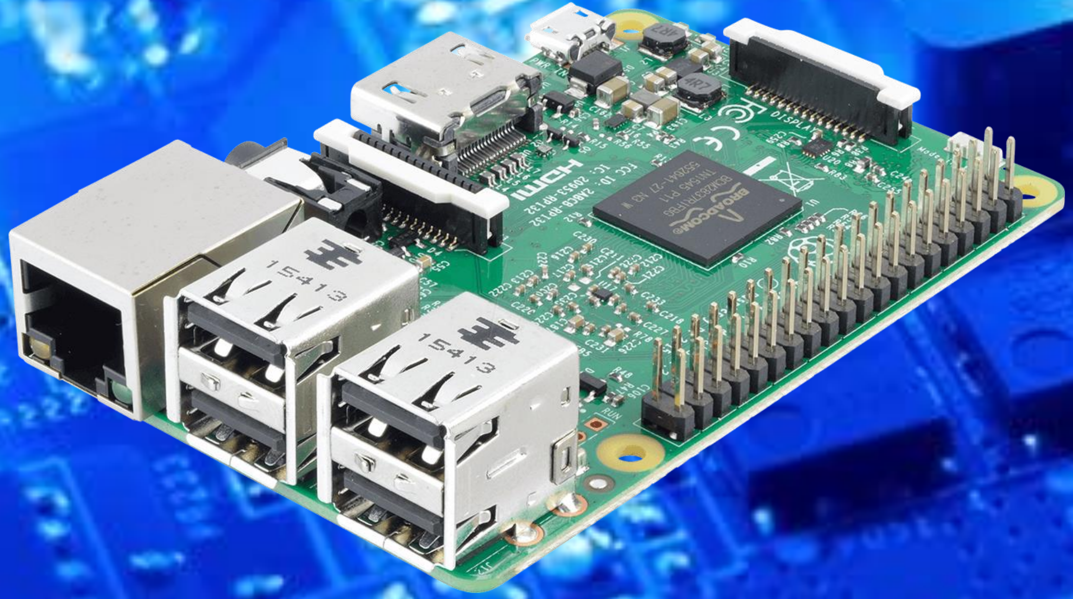
EEE3096S



# Specifications and Modeling (4)

Section 2.4 of textbook relates

L8

# Embedded Systems II


Dr Simon Winberg

Electrical Engineering
University of Cape Town

# Outline of Lecture

- Programming
  – State Machine → Code
- Modelling
  – StateCharts

We starting with moving out of specifications into design…
(but then a little revisit to requirements)

# State Machines

How to convert a state machine to code

# Example State Machine: Water Heater System

**Inputs**

temp : current water temperature

tunon : switch to turn system on

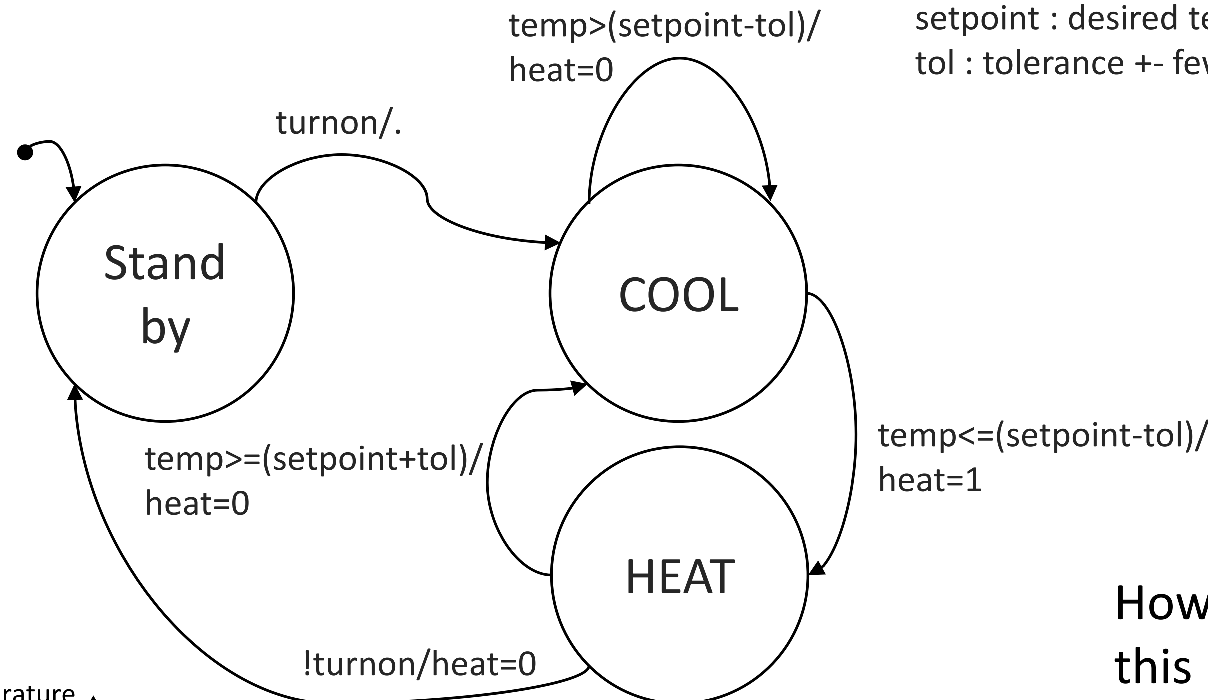
**Outputs**

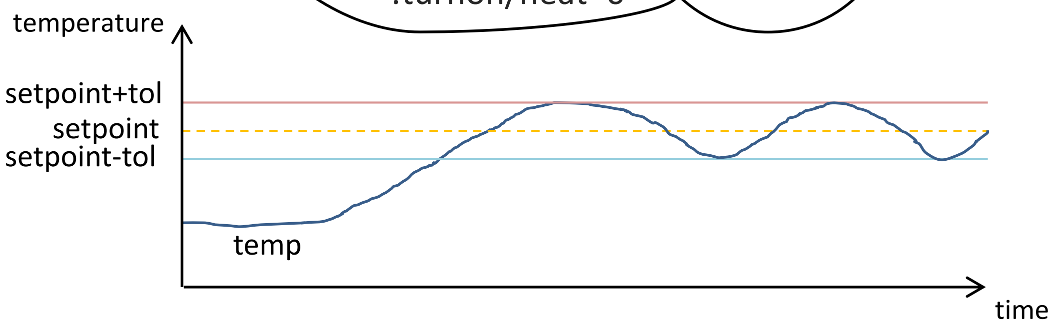
heat : turn heater on (1) or off (0)

**Parameters**

setpoint : desired temperature

tol : tolerance +- few degrees




How do we turn this into code?

setpoint = 60 degC

tol = 10 degC

```c
/*  EEE3096S Lecture 8
     Implementation of water heating state machine.
    Using Mealy machine model.    */

// Include Libraries
#include <stdio.h>
#include <stdlib.h>
#include "util/timer.h"
#include "sim_inputs.h"
// set the debug level
int debugon = 1;

// Define the states
enum STATES {
     STATE_Standby,
     STATE_Cool,
     STATE_Heat
     };
```

0. Might need various includes

1. Define your states

To code state machines or statecharts you could use a tool such as Yakindu (see [https://www.itemis.com/en/yakindu/state-machine/](https://www.itemis.com/en/yakindu/state-machine/)) or you could do it manually in the code directly, this is more common.

```
// Handle reading from input ports

// e.g. to simulate inputs or do real one

int input_temp ()
{
#ifdef __WIN32__
    // simulates reading a temperature
    static int temp_i = 0;
    if (temp_i>=sizeof(sim_temp)/sizeof(int)) temp_i =
0;
    return sim_temp[temp_i++];
#else
  return *((int*)0xFFFF0001)&0xFF; // read port
#endif
}
```

Implement the way inputs are received

You might do a version to test on a PC (e.g. __Win32__
and a version to test on the embedded side, the #else)

```
// Implement the state machine in a suitable function …
int main()
{
…
    int loops = 80;
```

If simulating think of using a limited num of loops

```
    // print column headings for printing debug info
    if (debugon) printf("turnon\ttemp\theat");
```

Maybe you want debug info to output

```
    // set up parameters
    int setpoint = 60; // degrees
    int tol      = 10;  // degrees
```

Set up parameters (good programming)

```
    /////// STATE MACHINES ////////////////////////

    // initialize the state machine
    int state = STATE_Standby;
```

Set starting state

```
    // set up shadown registers
```

… discussed in a moment …

```
    while (loops) {
```

… outline for the state machine …

```
        tic(); // get the current timer value
        // can print out the state and all the inputs/outputs
        if (debugon) printf("%d\t%d\t%d\n",turnon,temp,heat);
        // count down loops
        loops--;
        // wait for a little while (i.e. Sleep(100ms)
        while (toc()<0.10); // busy wait until timer is ready
    }
…
```

It is a synchronous design, transitions every 100ms

# Shadow registers

- These are essentially just copies of port registers
- Shadow input
  - You read (or poll) the input port at a certain rate and keep the latest input value in the shadow input register
- Shadow output
  - You assign the shadow register to the value you are planning to output, then write the output (possibly at specific intervals)
  - Often need a method to detect change of shadow output to avoid unneeded writes

Set up your shadow registers (if you are using them)

```
// set up shadown registers
int temp   = input_temp();
int turnon = input_turnon();
// make copy of prev value of shadow output registers
int old_heat = -1; // set as invalid to force write
int heat     = 0;  // assume heating element off at start
```

You might do a version to test on a PC (e.g. __Win32__
and a version to test on the embedded side, the #else)

Not the rest of the state machine

```
while (loops) {
tic(); // get the current timer value
// set shadow registers as copies of inputs
temp   = input_temp();
turnon = input_turnon();
// now check the state
switch (state) {
case STATE_Standby: if (turnon) state = STATE_Cool;
                                break; // marks end this state
case STATE_Cool   : if (temp >(setpoint-tol)) {
                        heat = 0; state = STATE_Cool; }
               else if (temp<=(setpoint-tol)) {
                        heat = 1; state = STATE_Heat; }
                    break;
case STATE_Heat   : if (temp>=(setpoint+tol)) {
                        heat = 0; state = STATE_Cool; }
               else if (!turnon) {
                        heat = 0; state = STATE_Standby; }
                     break;
        };
```

You need to update shadowed outputs

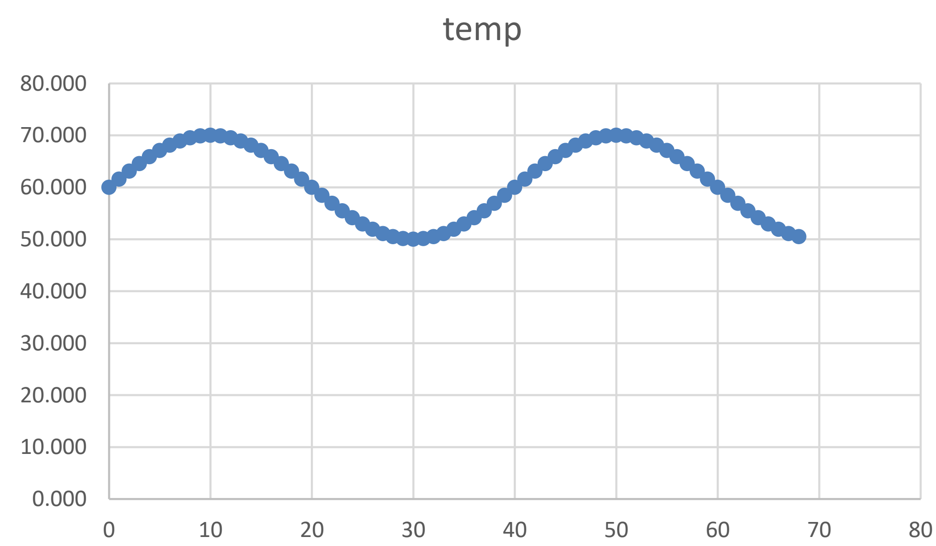
```
// update the outputs
if (old_heat != heat) output_heat(heat);
old_heat = heat;


…


} // end off the while loop
```

# Test Data

| in_temp |
| --- |
| 60.000, |
| 61.564 |
| 63.090, |
| 64.540, |
| 65.878, |
| 67.071, |
| 68.090, |
| 68.910, |
| 69.511, |
| 69.877, |
| 70.000, |
| 69.877, |
| 69.511, |
| 68.910, |
| … |




Stored in file: sim_inputs.h

# Quick Run of the Simulated Program

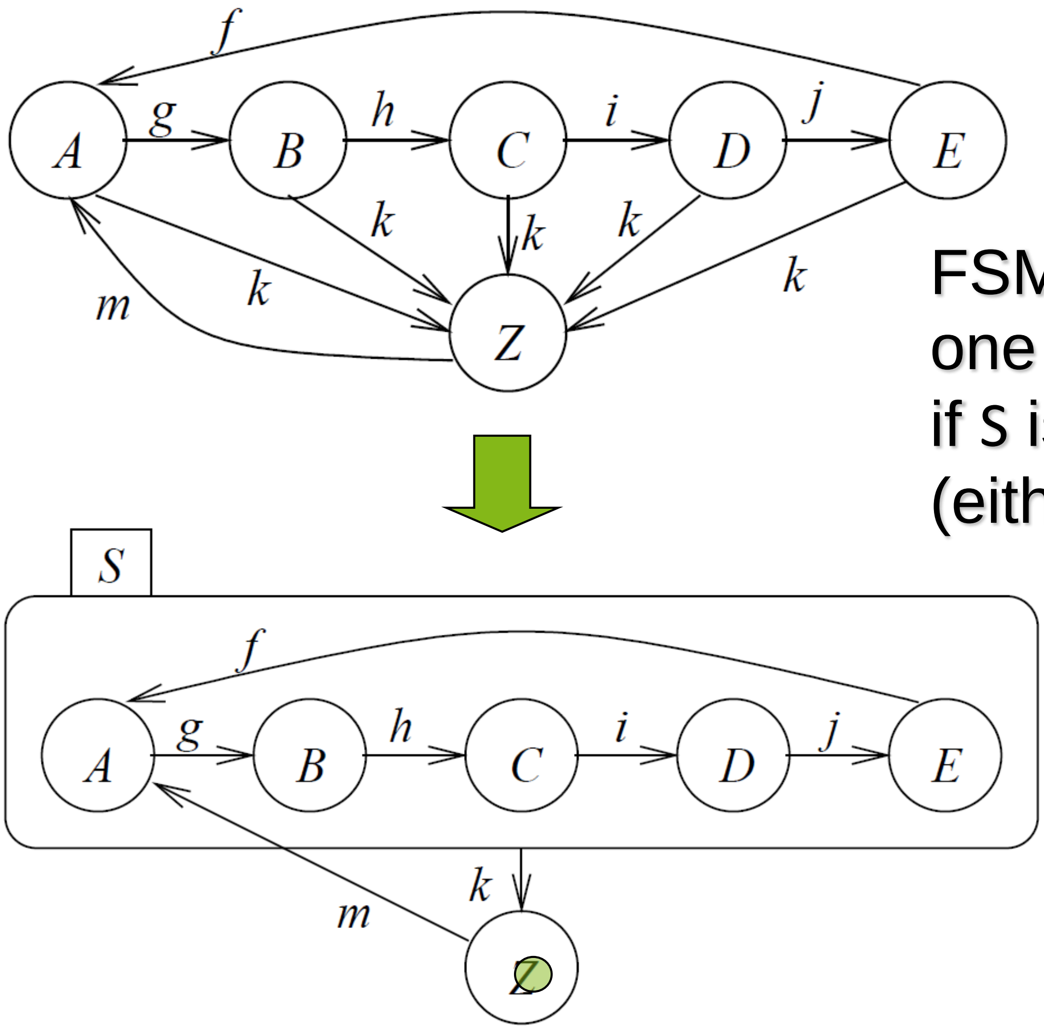
## Modeling

# UML State Charts

Essentially hierarchical state machines that support timing constraints

They chose the name “State Chart” because that was the only remaining combination of the words “flow”, “state”, “diagram” or “chart” that hadn’t already been taken.

# Introducing State Hierarchy



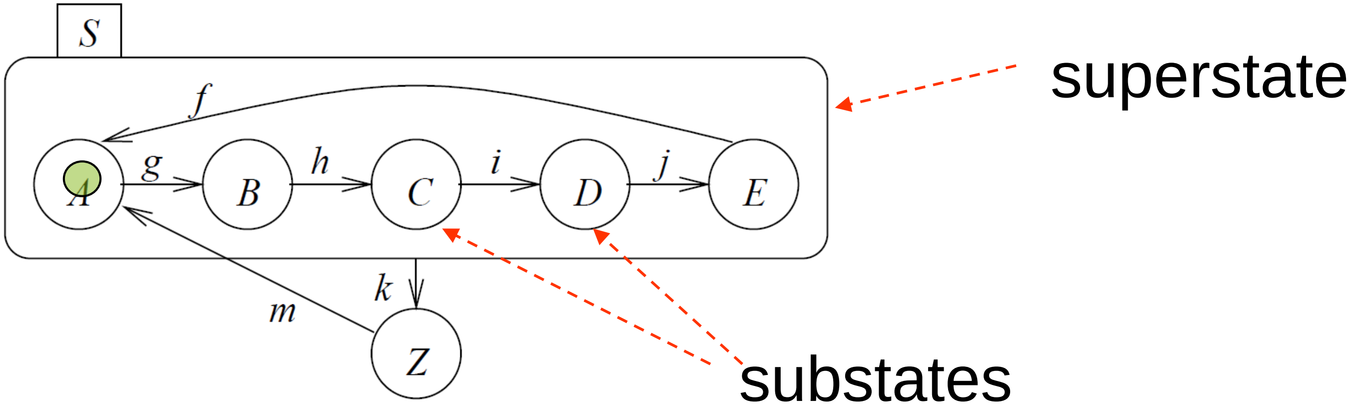

FSM will be **in** exactly one of the substates of S if S is **active**
(either in A or in B or ..)

Short animation to illustrate execution behaviour …

= indicates active state

# State Chart Terminology

- Current state of FSM is also called the **active** state.
- States which are not composed of other states are called **basic states.**
- States containing other states are called **super-states**.
- Super-states $S$ are called **OR-super-states**, if exactly one of the sub-states of $S$ is active whenever $S$ is active.



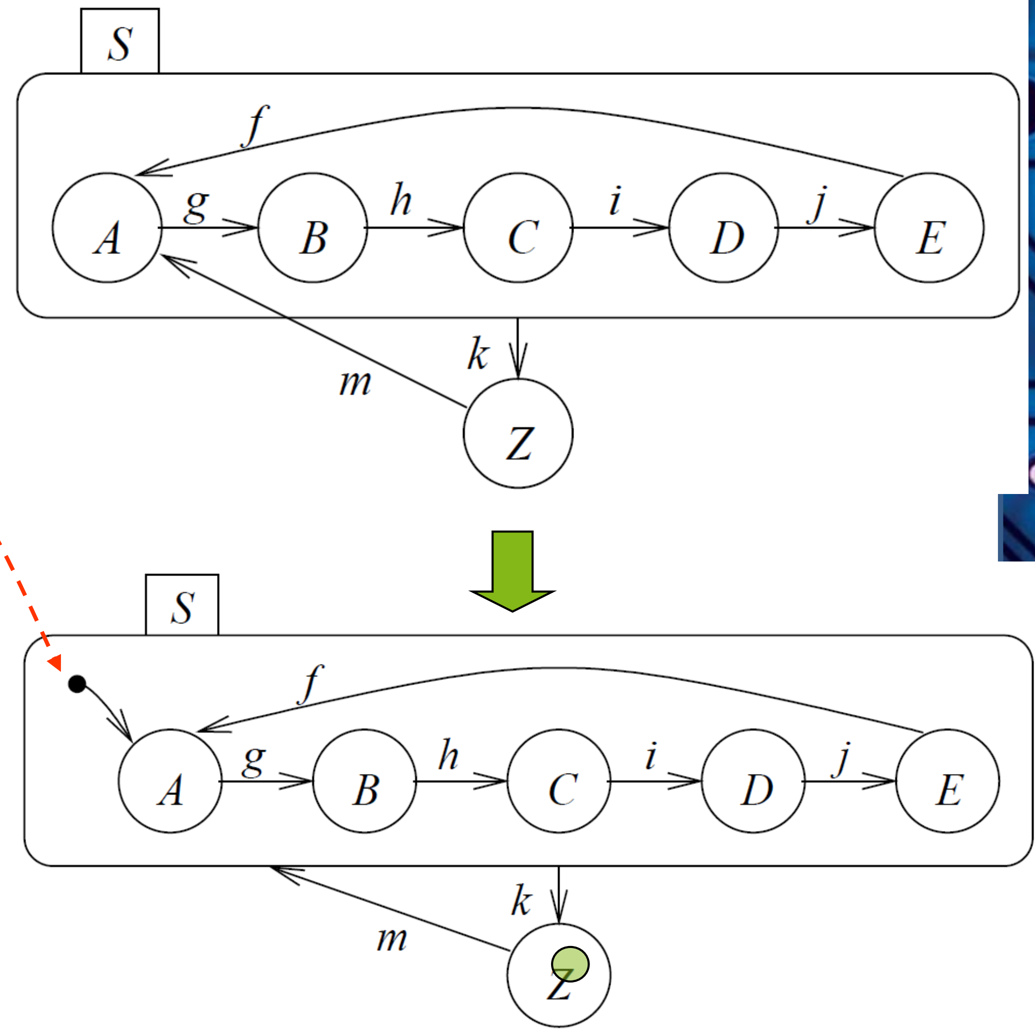
# Default state mechanism

Try to hide internal structure from outside world!

☞ Default/initial state

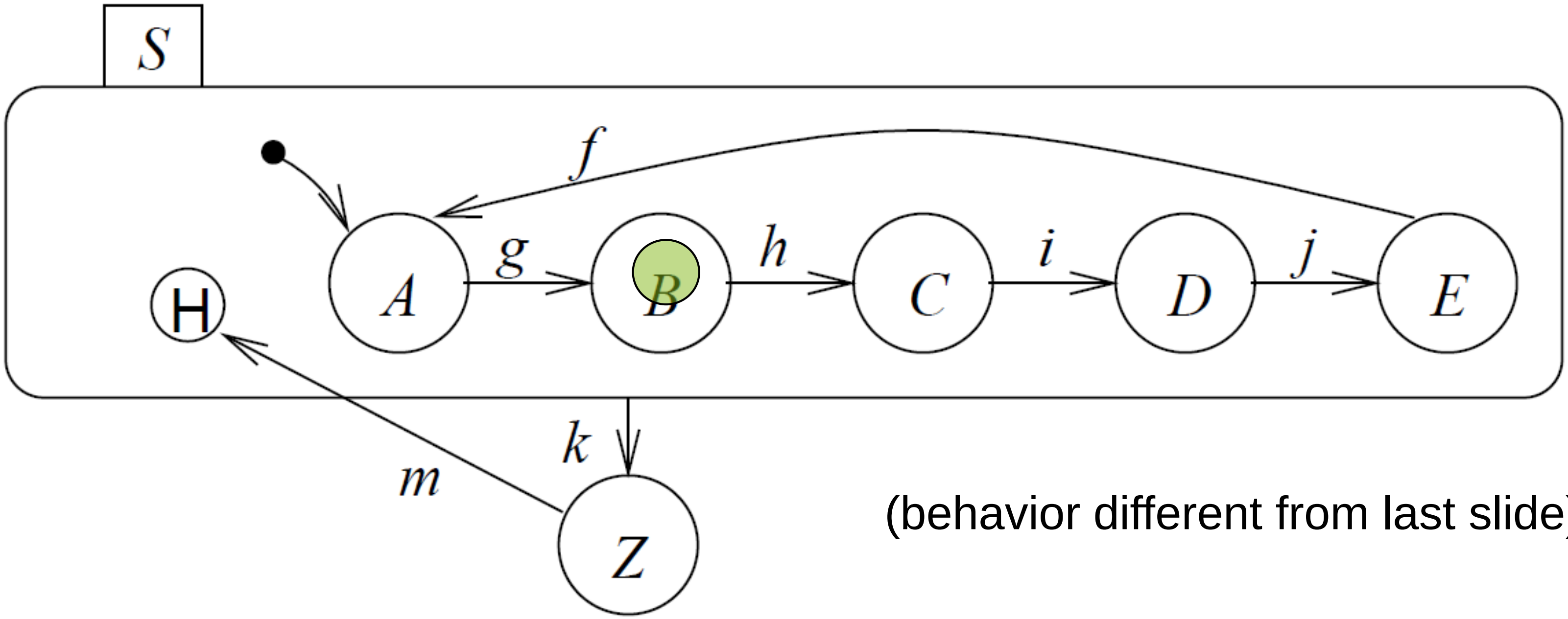
Filled circle indicates sub-state entered whenever super-state is started.

Not a state by itself!




Short animation to illustrate execution behaviour …

# State Chart History Mechanism

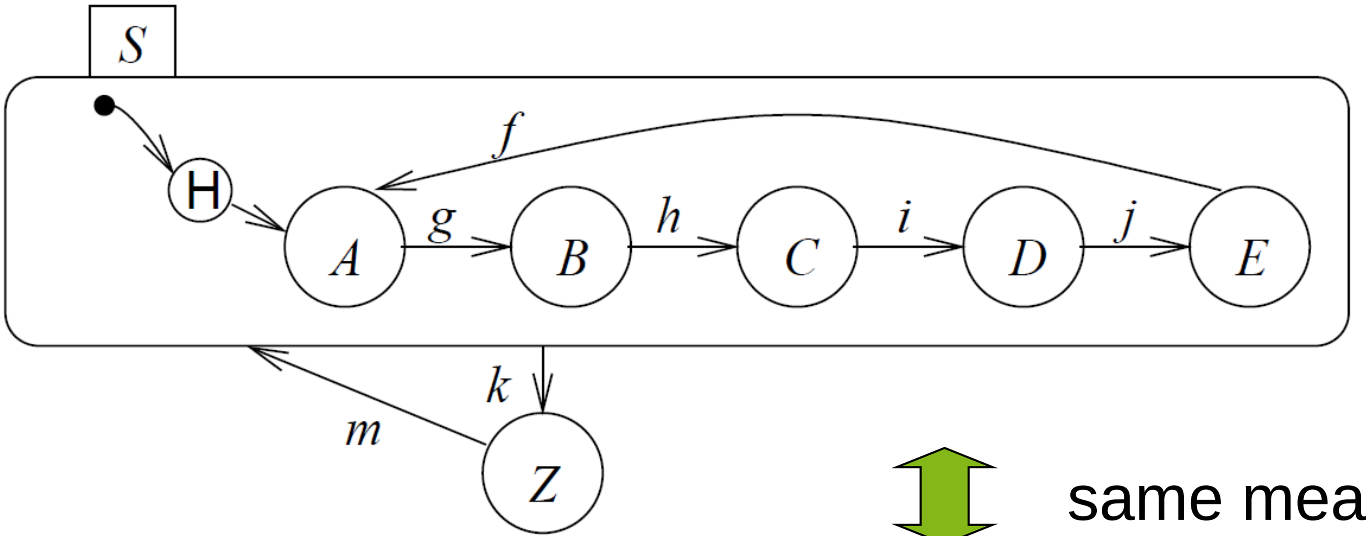



(behavior different from last slide)

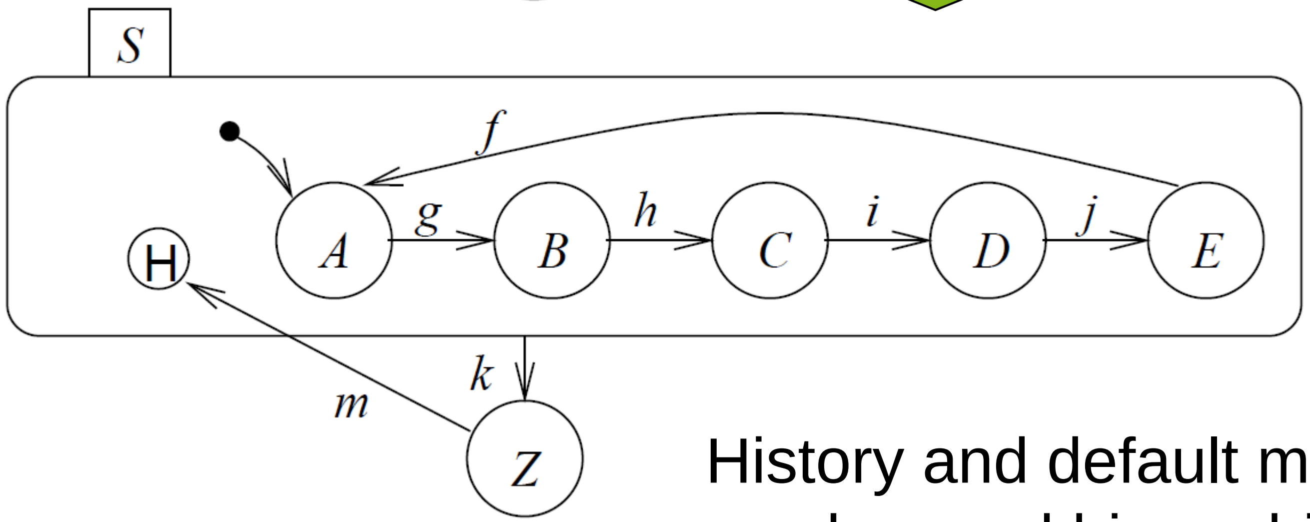
- For input $m$, $S$ enters the state it was in before $S$ was left (can be $A$, $B$, $C$, $D$, or $E$).
- If $S$ is entered for the first time, the default mechanism applies.
- Could also have other transition to $H^{-1}$ state, $H^{-2}$ state etc. for prior histories (not used often as it can be confusing to work with)

Short animation to illustrate execution behaviour …

# Combining history and default state mechanism




same meaning




History and default mechanisms can be used hierarchically.

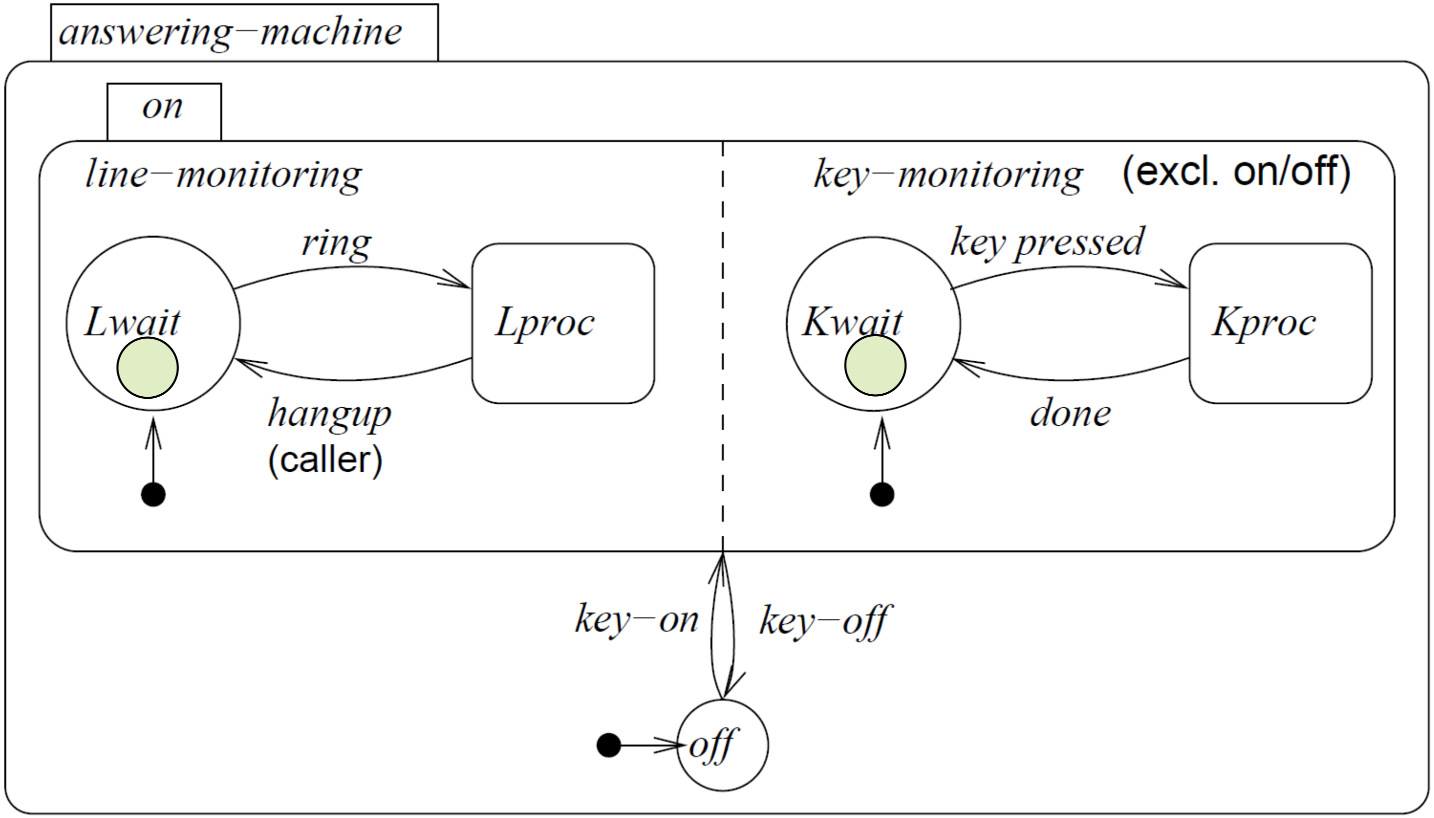
# State Chart Concurrency

- Convenient ways of describing concurrency requirement

**AND-super-states**: FSM is in **all** (immediate) sub-states of a super-state;

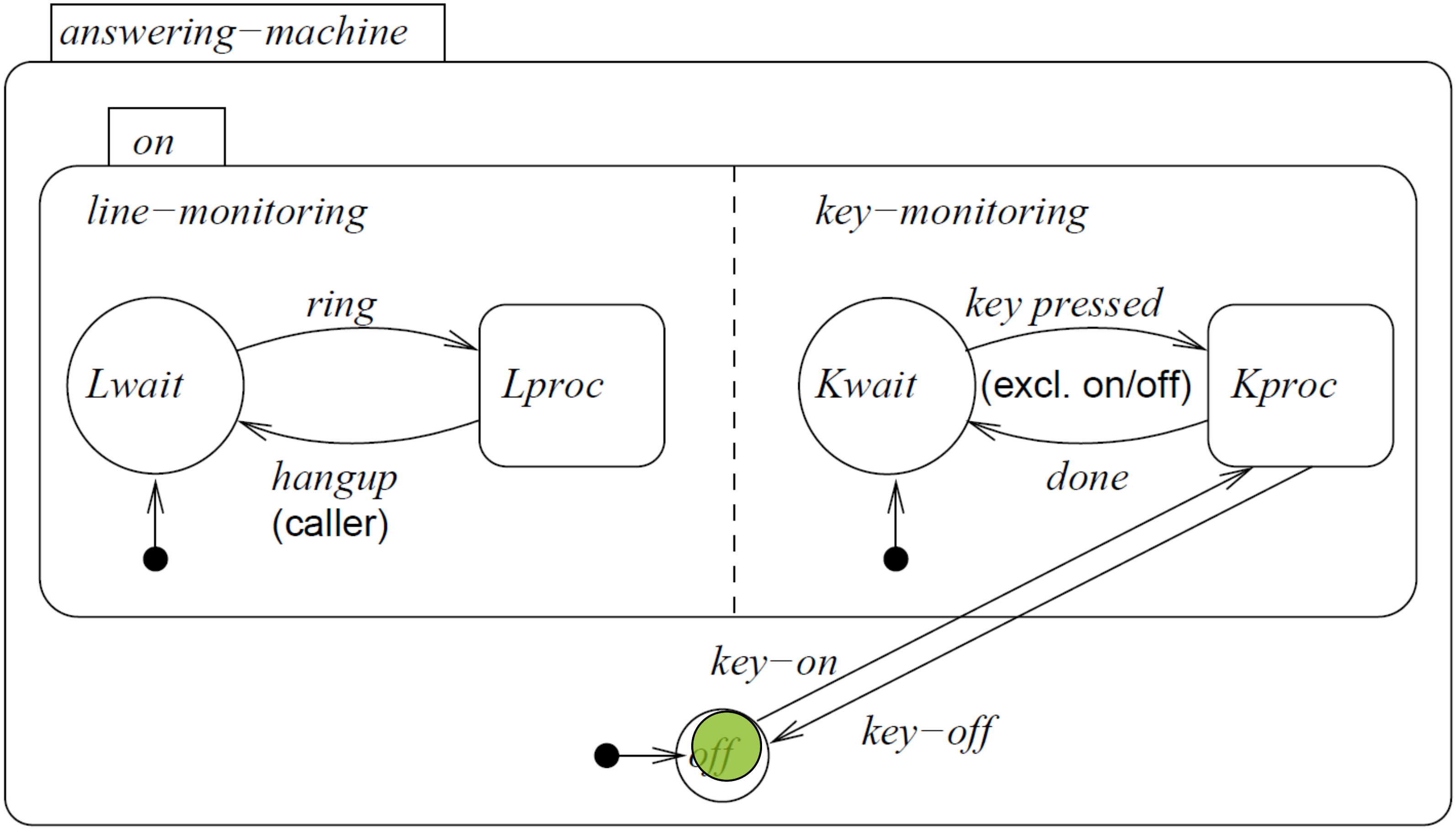
Example:

i.e. both left and right states are running concurrency




Short animation to illustrate execution behaviour …

# Entering and leaving AND-super-states




- Line-monitoring and key-monitoring are entered and left, when service switch is operated.

i.e. if you start the on super-state both states have to activate

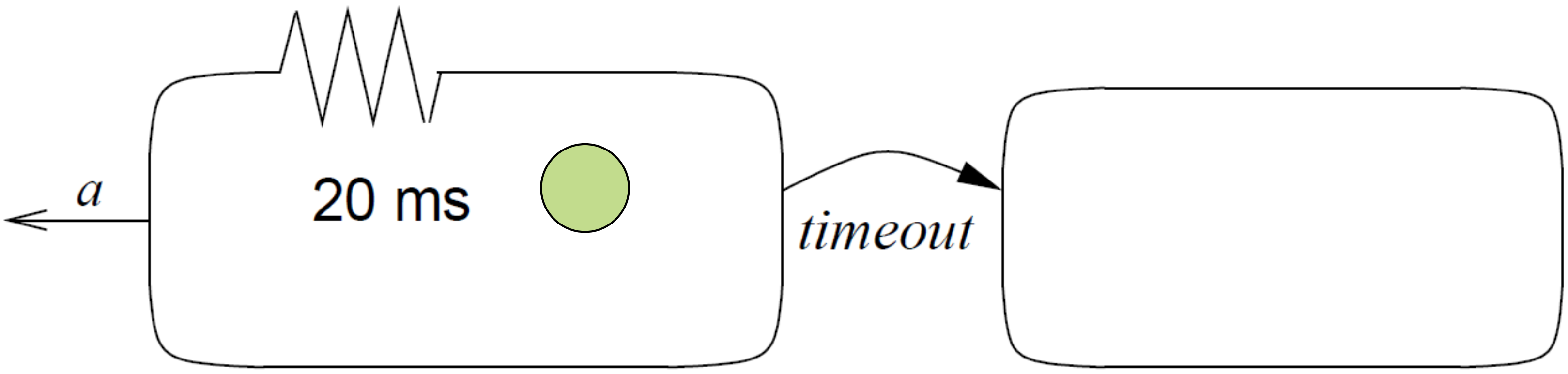
Short animation to illustrate execution behaviour …

# Types of states

- In StateCharts, states are either
  - **basic states, or**
  - **AND-super-states, or**
  - **OR-super-states.**

# Timers

- Since time needs to be modeled in embedded & cyber-physical systems, timers need to be modeled.
- In StateCharts, special edges can be used for timeouts.



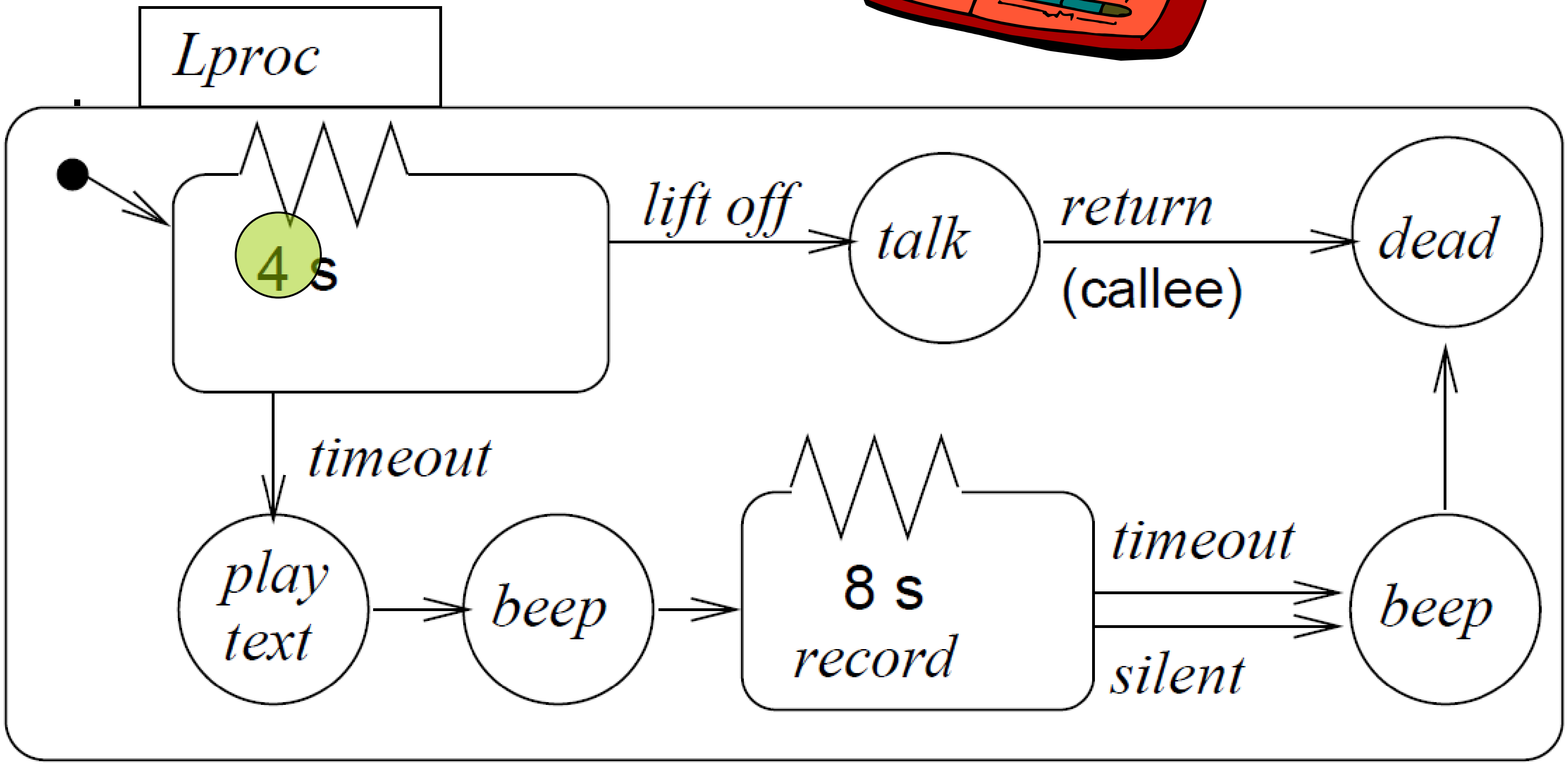

If event a does not happen while the system is in the left state for 20 ms, a timeout will take place.

The state with /\/\/\ symbol indicates a timed state.
This help to make these states easier to spot in a design.

Short animation to illustrate execution behaviour …

# Using timers in an answering machine

A short animation to emphasise operation…




(6) Short animation to illustrate execution behaviour …

# General form of state chart edge labels

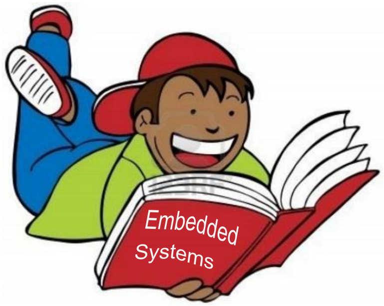

- **Events:**
  - Exist only until the next evaluation of the model
  - Can be either internally or externally generated
- **Conditions:**
  - Refer to values of variables that keep their value until **they are reassigned**
- **Reactions:**
  - Can either be assignments for variables
  - or creation of events
- **Example:**
  - *service-off* [not in *Lproc*] / *service*:=0

It's essentially a Mealy machine with some extras

# The Next Episode…

## Lecture L09

L09: FDM timing constraints, Dataflow, ICD




**Reminder:** Read section 2.5