1.0.0
20251105

Embedded Systems and IoT

Ingegneria e Scienze Informatiche - UNIBO

a.a 2025/2026

Lecturer: Prof. Alessandro Ricci

[module-2.2]

MODELLING EMBEDDED SYSTEMS WITH FINITE STATE MACHINES

OUTLINE

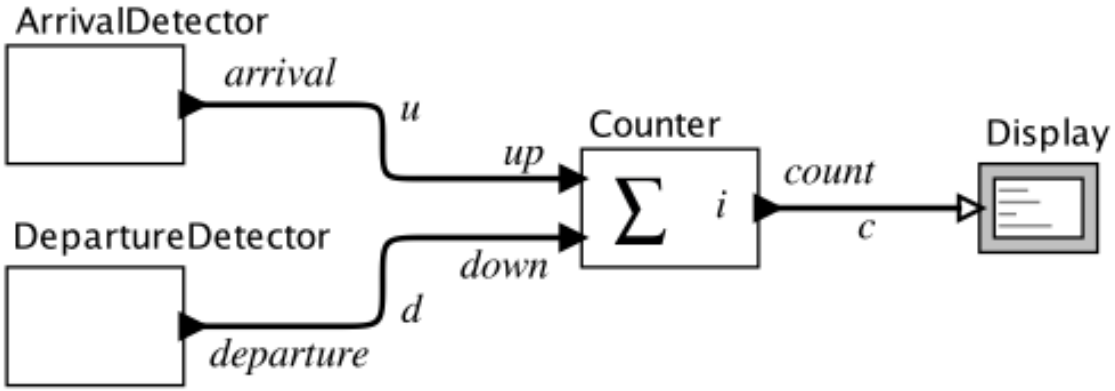
- In this model we define in a rigorous way the model based on finite state machines (FSM)
- The content is based on chapter 3 of **[IES]** e on the full textbook **[PES]**
  - in bibliography

MODELS FOR EMBEDDED SYSTEMS

- Models of embedded systems include both *discrete* and *continuous* components.
  - continuous components evolve smoothly, while discrete components evolve abruptly, with ~atomic changes
- The physical dynamics of the system can often be modeled with ordinary differential or integral equations
- Instead, discrete components featuring a *discrete dynamics* instead can be effectively modelled by **state machines**
- In this course we will consider mainly embedded systems composed by discrete components, adopting **Finite State Machines** as main modelling formalism

EXAMPLE: PARKING GARAGE

- Consider a system that counts the number of cars that enter and leave a parking garage in order to keep track of how many cars are in the garage at any time ([IES], chapter 3, p. 42)
- It can be modelled by three main interacting subsystems:
  - *ArrivalDetector*, produces an event when a car arrives
  - *DepartureDetector*, produces an event when a car departs
  - *Counter*, keeps a running count, starting from an initial value i. Each time the count changes, it produces an output event that updates a display.



- In the above example, each entry or departure is modeled as a **discrete event**

COUNTER

- The *Counter* subsystem reacts to the sequence of input events and produce an output event
  - the input is represented by a pair of discrete signals (up/down) that in some specific moments may have an event (i.e. they are on) and in other moment not (i.e. they are absent).
  - the output is a discrete signal that, when the input is present, has a value and it is a natural number, and in other moments it is absent
- When an event is present at the up/down port, *Counter* increments/ decrements the count and produces its value in output.
  - in all other cases the output is absent

INPUT AND OUTPUT SIGNAL MODEL

- The u and d input signals (up e down) that represents the occurrence of events can be modelled as the functions:

  u: $\mathbb{R} \longrightarrow$ { absent, present }

  At each time t, the signal u(t) is evaluated either as “absent" (no events in that moment), or “present”, meaning that there is an event in that moment
  - these signals are called “pure”
    - they do not carry any information content but the presence or absence
- The output signal c can be represented by a function:

  c: $\mathbb{R} \longrightarrow$ {absent} U $\mathbb{N}$

  This is not a pure signal: it carries also an information content (besides possibly being “absent” like u and d)

DYNAMICS: REACTIONS

- The dynamics of a discrete system like in this case can be described as a sequence of steps called ***reactions,*** being each reaction instantaneous (duration equals to zero)
- The reactions of a discrete system are *triggered* by the environment where the system works
  - when they are triggered by input events they are called *event-triggered*
  - parking example:
    - the reactions of the Counter subsystem are triggered when one or more input events about the arrival or departure of cars are present
    - when both the input events are absent, then no reactions are triggered

VALUATION OF INPUTS AND OUTPUTS

- The execution of a reaction lead to a ***valuation*** of the inputs and outputs, that is:
  - a **variable** is associated to each *input signal* and the valuation of the input consists in assigning the value of the input signal at that time to the variable
  - the same applies for *output signals*
    - for each output signal we use a variable that is assigned to the value of the output function in that moment
  - the values can include also “absent”, meaning no signals

THE NOTION OF STATE

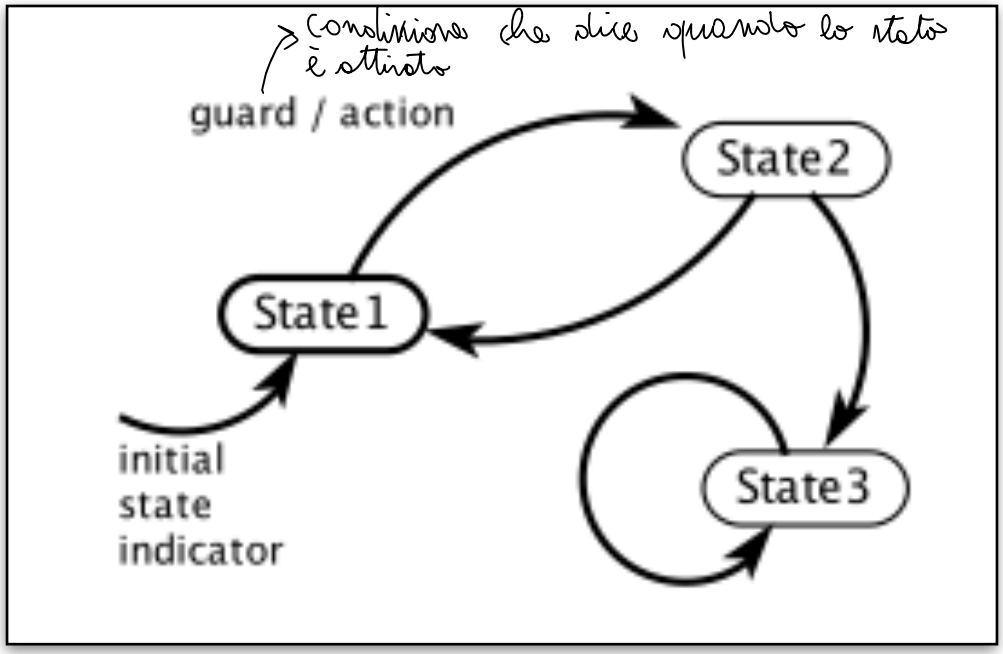
- Intuitively, the **state** of a system is its condition at a particular point in time and, in general, the state affects how the system reacts to inputs
- Formally, we define the state to be *an encoding of everything about the past that has an effect on the system's reaction to current or future inputs*
  - the state is a summary of the past
- The the *Counter* subsystem of the example, *state(t)* — i.e. the state at time t — can be represented by an integer in the range between 0 and M, where M is the max number of spaces
- Given the set `States` = { 0, 1, 2, ... M }, then the state can be modelled as the function:

  $\texttt{state}: \mathbb{R} \longrightarrow \texttt{States}$

FINITE STATE MACHINES

IMPORTANT

- A **state machine** in general is a model of a system with discrete dynamics that at *each reaction maps valuations of the inputs to valuations of the outputs, where the map may depend on its current state*.
- A **finite-state machine** (**FSM**) is a state machine where the set States of possible states is finite.
- If the number of states is reasonably small, then FSMs can be conveniently drawn using *state diagrams:*



States =
{ State1, State2, State3 }

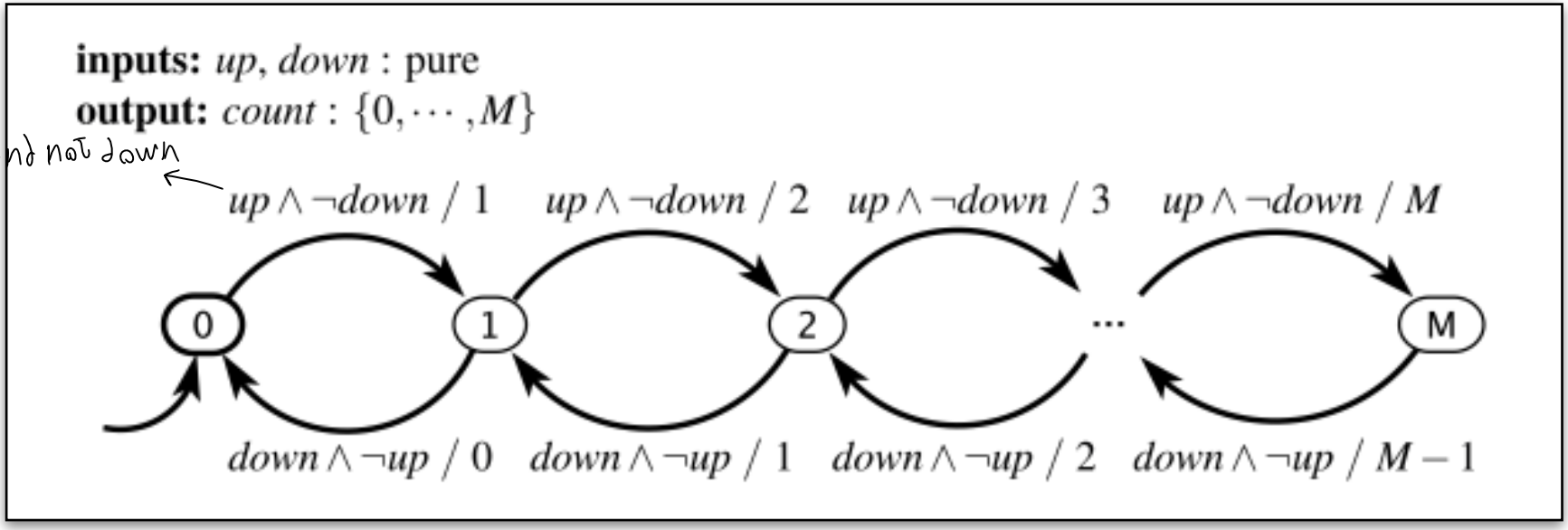
Initial state = State1

TRANSITIONS

IMPORTANT

- **Transitions** between states govern the discrete dynamics of the state machine and the mapping of input valuations to output valuations
  - a transition is represented as a curved arrow, going from one state to another
  - a transition may also start and end at the same state (es: State3)
    - in this case, the transition is called a self transition.
- Transitions are characterised by a **guard** and an **action**
  - the *guard* determines whether the transition may be taken on a reaction.
    - it is represented by a predicate (a boolean-valued expression) that evaluates to true when the transition should be taken
    - when a guard evaluates to true we say that *the transition is enabled*
  - the *action* specifies what outputs are produced on each reaction
    - it is an assignment of values (or absent) to the output ports
    - any output port not mentioned in a transition that is taken is implicitly absent

PARKING GARAGE EXAMPLE



- The predicates in the guards involve the valuation of pure functions that are treated as boolean expressions
  - the expression *up* means that the valuation of the *up* signal is “present”, while ¬*down* means that the valuation of the *down* signal is “absent”, i.e. signal not present

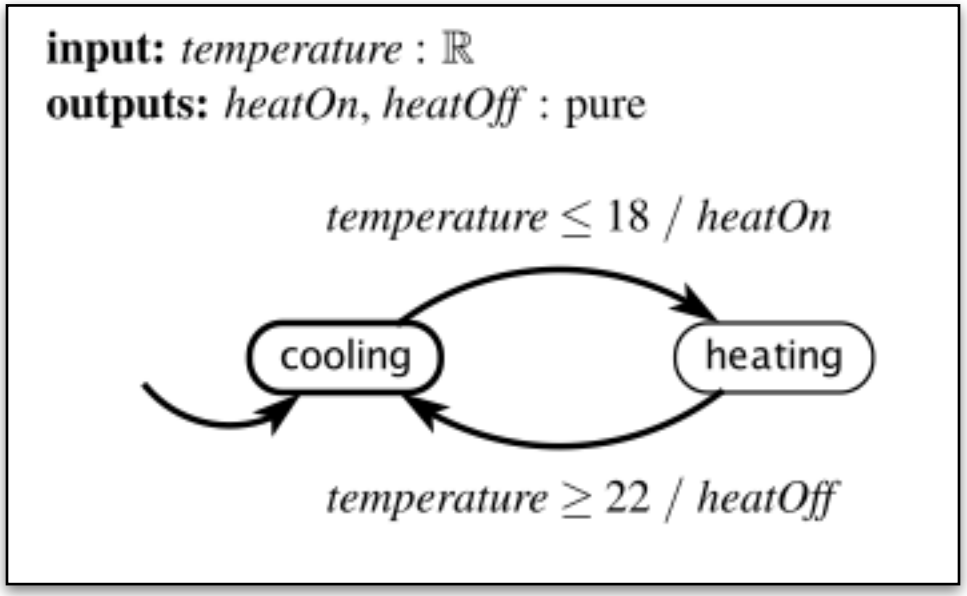
ASYNCHRONOUS AND SYNCHRONOUS FSM

IMPORTANT

- A FSM does not specify when valuation should be carried on to possible trigger the a reaction
- Two possibilities
  - **Asynchronous FSM, also called Event-triggered FSM**
    - in this case, the valuation occurs anytime there is an input event
    - in this case, the environment where the system works establish when the reactions are evaluated and triggered
  - **Synchronous FSM, also called Time-triggered FSM**
    - in this case, valuations occur periodically, at a regular interval of of time called *period*
      - the period determines the working frequency/rate of the machine

HVAC EXAMPLE

- HVAC system (heating, ventilation, air conditioning)



- This FSM can be conceived either as *event-triggered* — if the machine reacts each time the input temperature changes — or *time-triggered*, by evaluating (and reactive) at a regular pace
- NOTE: the FSM structure is the same

INPUT/OUTPUT MODELLING

- When applying FSM to embedded systems, the input/output are then mapped variables that are used to define guards and actions [PES book]
  - **input variables**
    - modified by the external environment where the machine operates
  - **output variables**
    - modified by the machine by means of actions, to control the external environment
  - besides I/O, variables can be used to flexibly define the global state of the machine
  - guards in transitions are defined using these variables (both input and state variables)
- Examples
  - blinking, button-led

IMPLEMENTING A FSM

- General schema to implement a FSM as a program controlling the embedded system behaviour
- Full process from design to code:
  - first the FSM is defined, representing a model of the behaviour (design stage)
  - then it is converted/implemented into a program to be run on microcontrollers
- In *model-driven engineering*, this conversion is automated by means of proper tools

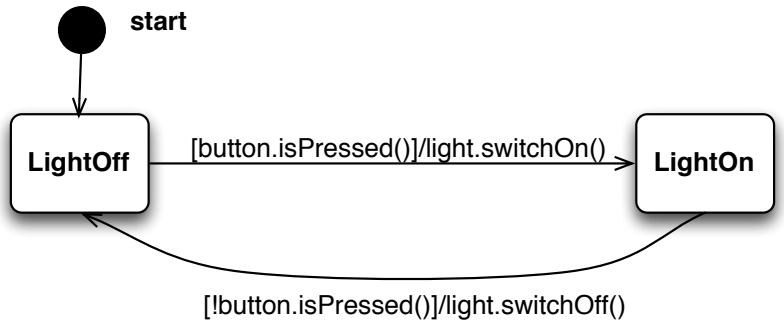
```
/* global vars used by the machine
*/
int a0, b0, …

/* variable keeping track
   of the state */
enum States {…} state;

/* procedure implementing
  the step of the state machine */

void step(){
  switch(state) {
      case …
      case …
  }
}

loop(){
 step();
}
```

IMPLEMENTING A FSM: NOTES

- Remarks
  - explicit representation of the states
    - e.g. by means of enum constants
  - there is a variabile that keeps track of the current state
    - “state” in the example
  - step() of the machine in the main loop
    - it checks which transitions are enabled, depending on the current state
    - it executes the actions that are specified by the transition and the state variable is changed accordingly
  - when working with synchronous FSM, then the loop should be executed with a specified period
- UML representation by means of *statecharts*

BUTTON-LED EXAMPLE

- General schema to implement a FSM upon a control loop architecture
  - explicit state representation



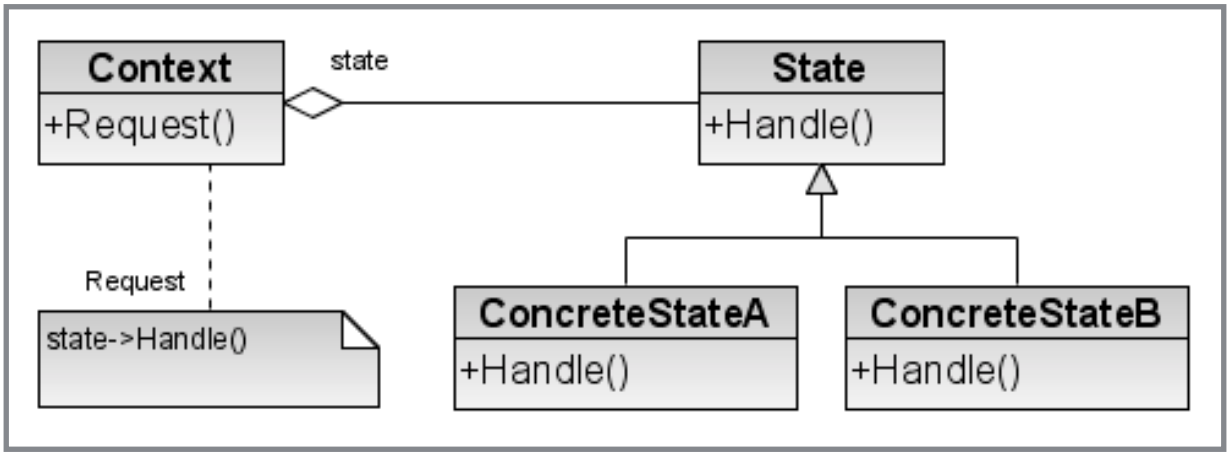

```
enum States { LightOn, LightOff } state

setup(){
  state = LightOff
}

step() {
  switch (state){
    case LightOff:
      if (button.isPressed()){
        light.switchOn()
        state = LightOn
      }
    case LightOn:
      if (!button.isPressed()){
        light.switchOff()
        state = LightOff
      }
  }
}

loop {
  step()
}
```

DISCIPLINE

- In a correct FSM model and implementation:
  - the computations related to actions should always terminate
    - there should be no infinite loop
    - in theory they should be instantaneous
  - the valuation of a condition should not change the state of variables

STATE PATTERN

- State pattern [GOF]



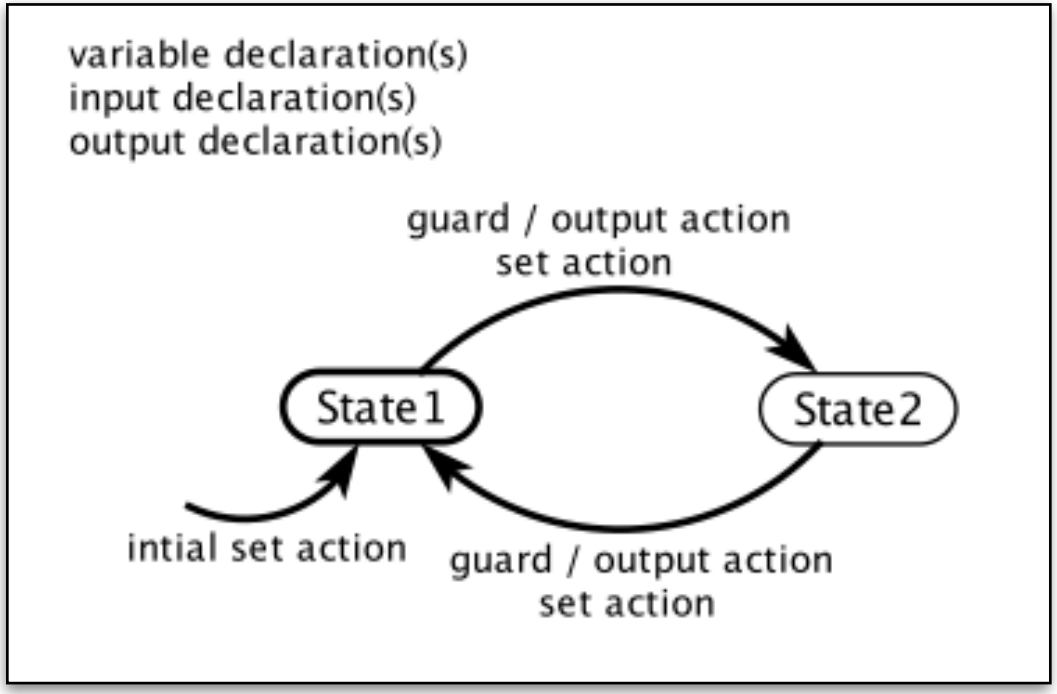
- Loop + State

```
interface State {
  do()
  nextState(): State
}

class MyState1 implements State {…}
class MyState2 implements State {…}
```

```
…
state = initialState
loop {
 state.do()
 state = state.nextState()
}
```

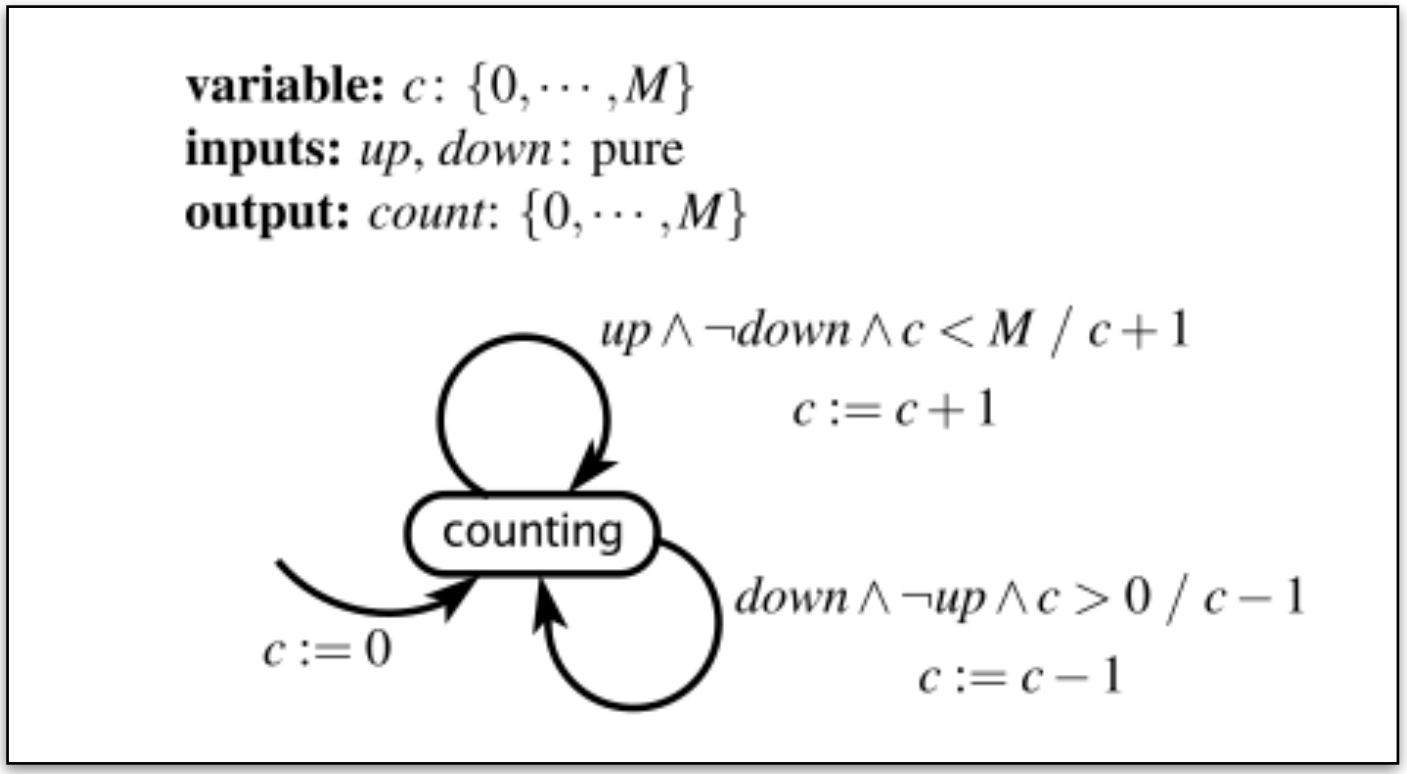
EXTENDED FSM

- Extended FSM are FSM machines in which variables are flexibly used also to describe the state, to make the overall description more concise and effective
  - intensional state representation
- In this case, actions may include also changes over the state variables



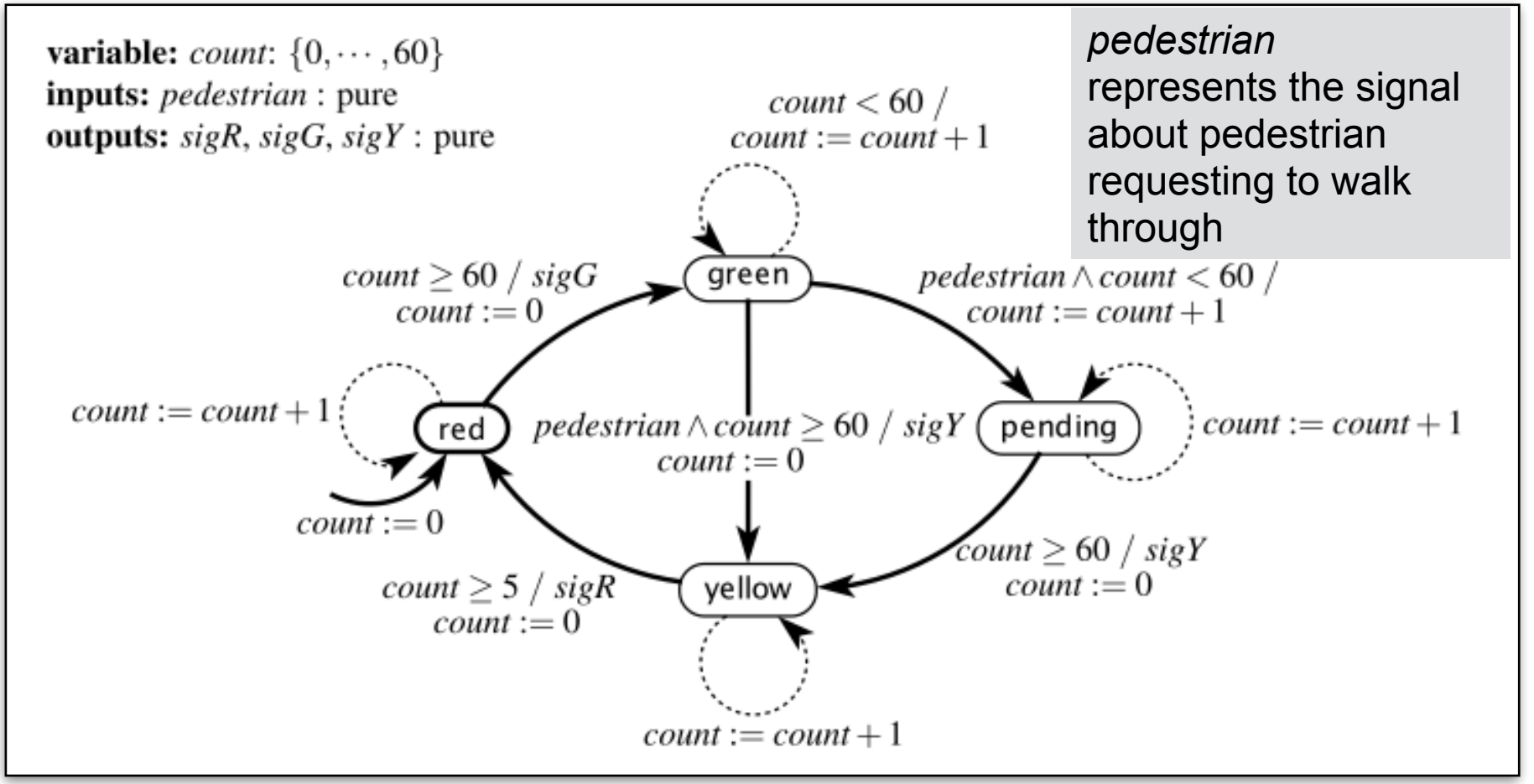
PARKING GARAGE EXAMPLE

- System modelled as an Extended FSM



TIME TRIGGERED EFSM EXAMPLE

- FSM for a pedestrian traffic light
- We consider a time-triggered version with period = 1 second
  - so it reacts one time per second



STATE SPACE SIZE IN EFSM

- The total number of states in a EFSM can be computed by considering all possible admissible configurations of state variables

  $|States| = n*p^m$

  where

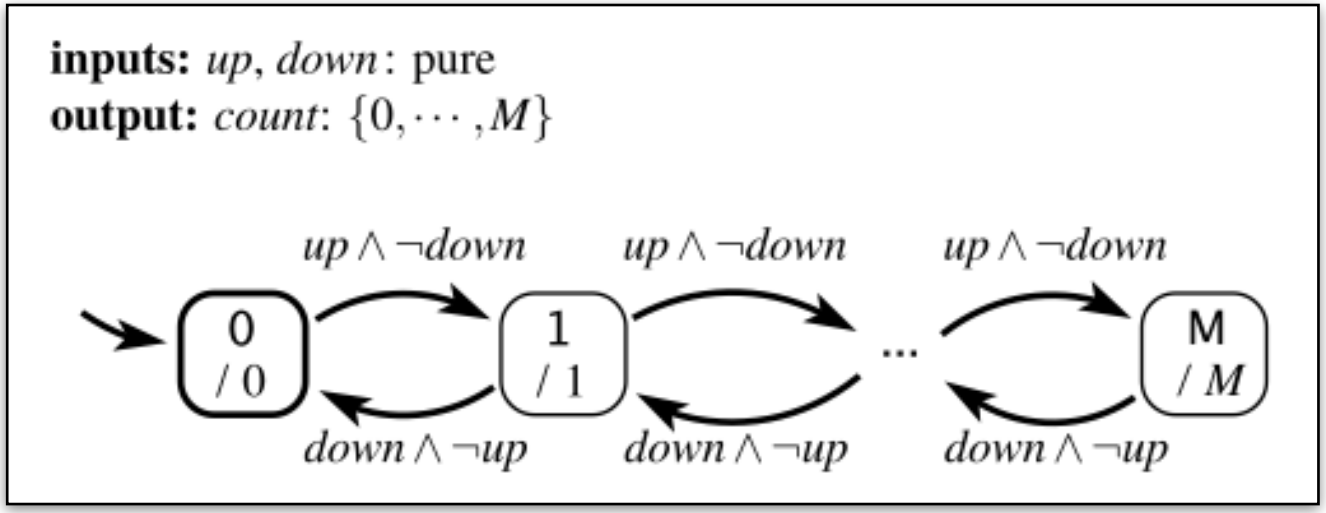
  n = number of discrete states
  m = number of variables used to describe the state
  p = possible set of values that can be assigned to variables

- Parking garage example:
  - $|States| = 1*(M+1)^1 = M+1$

MEALY AND MOORE MACHINES

- State machines described so far are also called **Mealy machines**
  - in honour of Gordon Mealy, Bell Labs engineer that published for the first time this formalism in 1955
  - key characterisation: they produce outputs (actions) when the transition is triggered
- Alternative approach: **Moore machines**
  - in this case, the output is bound to states and is produced when the machine enters into that state
  - in honour of Edward Moore, Bell Labs, who introduced them in 1956

PARKING GARAGE AS A MOORE MACHINE



- The output is determined by the state, not by the transitions
  - the input determines which transition is going to be triggered, but not which output
  - these machines are also called *causal machines*
- Given a Moore machine it is possible to define an equivalent Mealy machine
  - viceversa, it is possible to define a Moore machine from a Mealy machine, in which the output is produced in next reaction, not in current one

FOCUS: TIME-TRIGGERED FSMs

IMPORTANT

- The behaviour of an embedded system is typically **time-oriented**
  - **time** is often part of the specification in guards and actions (e.g. as deadlines), as well as to define regular/periodic behaviours
    - es: blinking
      - switch on and off a led every 500 ms
- Synchronous **FSM** have been introduced for this purpose [PES], to provide an effective way to handle time and time passing, to ease the specification of time-oriented behaviours
  - in this machine, the “step” of the machine is executed periodically, considering some specified **period**
  - it is like that FSM has an internal *clock* and transitions occur only at the clock tick

IMPLEMENTING TIME-TRIGGERED FSM: TIMERS

- Programmable Timers are typically used to implement time-triggered FSM
  - programmed to generate an interrupt at the desired frequency = with the specified period
  - the interrupt then should lead to execute a step of the machine

EXAMPLE IN PSEUDO-CODE [PES]

```
volatile int timerFlag = 0;

void timerISR(){
  timerFlag = 1;
}

/* procedure implementing the step of the state machine */
void step(){…}

loop(){
  while (!timerFlag){}; /* wait for a tick for doing the next step */
  timerFlag = 0;
  step();
}
```

SYNCHRONOUS BLINKING

- Arduino Blinking example realised by a synchronous FSM, period 500 ms

```
#include "Led.h"
#include "Timer.h"

#define LED_PIN 13

Light* led;
Timer timer;

enum { ON, OFF} state;

void step(){
  switch (state){
    case OFF:
      led->switchOn();
      state = ON;
      break;
    case ON:
      led->switchOff();
      state = OFF;
      break;
  }
}
```

```
…

void setup(){
  led = new Led(LED_PIN);
  state = OFF;
  timer.setupPeriod(500);
}


void loop(){
  timer.waitForNextTick();
  step();
};
```

Timer is a OO class using a Timer lib in its implementation (included in the repo sources)

MULTIPLE TIME-SCALES

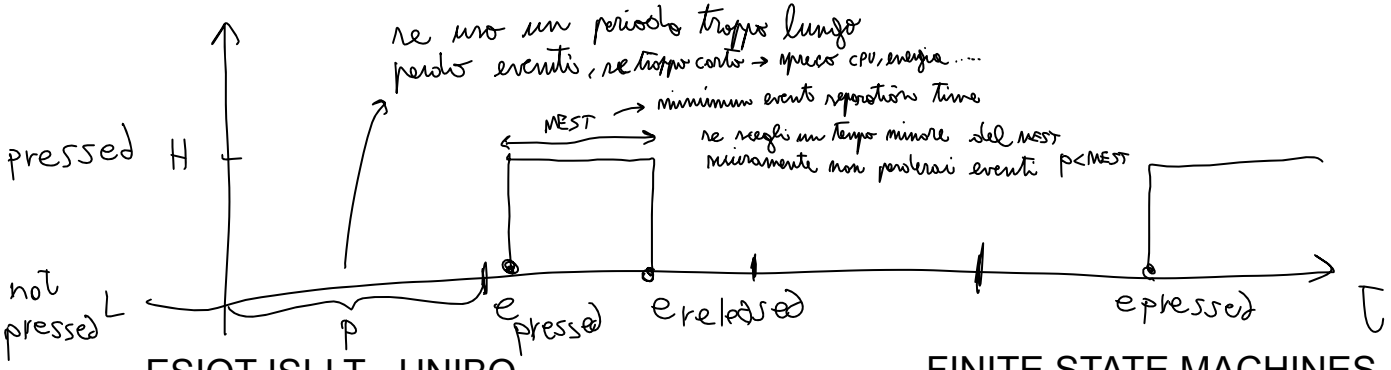
- A system can involve different timings and temporal intervals to be managed
  - e.g. blinking, with LED on for 500 ms and off for 750 ms
- The general strategy is to consider the greatest common divisor (GCD) as period of the machine
  - blinking example => period = 250 ms

INPUT SAMPLING

IMPORTANT

- **Sampling** is term used to refer to the periodic reading of sensors, at some frequency (i.e. with some period)
  - *sampling rate* = frequency of the periodic reading
    - *sampling rate = 1 / period*
    - measured in Hertz (Hz)

- **Choosing the period is a critical choice in the design of a synchronous FSM**
  - the period should be *small enough* not to loose events
    - button-led example
  - the period should be *big enough* to avoid overrun exceptions (discussed in next module)



BUTTON-LED EXAMPLE

- Synch FSM for a button-led — choosing a period = 500 ms
  - *too big,* we **loose events**

```
#include "Led.h"
#include "Timer.h"

#define LED_PIN 13
#define BUTTON_PIN 2

Light* led;
Button* button;
Timer timer;

enum { ON, OFF} state;

void setup(){
  led = new Led(LED_PIN);
  button = new ButtonImpl(BUTTON_PIN);
  state = OFF;
  timer.setupPeriod(500);
}

void loop(){
  timer.waitForNextTick();
  step();
};
```

```
…
void step(){
  switch (state){
    case OFF:
      if (button->isPressed()){
        led->switchOn();
        state = ON;
      }
      break;
    case ON:
      if (!button->isPressed()){
        led->switchOff();
        state = OFF;
      }
      break;
  }
}
```

CHOOSING THE SAMPLING RATE

IMPORTANT

- The greater is the frequency (i.e. the smaller is the period) and:
  - the higher is the ***reactivity***, on the one hand
  - the bigger is the use of the microcontroller and then power consumption, on the other hand
- *Then, as a general strategy, the period should be the greatest value of that range of values that guarantee not to loose events and so the correct functioning of the system*

MINIMUM EVENT SEPARATION TIME

IMPORTANT

- The **minimum event separation time** (MEST) is defined as the smallest interval of time that can occur between two input events, given the environment in which our system is operating

Theorem

- *in a sync FSM, by choosing a period less than MEST, we have the guarantee that all events will be detected, i.e. no events will be lost*

- Button-Led example
  - the MEST is the interval between pressing the button and release the button events
  - by choosing a period which is smaller than MEST, we are sure to get all pressing button events and therefore all the states in which the button is pressed

LATENCIES

- Some input events may be required to generate some output events or actions, within a certain amount of time
- The interval between the occurrence of the input event and the corresponding generation of the output event is called **latency**
  - in the button-led example: the time between pressing the button and the switching on of the led is a latency
- A typical objective in embedded systems is to **minimise latencies**
  - there is a relationships between the period and latencies
    - the smaller is the period, the smaller are latencies
  - e.g. button-led example
    - a 300ms latency would not be acceptable - it would be perceived as an unwanted delay by a user
    - a 50 ms instead would be acceptable

INPUT CONDITIONING

- Sensors in general can be affected by physical or HW imperfections, so that we need to apply some ***input conditioning*** in order to avoid errors in sampling the input signals
- An example is **bouncing** in tactile buttons
  - bouncing is the phenomenon happening internally to buttons (in particular in low quality ones..) so that a single pressure of the button can lead to an input signal switching fast multiple times between LOW and HIGH, before being stable on HIGH, because of mechanical bouncing of the inner part of the button
  - in this case, if the sampling rate is high (i.e. the period is small), then these bounces can be detected by sampling, resulting in sampling multiple pressures of the button

BUTTON DE-BOUNCING

- **Button debouncing** is about ignoring bounces i.e. spurious pressures, so that a single pressure is detected
- It can be done both via HW and SW
  - at a software level,
    - we could choose a sampling rate less than the bouncing frequency — i.e. a sampling period > bouncing period
      - e.g. for bouncing period could be ~20ms, we choose a sampling period of ~50ms
    - we could track the time of input events, discarding those events that are judged as not good
- Besides bouncing, filtering techniques can be applied to discard any spurious input events or signals (**glitches**, or **spikes)**

SYNC BUTTON-LED

- Sync FSM del button-led, with a proper period

```
#include "Led.h"
#include "Timer.h"

#define LED_PIN 13
#define BUTTON_PIN 2

Light* led;
Button* button;
Timer timer;

enum { ON, OFF} state;

void setup(){
  led = new Led(LED_PIN);
  button = new ButtonImpl(BUTTON_PIN);
  state = OFF;
  timer.setupPeriod(50);
}

void loop(){
  timer.waitForNextTick();
  step();
};
```

```
…
void step(){
  switch (state){
    case OFF:
      if (button->isPressed()){
        led->switchOn();
        state = ON;
      }
      break;
    case ON:
      if (!button->isPressed()){
        led->switchOff();
        state = OFF;
      }
      break;
  }
}
```

BIBLIOGRAPHY

- **[IES]** Introduction to Embedded Systems. Lee, Sheshia.
  - capitolo 3
- **[PES]** Frank Vahid, Tony Vargis, Bailey Miller. *Programming Embedded Systems: An introduction to Time-Oriented Programming*. University of California, Riverside.
- **[GOF]** Erich Gamma; Richard Helm, Ralph Johnson, John M. Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley