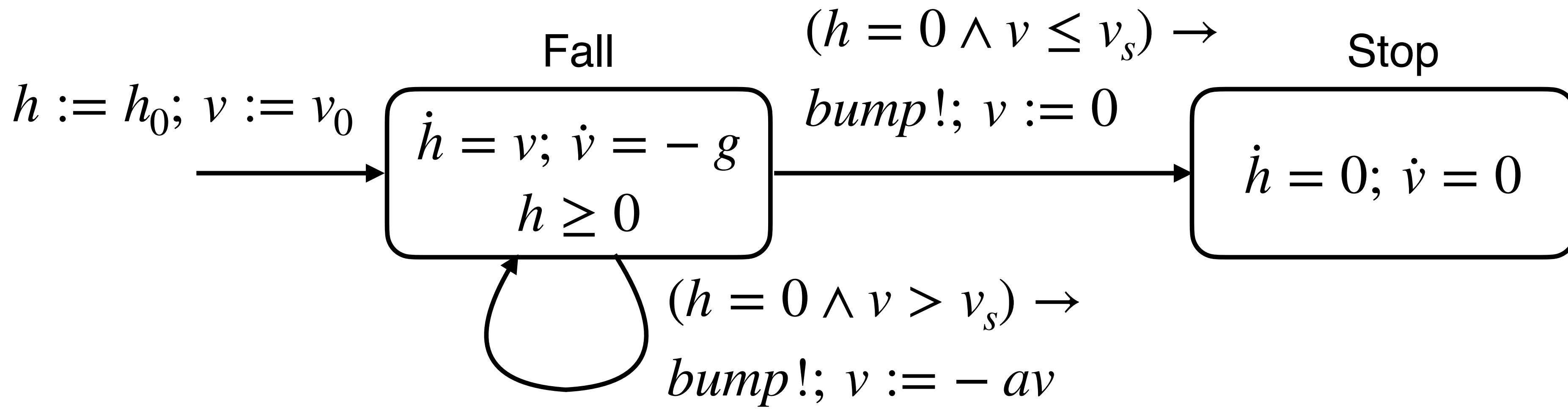
$(x = d) \rightarrow a!; d := d/2; x := 0$

$(x > 0) \rightarrow a!; x := 0$

# Zeno Process and Reachability



# A Non-Zeno Model of the Bouncing Ball



- A Zeno process can be converted into a non-Zeno process by modifying the model so that the model does not force mode-switching after a certain short durations.

# Stability



A

$$\begin{aligned}\dot{s}_1 &= -s_1 - 100s_2 \\ \dot{s}_2 &= 10s_1 - s_2 \\ s_2 &\geq -0.2s_1\end{aligned}$$



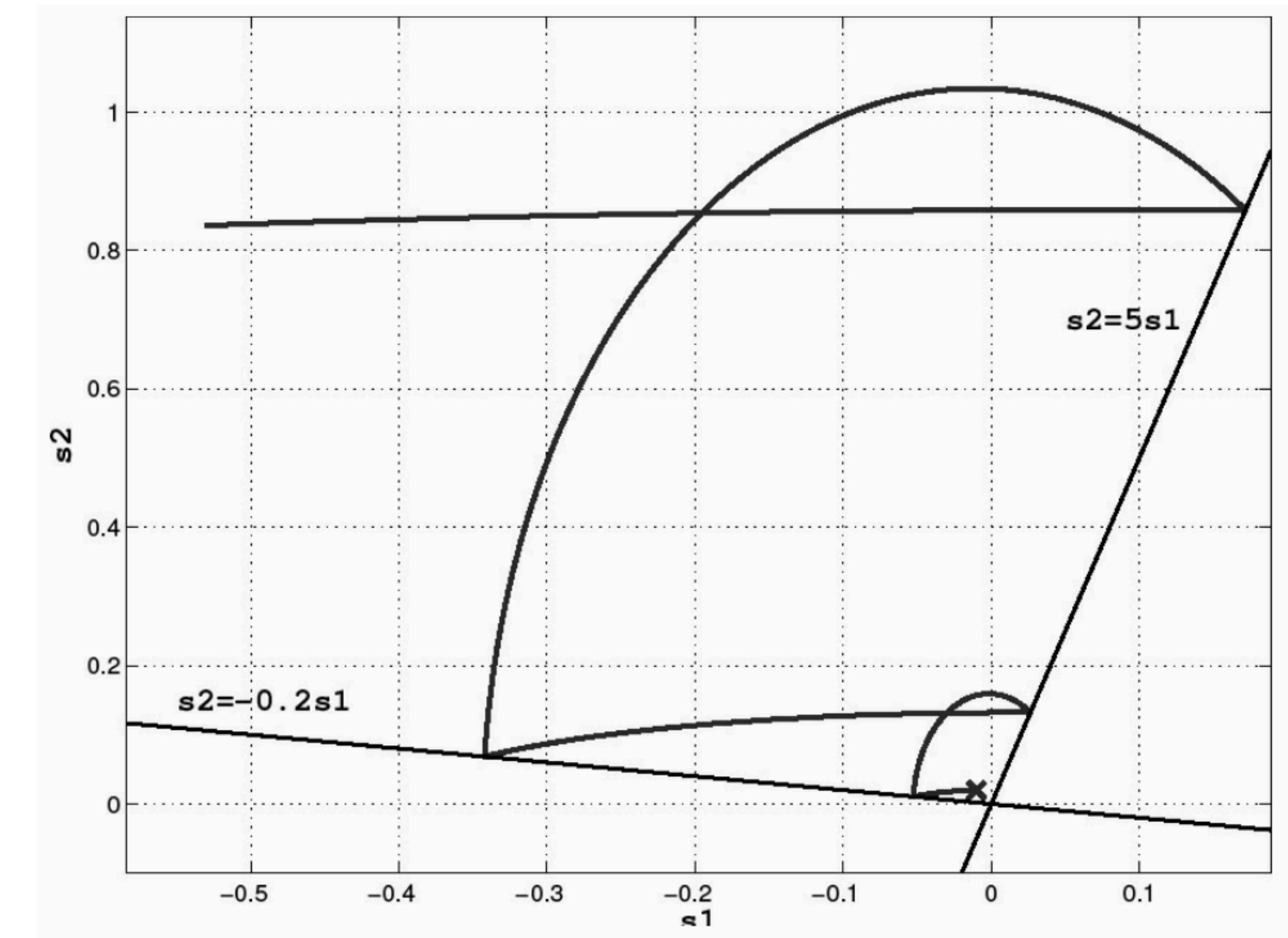
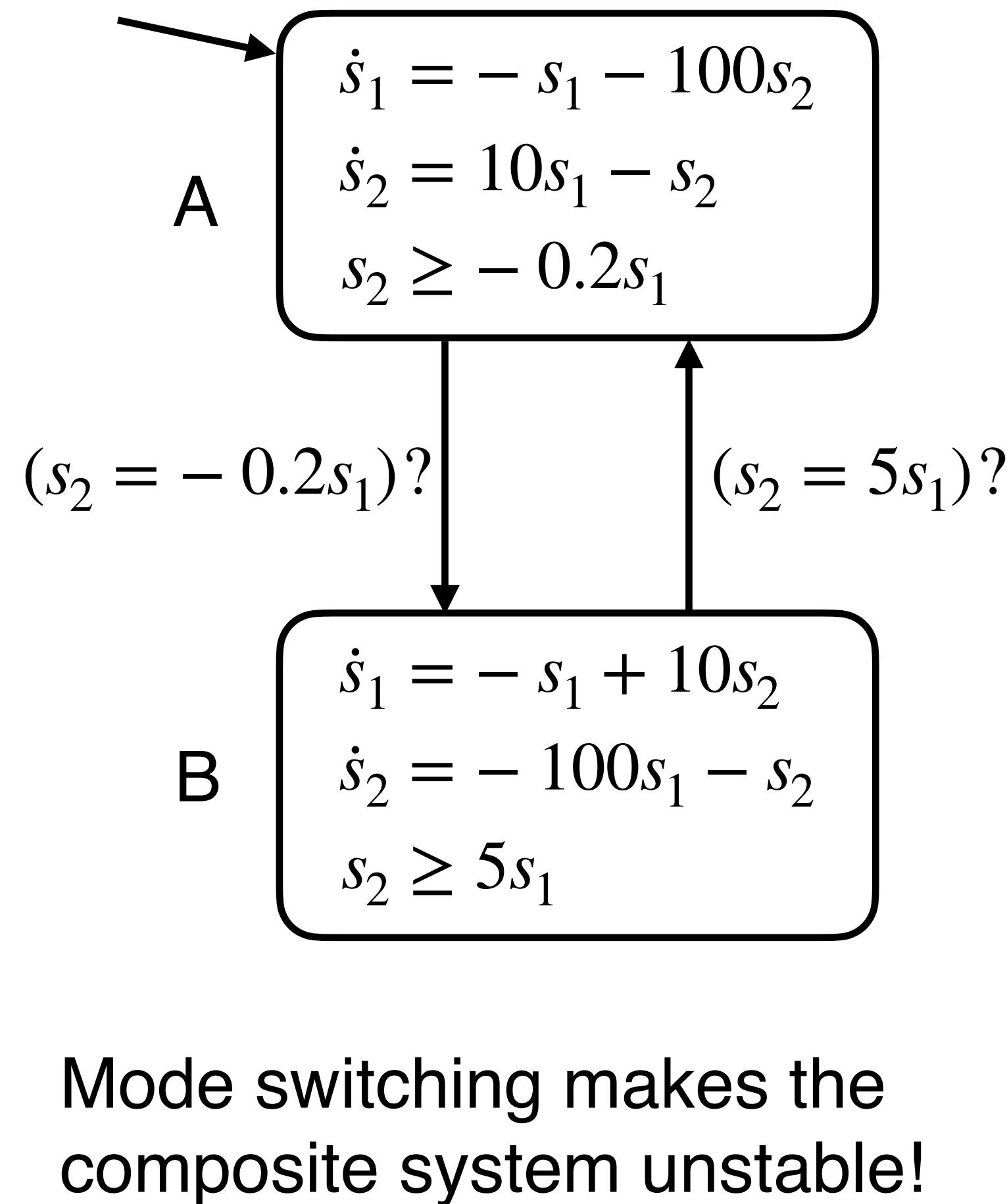
B

$$\begin{aligned}\dot{s}_1 &= -s_1 + 10s_2 \\ \dot{s}_2 &= -100s_1 - s_2 \\ s_2 &\geq 5s_1\end{aligned}$$

- **Stability of Dynamical Systems**

- A state  $s_e$  of a dynamical system is an equilibrium if the system continues to stay in the state (without external inputs).
  - An equilibrium  $s_e$  is stable if we perturb the system state slightly, then the state of the system stays within a bounded distance from  $s_e$ .
  - An equilibrium  $s_e$  is asymptotically stable if the state of the system converges to  $s_e$ .
- Ex. Both A and B are asymptotically stable.

# Instability Due to Mode Switching



# Programming Reactive Systems

# Programming Reactive Systems

- Reactive Systems: A computational system that reacts with its environment
  - Ex. Embedded Systems, GUI, etc.
- Patterns for Programming Reactive Systems
  - Polling
  - Callback (Interrupts)

# Ex. Fan Controller

- Two environmental sensors: Temperature & Humidity
- Controlling policy
  - Turns the fan ON when  $DI \geq 75$
  - OFF otherwise
- $DI$  : Discomfort Index  $(0.81T + 0.01H(0.99T - 14.3) + 46.3)$ 
  - $T$  : temperature (Celsius)
  - $H$  : relative humidity (%)
  - Empirically, 50% of people feel uncomfortable if  $DI$  reaches 75.

# Typical Pattern: Polling (a)

```
#define FAN_PIN 21
DHT12 dht12;

void setup() {
    Wire.begin();
    pinMode(FAN_PIN, OUTPUT);
}

// Main polling loop
void loop () {
    // read sensor values via I2C
    dht12.update();
    float tmp = dht12.temperature();
    float hmd = dht12.humidity();
    // calculate current discomfort index
    float di = 0.81 * tmp + 0.01 * hmd
                * (0.99 * tmp - 14.3) + 46.3;
    // turns the fan on if di >= 75, off otherwise
    digitalWrite(FAN_PIN, di >= 75.0);
}
```

# Typical Pattern: Polling (b)

```
#define FAN_PIN 21
TMP11 tmp11; // temperature sensor
HMD22 hmd22; // humidity sensor

void setup() {
    Wire.begin();
    pinMode(FAN_PIN, OUTPUT);
}

// Main polling loop
void loop () {
    // read sensor values via I2C
    tmp11.update(); // 3ms
    hmd22.update(); // 15ms
    float tmp = tmp11.read();
    float hmd = hmd22.read();
    // calculate current discomfort index
    float di = 0.81 * tmp + 0.01 * hmd
                * (0.99 * tmp - 14.3) + 46.3;
    // turns the fan on if di >= 75, off otherwise
    digitalWrite(FAN_PIN, di >= 75.0);
}
```

```
TMP11 tmp11;  
HMD22 hmd22;  
  
void do_action() {  
    float di = 0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3;  
    digitalWrite(FAN_PIN, di >= 75.0);  
}  
  
void tmp_handler () {  
    tmp = tmp11.read();  
    do_action();  
}  
  
void hmd_handler () {  
    hmd = hmd22.read();  
    do_action();  
}  
  
void setup() {  
    Wire.begin();  
    tmp11.attach(tmp_handler);  
    hmd22.attach(hmd_handler);  
}  
  
void loop () { vTaskDelay(portMAX_DELAY); }
```

Typical Pattern:  
Callbacks (a)

```

TMP11 tmp11;
HMD22 hmd22;
float tmp, hmd;
SemaphoreHandle_t sem;
TaskHandle_t tmp_reader_task, hmd_reader_task;

void tmp_reader_task_fun() {
    for (;;) {
        xTaskNotifyWait(0, 0, NULL, portMAX_DELAY);
        float tmp0 = tmp11.read();
        xSemaphoreTake(sem, portMAX_DELAY);
        tmp = tmp0;
        xSemaphoreGive(sem);
    }
}

void hmd_reader_task_fun() {
    for (;;) {
        xTaskNotifyWait(0, 0, NULL, portMAX_DELAY);
        float hmd0 = hmd22.read();
        xSemaphoreTake(sem, portMAX_DELAY);
        hmd = hmd0;
        xSemaphoreGive(sem);
    }
}

```

Typical Pattern:  
Callbacks (b)

```

void tmp_handler () { xTaskNotifyFromISR(tmp_reader_task); }

void hmd_handler () { xTaskNotifyFromISR(hmd_reader_task); }

void setup() {
    Wire.begin();
    sem = xSemaphoreCreateBinary();
    xTaskCreatePinnedToCore(tmp_reader_task_fun, "tmp_reader_task",
                           4096, NULL, 1, &tmp_reader_task, 1);
    xTaskCreatePinnedToCore(hmd_reader_task_fun, "hmd_reader_task",
                           4096, NULL, 1, &hmd_reader_task, 1);
    tmp11.attach(tmp_handler);
    hmd22.attach(hmd_handler);
}

void loop () {
    float di;
    xSemaphoreTake(sem, portMAX_DELAY);
    di = 0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3;
    xSemaphoreGive(sem);
    digitalWrite(FAN_PIN, di >= 75.0);
    delay(100);
}

```

**Typical Pattern:  
Callbacks (b)**

# Polling and Callbacks

- Polling
  - OK for simple system
  - The slowest input govern the input timing
- Callbacks
  - Can handle different timing inputs
  - Split the code into small pieces
  - Callback functions (handlers) may have constraints
- In general, the above patterns are used with threads and/or state machines. This complicates the code and lowers the readability, maintainability, etc.

# Functional Reactive Programming (FRP)

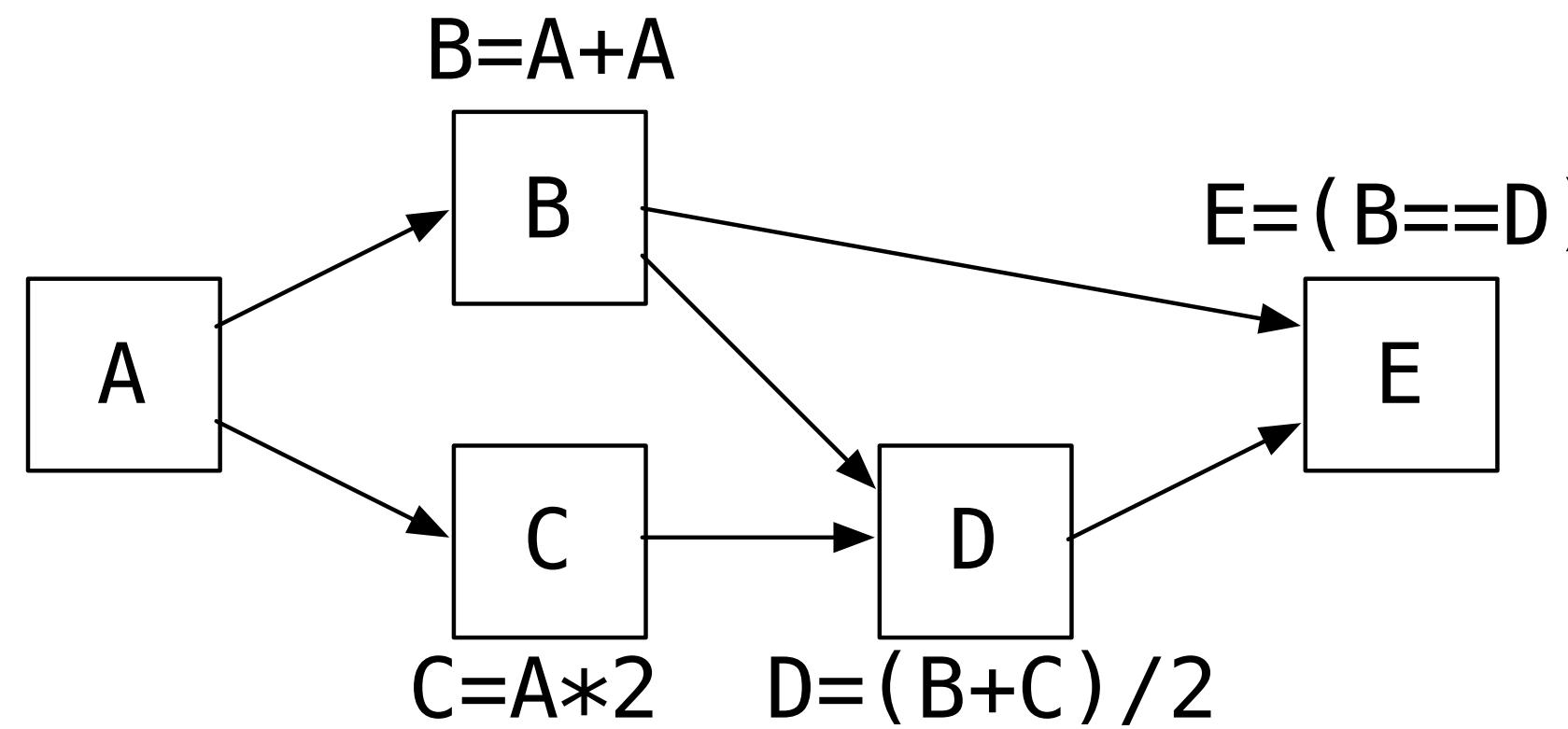
- A programming paradigm for reactive systems based on functional (declarative) abstractions to express time-varying values and events.
- Time-Varying Values (aka Signals) : Signal  $\alpha = \text{Time} \rightarrow \alpha$ 
  - First class entities representing continuously changing data (of type  $\alpha$ ) over time
  - Ex. temperature :: Signal Float
- Events: Event  $\alpha = [(\text{Time}, \alpha)]$ 
  - List of discrete events of type  $\alpha$
  - Ex. button :: Event Bool

# Reactive Systems in FRP

- A reactive system is a function of type  $\text{Signal } \alpha \rightarrow \text{Signal } \beta$ 
  - $\alpha$  : input type,  $\beta$  : output type
- Causality Requirement
  - The value of the output at time  $t$  must be determined by input on interval  $[0, t]$ .
- pure and stateless
  - if output at time  $t$  only depends on input at time  $t$
- impure and stateful
  - if output at time  $t$  depends on input over the interval  $[0, t]$

# Glitch Freedom

- A program can be represented as a directed graph without cycles (= DAG) whose nodes and edges correspond to the signals and their dependencies.



- In this example, a *glitch* is the phenomenon that there is an observable difference between B and D due to the delays in updating B, C and D.
- Glitch freedom is a desired property of FRP.

# Time-Leak / Space-Leak

- Unrestricted access to a time varying values leads to time/space-leaks
- $x :: \text{Signal } \alpha = \text{Time} \rightarrow \alpha$ 
  - The value of  $x$  at time  $t$  (i.e.,  $x(t)$ ) depends on the values of  $x$  in the interval  $[0,t]$ .
- e.g., If  $x(t)$  is allowed to be accessed and
  - If  $x(t)$  can always be calculated using  $\{x(t') \mid t' \in [0,t)\}$ , it may take unexpectedly long time (time-leak).
  - If  $\{x(t') \mid t' \in [0,t)\}$  is stored in somewhere to calculate  $x(t)$ , it may require unexpectedly large memory (space-leak)

# To Avoid Time-/Space-Leaks

- Restricted representation of Signal  $\alpha$ 
  - Signal  $\alpha$  as a (primitive) type other than  $\text{Time} \rightarrow \alpha$
- Arrowization
  - Abandon time-varying values themselves and use "functions" on time-varying values
  - Arrows are used to construct such "functions"
  - Ex. Yampa (FRP library for Haskell)

# FRP Languages / DSLs / Libraries

- Fran [Elliot97]
  - Haskell FRP library for animation
- Yampa [Hudak03][Coutney03]
  - Haskell FRP library based on the notion of Arrows
- FrTime [Cooper04]
  - Extension of Scheme for GUI, simulation
- Flapjax [Meyerovich09], Elm [Czaplicki13]
  - Library/language for client-side web programming
- The majority of the FRP systems proposed so far are based on Haskell or other languages that require rich runtime resource.

# Arrowized FRP

- Compose FRP program using *signal functions*
  - Signals are hidden from the code
- Signal Functions
  - $SF \alpha \beta \simeq \text{Signal } \alpha \rightarrow \text{Signal } \beta$
  - $SF \alpha \beta$  is an *abstract type*
    - Operations on it provide a disciplined way to compose signals
    - The notion of *Arrows* provides the discipline.

# Arrow Operations (1)

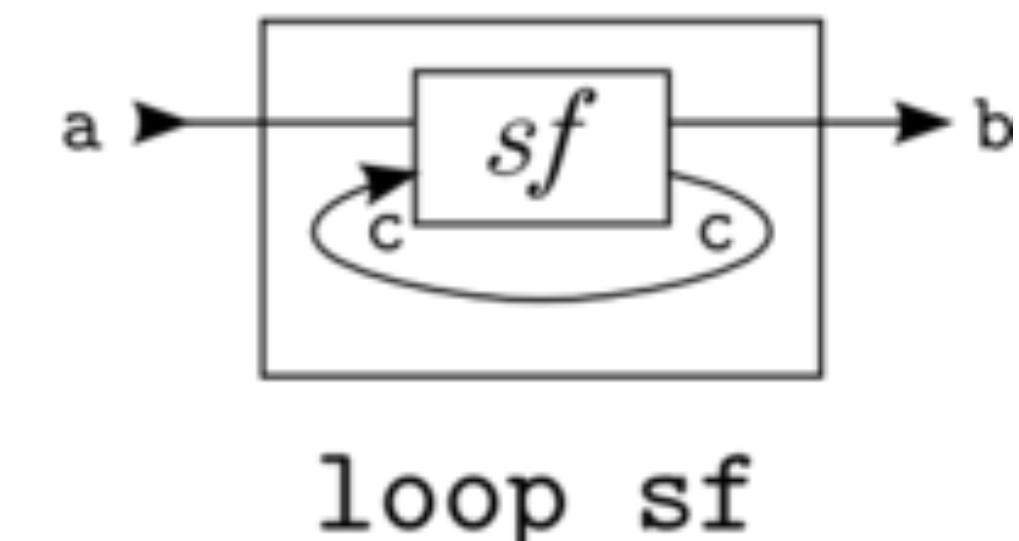
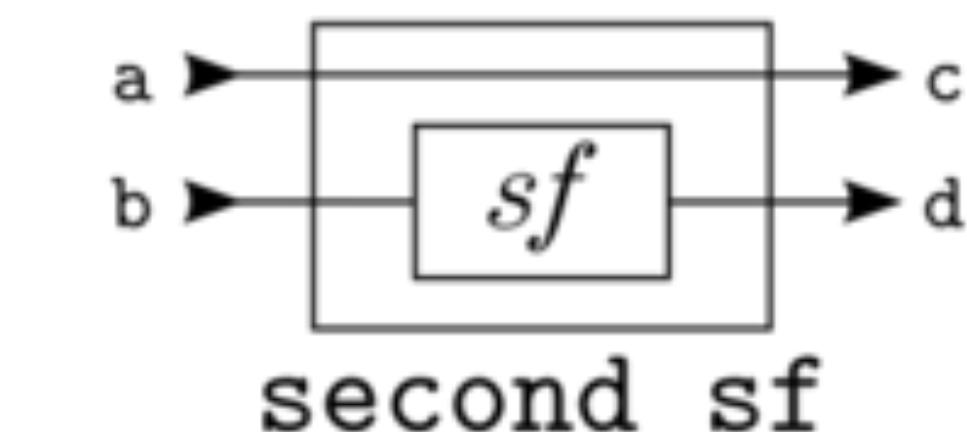
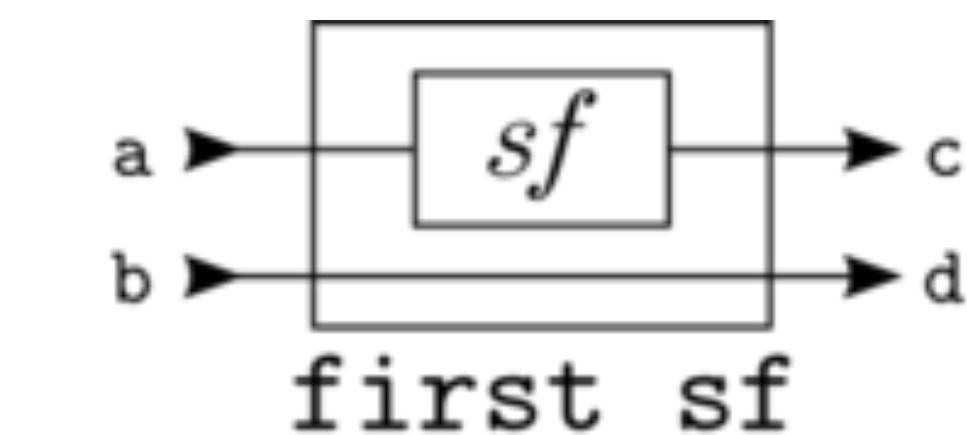
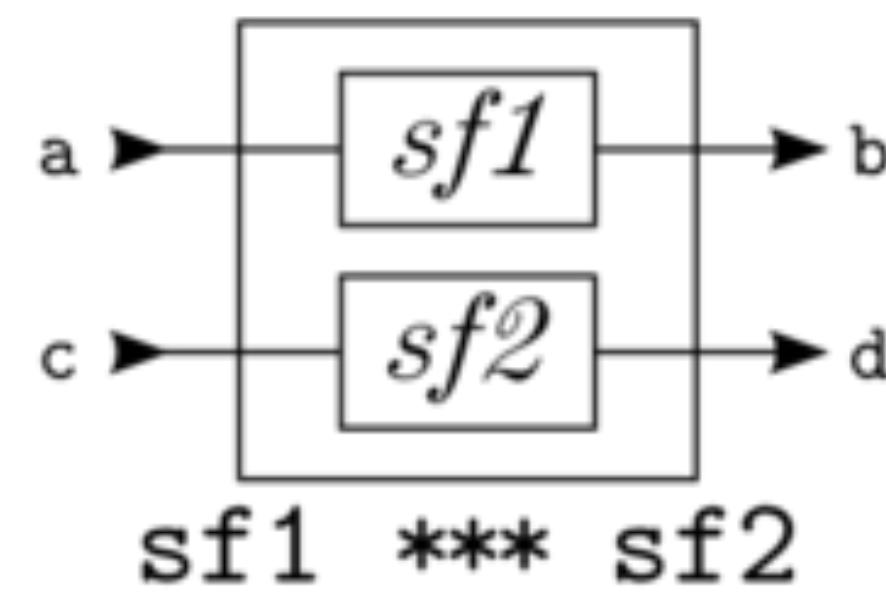
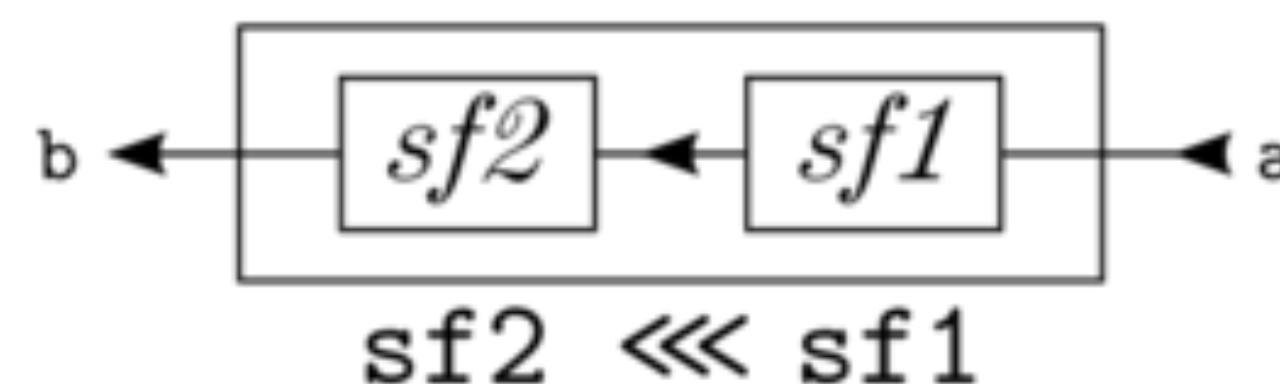
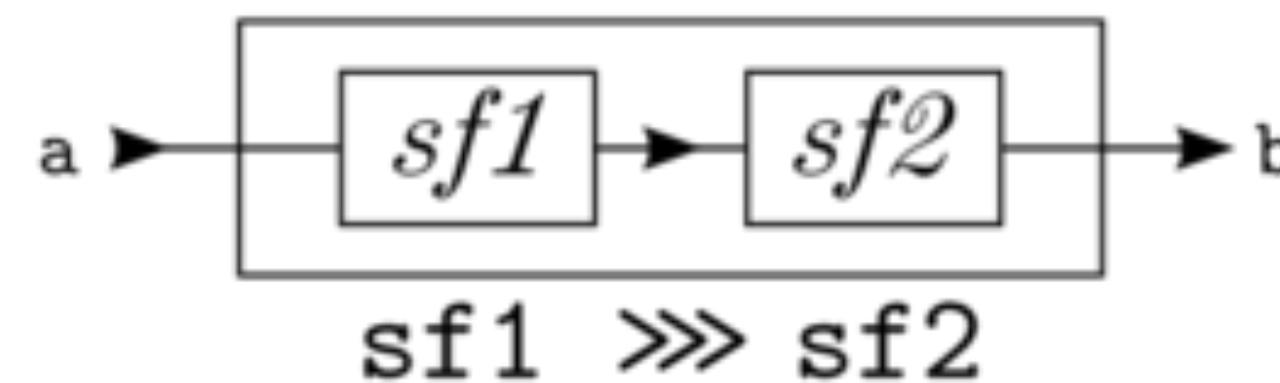
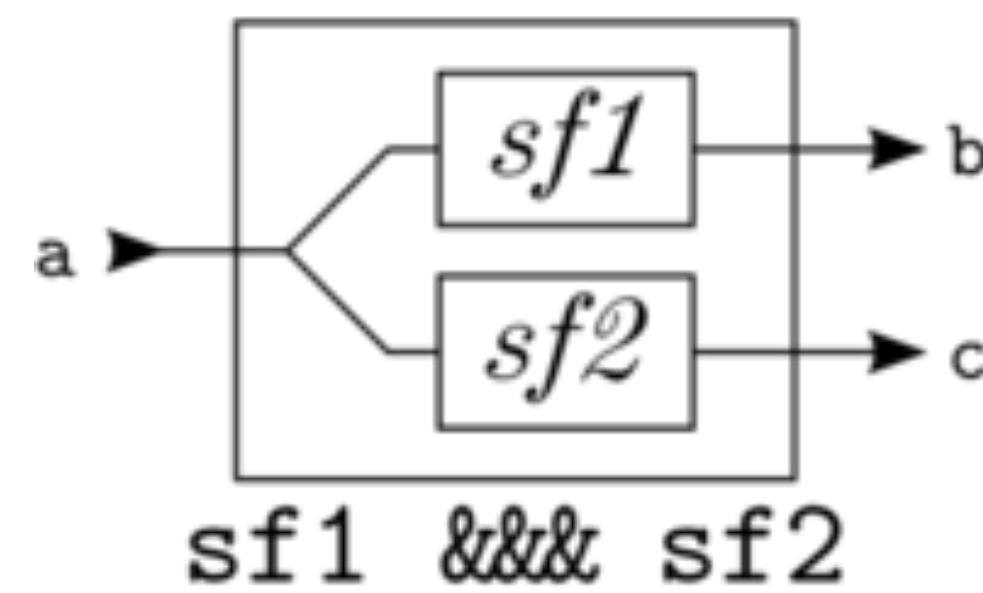
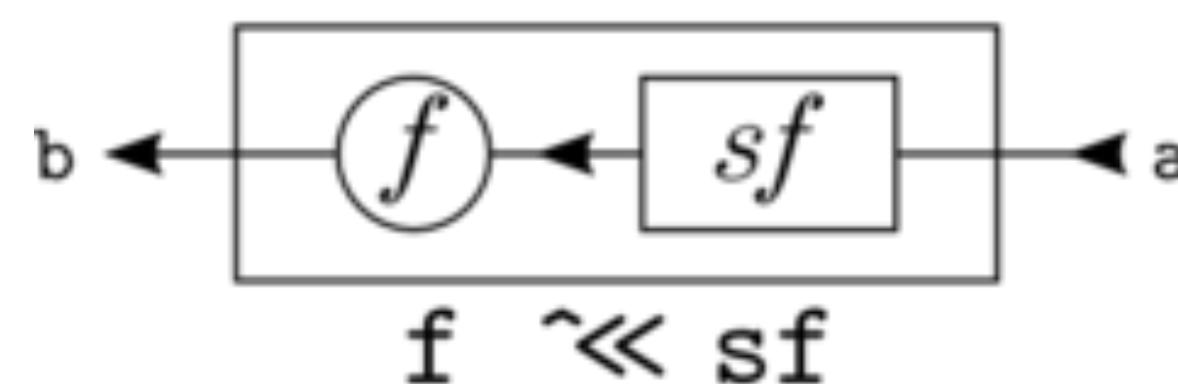
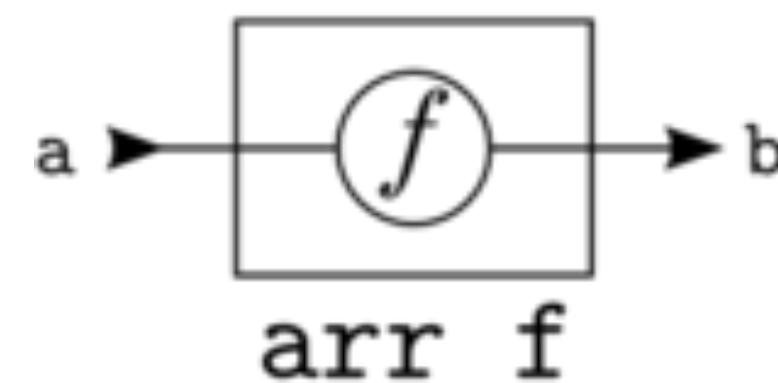
- Lifting :  $\text{arr} :: (\text{a} \rightarrow \text{b}) \rightarrow \text{SF a b}$
- Composition :  $(\ggg) :: \text{SF a b} \rightarrow \text{SF b c} \rightarrow \text{SF a c}$ 
  - Note:  $f \ggg g = (\ggg) f g$
- The following equations should hold.
  - $\text{arr } f \ggg \text{arr } g = \text{arr } (g . f)$ 
    - Note:  $(g . f) x = g (f x)$
  - $(f \ggg g) \ggg h = f \ggg (g \ggg h)$

# Arrow Operations (2)

- (`&&&`) ::  $SF\ a\ b \rightarrow SF\ a\ c \rightarrow SF\ a\ (b, c)$ 
  - $\text{arr } f \&\& g = \text{arr } (f \& g)$ 
    - where  $(f \& g) x = (f x, g x)$
- (`***`) ::  $SF\ a\ b \rightarrow SF\ c\ d \rightarrow SF\ (a, c)\ (b, d)$ 
  - $f *** g = (\text{arr } \text{fst} \ggg f) \&\& (\text{arr } \text{snd} \ggg g)$ 
    - where  $\text{fst } (x, y) = x, \text{snd } (x, y) = y$
- `first` ::  $SF\ a\ b \rightarrow SF\ (a, c)\ (b, c)$ 
  - $\text{first } f = f *** \text{arr } \text{id}$ 
    - where  $\text{id } x = x$
- `second` ::  $SF\ a\ b \rightarrow SF\ (c, a)\ (c, b)$ 
  - $\text{second } f = \text{arr } \text{id} *** f$

# Arrow Operations Visualized

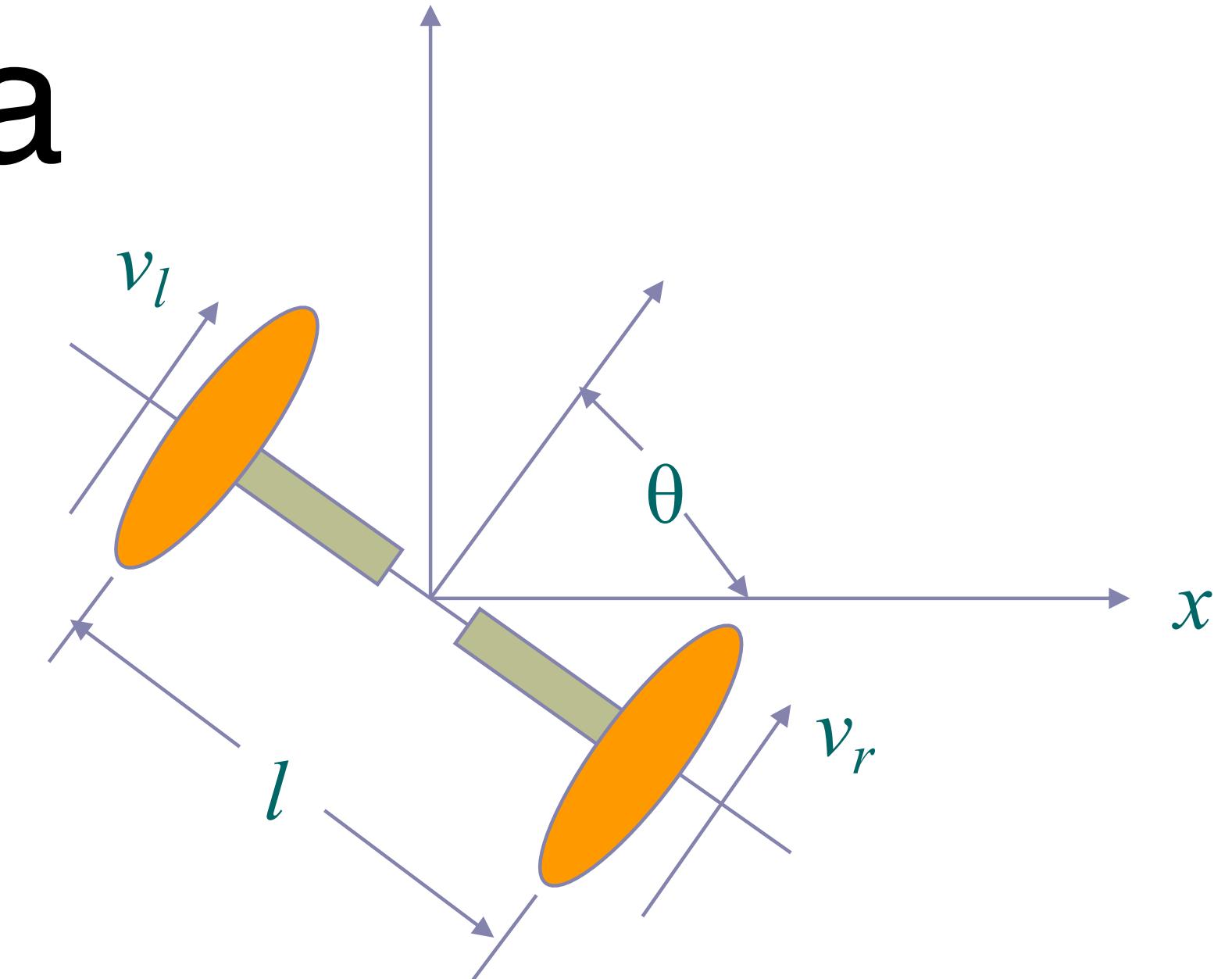
<https://wiki.haskell.org/Yampa>



# Ex. Two-Wheel Robot in Yampa

- Calculate the x position of the robot.

$$x(t) = \int_0^t \frac{vr(t) + vl(t)}{2} \cos \theta(t) dt$$



```
vrSF, vlSF, thetaSF :: SF Input Float  
  
x :: SF Input Float  
x = let v = (vrSF &&& vlSF) >>> arr2 (+)  
     t = thetaSF >>> arr cos  
   in (v &&& t) >>> arr2 (*) >>> integral >>> arr (/ 2)
```

<https://wiki.haskell.org/Yampa>

```
arr2 :: (a -> b -> c) -> SF (a, b) c  
arr2 = arr . uncurry  
uncurry :: (a -> b -> c) -> (a, b) -> c  
uncurry f (x, y) = f x y
```

# Using Arrow Syntax

```
vrSF, vlSF, thetaSF :: SF Input Float  
  
x :: SF Input Float  
x = proc input -> do  
    vr <- vrSF -< input  
    vl <- vlSF -< input  
    theta <- thetaSF -< input  
    i <- integral -< (vr + vl) * cos theta  
    returnA -< (i / 2)
```

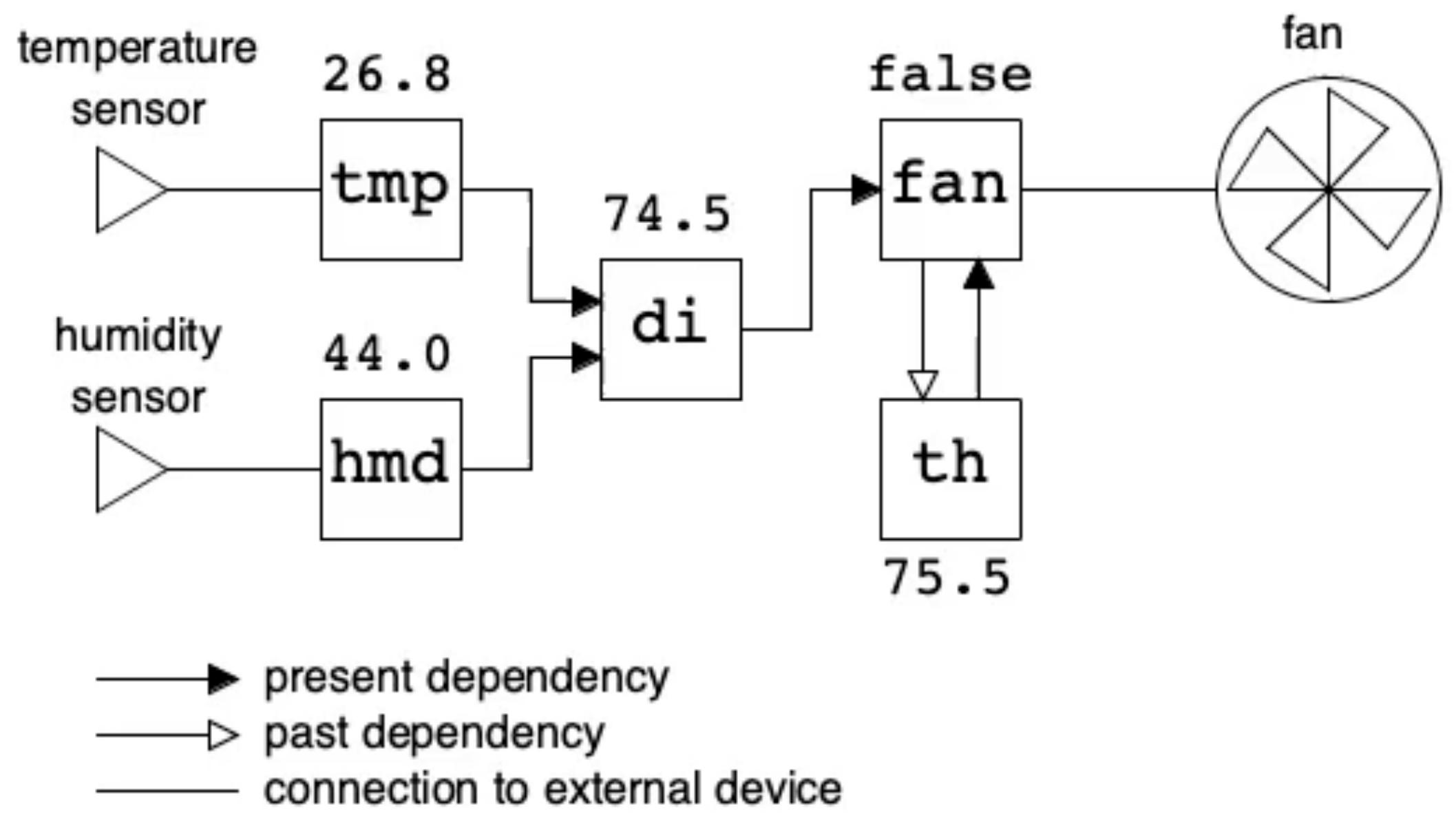
# Emfrp [Sawada et al, '16]

## A Pure FRP Language for Resource-Constrained Embedded Systems

- Statically-typed pure FRP language
  - Parametric polymorphism, Type inference, Algebraic types & Pattern matching
- Lifting-free, Glitch-free, Space/Time-leaks-free
- Statically determined runtime memory size
- Small memory footprint
  - Can run even in a 8bit microcontrollers such as ATmega32

# Emfrp Example

## FanController



A program can be seen as a directed graph of time-varying values and their dependencies. Without past dependencies, the graph forms a DAG.

```

module FanController % module name
in tmp : Float,      % temperature sensor
      hmd : Float       % humidity sensor
out fan : Bool       % fan switch

% discomfort (temperature-humidity) index
node di = 0.81 * tmp + 0.01 * hmd *
          (0.99 * tmp - 14.3) + 46.3

% fan status
node init[False] fan = di >= th

% threshold
node th = 75.0 +
           if fan@last then -0.5 else 0.5
    
```

- A fan controller with temperature and humidity sensors.
- Turns ON the fan while the current discomfort index is larger than or equal to the threshold.
- Does a simple hysteresis control to avoid frequent switching of the fan.

```

module Balancer
in gyroY : Int, # gyroscope (y-axis)
    accX : Int, # accelerometer (x-axis)
    encL : Int, # left motor encoder
    encR : Int # right motor encoder
out motorL : Int, # left motor
    motorR : Int # right motor
use Std
...
# definitions of constants

node init[0] angle =
    (angle@last + gyroY * update_time_ms) * 99 / 100

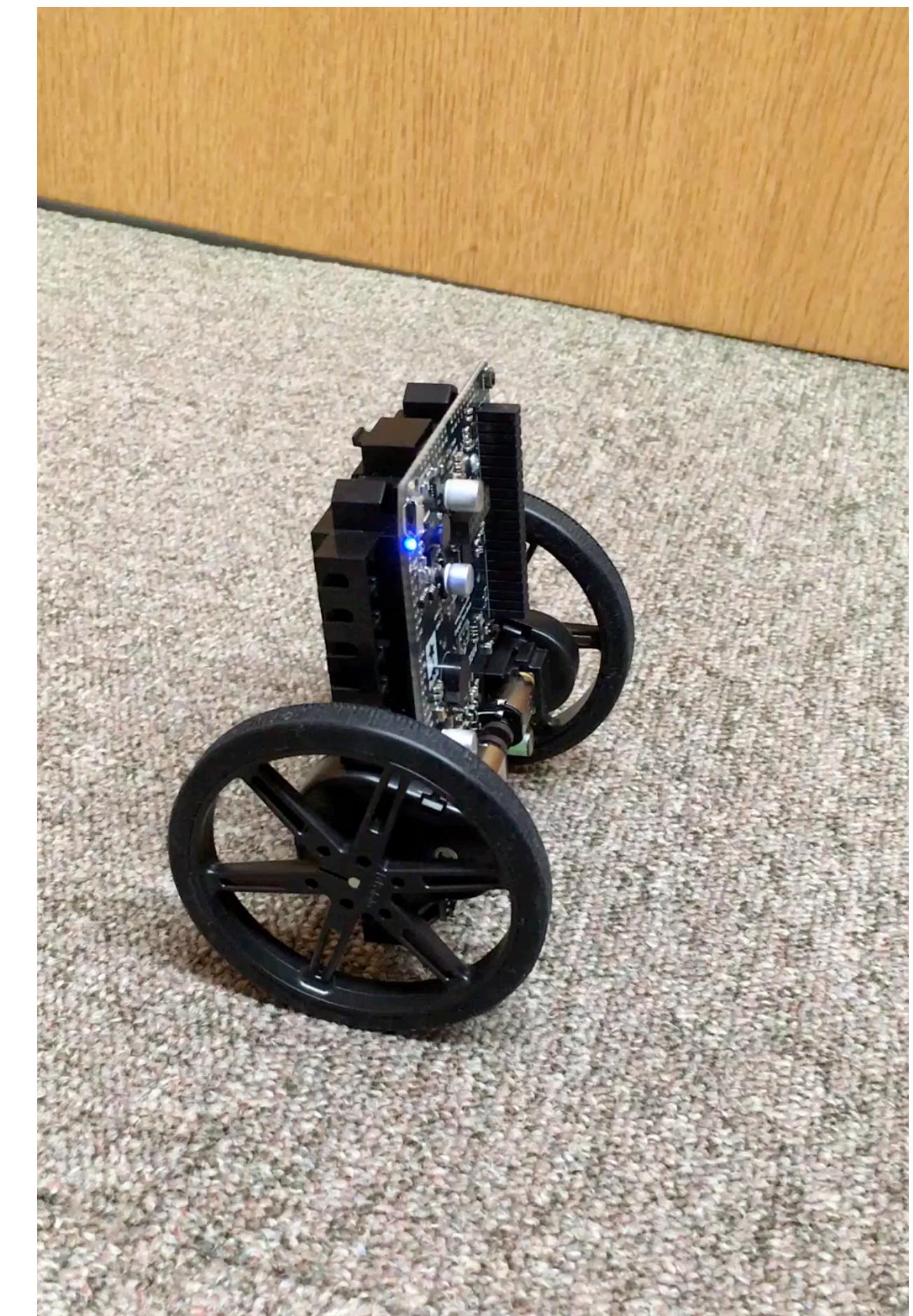
node speedL = encL - encL@last
node speedR = encR - encR@last
node init[0] distL = distL@last + speedL
node init[0] distR = distR@last + speedR

node risingAngleOff = gyroY * angle_rate_ratio + angle
node init[0] motor =
    motorSpeed(motor@last +
        (angle_resp * risingAngleOff +
        dist_resp * (distL + distR ) +
        speed_resp * (speedL + speedR)) / 100 / gear_ratio)

node diffSpeed = (distL - distR) * dist_diff_resp / 100
node motorL = if accX > 0 then motor + diffSpeed else 0
node motorR = if accX > 0 then motor - diffSpeed else 0

```

# Ex. Inverted Pendulum



Pololu Balboa 32U4  
(ATmega 32U4, 32KB Flash,  
2.5KB RAM)

# Summary

- Hybrid Systems
  - Hybrid Process
  - Zeno Behaviors
  - Stability
- Programming Reactive Systems
  - Functional Reactive Programming