



**Sharif University of Technology**  
**Department of Computer Science and Engineering**

**Lec. 3:**  
**Specification and Modeling**



M. Ansari

Spring 2025

According to Peter Marwedel's Lectures

# Motivation for considering specs & models

---

- ❖ Why considering specs and models in detail?
- ❖ If something is wrong with the specs, then it will be difficult to get the design right, potentially wasting a lot of time.
- ❖ Typically, we work with **models** of the **system under design** (SUD)
- ❖ What is a *model* anyway?



# Models

---

**Definition:** *A model is a simplification of another entity, which can be a physical thing or another model. The model contains exactly those characteristics and properties of the modeled entity that are relevant for a given task. A model is minimal with respect to a task if it does not contain any other characteristics than those relevant for the task.*

[Jantsch, 2004]:

Which requirements do we have for our models?

# System Specification

---

- ❖ The first and the most important step in the design flow.
  - Requires human intelligence
  - Can we use natural language?
  - It is necessary to check specifications for
    - Completeness
    - Absence of contradictions
  - It should be possible to derive implementations from the specification in a systematic way.

# Required Features

---

## ❖ **Specification languages for ES should have the following features:**

- Hierarchy
  - Behavioral Hierarchies
    - ✓ e.g., Super-states
  - Structural Hierarchies
    - ✓ Like what is supported by VHDL and Verilog
- Timing Behavior
  - Delay
  - Cause and effect relationship

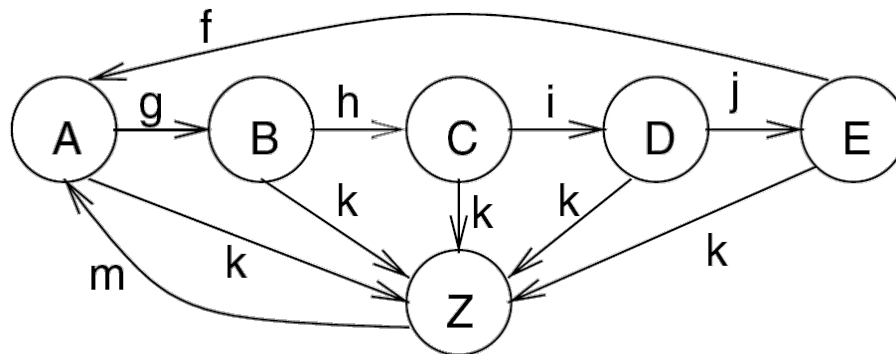
# Required Features (Cont.)

---

- ❖ State-oriented behavior
  - Automata provide a good mechanism for modeling reactive systems.
- ❖ Event handling
  - The reactive nature of ES
  - Mechanisms for describing events must exist
    - External events (Caused by environment)
    - Internal events (Caused by components of the system)
- ❖ Support for efficient implementation
  - e.g. Hardware/Software Co-design

# Required Features (Cont.)

- ❖ Support for dependable system design
  - Unambiguous semantics
  - Facilitate formal verification
- ❖ Exception-oriented behavior
  - It is not acceptable that exceptions have to be indicated for each and every state



State diagram with exception k

# Required Features (Cont.)

---

## ❖ Concurrency

- Real-life systems are concurrent systems.
- It is necessary to be able to specify concurrency **conveniently**.

## ❖ Synchronization and communication

- Concurrent actions have to be able to **communicate** and it must be possible to **agree** on the use of **resources** (e.g., mutual exclusion).



# Required Features (Cont.)

---

- ❖ Presence of programming elements
  - Usual programming languages have proven to be a **convenient** means of **expressing computations**.
  - Classical **hardware description** techniques (e.g., state diagrams) do not meet this requirement.
- ❖ Executability
  - Simulation (Design Verification)

# Required Features (Cont.)

---

- ❖ Readability and flexibility
  - Readable by human
  - Small changes of the system → Small changes of the specification
- ❖ Support for non-standard I/O-devices
  - to describe inputs and outputs for non-standard I/O-devices conveniently.

# Required Features (Cont.)

---

- ❖ Non-functional properties
  - Reliability
  - Size
  - Power consumption
- ❖ Appropriate model of computation
  - Von Neumann paradigm is not suitable

**There is no hope to develop a formal language capable of meeting all these requirements.**

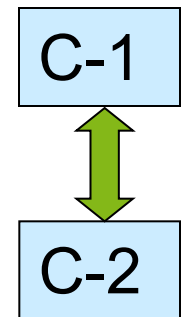
# Models of computation

---

## What does it mean, “to compute”?

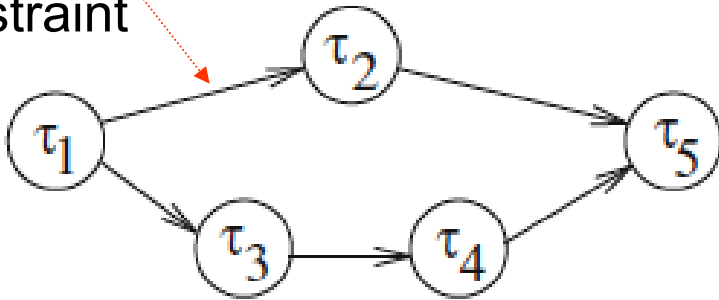
### Models of computation define:

- Components and an execution model for computations for each component
- Communication model for exchange of information between components.



# Dependence graph: Definition

Sequence  
constraint



Nodes could be programs  
or simple operations

**Def.:** A **dependence graph** is a directed graph  $G=(\tau,E)$  in which  $E \subseteq \tau \times \tau$  is a relation.

If  $(\tau_1, \tau_2) \in E$ , then  $\tau_1$  is called an **immediate predecessor** of  $\tau_2$  and  $\tau_2$  is called an **immediate successor** of  $\tau_1$ .

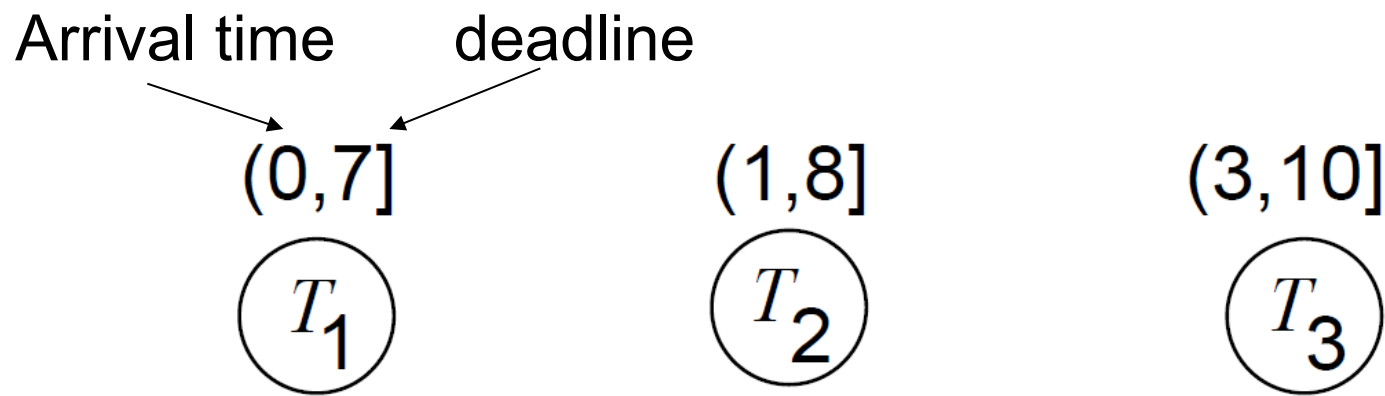
Suppose  $E^*$  is the transitive closure of  $E$ .

If  $(\tau_1, \tau_2) \in E^*$ , then  $\tau_1$  is called a **predecessor** of  $\tau_2$  and  $\tau_2$  is called a **successor** of  $\tau_1$ .

# Dependence graph: Timing information

---

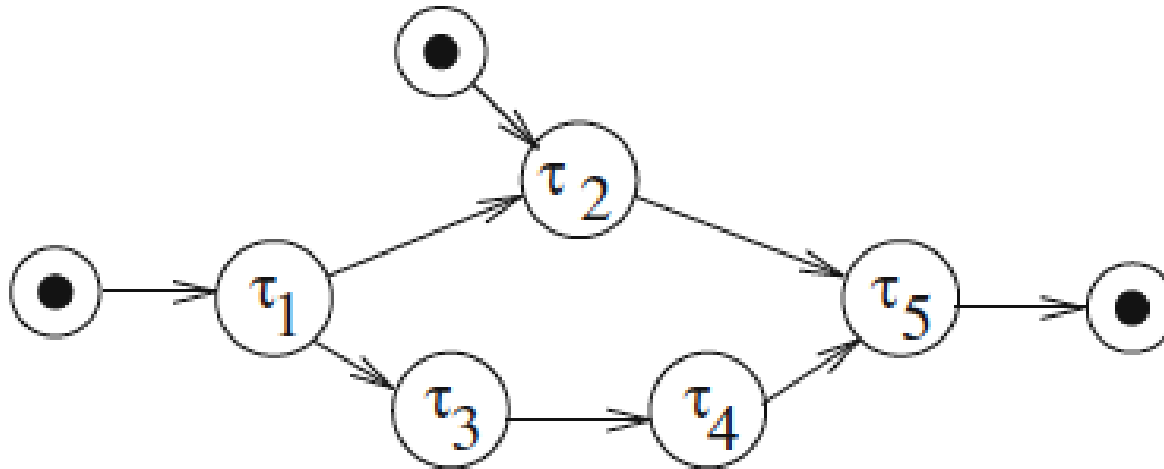
- ❖ Dependence graphs may contain additional information, for example:
  - Timing information



# Dependence graph: I/O-information

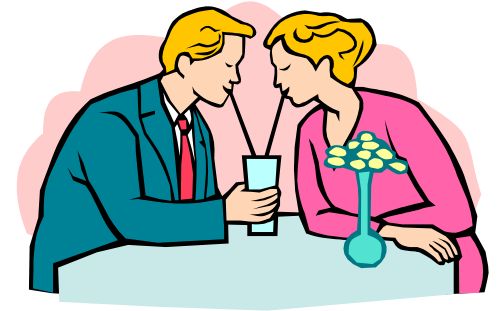
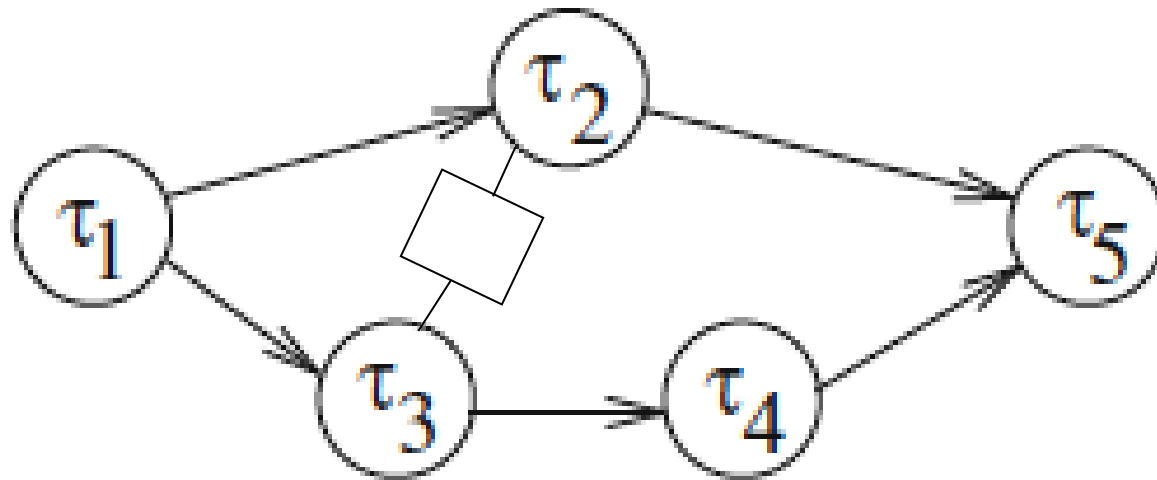
---

❖ Graph including I/O nodes and edges



# Dependence graph: Shared resources

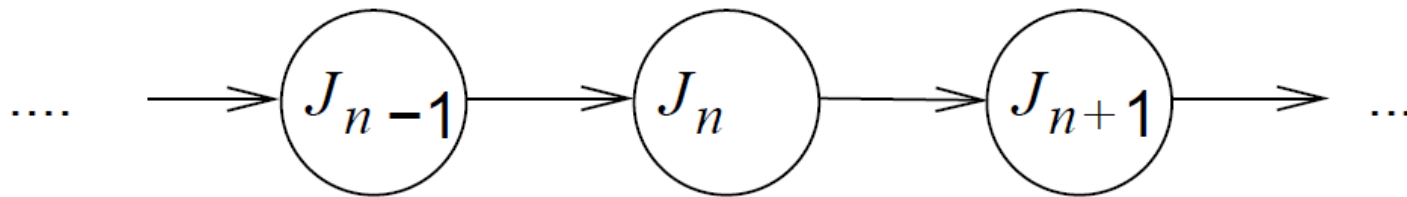
---





# Dependence graph: Periodic schedules

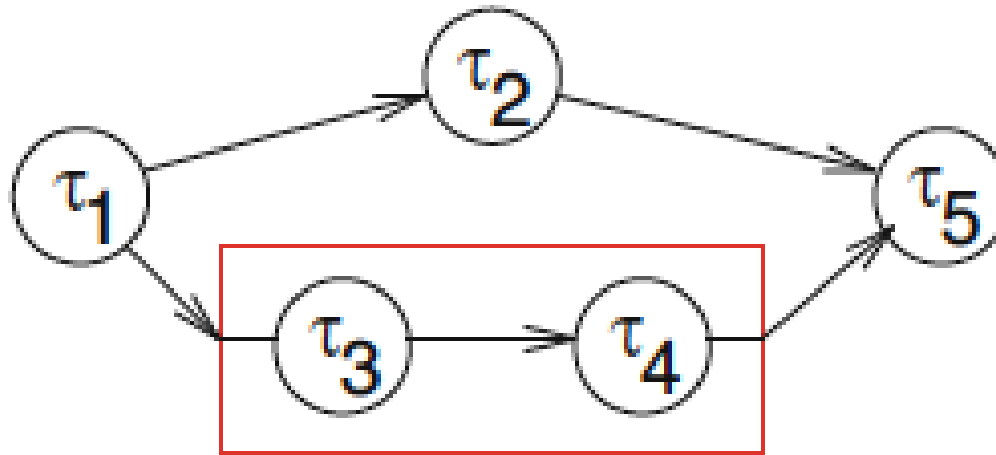
---



- A **job** is single execution of the dependence graph
- Periodic dependence graphs are infinite

# Dependence graph: Hierarchical task graphs

---



# Models of Communication: 1) Shared memory

---

## ❖ Shared memory



Variables accessible to several components/tasks.

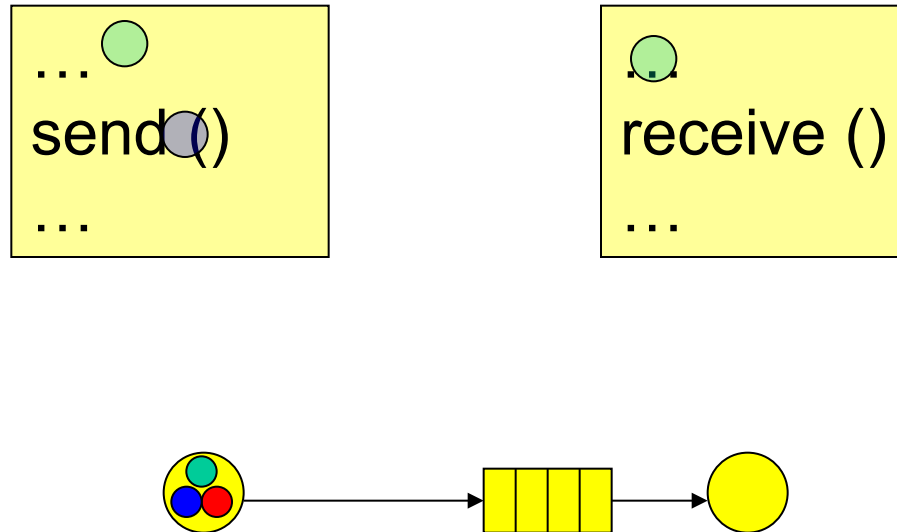
Model mostly restricted to local systems.

# Models of Communication: 2) message passing

## a) Non-blocking/asynchronous message passing

---

Sender does not have to wait until message has arrived;



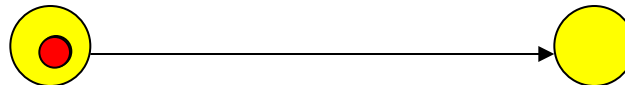
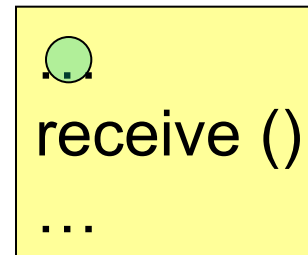
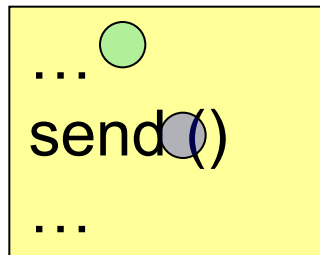
Potential problem: buffer overflow

# Models of Communication: 2) message passing

## b) Blocking/synchronous message passing

---

Sender will wait until receiver has received message



No buffer overflow, but reduced performance.

# Models of Communication: 2) message passing

## c) Extended rendezvous, remote invocation

---

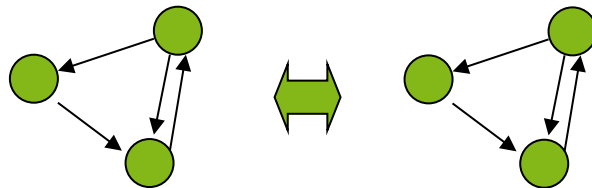
- ❖ The sender is allowed to continue only after an acknowledgment has been received from the recipient.
- ❖ The recipient does not have to send this acknowledgment immediately after receiving the message but can do some preliminary checking before actually sending the acknowledgment.

# Organization of computations within the components (1)

---

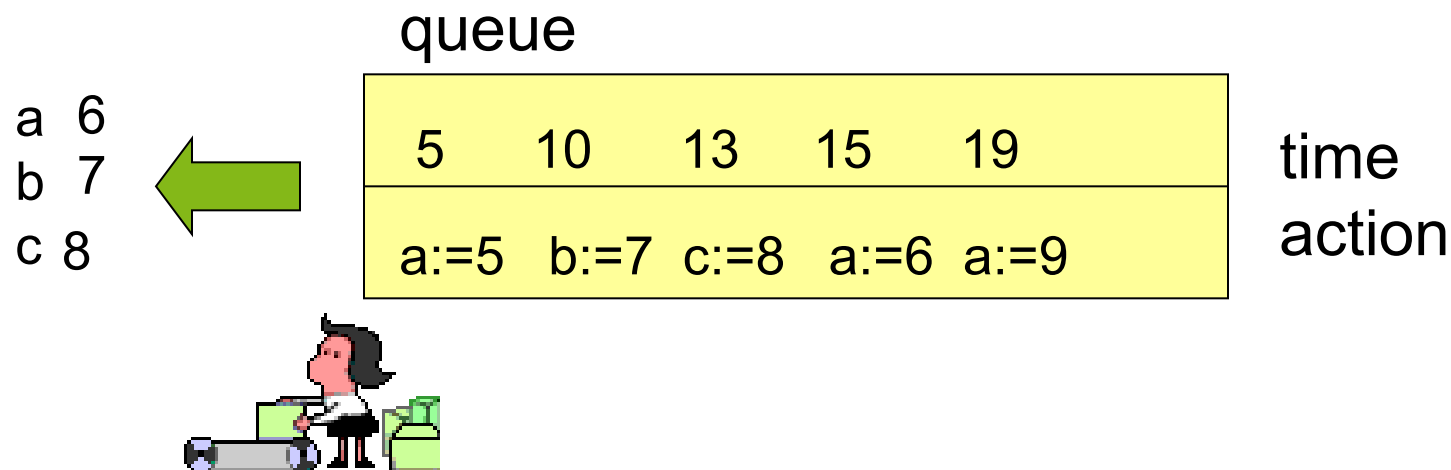
## ❖ Finite State Machines (FSMs)

- This model is based on the notion of a finite set of states, inputs, outputs, and transitions between states.
- Several of these machines may need to communicate, forming so-called **communicating finite state machines (CFSMs)**.



# Organization of computations within the components (2)

## ❖ Discrete Event Model



## ❖ Von Neumann Model

- Sequential execution, program memory etc.



# Organization of computations within the components (3)

---

## ❖ Differential Equations

- Differential equations are capable of modeling analog circuits and physical systems.
- They can find applications in cyber-physical system modeling.

$$\frac{\partial^2 x}{\partial t^2} = b$$



## ❖ Data Flow

- The availability of data triggers the possible execution of operations.

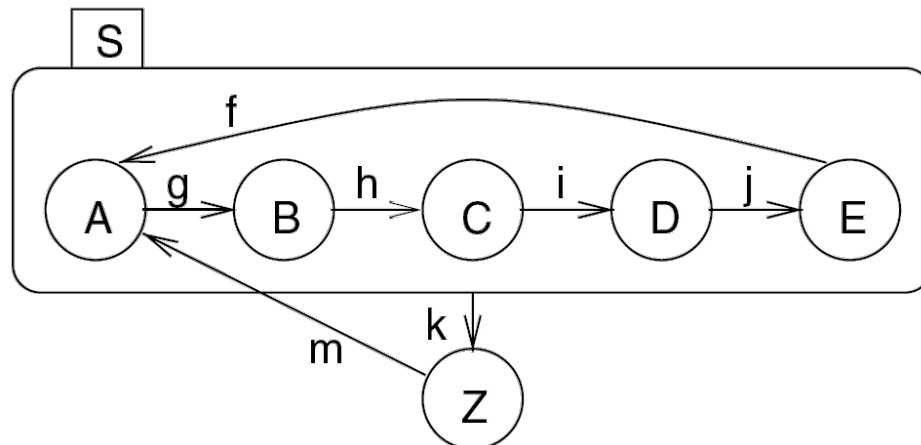
# An Overview of StateCharts

---

- ❖ StateCharts is a **language**.
- ❖ CFSM MoC
- ❖ Communication
  - Shared Memory
- ❖ Deterministic FSM

# Modeling of Hierarchy

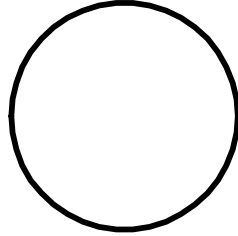
- ❖ The key extension is hierarchy.
  - States comprising other states are called **super-states**.
  - States included in super-states are called **sub-states** of the super-states.
  - Each state which is not composed of other states is called a **basic state**.



# Super Nodes vs. Basic States

---

❖ Basic states

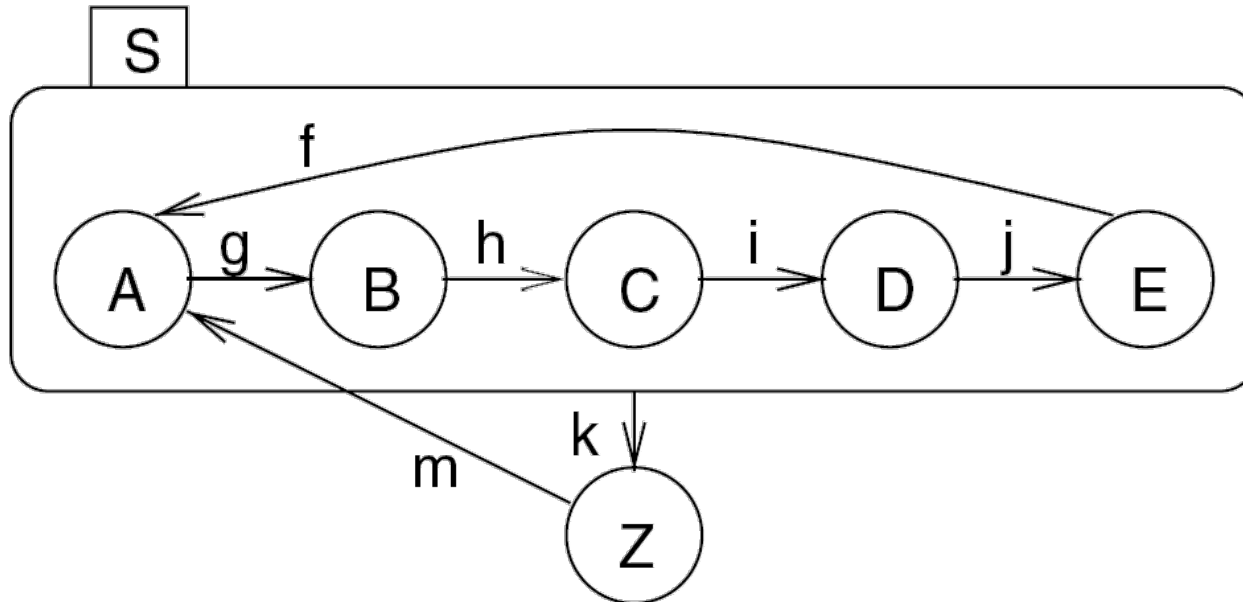


❖ Super nodes



# OR-Super-States

- ❖ The FSM can only be in one of the sub-states of super-state S at any time.



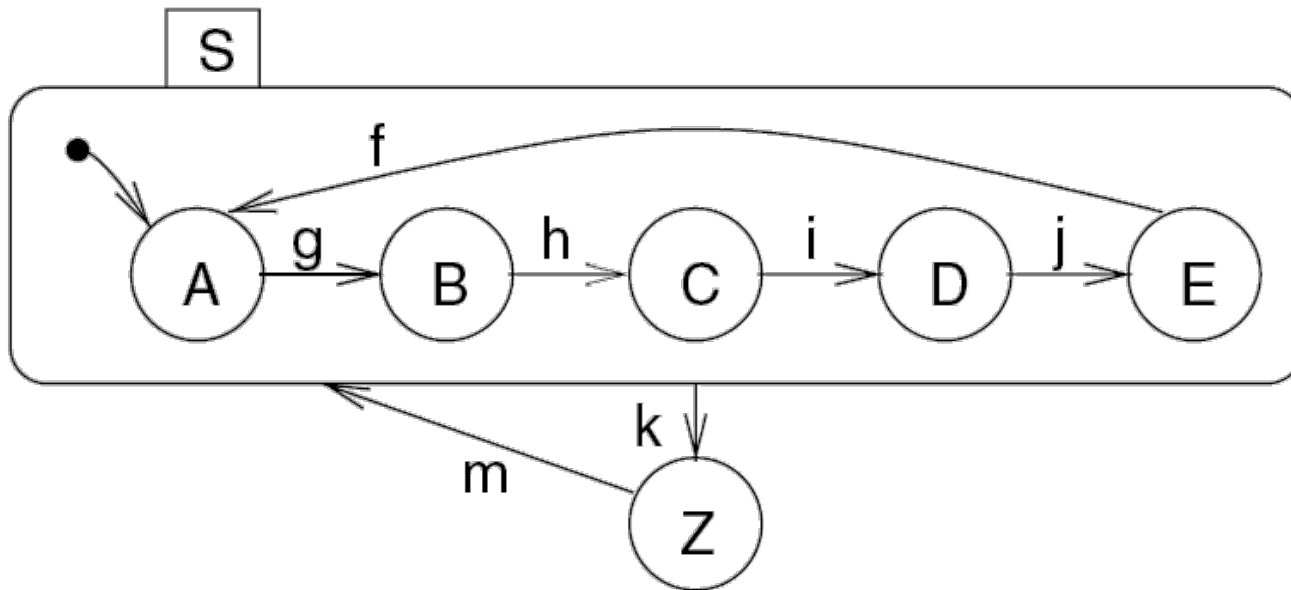
# Design Modularity

---

- ❖ There are two mechanisms to **hide** the internal structure of super-states from the environment.
  - Default State Mechanism
  - History Mechanism

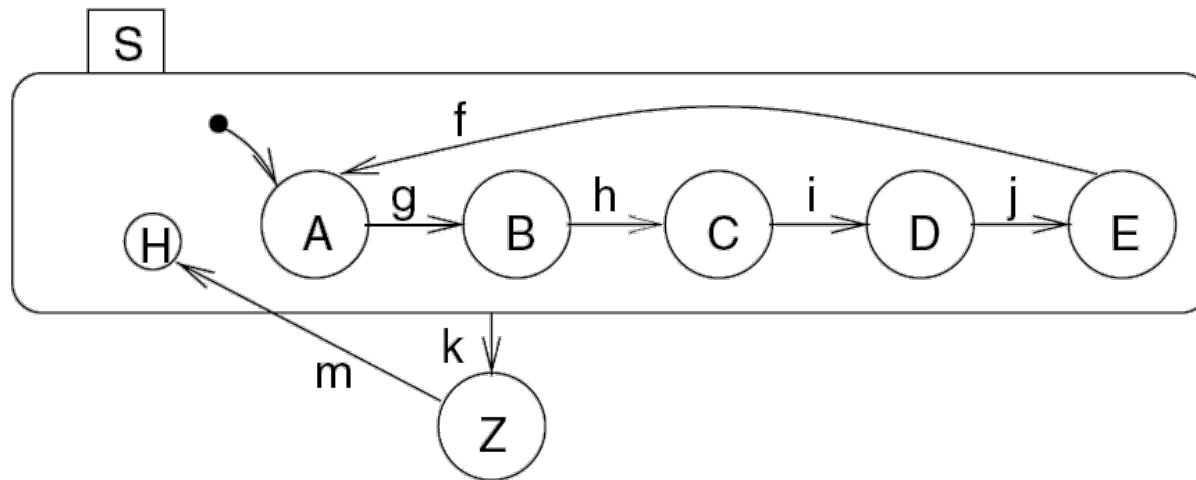
# Default State Mechanism

- ❖ Note that the **filled circle** does not constitute a state itself.



# History Mechanism

- ❖ It is possible to return to the last sub-state that was active before the super-state was left.
- ❖ The filled circle defines the next state for the very initial transition into the super-state.

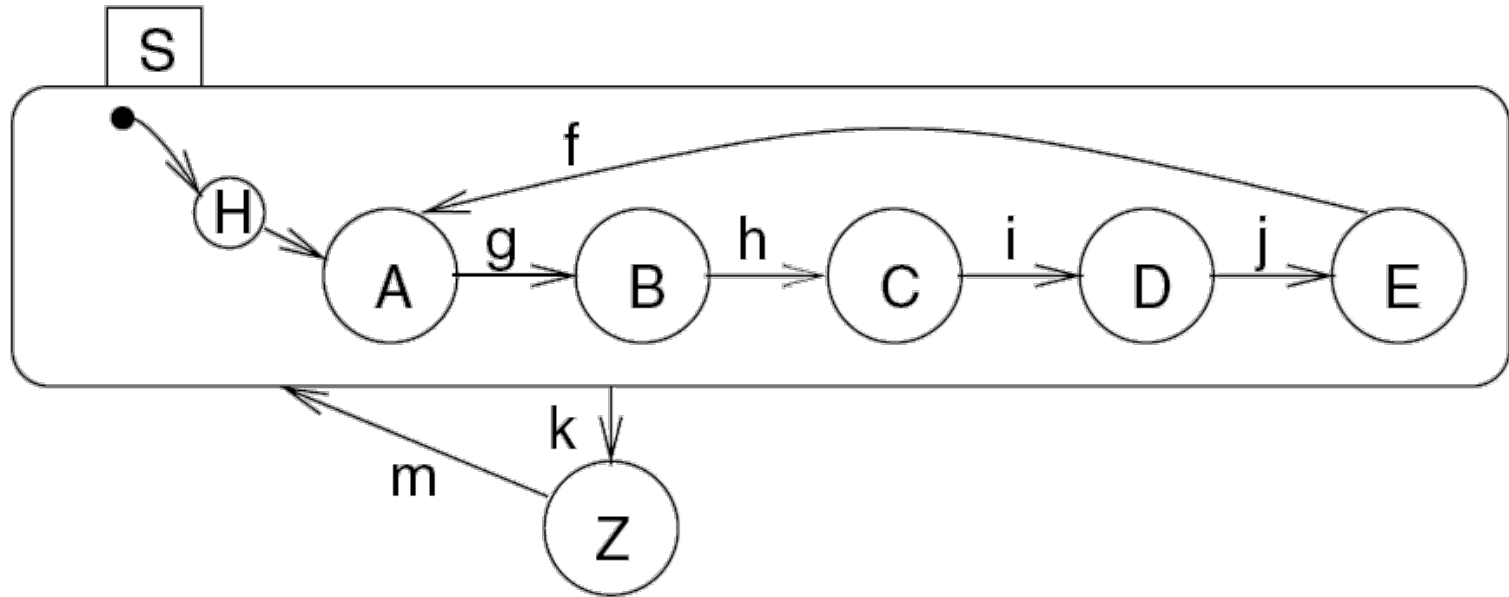




# History Mechanism

---

- ❖ Another notation (Equivalent to the previous one)

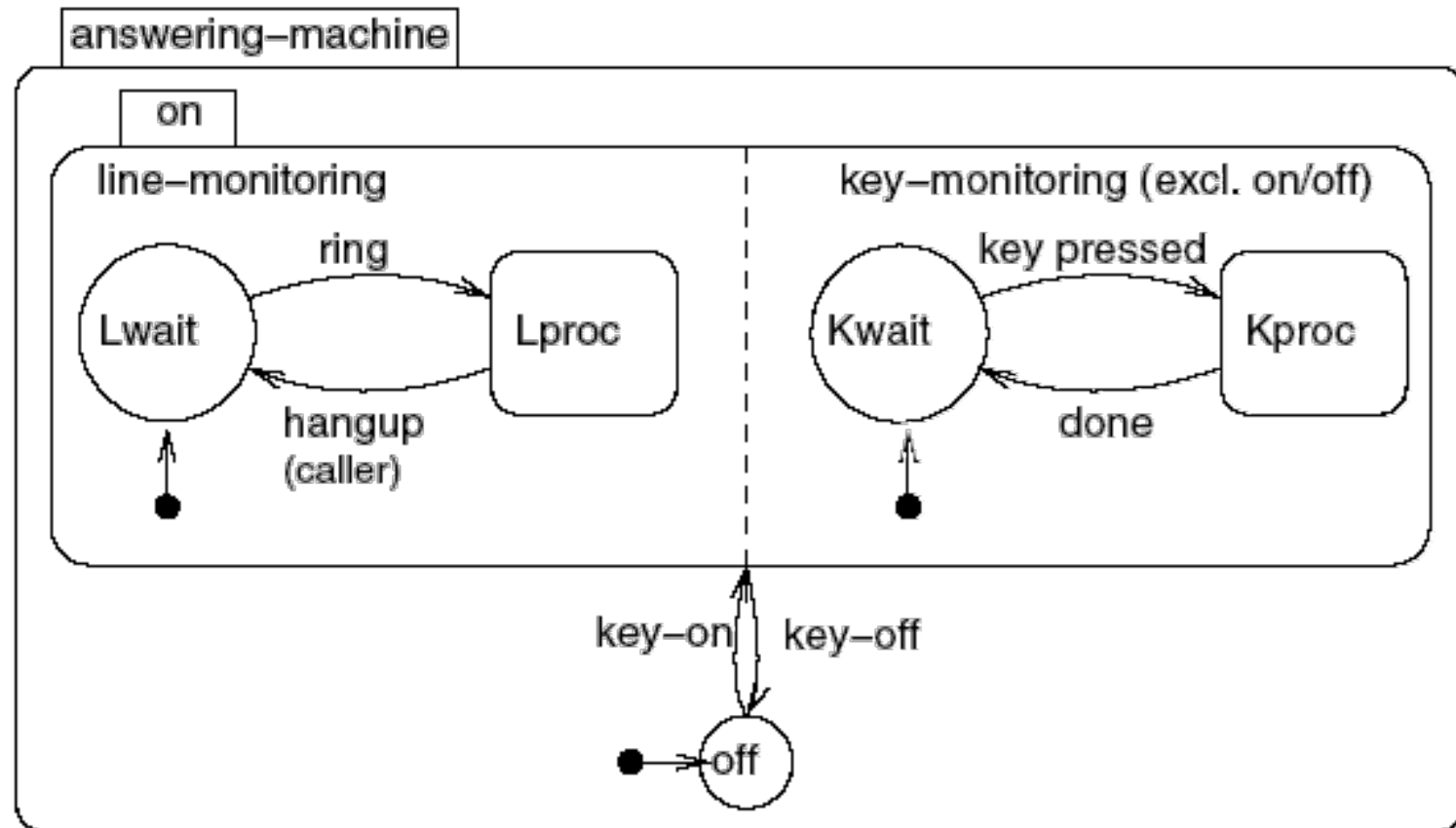


# AND-Super-States

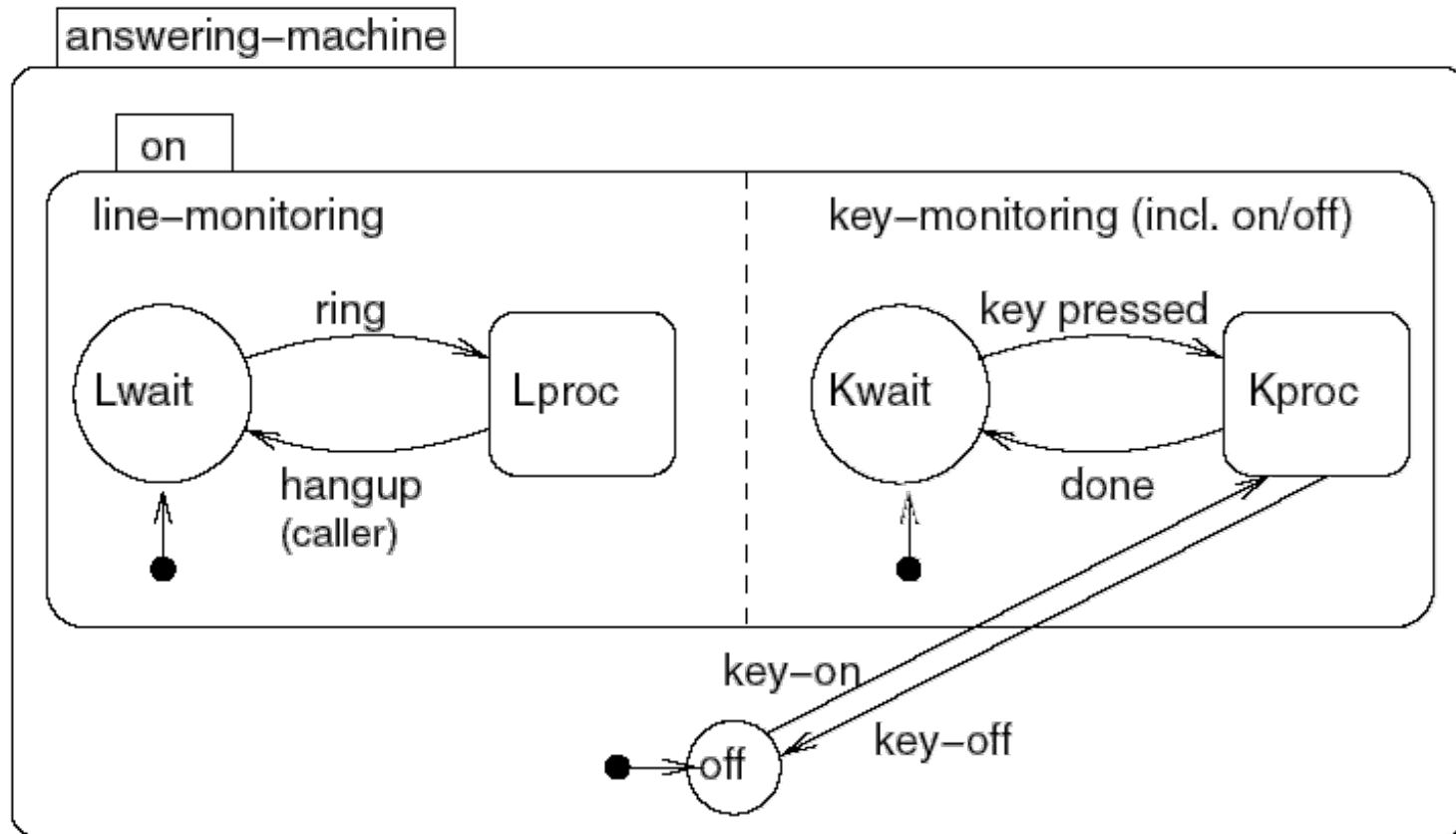
---

- ❖ The system containing an AND-super-state  $S$  is in **all** of the sub-states of  $S$  whenever it is in  $S$ .
- ❖ Also, transitions out of  $S$  always result in leaving **all** the sub-states of  $S$ .
- ❖ AND-super-states are used to describe **concurrency**.

# Example: Answering Machine



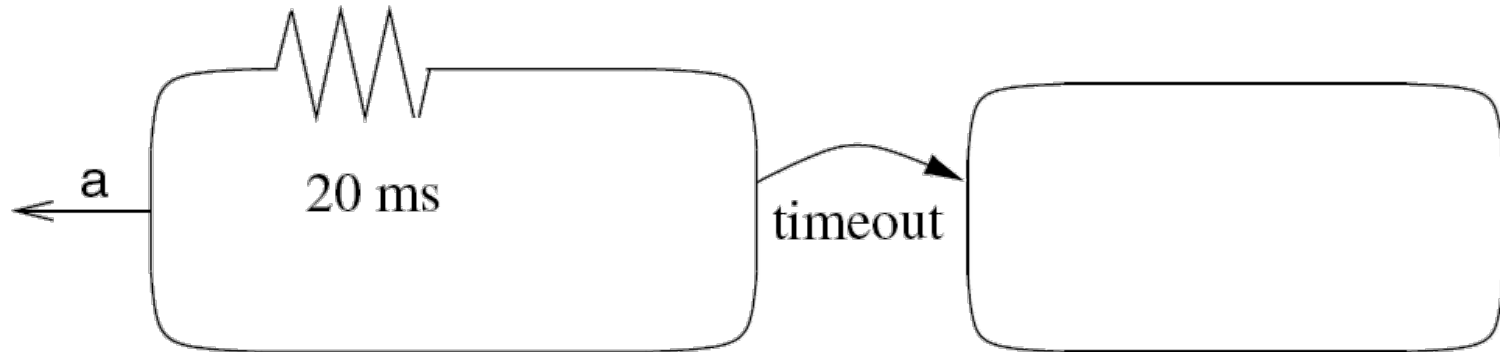
# Example (Cont.)



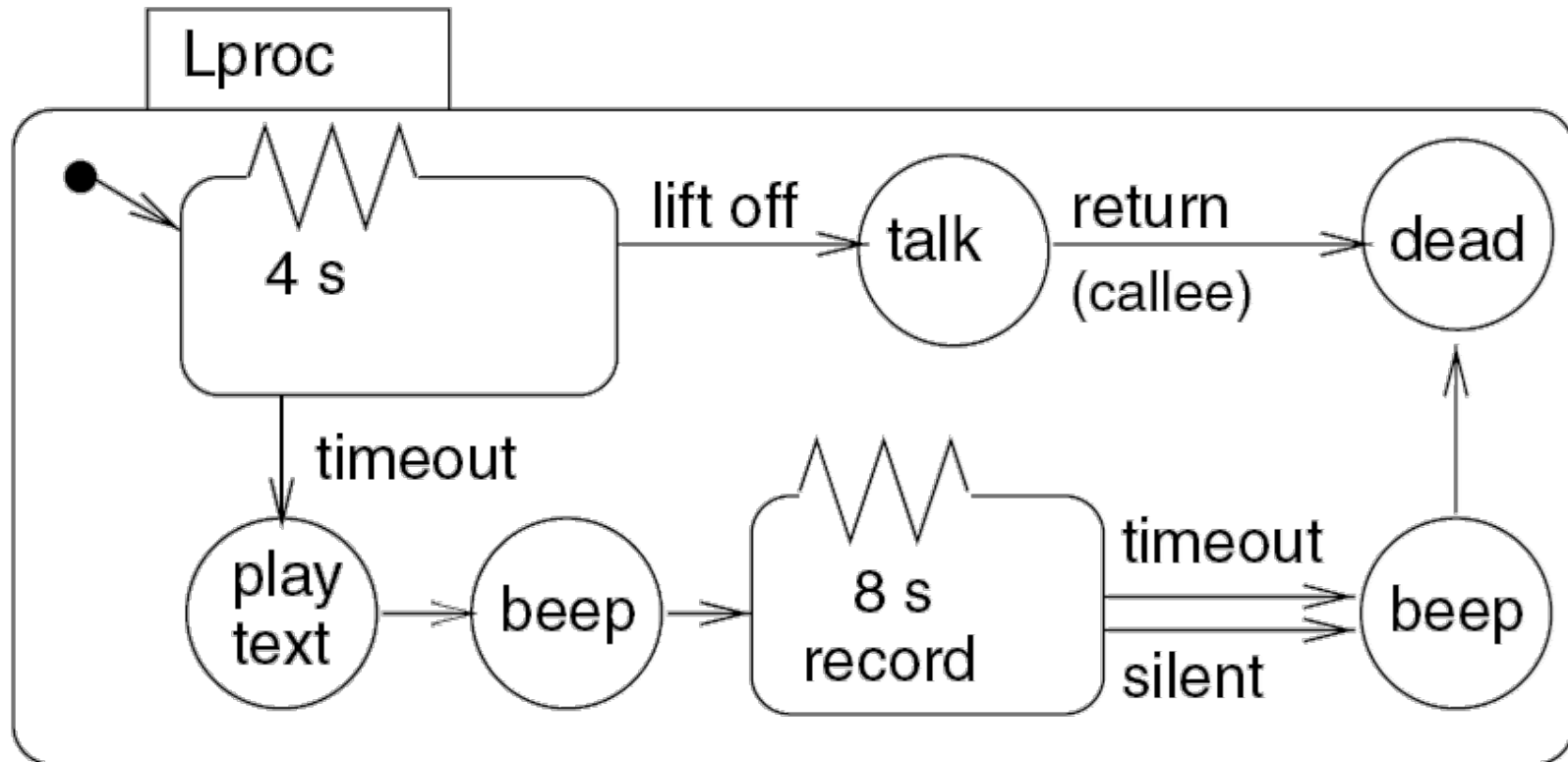
# Timers

---

- ❖ Timing
- ❖ Real-time systems



# Example



# Edge Labels

---

## ❖ Event [condition] / Reaction

- A proper reaction may depend on both the **system state** and the **event** that has happened.

## ❖ Examples:

- On-key/on:=1
- Off-key [Battery = Charged] / on:=0

# Execution of StateCharts

---

❖ Execution = Simulation

❖ Atomic Execution

- Shadow and Main Variables

- Simulation of Concurrency

- Design (Specification) Fault Avoidance for Concurrent Systems



# Assignment

---

- ❖ Incubator Temperature Control
  - Find a model for space heating:
    - You can use existing models (usually given in the form of a transfer function)
  - Provide the Statechart code for your temperature control system
  - Simulate the whole system (Simulink Stateflow) and report the results

# Moore and Mealy Automata

---

## ❖ Moore:

- Output reactions are assigned to the nodes of an automaton

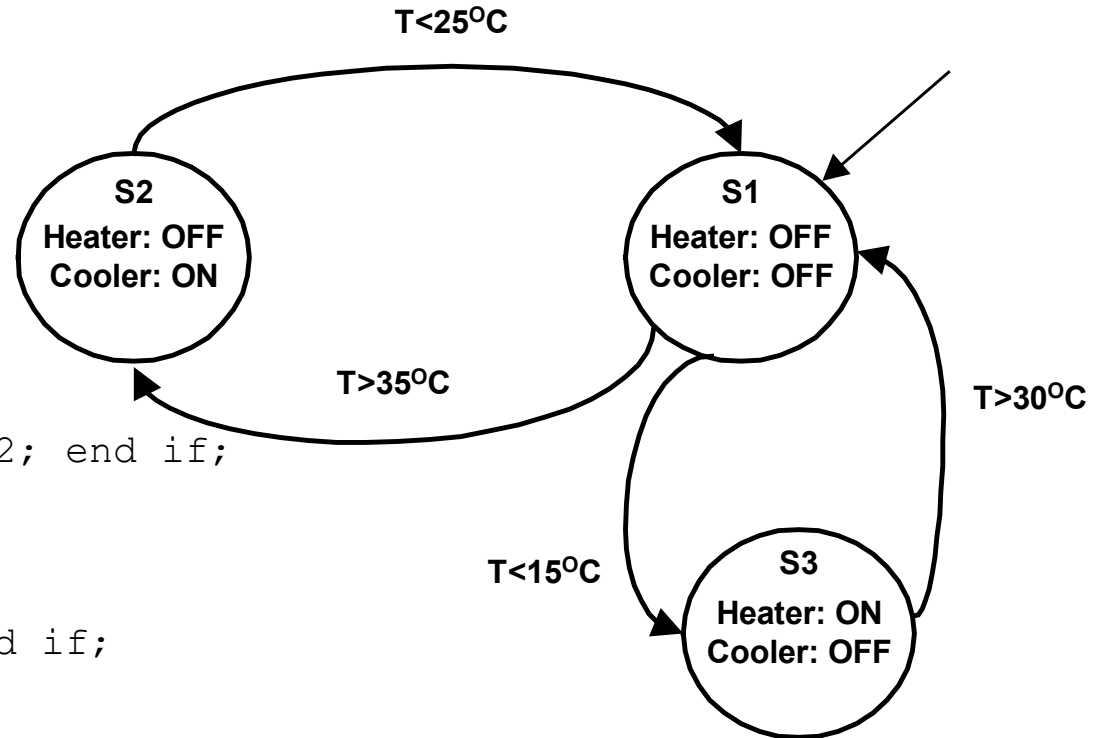
## ❖ Mealy:

- Output reactions are assigned to the arcs of an automaton
- Mixed Moore-Mealy automata are also possible.

# Example: Moore

```
State PS=S1,NS;  
Event e;
```

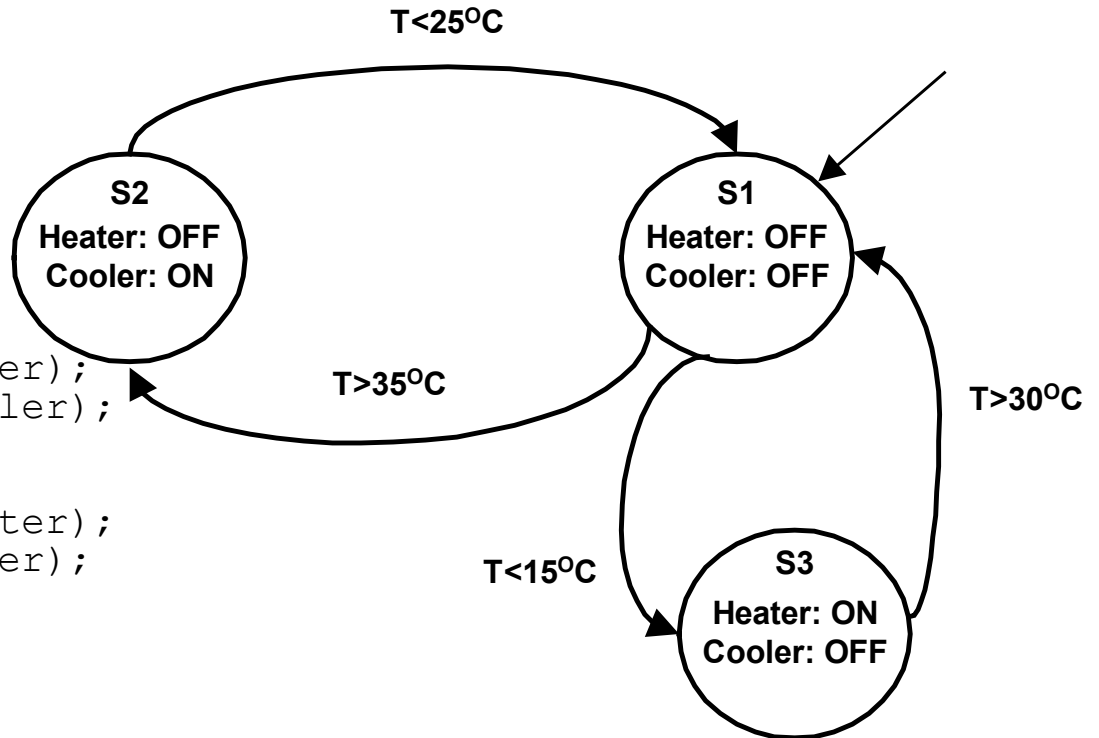
```
while(1){  
  case(PS){  
    S1:  
      Turn_off(Heater);  
      Turn_off(Cooler);  
      e=Wait for event();  
      if(e=='T<15') NS=S3;  
      else if(e=='T>35') NS=S2; end if;  
    S2:  
      Turn_off(Heater);  
      Turn_on(Cooler);  
      e=Wait for event();  
      if(e=='T<25') NS=S1; end if;  
    S3:  
      Turn_on(Heater);  
      Turn_off(Cooler);  
      e=Wait for event();  
      if(e=='T>30') NS=S1; end if;  
  }  
  PS=NS;  
}
```



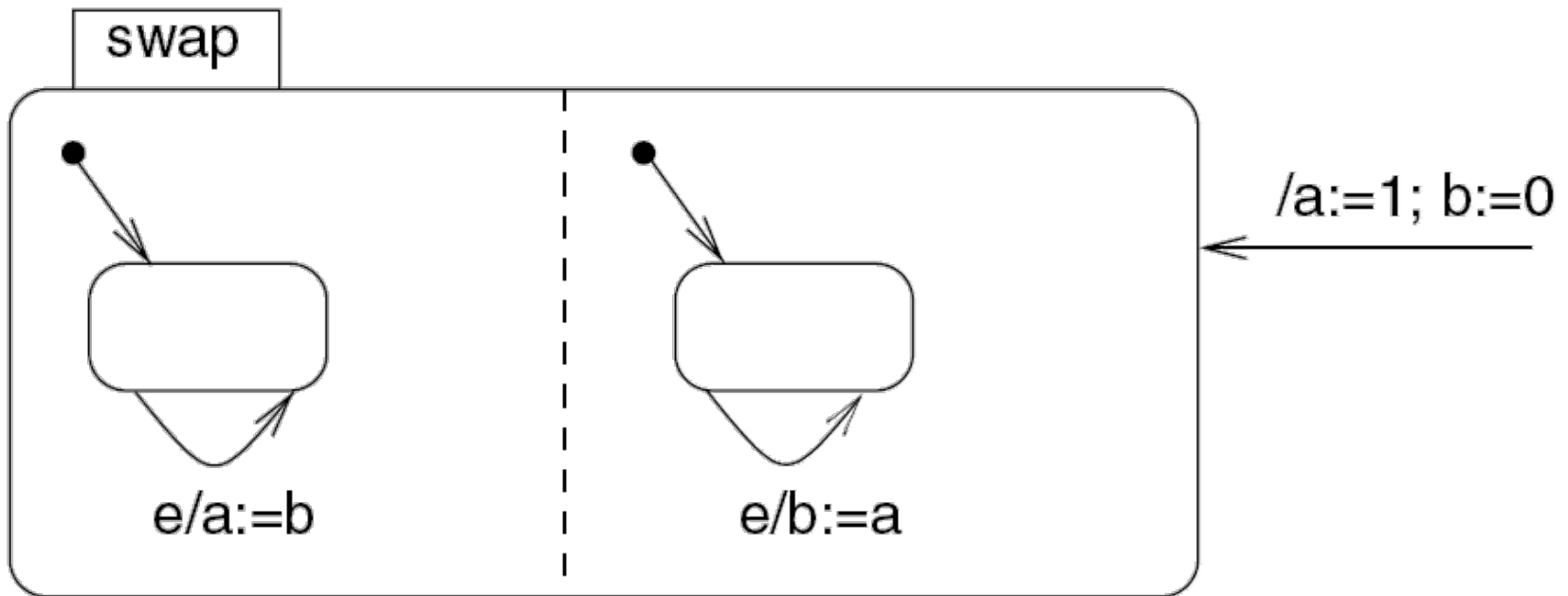
# Example: Mealy

```
State PS=S1, NS;  
Event e;
```

```
while(1) {  
  case(PS) {  
    S1:  
      e=Wait_for_event();  
      if(e=='T<15')  
        NS=S3;  
        Turn_on(Heater);  
        Turn_off(Cooler);  
      else if(e=='T>35')  
        NS=S2;  
        Turn_off(Heater);  
        Turn_on(Cooler);  
      end if;  
    S2:  
      ...  
    S3:  
      ...  
  }  
  PS=NS;  
}
```



# Mutually Dependent Assignments



# Summary

---

- ❖ Models of computation
- ❖ System Specification
- ❖ Required Features
- ❖ Modeling of Hierarchy
- ❖ Design Modularity
- ❖ Moore and Mealy Automata