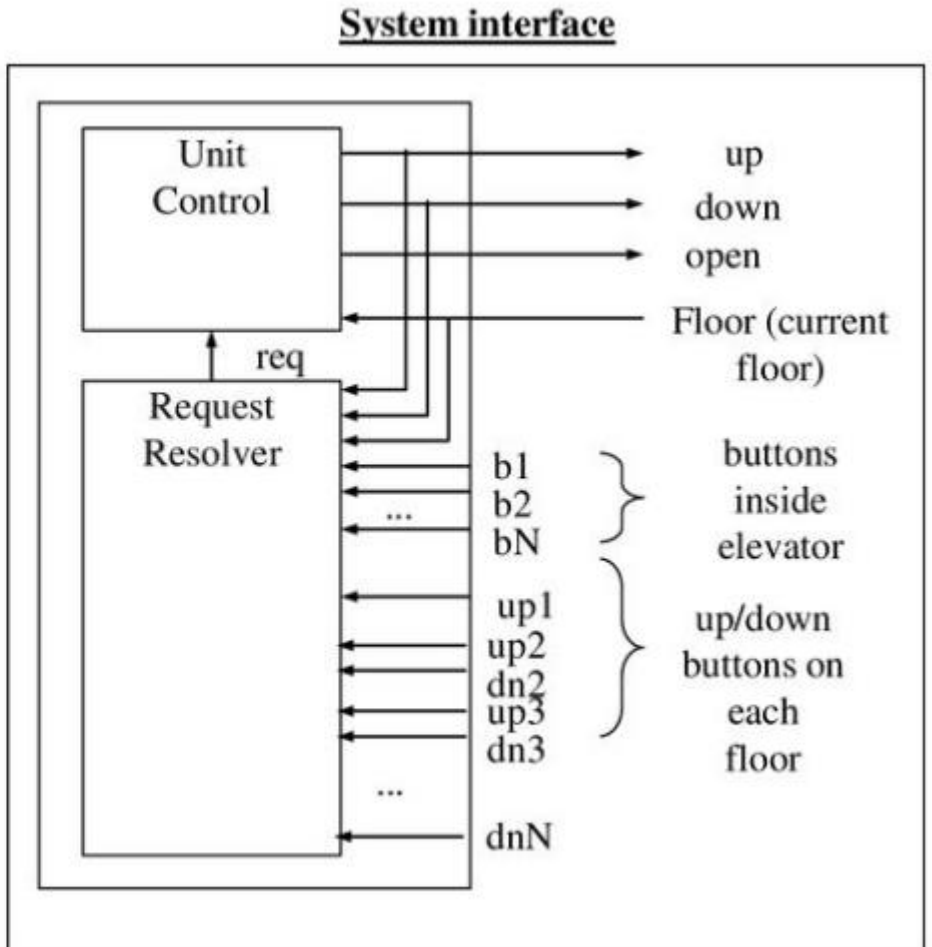
# CSE 211: Introduction to Embedded Systems

Section 8

# Example: Elevator controller

- Simple elevator controller
- Two major blocks:
  - *Request Resolver* resolves various floor requests into single requested floor
  - *Unit Control* moves elevator to this requested floor
- Simple English description….

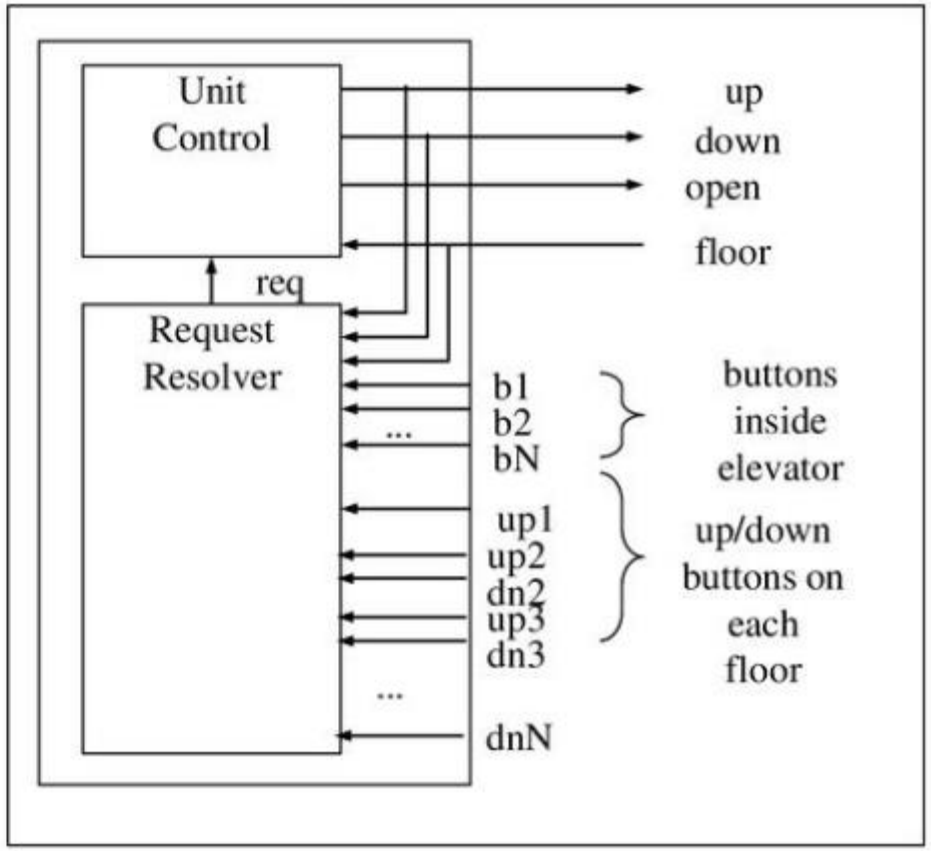


# Example: Elevator controller

**Partial English description**

"Move the elevator either up or down to reach the requested floor. Once at the requested floor, open the door for at least 10 seconds, and keep it open until the requested floor changes. Ensure the door is never open while moving. Don't change directions unless there are no higher requests when moving up or no lower requests when moving down…"

**System interface**



# Elevator controller using sequential programming

```
Inputs: int floor;  bit b1..bN;   up1..upN-1;    dn2..dnN;
Outputs: bit up, down, open;
Global variables: int req;

void Unit_Control ( )
 {
 up = down = 0; open = 1; //Initial condition
 while (1)    {        // Infinite / super loop
   while (req != floor)
   open = 0; //close the door
   if (req > floor) { up = 1 ; }
   else {   down = 1;   }
   while (req = = floor); // Same floor
   up = down = 0;
   open = 1; // Keep the door open
   delay(10) ; // Wait for 10 seconds
 }              }
```

```
void Request_Resolver ( )
{
  while (1)
  ...
    req = ...
  ...
}

void main ( )
{
  Call concurrently:
    Unit_Control( ) and
    Request_Resolver( )
}
```

# Finite State Machine (FSM)

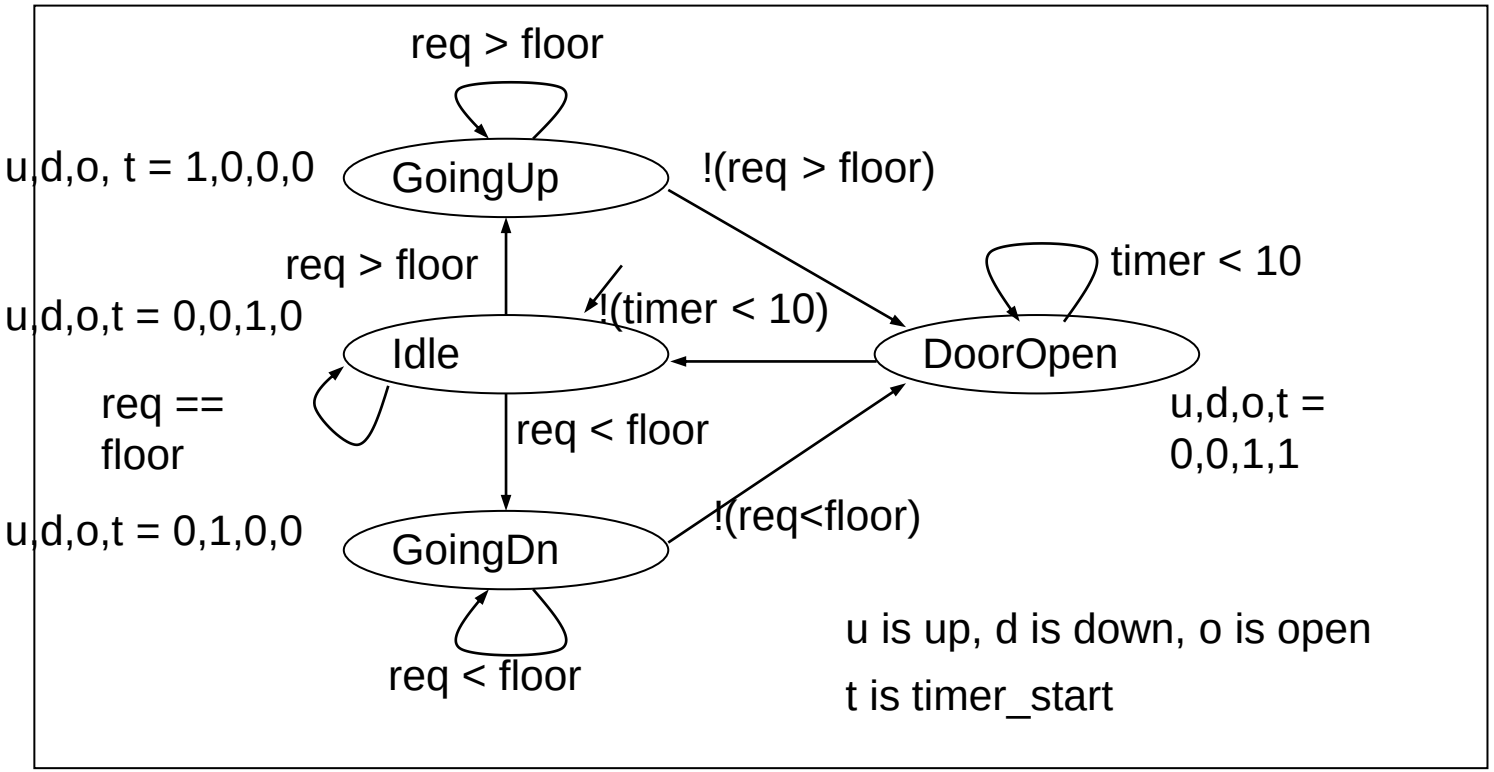
- In a finite state machine (FSM) model we describe the system behaviour as a set of possible states
- The system can only be in one of these states at a given time
- It describes the possible state transitions from one state to another depending on input values
- It also describes the actions that occur when in a state or when transitioning between states

# Finite State Machine (FSM)

- Trying to capture this behavior as sequential program is a bit awkward
- Instead, we might consider an FSM model, describing the system as:
  - Possible states
    - E.g., *Idle*, *GoingUp*, *GoingDn*, *DoorOpen*
  - Possible transitions from one state to another based on input
    - E.g., req > floor
  - Actions that occur in each state
    - E.g., In the *GoingUp* state, u,d,o,t = 1,0,0,0 (up = 1, down, open, and timer_start = 0)
- Try it...

# Finite-state machine (FSM) model

***UnitControl* process using a state machine**



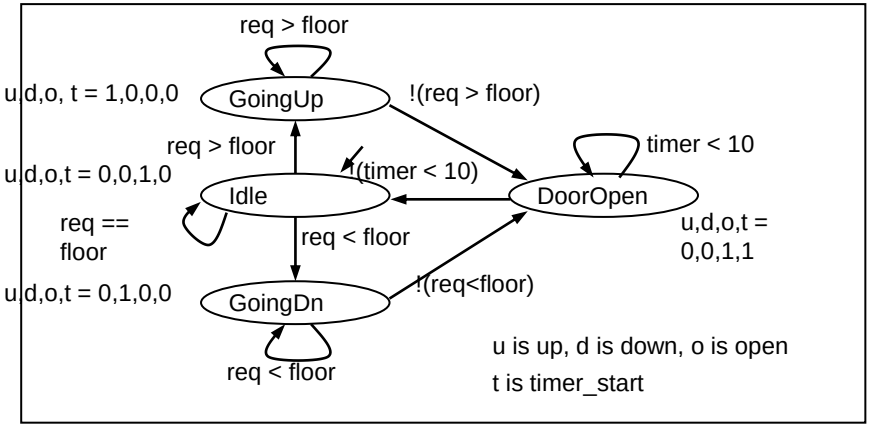
# Formal definition

- An FSM is a 6-tuple $F<S, I, O, F, H, s_0>$
  - $S$ is a set of all states $\{s_0, s_1, \ldots, s_l\}$
  - $I$ is a set of inputs $\{i_0, i_1, \ldots, i_m\}$
  - $O$ is a set of outputs $\{o_0, o_1, \ldots, o_n\}$
  - $F$ is a next-state function ($S$ x $I \rightarrow S$)
  - $H$ is an output function ($S \rightarrow O$)
  - $s_0$ is an initial state
- Moore-type
  - Associates outputs with states (as given above, $H$ maps $S \rightarrow O$)
- Mealy-type
  - Associates outputs with transitions ($H$ maps $S$ x $I \rightarrow O$)
- Shorthand notations to simplify descriptions
  - Implicitly assign 0 to all unassigned outputs in a state
  - Implicitly AND every transition condition with clock edge (FSM is synchronous)

# Finite-state machine with datapath model (FSMD)

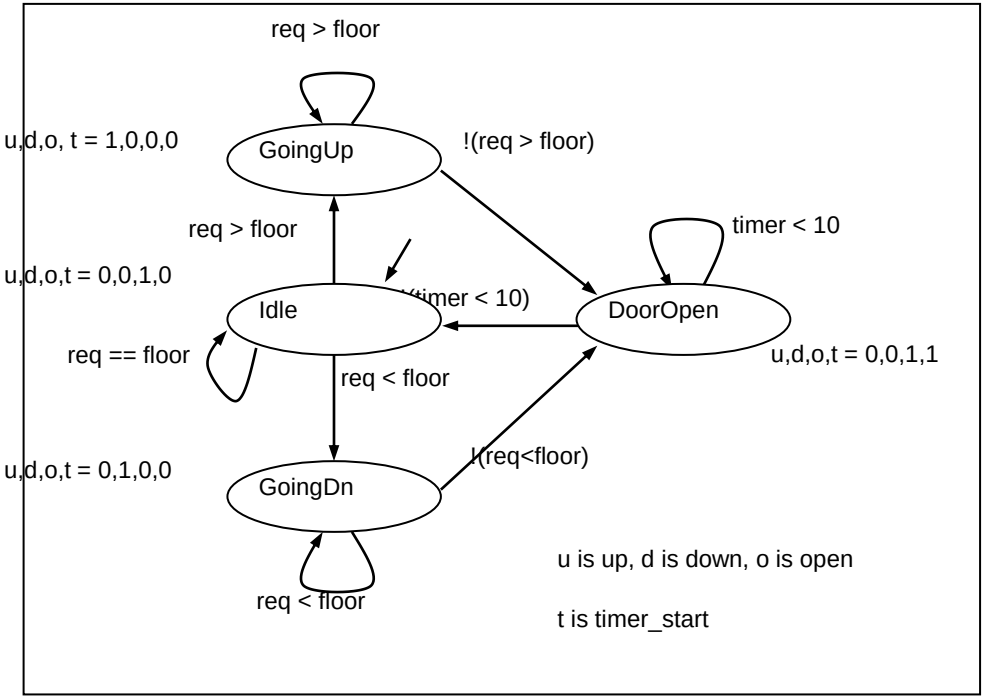
- FSMD extends FSM: complex data types and variables for storing data
  - FSMs use only Boolean data types and operations, no variables
- FSMD: 7-tuple $<S, I, O, \underline{V}, F, H, s_0>$
  - $S$ is a set of states $\{s_0, s_1, \ldots, s_l\}$
  - $I$ is a set of inputs $\{i_0, i_1, \ldots, i_m\}$
  - $O$ is a set of outputs $\{o_0, o_1, \ldots, o_n\}$
  - **$\underline{V}$ is a set of variables $\{\underline{v_0}, \underline{v_1}, \ldots, \underline{v_n}\}$**
  - $F$ is a next-state function ($S$ x $I$ x $V \rightarrow S$)
  - $H$ is an **action** function ($S \rightarrow O + V$)
  - $s_0$ is an initial state
- $I, O, V$ may represent complex data types (i.e., integers, floating point, etc.)
- $F, H$ may include arithmetic operations
- $H$ is an action function, not just an output function
  - Describes variable updates as well as outputs
- Complete system state now consists of current state, $s_i$, and values of all variables

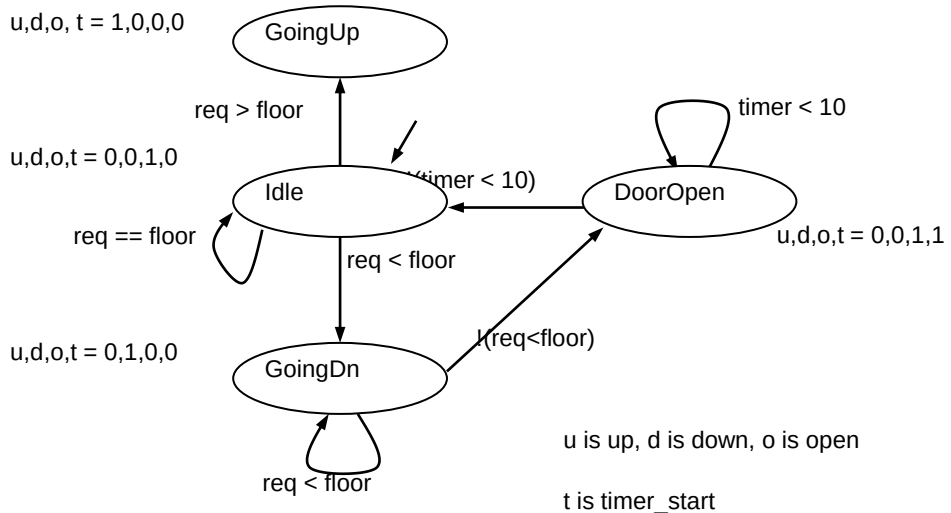
*We described UnitControl as an FSMD*



# Describing a system as a state machine

1. List all possible states
2. Declare all variables (none in this example)
3. For each state, list possible transitions, with conditions, to other states
4. For each state and/or transition, list associated actions
5. For each state, ensure exclusive and complete exiting transition conditions
   - No two exiting conditions can be true at same time
     - Otherwise nondeterministic state machine
   - One condition must be true at any given time
     - Reducing explicit transitions should be avoided when first learning






```
#define IDLE0
#define GOINGUP1
#define GOINGDN2
#define DOOROPEN3
void UnitControl() {
   int state = IDLE;
   while (1) {
      switch (state) {
   IDLE: up=0; down=0;
      open=1; timer_start=0;
      if (req==floor)
          {state = IDLE;}
       if(req > floor)
          {state = GOINGUP;}
       if(req < floor)
          {state = GOINGDN;}
       break;
```

```
GOINGUP:  up=1; down=0; open=0;
          timer_start=0;
  if (req > floor) {state = GOINGUP;}
  if (!(req>floor)){state =DOOROPEN;}
    break;
GOINGDN:  up=0; down=1; open=0;
          timer_start=0;
  if (req < floor) {state = GOINGDN;}
  if (!(req<floor)){state =DOOROPEN;}
    break;
DOOROPEN: up=0; down=0; open=1;
          timer_start=1;
  if (timer < 10) {state = DOOROPEN;}
  if (!(timer<10)){state = IDLE;}
          break;
      }
   }
}
```

***UnitControl* state machine in sequential programming language**

# State machine vs. sequential program model

- Different thought process used with each model
- State machine:
  - Encourages designer to think of all possible states and transitions among states based on all possible input conditions
- Sequential program model:
  - Designed to transform data through series of instructions that may be iterated and conditionally executed
- State machine description excels in many cases
  - More natural means of computing in those cases
  - *Not* due to graphical representation (state diagram)
    - Would still have same benefits if textual language used (i.e., state table)
    - Besides, sequential program model could use graphical representation (i.e., flowchart)

# General template

```
#define S0  0
#define S1  1
...
#define SN  N
void StateMachine() {
   int state = S0; // or whatever is the initial state.
   while (1) {
      switch (state) {
         S0:
            // Insert S0's actions here & Insert transitions Ti leaving S0:
            if( T0's condition is true ) {state = T0's next state; /*actions*/ }
            if( T1's condition is true ) {state = T1's next state; /*actions*/ }
            ...
            if( Tm's condition is true ) {state = Tm's next state; /*actions*/ }
            break;
         S1:
            // Insert S1's actions here
            // Insert transitions Ti leaving S1
            break;
         ...
         SN:
            // Insert SN's actions here
            // Insert transitions Ti leaving SN
            break;
      }
   }
}
```

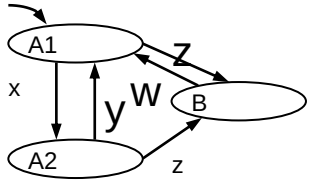
# General template

- Follow rules (template) for capturing state machine constructs in equivalent sequential language constructs
- Used with software (e.g.,C) and hardware languages (e.g.,VHDL)
- Capturing *UnitControl* state machine in C
  - Enumerate all states (#define)
  - Declare state variable initialized to initial state (IDLE)
  - Single switch statement branches to current state's case
  - Each case has actions
    - up, down, open, timer_start
  - Each case checks transition conditions to determine next state
    - if(…) {state = …;}

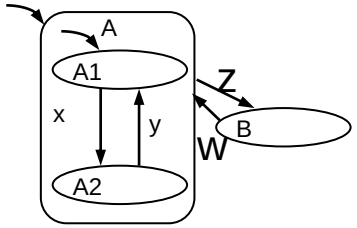
# HCFSM and the Statecharts language

- Hierarchical/concurrent state machine model (HCFSM)
  - Extension to state machine model to support hierarchy and concurrency
  - States can be decomposed into another state machine
    - *With hierarchy* has identical functionality as *Without hierarchy*, but has one less transition (*z*)
    - Known as OR-decomposition
  - States can execute concurrently
    - Known as AND-decomposition
- Statecharts
  - Graphical language to capture HCFSM
  - *timeout*: transition with time limit as condition
  - *history*: remember last substate OR-decomposed state *A* was in before transitioning to another state *B*
    - Return to saved substate of A when returning from B instead of initial state

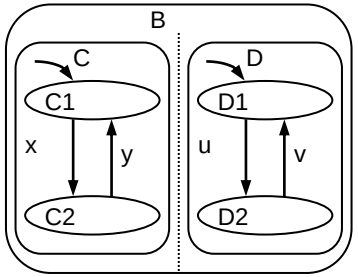
**Without hierarchy**



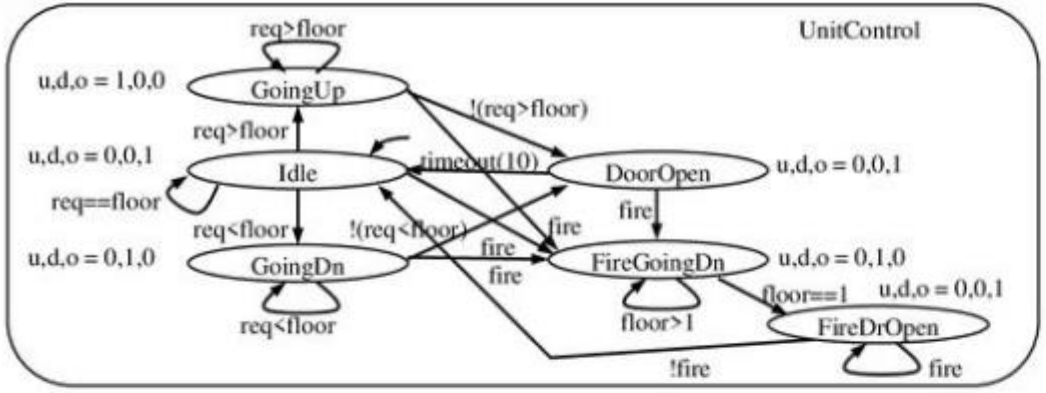

**With hierarchy**




**Concurrency**



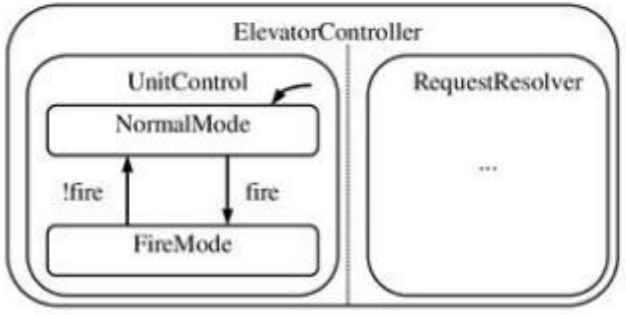
# Unit control with fire mode



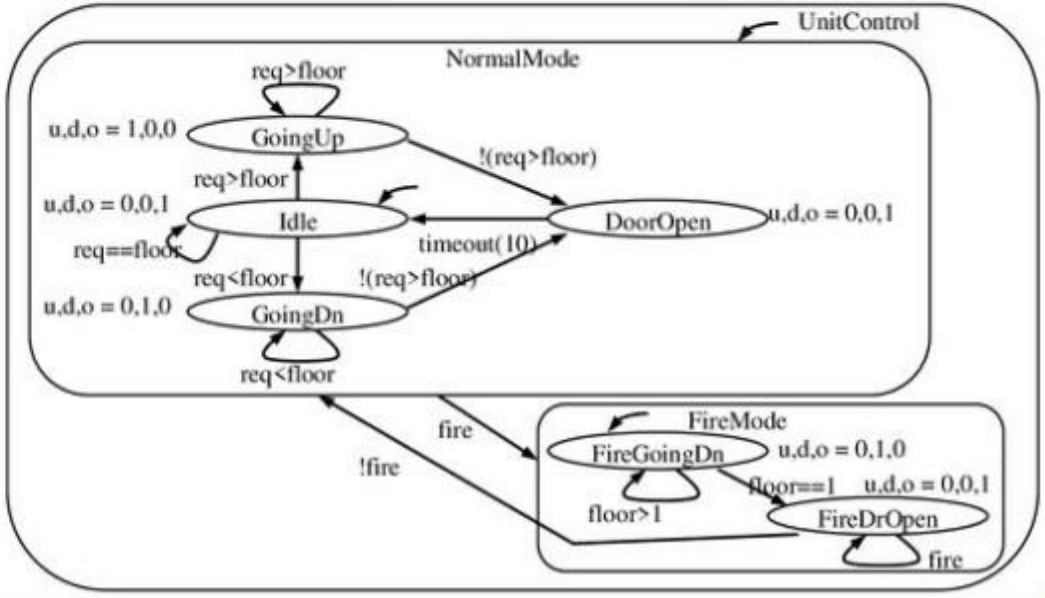

**Without hierarchy**

- FireMode
  - When *fire* is true, move elevator to 1st floor and open door
  - w/o hierarchy: Getting messy!
  - w/ hierarchy: Simple!

**With hierarchy**

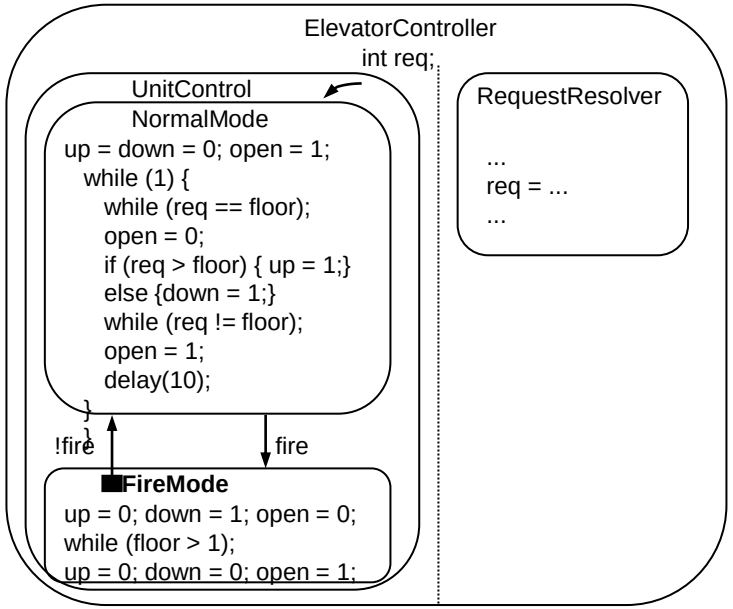



**With concurrent *RequestResolver***



# Program-state machine model (PSM): HCFSM plus sequential program model

- Program-state's actions can be FSM or sequential program
  - Designer can choose most appropriate
- Stricter hierarchy than HCFSM used in Statecharts
  - transition between sibling states only, single entry
  - Program-state may "complete"
    - Reaches end of sequential program code, OR
    - FSM transition to special *complete* substate
    - PSM has 2 types of transitions
      - Transition-immediately (TI): taken regardless of source program-state
      - Transition-on-completion (TOC): taken only if condition is true AND source program-state is complete
  - SpecCharts: extension of VHDL to capture PSM model
  - SpecC: extension of C to capture PSM model




- *NormalMode* and *FireMode* described as sequential programs
- Black square originating within *FireMode* indicates *!fire* is a TOC transition
  - Transition from *FireMode* to *NormalMode* only after *FireMode* completed

# Thank You