Lecture 11. State-Based Design and its Implementation

Valeriy Vyatkin

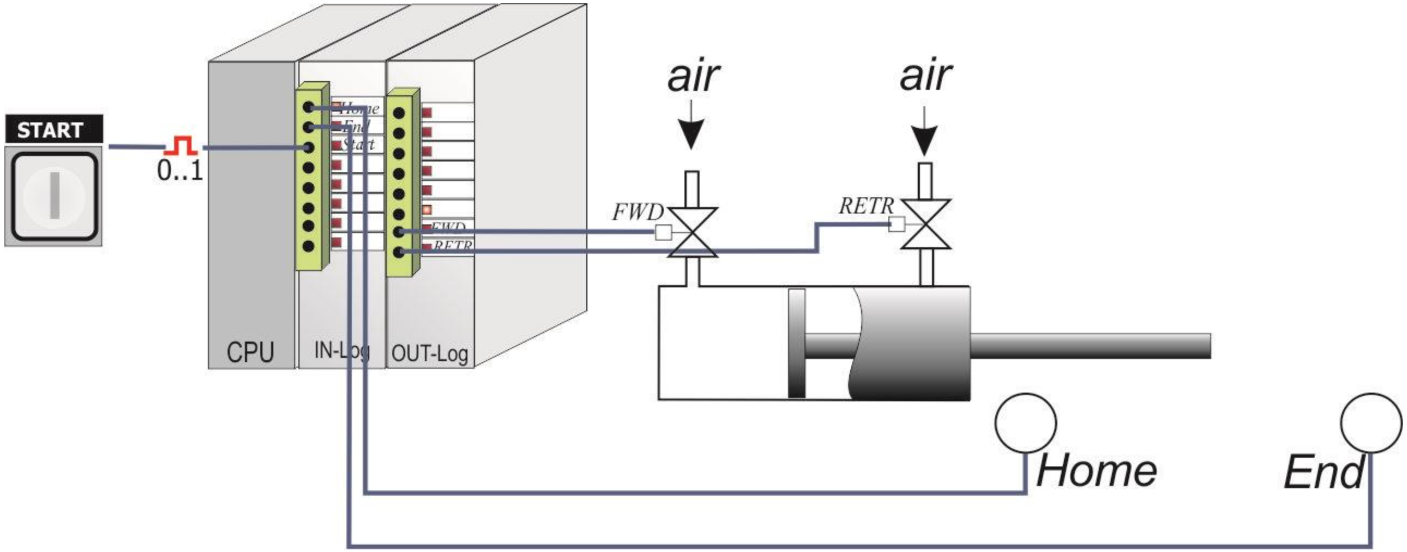
Aalto University

2022

State-based Logic Design

Example: Double acting Pneumatic Cylinder

Drawing a state machine for a Double Acting Pneumatic Cylinder



In this example, system has 2 control signals

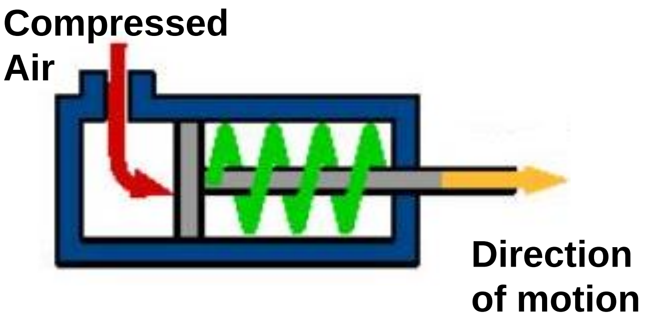
1. FWD (Forward)
2. RETR (Retract)

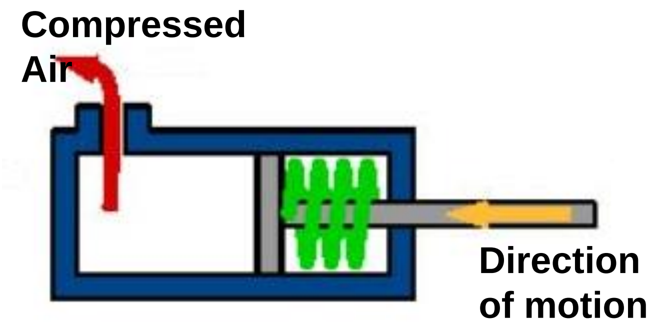
Pneumatic Cylinder

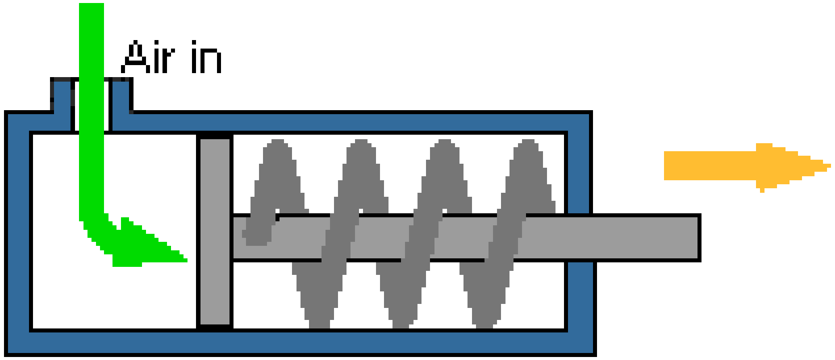
1. Converts the energy in the compressed air into linear motion.
2. The air enters the cylinder and pushes a piston from one end of the cylinder to the other.
3. There are two main types of cylinder - **Single acting** and **Double acting**

Single Acting Pneumatic Cylinder

1. In a **single acting cylinder**, the piston is forced out by the pressure of the air.
2. When the air supply is removed and the air inside the cylinder is allowed to escape, the piston moves back, driven by the force of a spring.
3. By restricting the escaping air *(exhaust)*, it is possible to slow down the return movement of the piston.

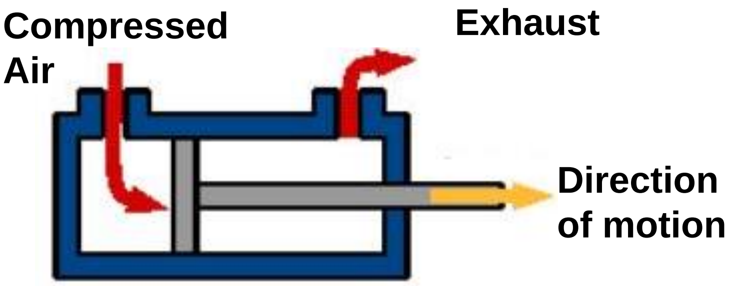


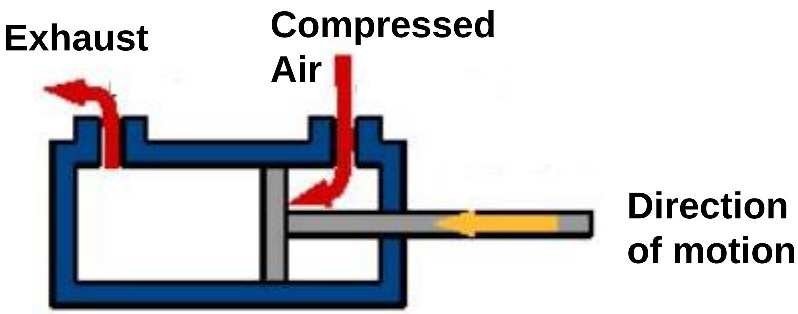


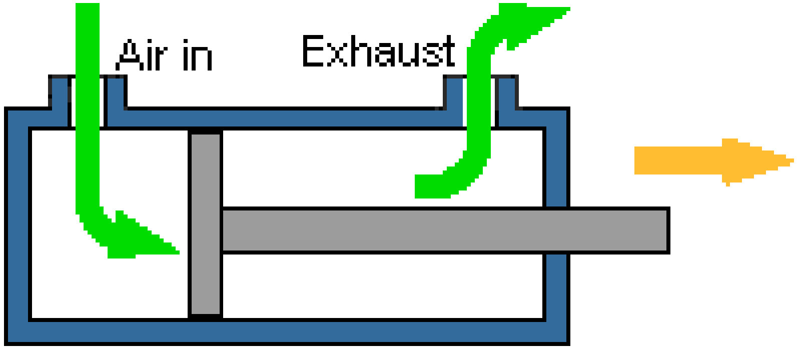


Double Acting Pneumatic Cylinder

1. **Double acting cylinder** has two air connections.
2. Compressed air is applied to one connector and the other connector is allowed to exhaust to atmosphere *(i.e. the air is allowed to escape freely)*.
3. the piston is driven to one end of the cylinder.
4. When air is then applied to the second connector and the first is allowed to exhaust to atmosphere, the piston is driven back.



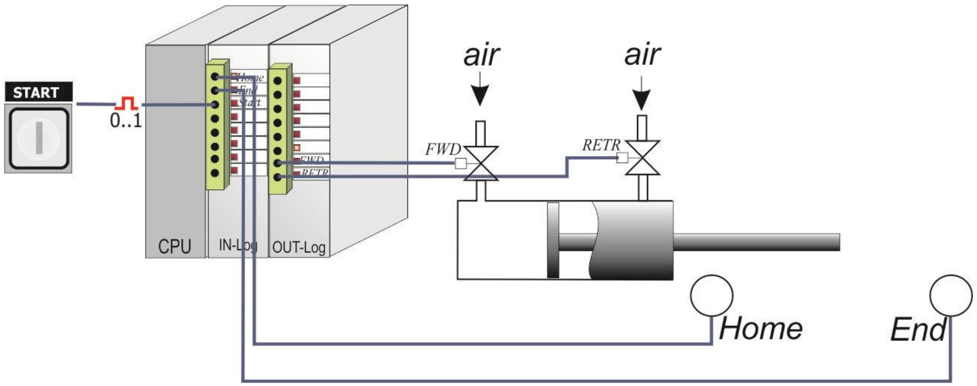




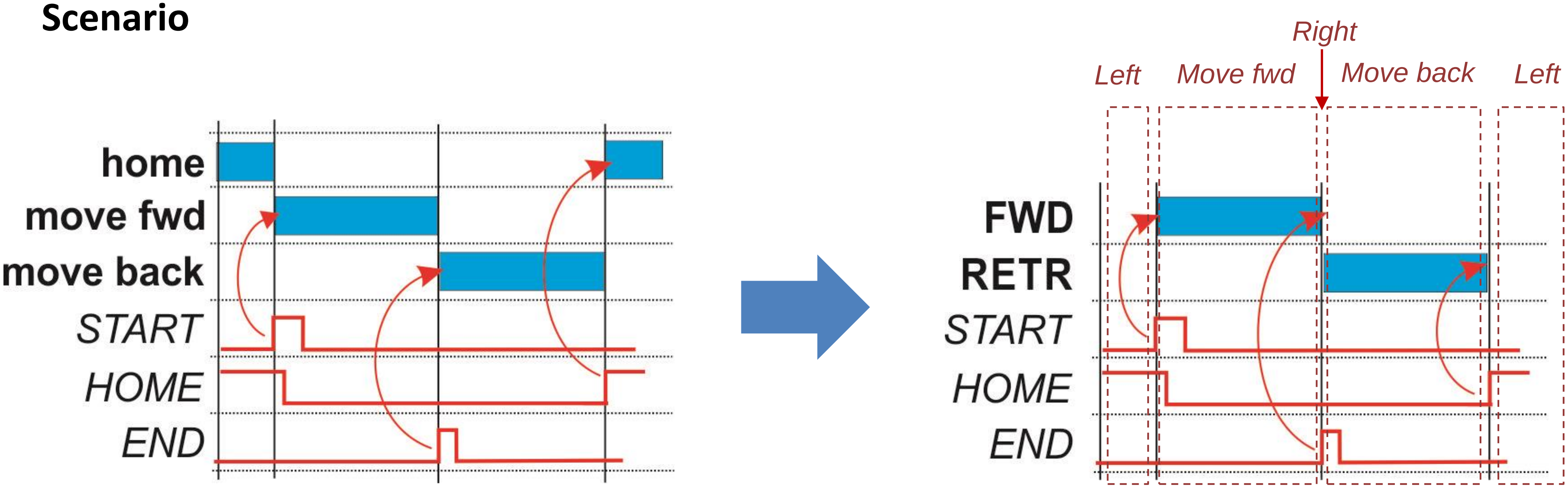
Aalto University

State-based Logic Design

Example: Double acting Pneumatic Cylinder



Scenario

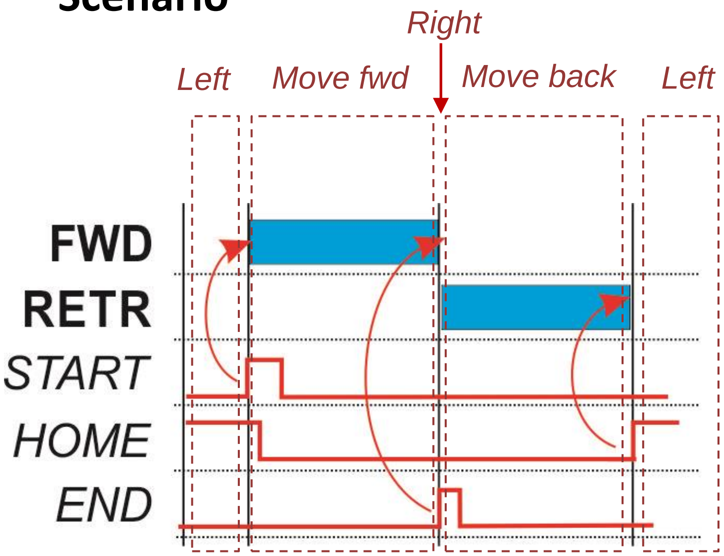


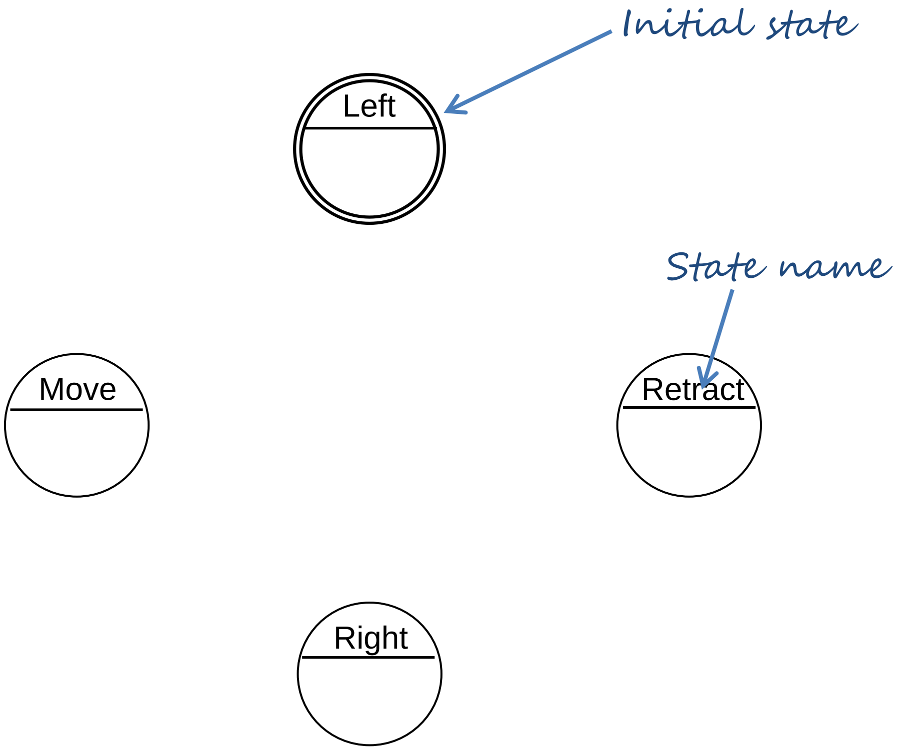
State-based Logic Design

Example: Double acting Pneumatic Cylinder

Step 1: Find stable states

Scenario



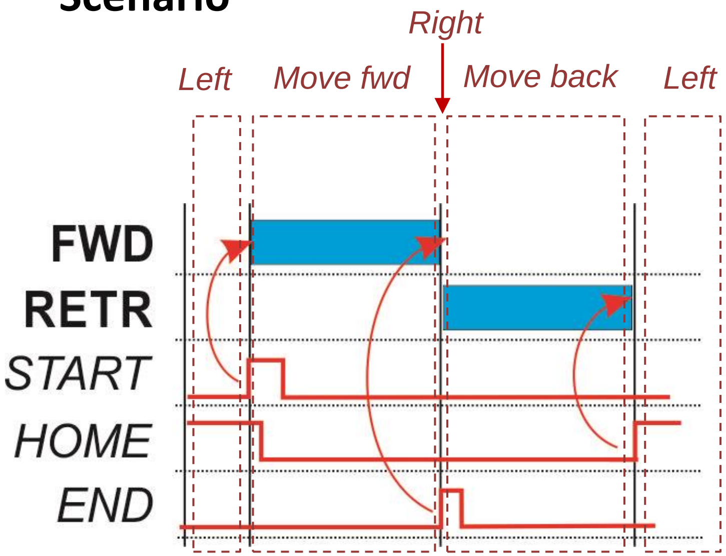


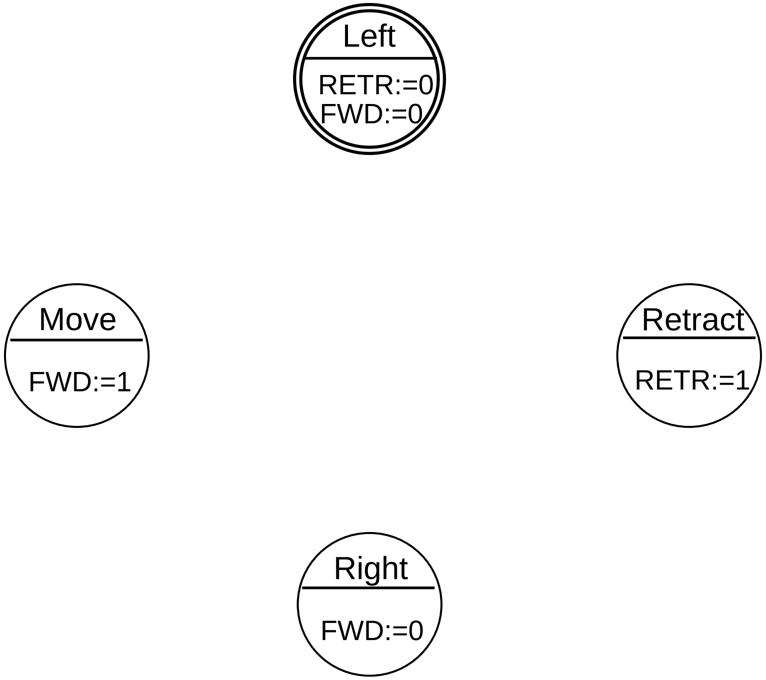
State-based Logic Design

Example: Double acting Pneumatic Cylinder

Step 2: What control signals are to be set ON in the states?

Scenario



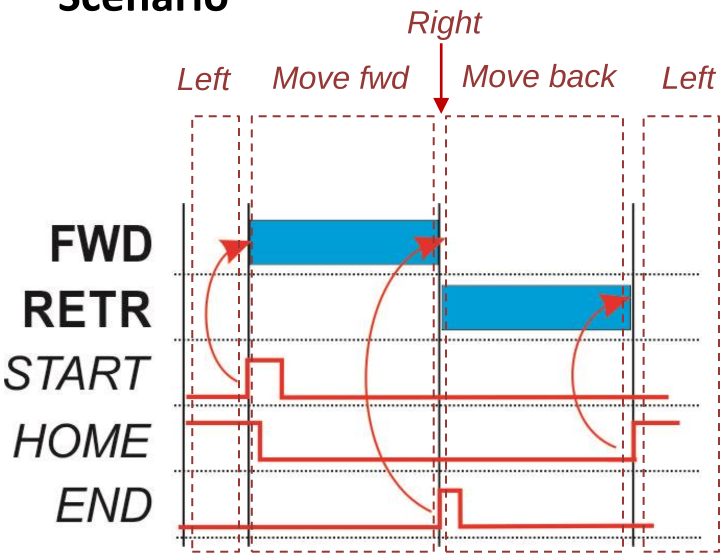


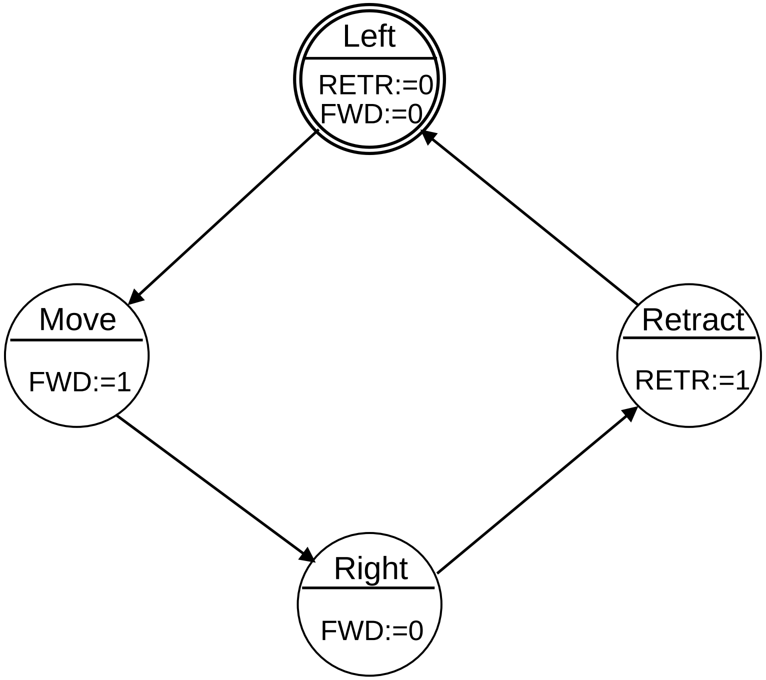
State-based Logic Design

Example: Double acting Pneumatic Cylinder

Step 3: Transitions between states

Scenario



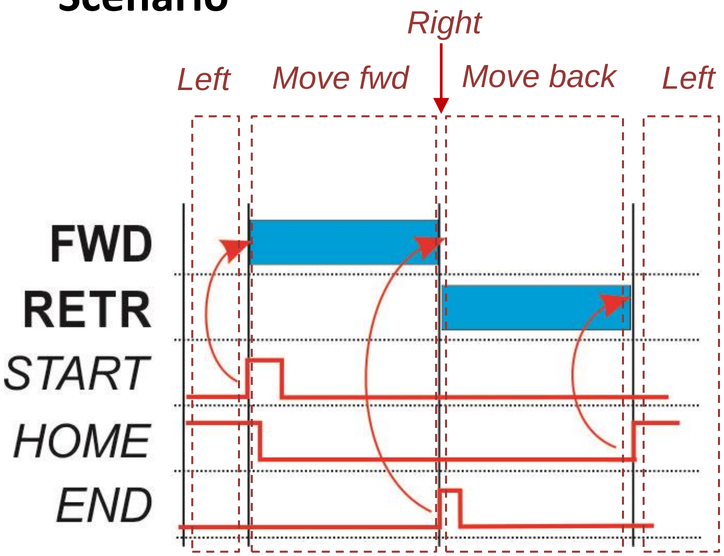


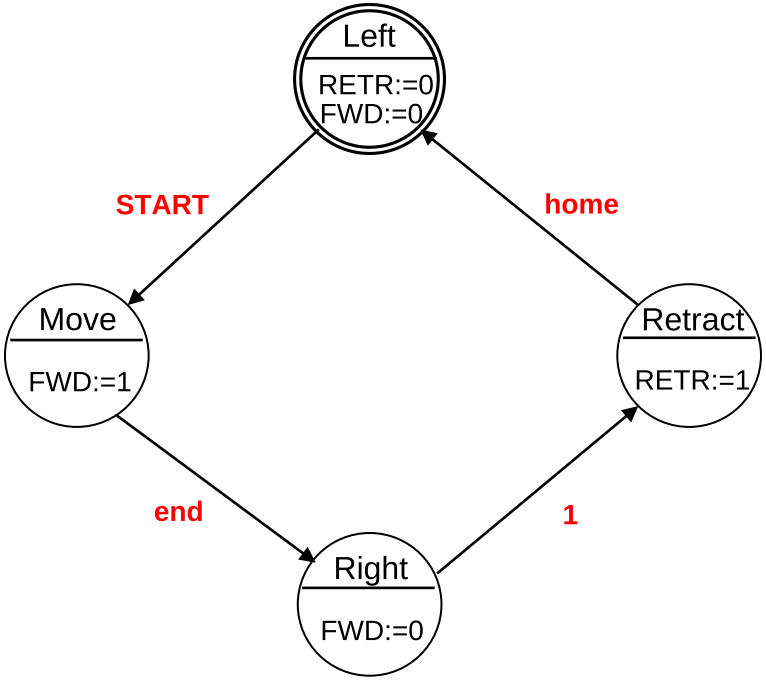
State-based Logic Design

Example: Double acting Pneumatic Cylinder

Step 4: Conditions of the transitions

Scenario

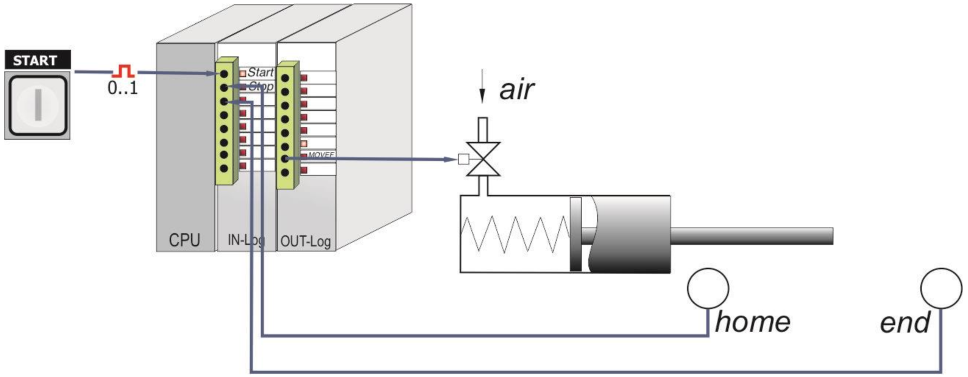


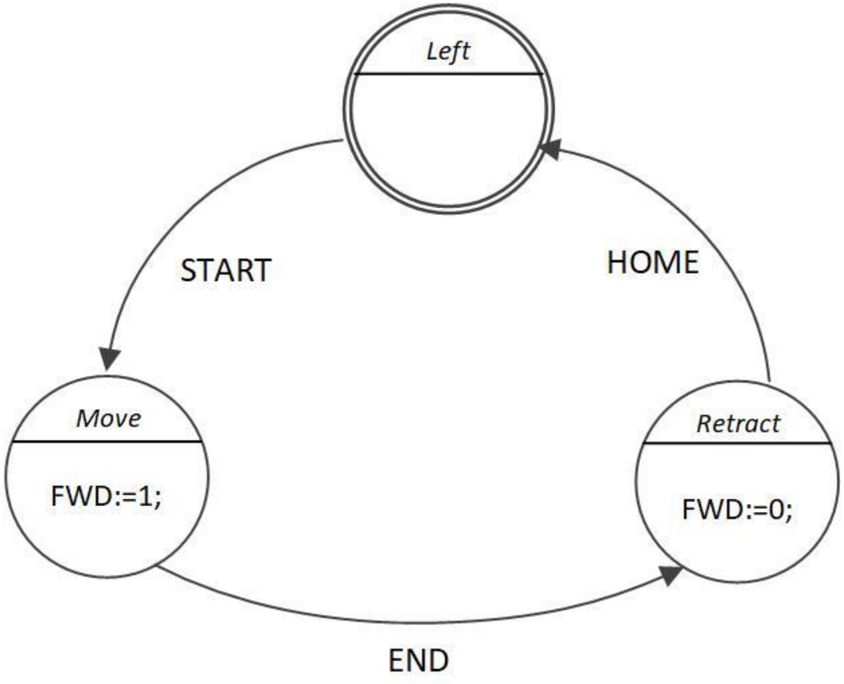


Aalto University

Finite-state machines (FSMs)

Single acting cylinder





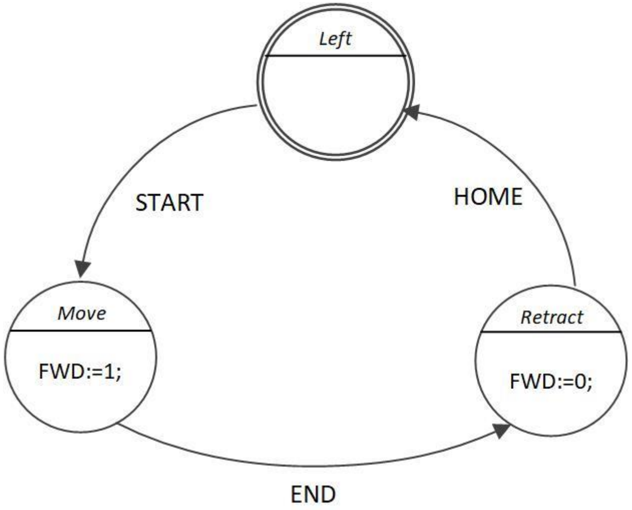
- Initial state (shown with double circle)
- The **next state** depends on the current state and current input signals
- The **outputs** either depend on the current state and input signals (Mealy machines) or on the current state only (Moore machines)
- Which kind of state machine is shown on the left?

Curiosity #1: Table implementation of FSM

State transition function T for the Cylinder

2^n for n binary inputs

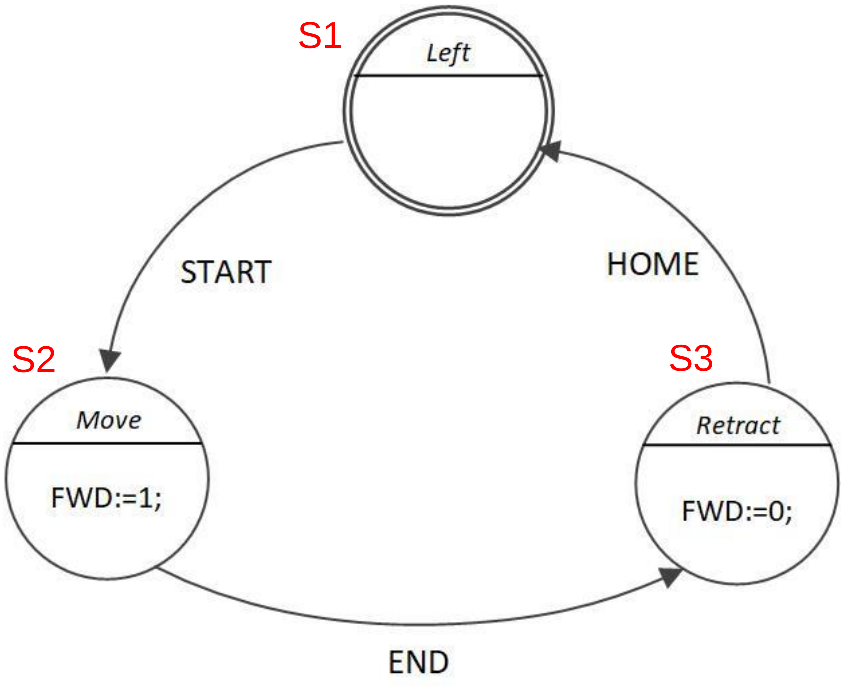
Inputs				**States**		
	START	HOME	END	S1	S2	S3
1	0	0	0	X	…	…
2	0	0	1	X		
3	0	1	0	S1		
4	0	1	1	X		
5	1	0	0	X		
6	1	0	1	X		
7	1	1	0	S2		
8	1	1	1	x		
Output					**FWD=1**	



Algorithm:
1. Fill the table and
2. Update states and outputs according to it

Curiosity #2: Boolean Functions -> Structured Text (or Ladder Diagrams)

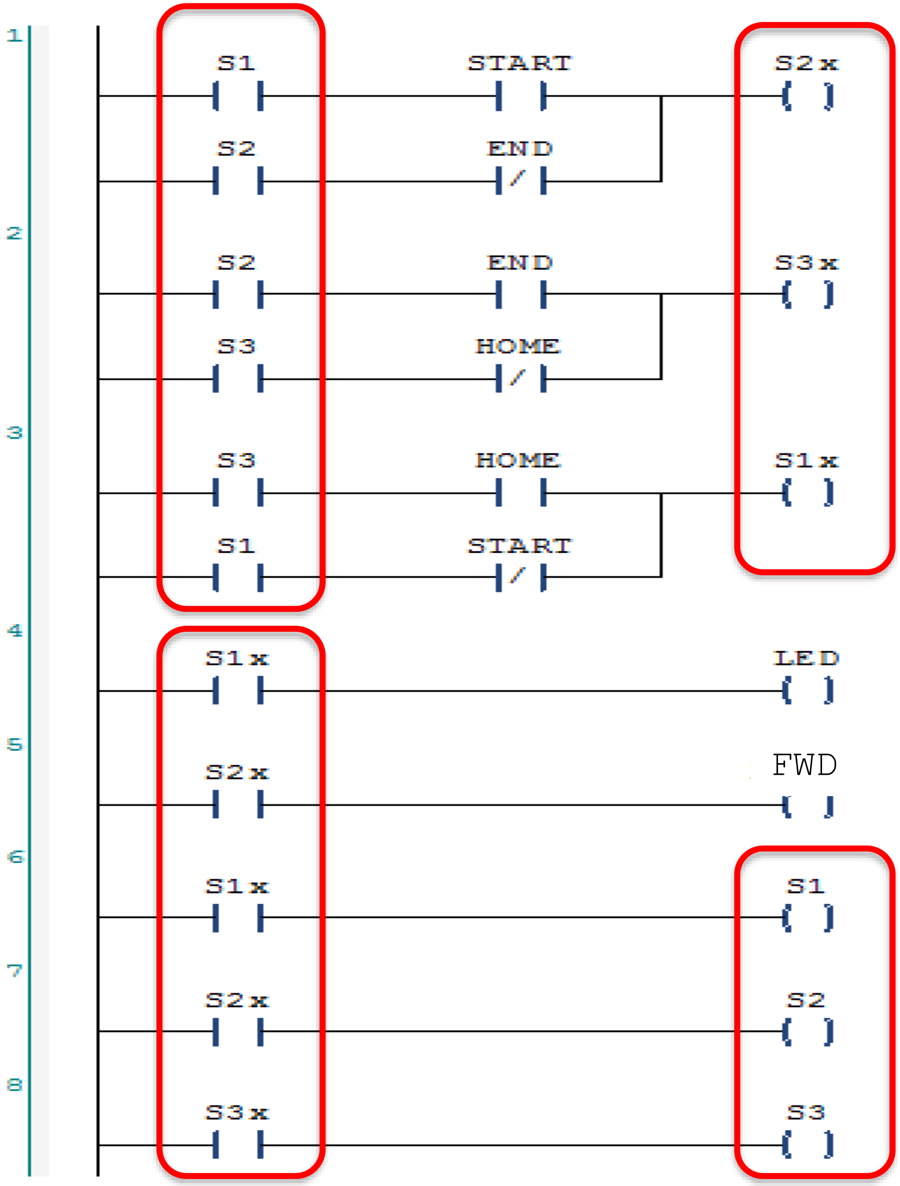
$$STATE' = STATE \cdot T_l + \sum_{j=0}^{M} STATE_j^i \cdot T_j^i$$



```
// Initialisation
IF FirstScan THEN
  S1  := 1; S2  := 0; S3  := 0;
  S1x := 1; S2x := 0; S3x := 0;
  FWD := 0;
  FirstScan := 0;
END_IF;
// State transition function
S2x := S2 AND NOT END
    OR S1 AND START;
S3x := S3 AND NOT HOME
    OR S2 AND END;
S1x := S1 AND NOT START
    OR S3 AND HOME;

// Outputs
FWD := S2;
// Next state variables
S1 := S1x; S2 := S2x; S3 := S3x;
```

Aalto University

Ladder logic representation



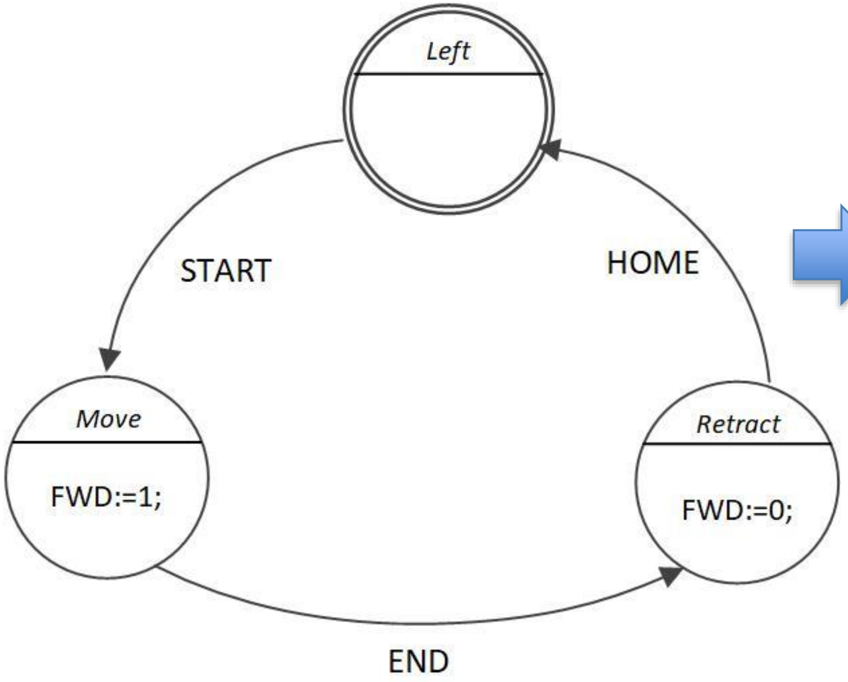
The same in Structured text:

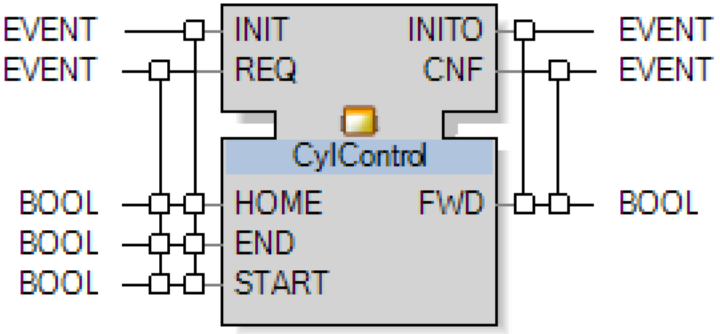
```
S2x  := S2 AND NOT END
        OR S1 AND START;
S3x  := S3 AND NOT HOME
        OR S2 AND END;
S1x  := S1 AND NOT START
        OR S3 AND HOME;
```

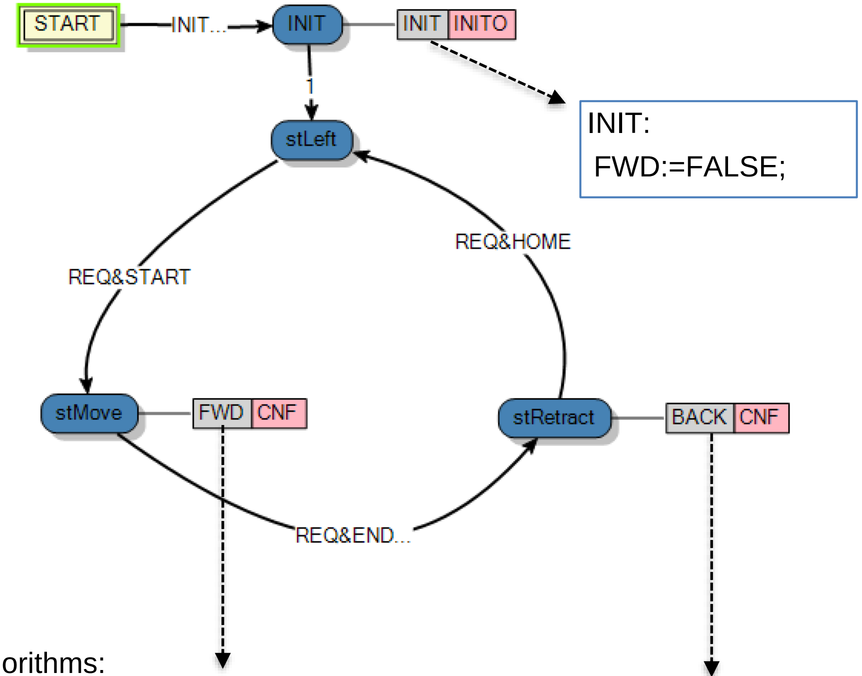
```
LED  := S1x;
FWD  := S2x;
```

```
S1  := S1x;
S2  := S2x;
S3  := S3x;
```

Implementation of Moore State Machine in Basic FB







```
INIT:
 FWD:=FALSE;
```

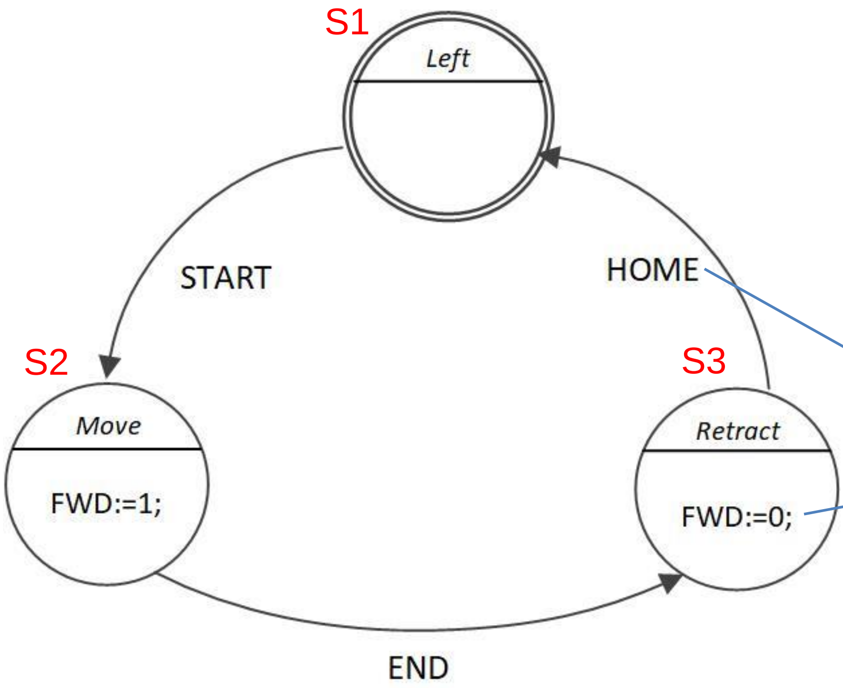
Algorithms:

```
FWD:

 FWD :=TRUE;
```

```
BACK:

FWD :=FALSE;
```

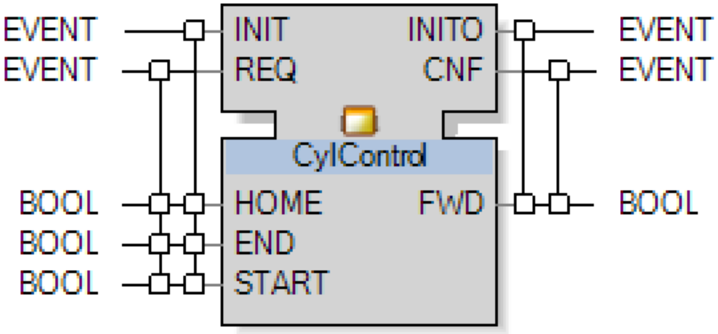
1. Added START state, where ECC is at the start up.
2. Event REQ activates all 3 transition conditions.
3. Setting of the control signal is implemented in algorithms.
4. Output events are emitted to make the control signal available outside the function block.

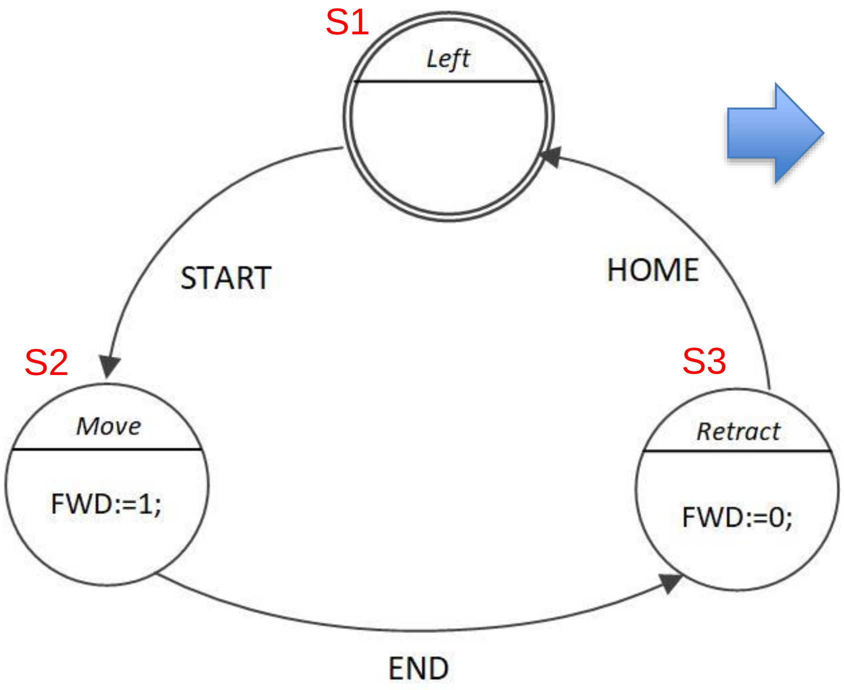
Implementation of Moore State Machine in ST language



```
IF State=1 THEN
        IF START THEN
            State:=2;
        END_IF;
ELSIF State=2 THEN
        FWD:=TRUE;
        IF END THEN
            State:=3;
        END_IF;
ELSIF State=3 THEN
        FWD:=FALSE;
        IF HOME THEN
            State:=1;
        END_IF;
END_IF;
```

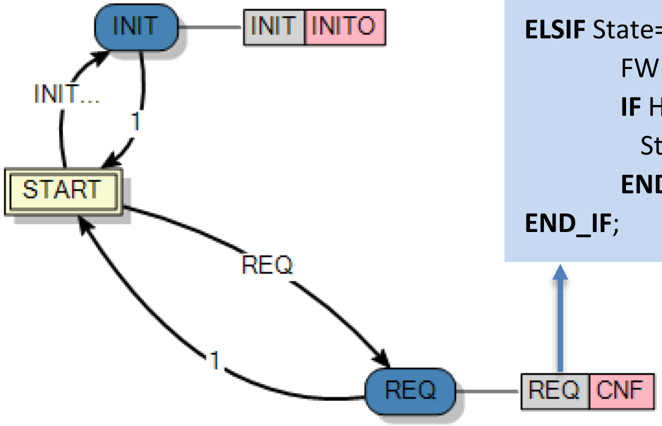
Encapsulate the ST code to function block



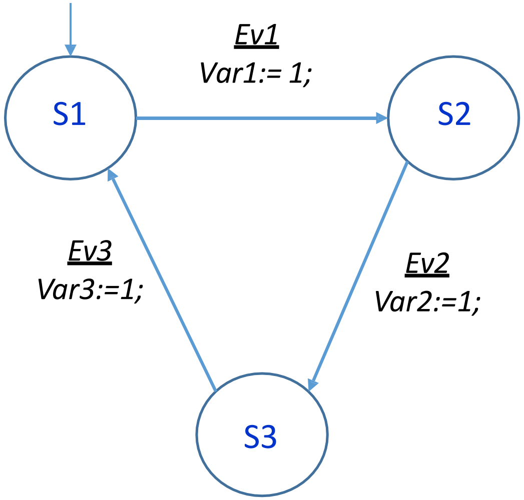


Q

```
IF State=1 THEN
        IF START THEN
          State:=2;
        END_IF;
ELSIF State=2 THEN
        FWD:=TRUE;
        IF END THEN
          State:=3;
        END_IF;
ELSIF State=3 THEN
        FWD:=FALSE;
        IF HOME THEN
          State:=1;
        END_IF;
END_IF;
```



Mealy Machine – Simple ST code example



Mealy State Machine example

Example: Mealy Machine in ST

```
//Example: Mealy Machine:
IF S1 THEN
    IF Ev1 THEN
        Var1:=1;
        S1:=0; S2:=1; S3:=0;
    END_IF
ELSIF S2 THEN
    IF Ev2 THEN
        Var2:=1;
        S1:=0; S2:=0; S3:=1;
    END_IF
ELSIF S3 THEN
    IF Ev3 THEN
        Var3:=1;
        S1:=1; S2:=0; S3:=0;
    END_IF
END_IF
```

(Here can be added possible state S1 action statements *)*

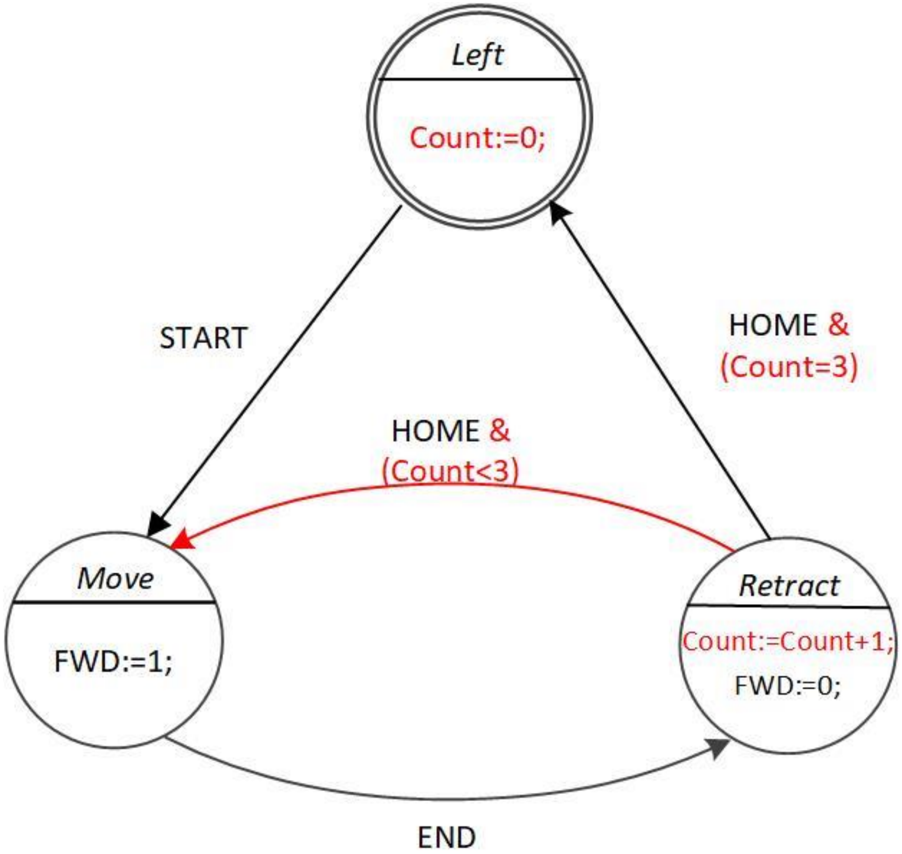
(Here can be added possible state S2 action statements *)*

(Here can be added possible state S3 action statements *)*

Extension 1: Counting

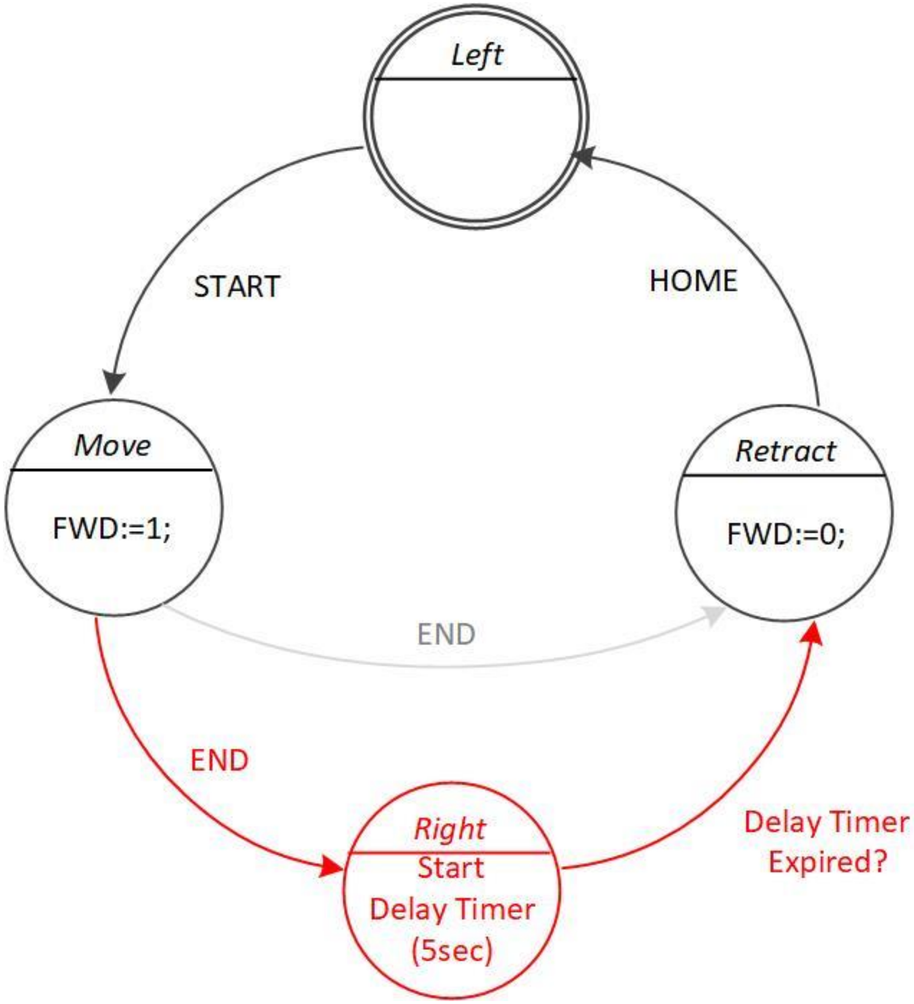
Requirement: Once the START button is pressed, the cylinder will need to shuttle back and forth 3 times.

To count the number of passes we need an integer variable (Count) which will be incremented every time cylinder reaches the “end” position. The variable will be compared with the desired number of passes (3) to decide whether to repeat or stop.
Transition arc from Retract to Move is added.



Aalto University

Extension 2: Hold cylinder for some time in the extended position



Requirement: Hold cylinder in the extended position for 5 sec.

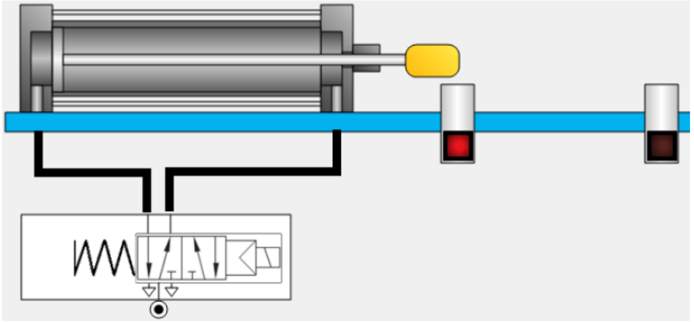
Solution: Use delay timer.

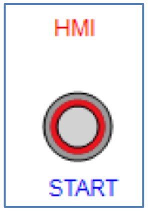
Delay timer is an object that can be:

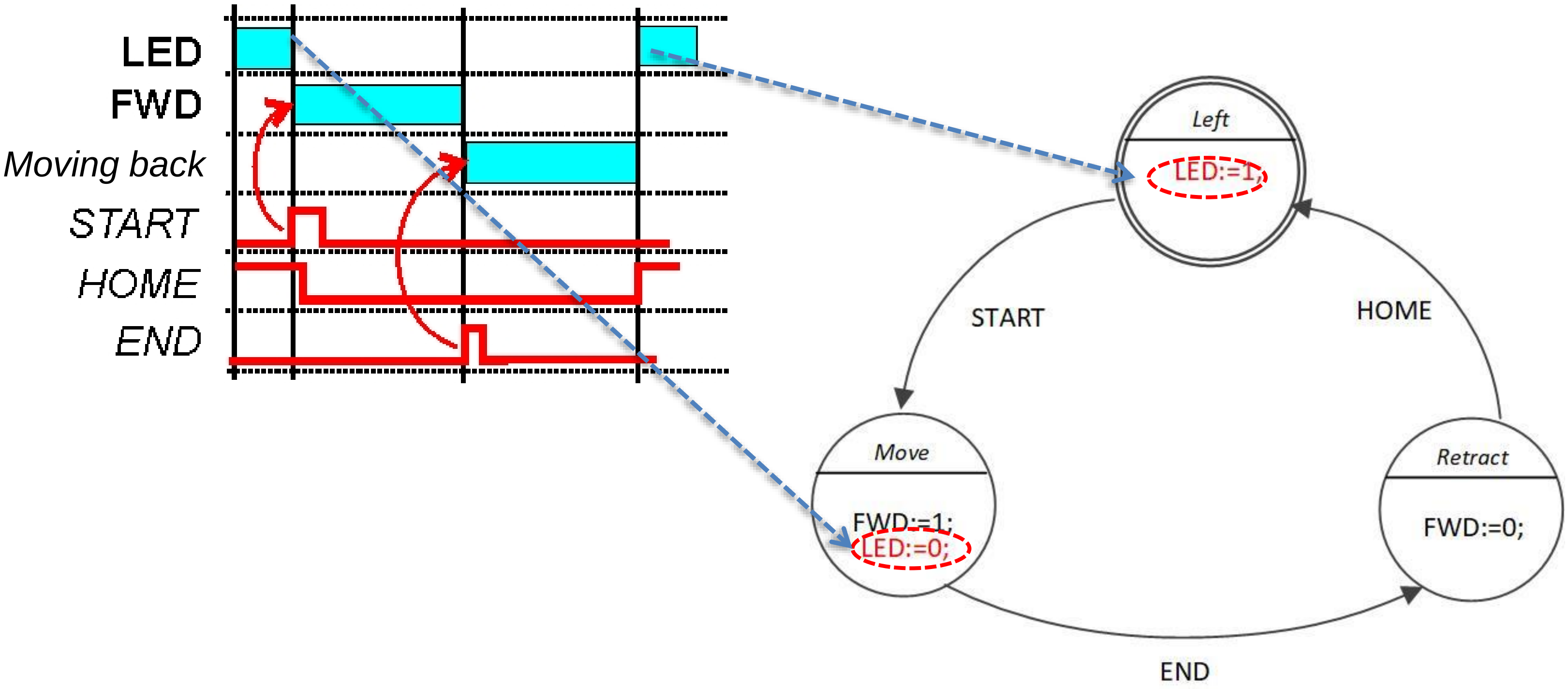
1) Started with some duration.
2) Checked if it has expired.

Extension 3: Adding HMI

HMI logic is implanted into the state machine

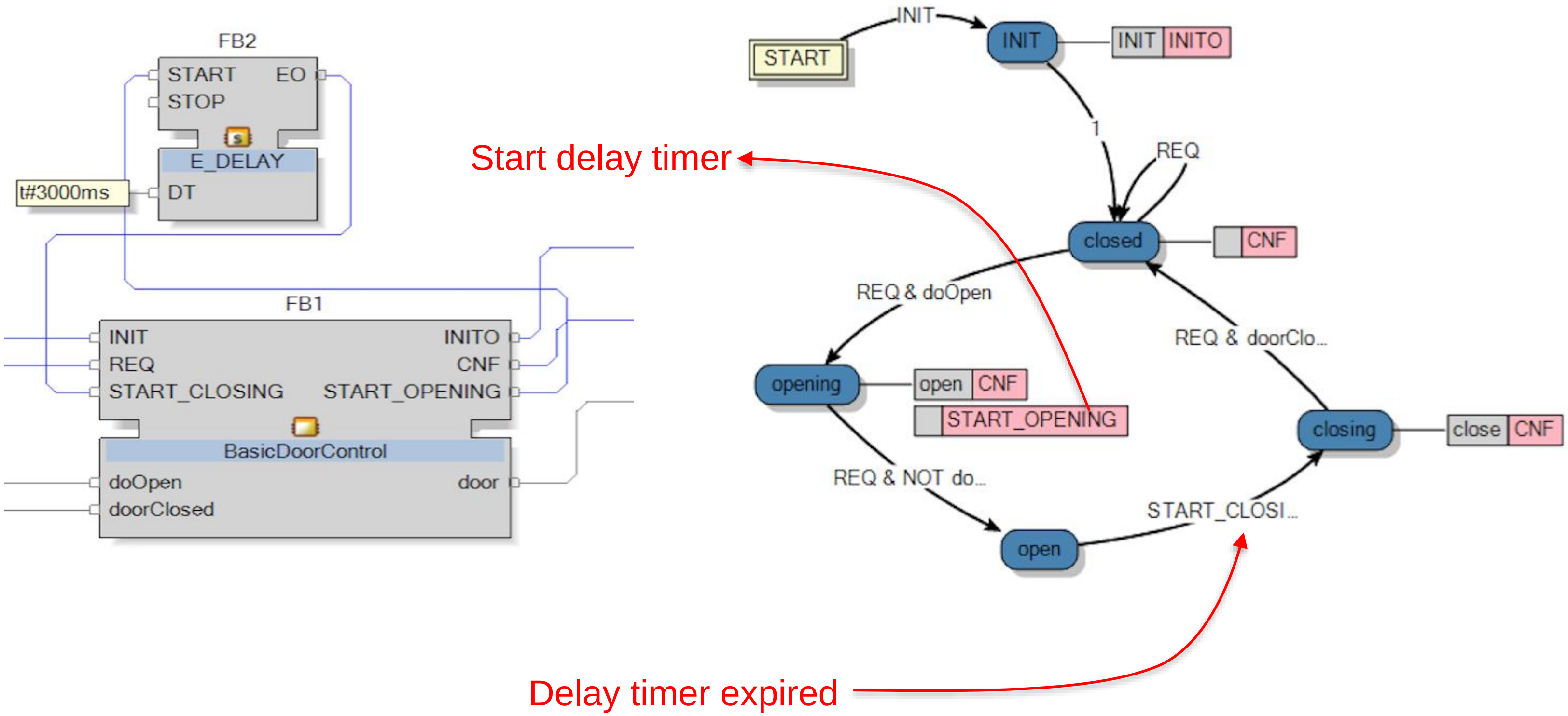






Use of timers in function blocks: delay in ECC

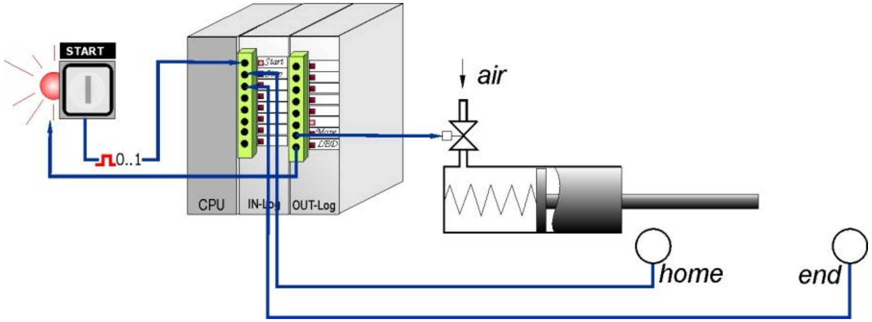
- There is no state timer in ECC.
- So one needs to use an external E_DELAY



Start delay timer

Delay timer expired

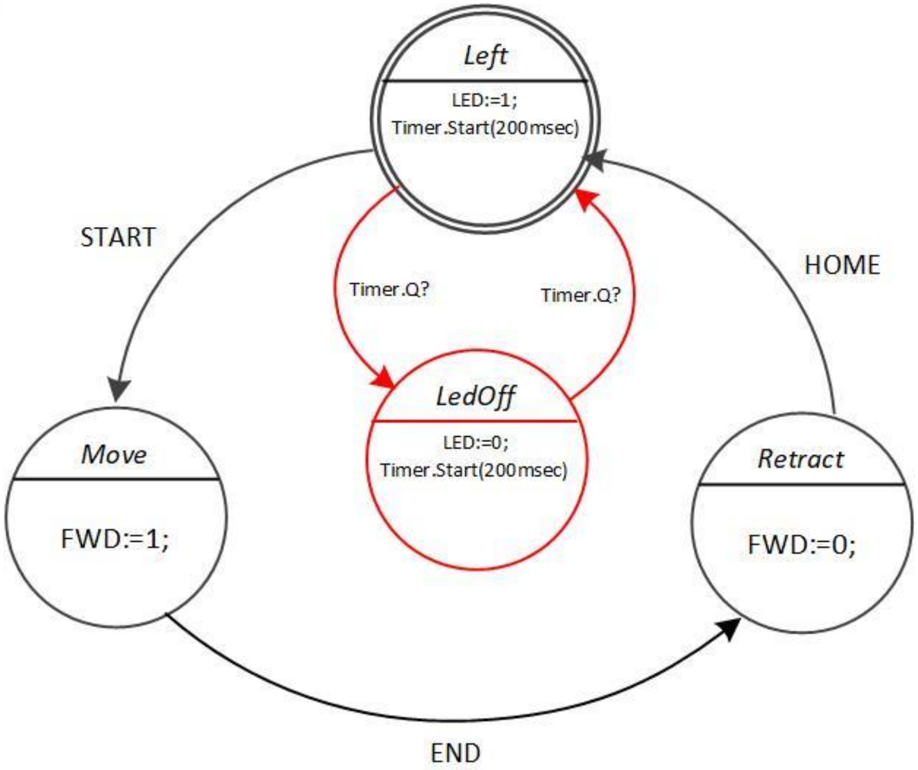
Extension 4: Blinking lamp



Requirement: There is an LED lamp under the button**.**

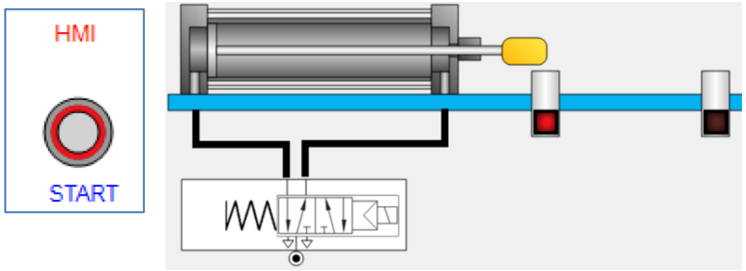
Make the LED of the START button blinking while the cylinder is in the leftmost state. Implement controller in LLD.

To implement the blinking, we need to introduce a “sister” state to S1, where the LED will be reset for some time, say 200 ms.

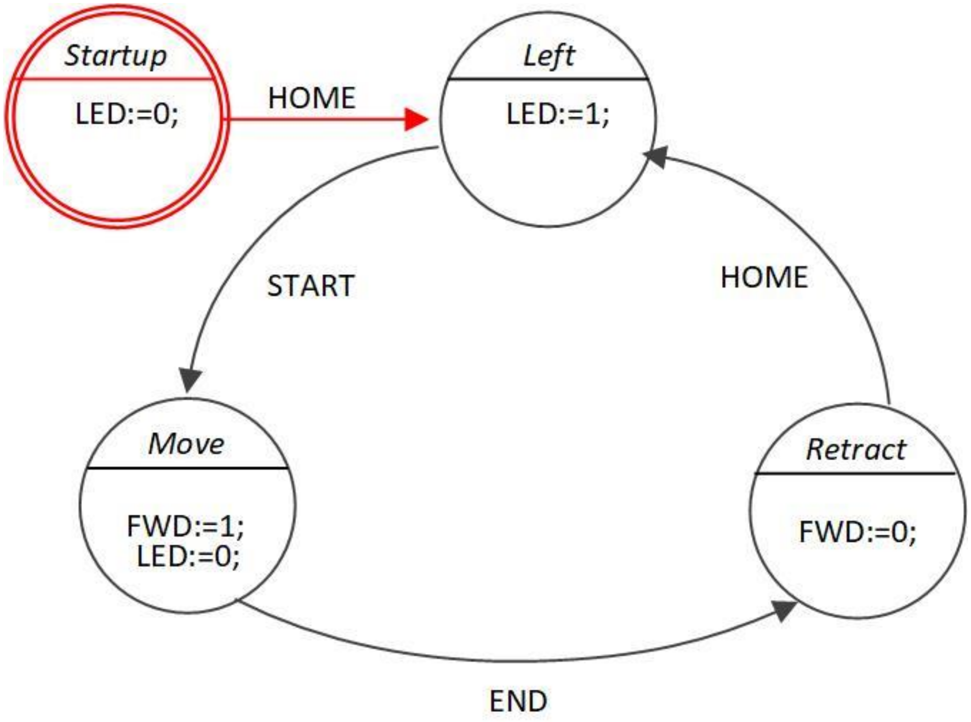


Extension 5: Initialization

Problem: At the power up the machine may be not in the initial state. The machine may require positioning to the intial state before starting the operation

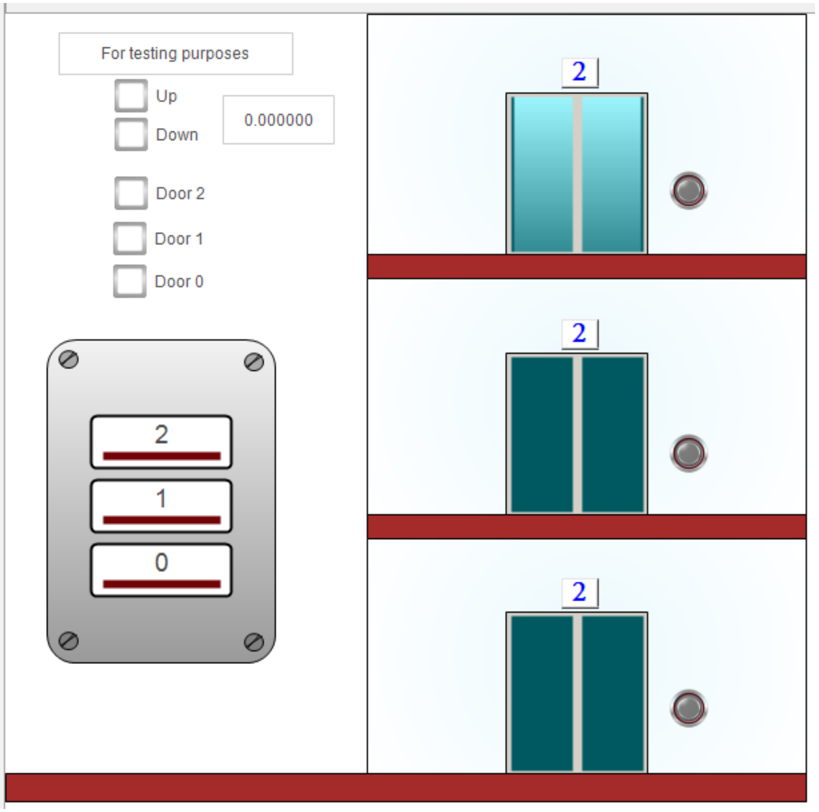


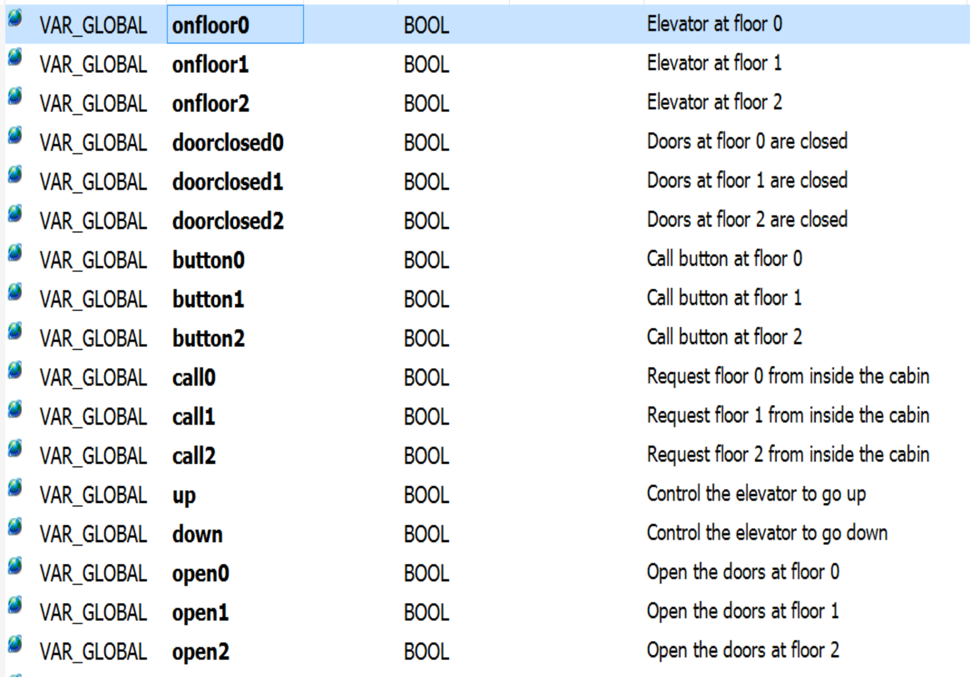
A new initial state Startup is introduced.



State-based design example: 3-Floor Elevator

- There is no weight sensor and no stop button in the elevator
- All call buttons are constantly active

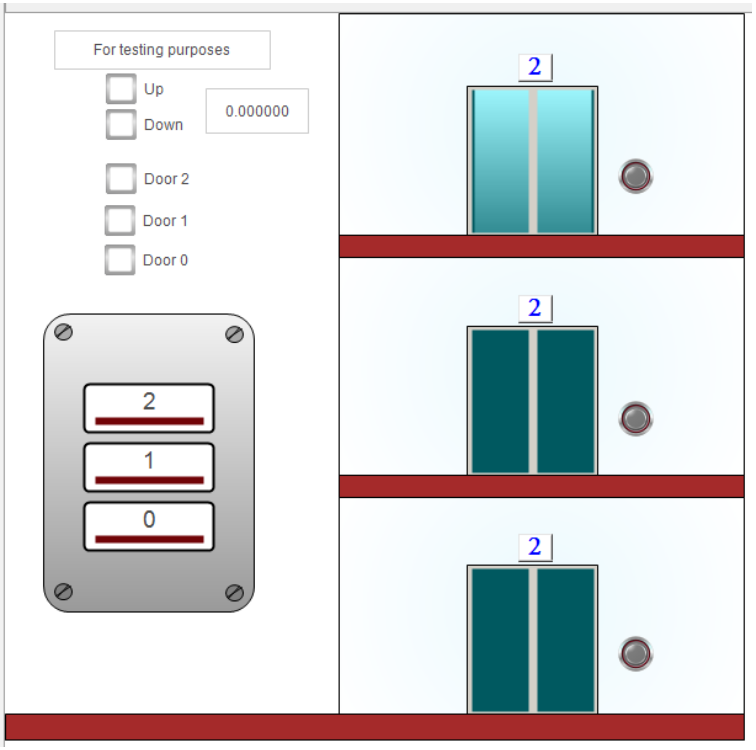


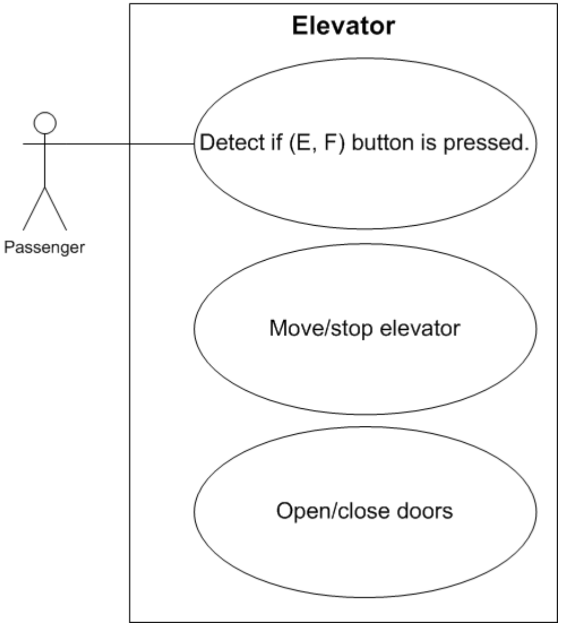


VAR_GLOBAL	**onfloor0**	BOOL	Elevator at floor 0
VAR_GLOBAL	**onfloor1**	BOOL	Elevator at floor 1
VAR_GLOBAL	**onfloor2**	BOOL	Elevator at floor 2
VAR_GLOBAL	**doorclosed0**	BOOL	Doors at floor 0 are closed
VAR_GLOBAL	**doorclosed1**	BOOL	Doors at floor 1 are closed
VAR_GLOBAL	**doorclosed2**	BOOL	Doors at floor 2 are closed
VAR_GLOBAL	**button0**	BOOL	Call button at floor 0
VAR_GLOBAL	**button1**	BOOL	Call button at floor 1
VAR_GLOBAL	**button2**	BOOL	Call button at floor 2
VAR_GLOBAL	**call0**	BOOL	Request floor 0 from inside the cabin
VAR_GLOBAL	**call1**	BOOL	Request floor 1 from inside the cabin
VAR_GLOBAL	**call2**	BOOL	Request floor 2 from inside the cabin
VAR_GLOBAL	**up**	BOOL	Control the elevator to go up
VAR_GLOBAL	**down**	BOOL	Control the elevator to go down
VAR_GLOBAL	**open0**	BOOL	Open the doors at floor 0
VAR_GLOBAL	**open1**	BOOL	Open the doors at floor 1
VAR_GLOBAL	**open2**	BOOL	Open the doors at floor 2

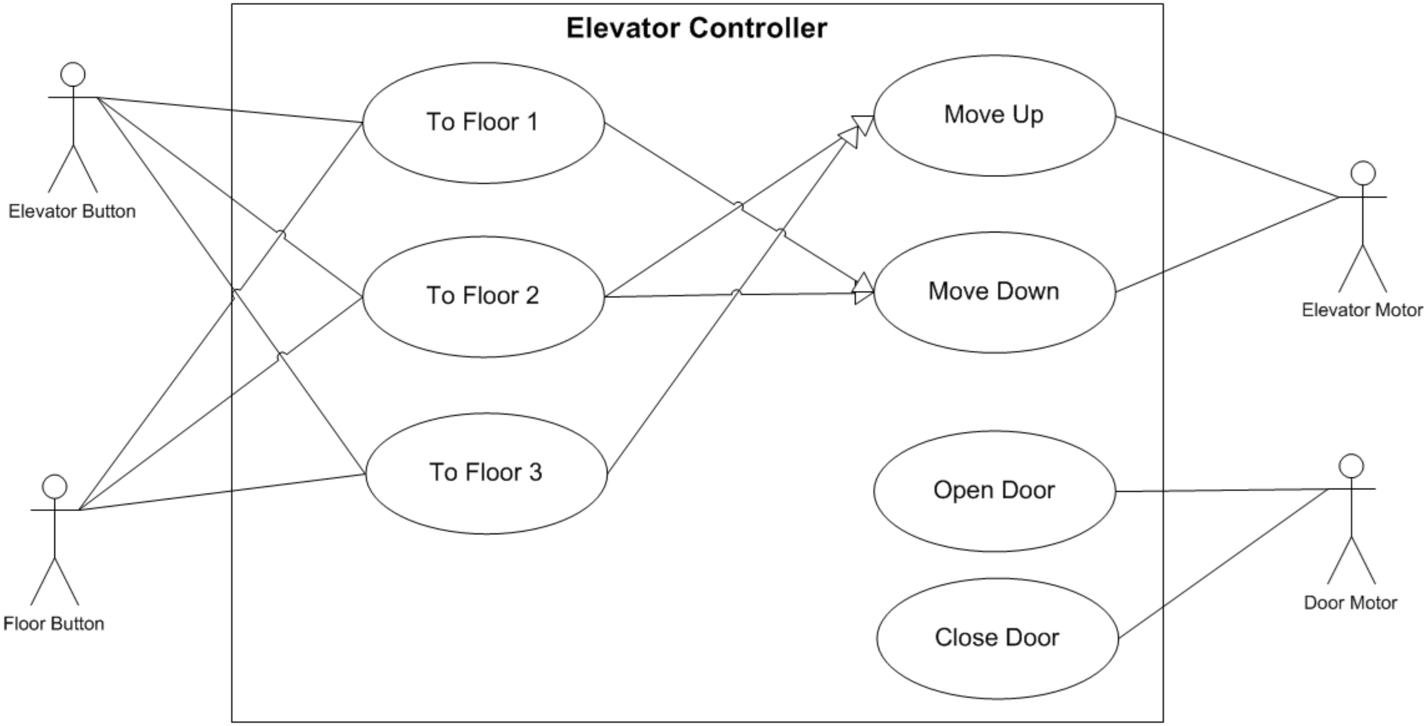
A Use Case Diagram of Elevator

A use case diagram at its simplest is a representation of a user's interaction with a system.

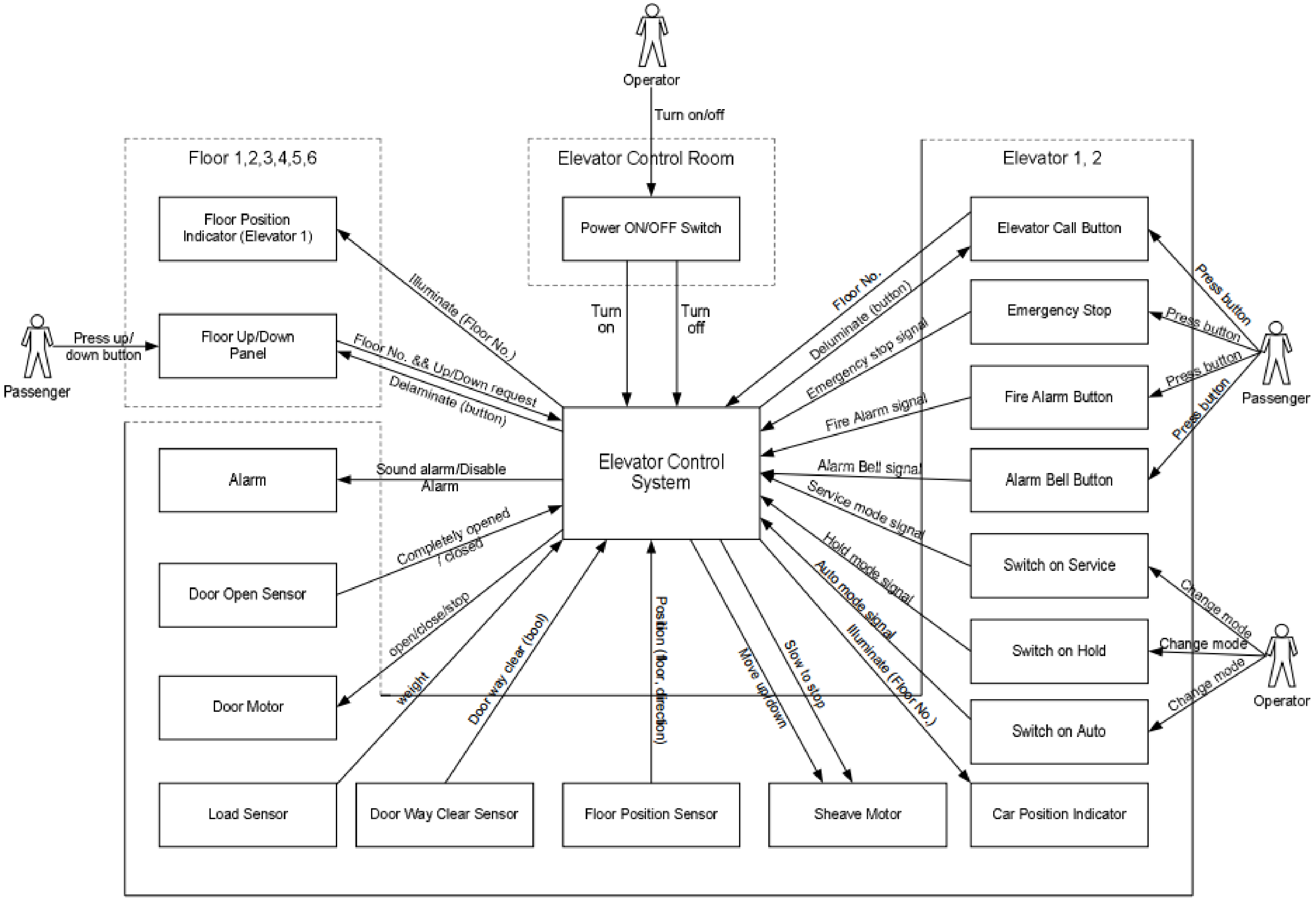




A More Detailed Use Case Diagram



A real-life elevator

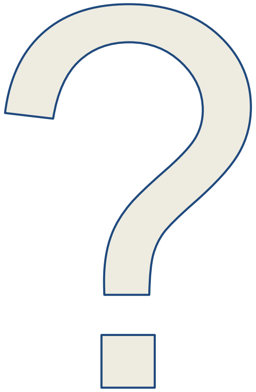


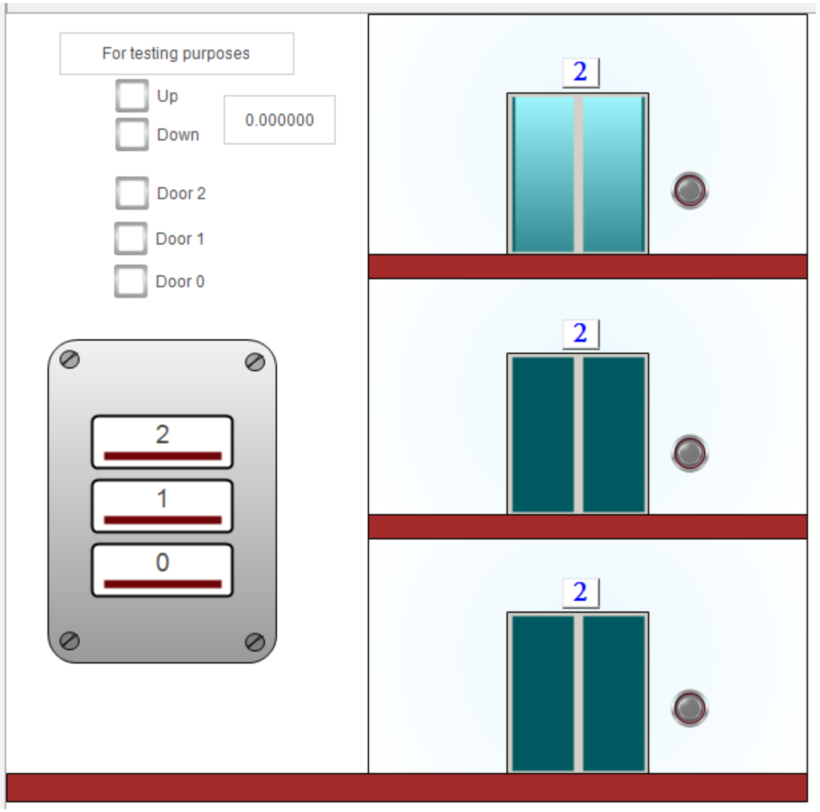
State-based controller design: algorithm

1. Determine modular decomposition.
2. For each module
   a. Identify stable states in the system’s behavior.
   b. Define for each state output signals that shall be true in this state.
   c. Define transition conditions from state to state.

Example: Modular State-based Controller for Elevator

Modular decomposition





For testing purposes

☐ Up

☐ Down

0.000000

☐ Door 2

☐ Door 1

☐ Door 0

Disclaimer

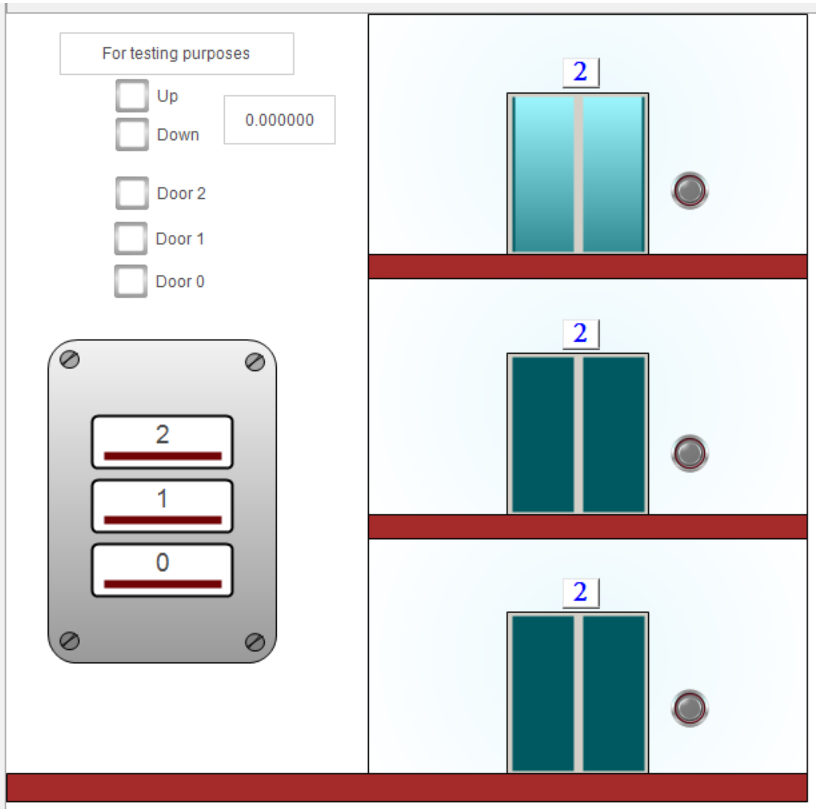
The elevator example is used in this course for illustrative purposes of hierarchical state-machine design.

In more detail and with hands on it will be investigated in the master course DIAS ELEC 8102

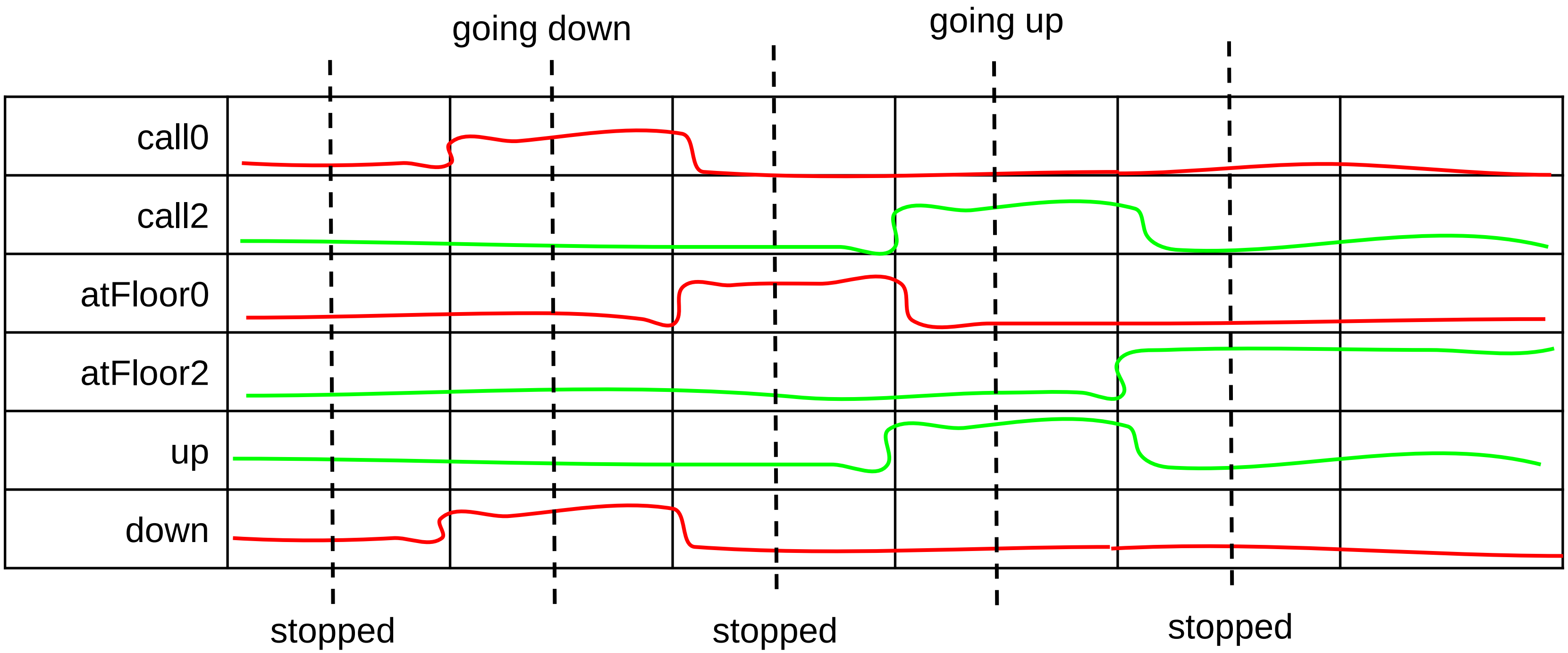
Example: Modular State-based Controller for Elevator

Modular decomposition

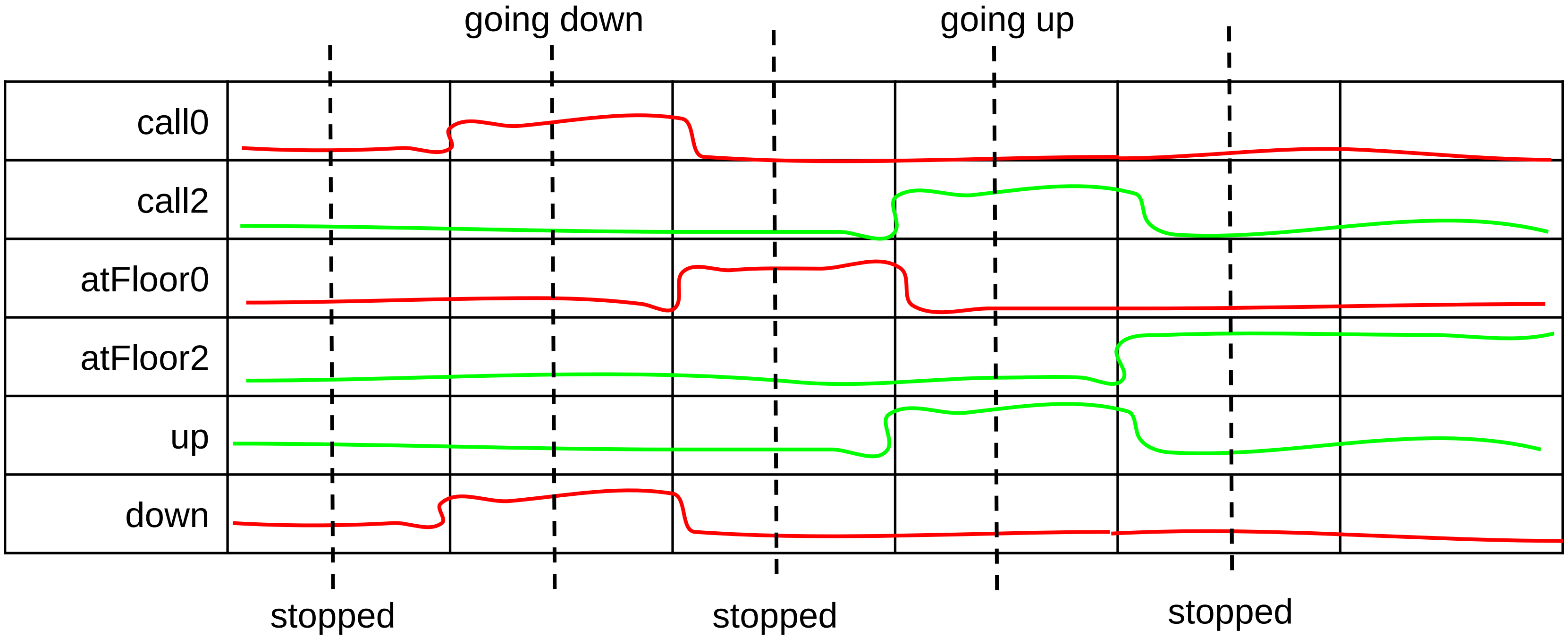
1. **FSM #1: Moving between floors**
    - control the elevator motor
2. **FSM #2: Opening/closing doors**
    - control the door motor



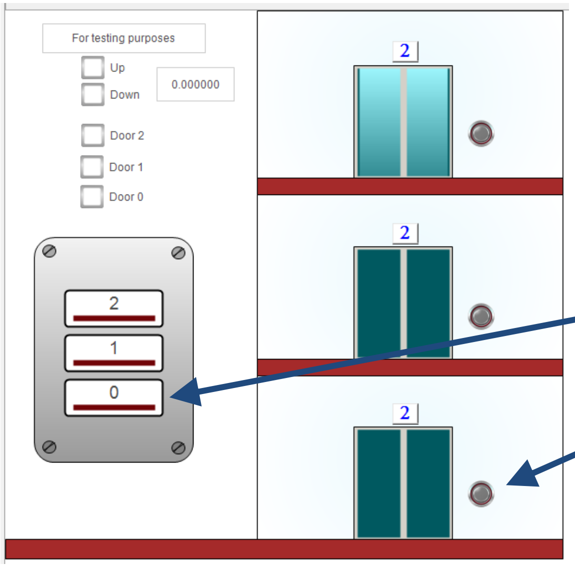
Moving between floors: determine states



Moving between floors: determine actions



Moving between floors: add transitions



Input variable	Description
onfloor(0, 1, 2)	Elevator is on floor 0/1/2
doorclosed(0, 1, 2)	Doors on floor 0/1/2 are closed
button(0, 1, 2)	button 0/1/2 is pressed
call(0, 1, 2)	call button on floor 0/1/2 is pressed

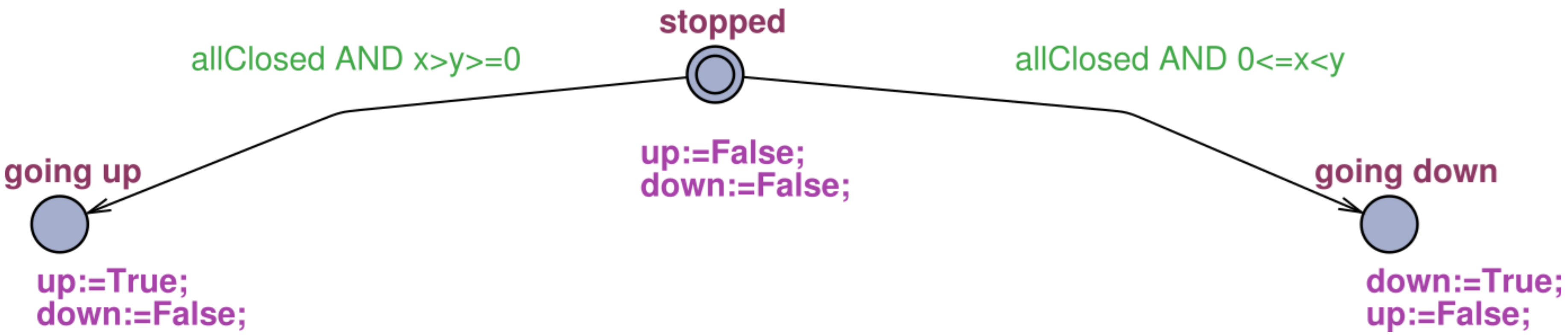
Moving between floors: add transitions

allClosed: BOOL	all doors closed	x: int	where elevator is called	y: int	current floor

```
allClosed:=doorclosed0
        AND doorclosed1
        AND doorclosed2;
```

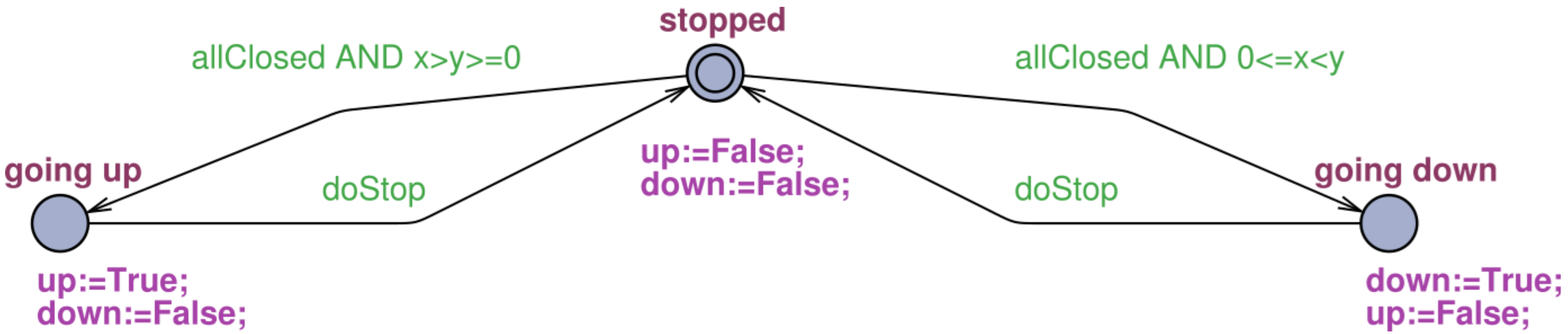
```
IF call0 OR button0 THEN
    x:=0;
ELSIF call1 OR button1 THEN
    x:=1;
ELSIF call2 OR button2 THEN
    x:=2;
ELSE
    x:=-2;
END_IF;
```

```
IF onfloor0 THEN
    y:=0;
ELSIF onfloor1 THEN
    y:=1;
ELSIF onfloor2 THEN
    y:=2;
ELSE
    y:=-2;
END_IF;
```



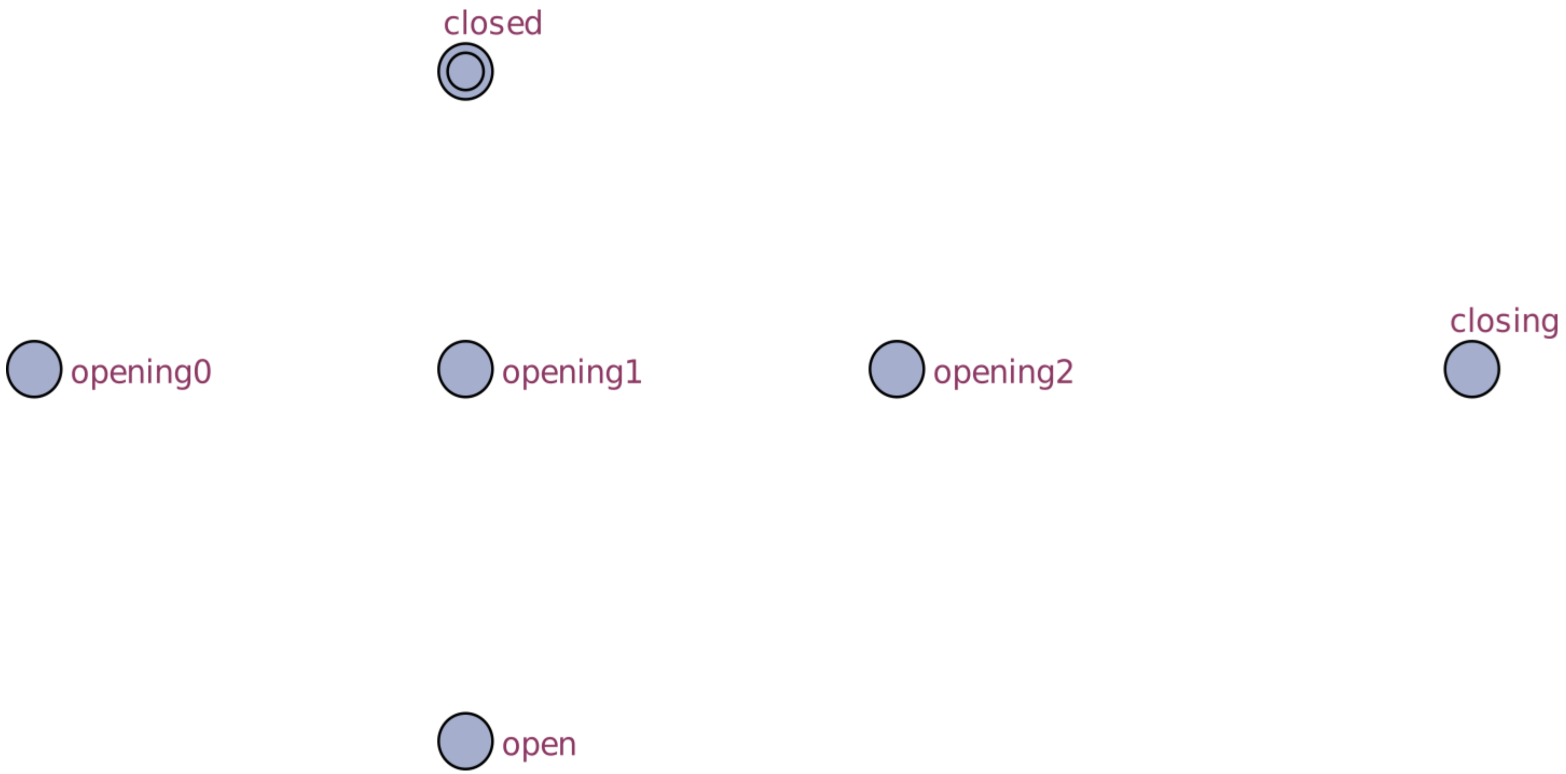
Moving between floors: add stopping conditions

```
doStop:=(y=0 AND (call0 OR button0))
        OR (y=1 AND (call1 OR button1))
        OR (y=2 AND (call2 OR button2));
```

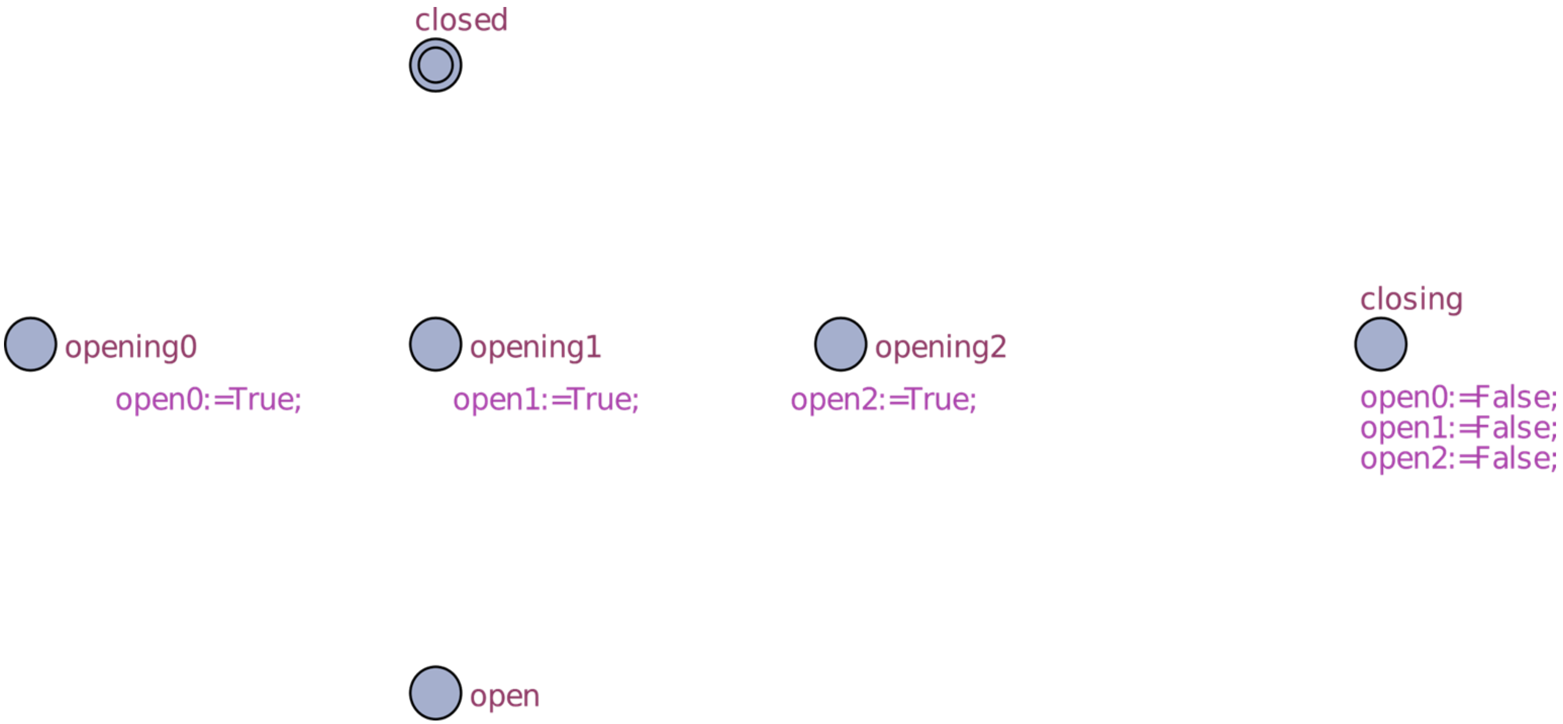


Done!

Opening/closing doors: states

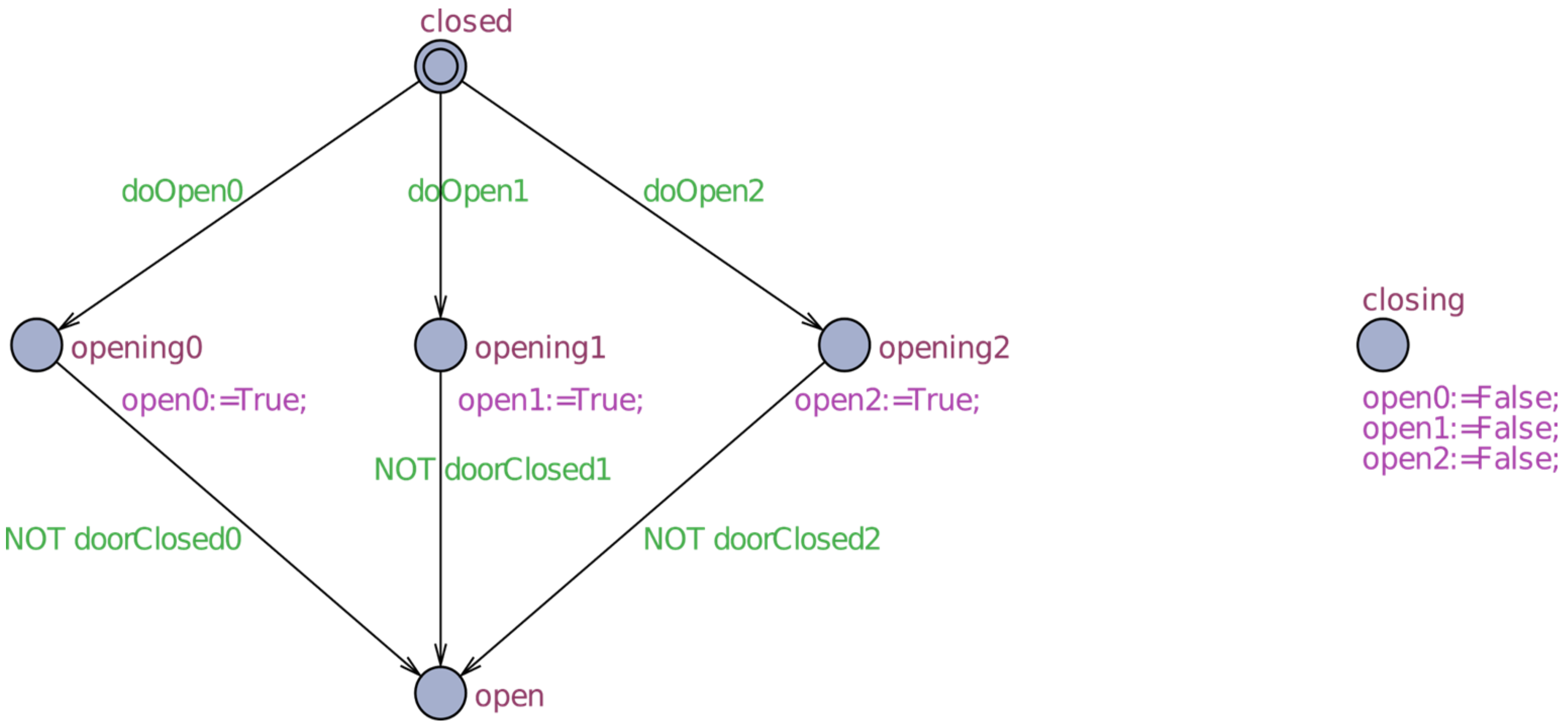


Opening/closing doors: actions



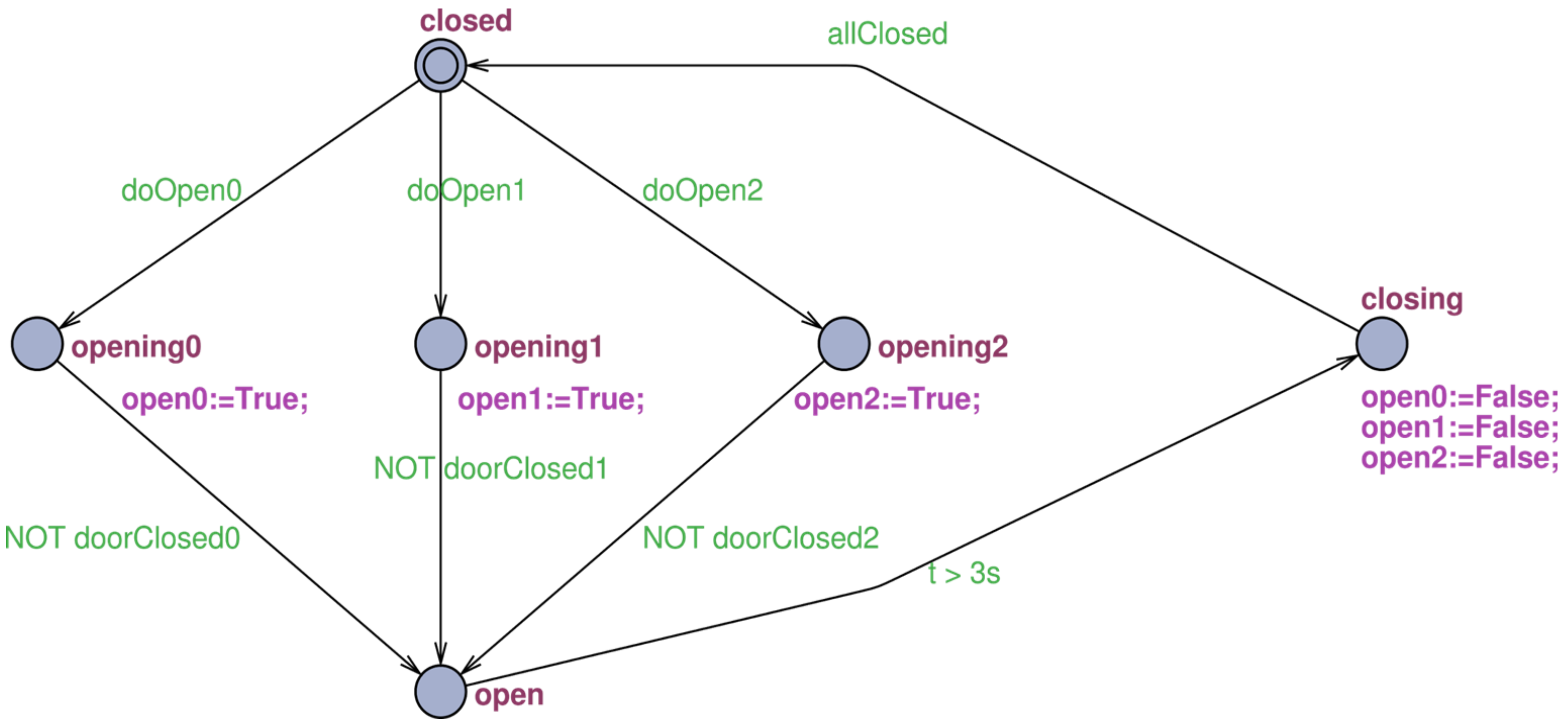
Opening doors

```
doOpen0:=(y=0 AND (call0 OR button0));
```

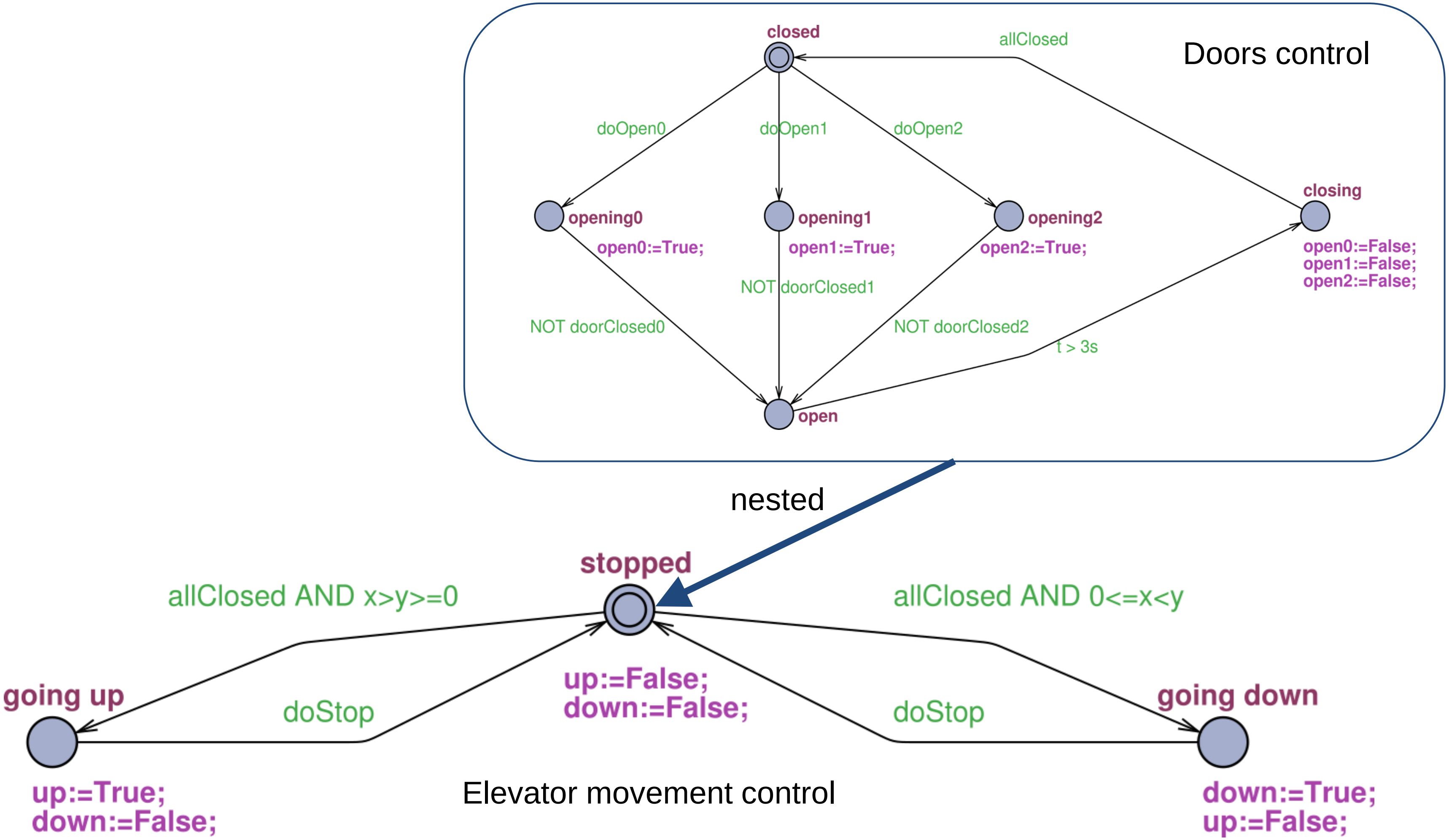


Closing doors

```
allClosed:=doorclosed0
        AND doorclosed1
        AND doorclosed2;
```



Final controller



Advantages of state-based controllers

- Clear step-by-step development process
  1. Determine modular decomposition
  2. Determine states
  3. Define actions
  4. Define transitions
- Self-annotated, easy-to-understand code
- First design, then just implement: minimal amount of debugging required

Limitations of State-based Design Approach

- Concurrent processes
  - need to decompose to smaller subsystems
  - how to handle their interaction and coordination?
- Sometimes it is difficult to define stable states

Summary

- State machine design reduces the effort of converting informal requirements written in natural language to the fully formal executable code.
- State-based design can be converted to code in many ways, e.g.
  - Look up table
  - Via Boolean logic
  - As IF-THEN-ELSE of a high-level programming language
  - In a graphical language of a similar structure
- Each way has different complexity of design, computation and life-time code maintenance
- State-based design can be extended with various enhancements, such as timing delays, arithmetic operations, etc.
- There are limitations of state-based design and some workarounds