EE414 Embedded Systems

Ch 9. State Machines

Byung Kook Kim
School of Electrical Engineering
Korea Advanced Institute of Science and Technology

Outline

State Machine Model

9.1 Introduction

- Describing embedded system's processing behavior
  - Can be extremely difficult
    - Complexity increasing with increasing IC capacity
      - Past: washing machines, small games, etc.
        - Hundreds of lines of code
      - Today: TV set-top boxes, Cell phone, etc. - Sophisticated appl.
        - Millions of lines of code
    - Desired behavior often not fully understood in beginning
      - Many implementation bugs due to description mistakes/omissions
  - English (or other natural language): common starting point
    - Precise description difficult to impossible
    - Example: Motor Vehicle Code – thousands of pages long...

Computation Model

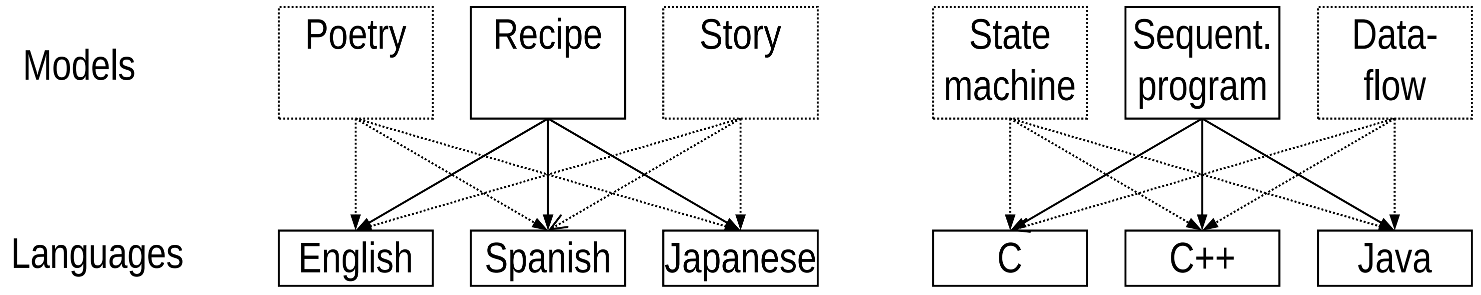
- How can we (precisely) capture behavior?
  - We may think of languages (C, C++), but ***computation model*** is the key.
  - Describe the behavior: Composed objects & execution semantics.
- **Common computation models:**
  - *Sequential program model*
    - Statements, rules for composing statements, semantics for executing them
  - *Communicating process model*
    - Multiple sequential programs running concurrently
  - *State machine model*
    - For control dominated systems, monitors control inputs, sets control outputs
  - *Dataflow model*
    - For data dominated systems, transforms input data streams into output streams
  - *Object-oriented model*
    - For breaking complex software into simpler, well-defined pieces.

9.2 Models vs. Languages, Text vs. Graphics

A. Models vs. Languages

- Computation models describe system behavior
  - Conceptual notion, e.g., recipe, sequential program
- Languages capture models
  - Concrete form, e.g., English, C

- *Variety of languages can capture one model*
  - E.g., sequential program model → C,C++, Java
- *One language can capture variety of models*
  - E.g., C++ → sequential program model, object-oriented model, state machine model
  - Certain languages better at capturing certain computation models.



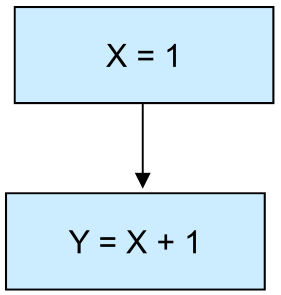
<u>Recipes vs. English</u> <u>Sequential programs vs. C</u>

B. Text versus Graphics

- Languages may use a variety of methods to capture models
  - Texts or graphics

- *Text and graphics* are just two types of *languages*
  - *Text*: letters, numbers
  - *Graphics*: circles, arrows (plus some letters, numbers).

```
X = 1;
Y = X + 1;
```



9.3 An Introductory Example

A simple elevator controller

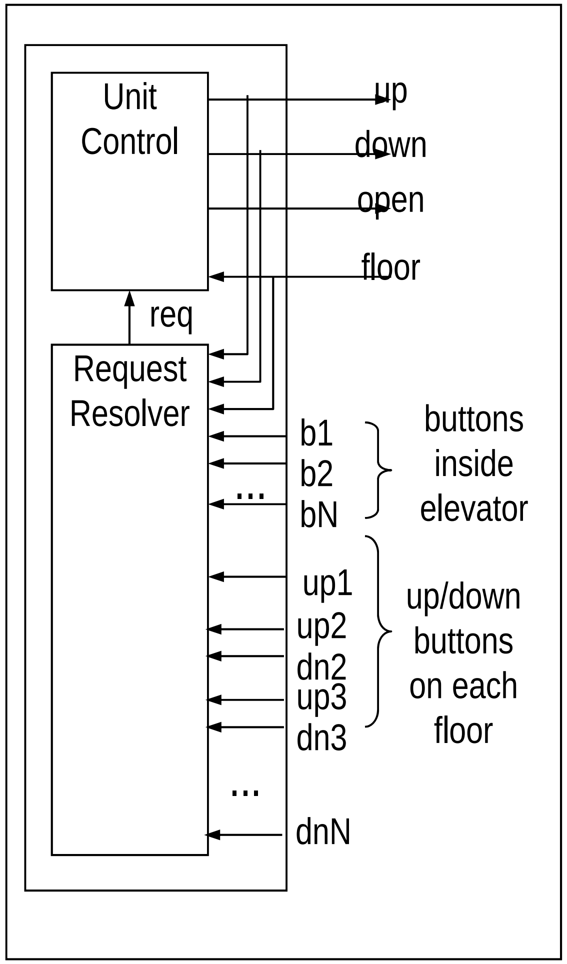
- **System interface →**
  - Inputs, outputs, and blocks.
  - *Request Resolver* resolves various floor requests into single requested floor
  - *Unit Control* moves elevator to this requested floor
- **A. Partial English description**
  - System's desired behavior

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…"

- Try capturing in C...

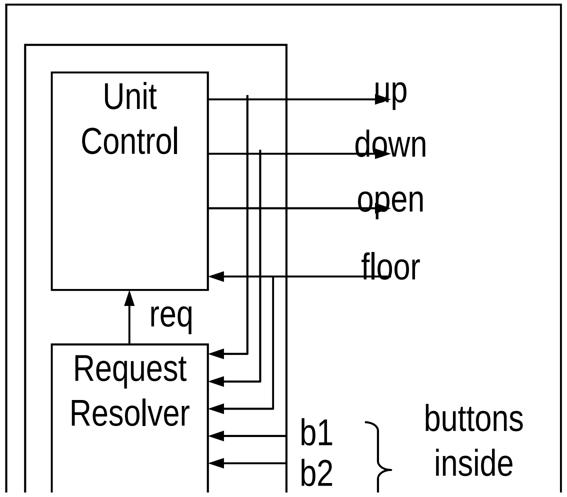
System interface



Elevator controller (II)

- **B. More precise - *sequential program model* →**
  - You might have come up with something having even more *if statements*.

System interface



Sequential program model

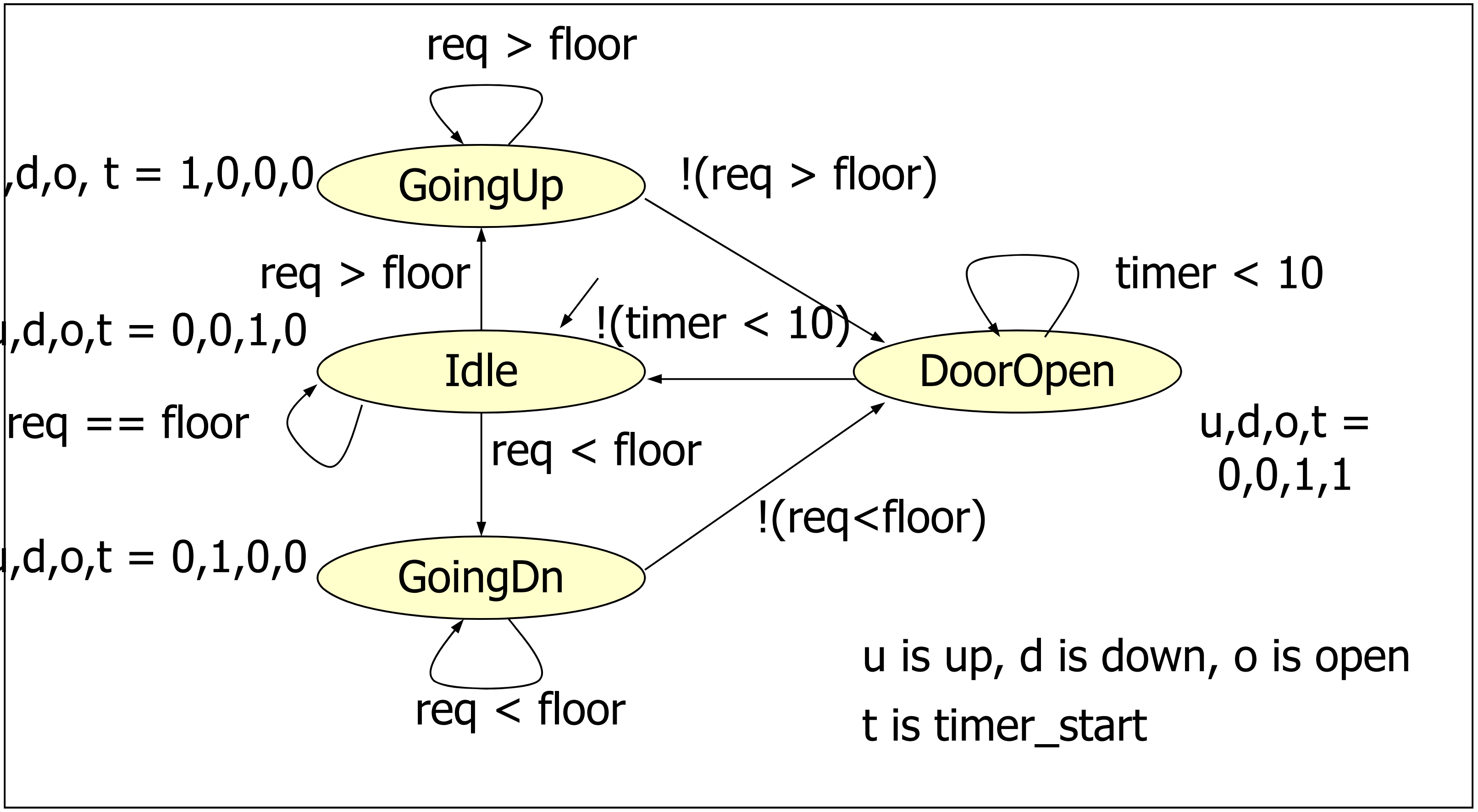
```
Inputs: int floor; bit b1..bN; up1..upN-1; dn2..dnN;
Outputs: bit up, down, open;
Global variables: int req;

void UnitControl()                      void RequestResolver()
{                                       {
  up = down = 0; open = 1;                while (1)
  while (1) {                             ...
    while (req == floor);                   req = ...
    open = 0;                             ...
    if (req > floor) { up = 1;}         }
    else {down = 1;}
    while (req != floor);               void main()
    up = down = 0;                      {
    open = 1;                             Call concurrently:
    delay(10);                              UnitControl() and
  }                                         RequestResolver()
}                                       }
```

9.4 Finite-State Machines (FSM) A Basic State Machine Model:

- Trying to capture this behavior as sequential program is a bit awkward.
  - Instead, we might consider an **FSM model**
- **FSM (Finite State Machine) model**
  - Describes the system behavior as a set of possible **states**
    - The system can only be in one of these states at a given time.
  - Describe the possible **transitions** from one state to another
    - *Depending on input values*
  - Describe **actions** that occur
    - In a *state* or when *transitioning between states*
- *Ex:* Elevator controller:
  - Possible states
    - E.g., *Idle*, *GoingUp*, *GoingDn*, *DoorOpen*
  - Possible transitions from one state to another based on input
    - E.g., req > floor (current), 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).

Finite-State Machine (FSM) Model

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



Formal Definition of FSM

- **An FSM is a 6-tuple *$F<S, I, O, F, H, s_0>$*,** where
  - 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 ($S \times I \rightarrow S$)
  - 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 $S \times I \rightarrow O$)

- Shorthand notations to simplify descriptions
  - Implicitly assign 0 to all unassigned outputs in a state
  - Implicitly AND every transition condition with clock edge: Synchronous.

9.5 Finite-State Machine with Datapath Model: FSMD

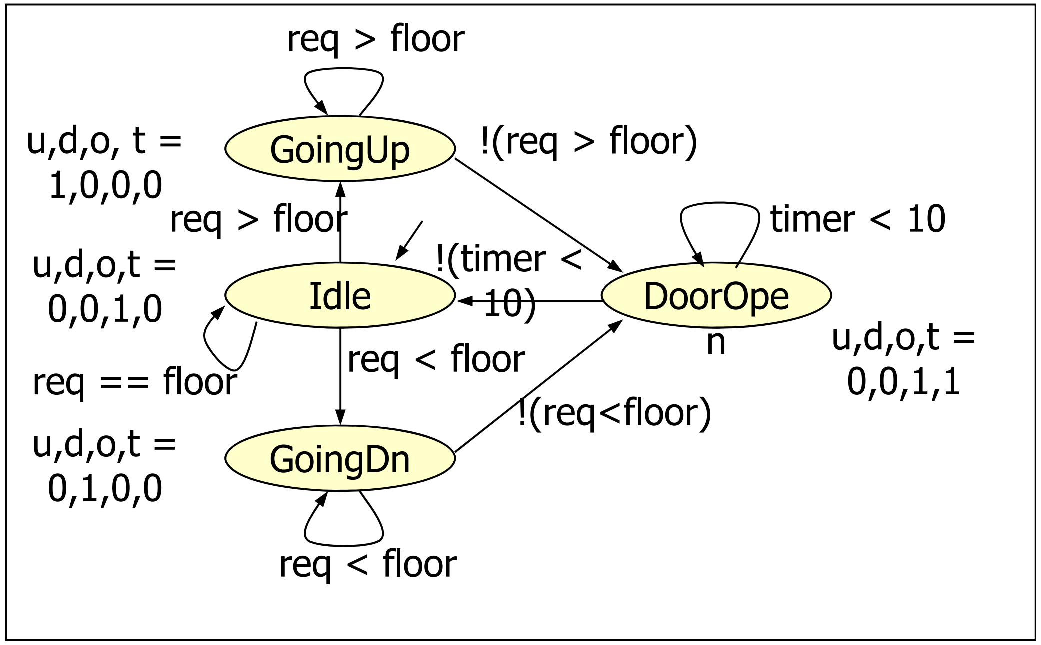
- FSMD (Finite-State Machine with Datapath) extends FSM
  - More complex data types and variables for storing data
  - FSMs use only Boolean data types and operations, no variables
- **FSMD: 7-tuple** $<S, I, O, \underline{V}, F, H, s_0>$
  - S is a set of states $\{s_0, s_1, \ldots, s_l\}$
  - I is a set of inputs $\{i_0, i_1, \ldots, i_m\}$
  - O is a set of outputs $\{o_0, o_1, \ldots, o_n\}$
  - **V is a set of variables $\{v_0, v_1, \ldots, v_n\}$**
  - F is a next-state function **($S \times I \times V \rightarrow S$)**
  - H is an **action function ($S \rightarrow O + V$)**
  - s_0 is an initial state. (Moore-type FSMD)

- I, O, V may represent more *complex data types* (integer, 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*.

9.6 Using State Machines

Describing a system as a state machine

- 1. List all possible *states*
- 2. Declare all *variables*
- 3. For each state,
  list possible *transitions*,
  with associated *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*
  - *Exclusive*: No two exiting conditions can be true at the same time
    - Otherwise *nondeterministic* state machine
  - *Complete*: One condition must be true at any given time
    - Reducing explicit transitions should be avoided when first learning.

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*
  - *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).

Try Capturing Other Behaviors with an FSM

- E.g., Answering machine blinking light when there are messages
- E.g., A simple telephone answering machine that answers after 4 rings when activated
- E.g., A simple crosswalk traffic control light

Capturing state machines in sequential programming language

- Despite benefits of state machine model, *most popular development tools use sequential programming language*
  - C, C++, Java, Ada, VHDL, Verilog, etc.
  - Development tools are complex and expensive, therefore not easy to adapt or replace
    - Must protect investment
- *Two approaches* to capturing state machine model with sequential programming language
  - **Front-end tool approach**
    - Additional tool installed to support state machine language
      - Graphical and/or textual state machine languages
      - May support graphical simulation
      - Automatically generate code in sequential programming language that is input to main development tool
    - Drawback: must support additional tool (licensing costs, upgrades, training, etc.)
  - **Language subset approach**
    - Directly capture state machine model in a sequential program language
    - Most common approach ...→

Language Subset Approach

- 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 = …;}

```
#define IDLE  0
#define GOINGUP  1
#define GOINGDN  2
#define DOOROPEN  3
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=1; down=0; 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;
    }
  }
}
```

General template

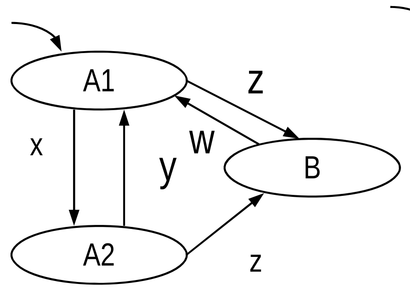
- General template →
  - For capturing a state machine in a sequential programming language.

```
#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 T_i leaving S0:
        if( T_0's condition is true ) {state = T_0's next state; /*actions*/ }
        if( T_1's condition is true ) {state = T_1's next state; /*actions*/ }
        ...
        if( T_m's condition is true ) {state = T_m's next state; /*actions*/ }
        break;
      S1:
        // Insert S1's actions here
        // Insert transitions T_i leaving S1
        break;
      ...
      SN:
        // Insert SN's actions here
        // Insert transitions T_i leaving SN
        break;
    }
  }
}
```

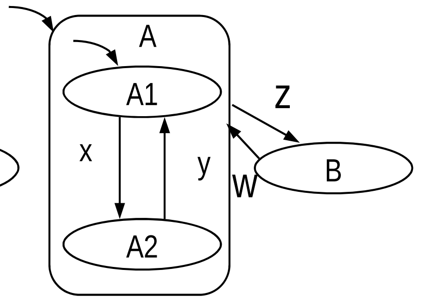
9.7 HCFSM and the Statecharts Language

- *Hierarchical/Concurrent* Finite-State Machine model (HCFSM)
  - Extension to state machine model to support *hierarchy* and *concurrency*
- *Hierarchy*
  - 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*
- *Concurrency*
  - States can execute concurrently →
    - Known as *AND-decomposition*

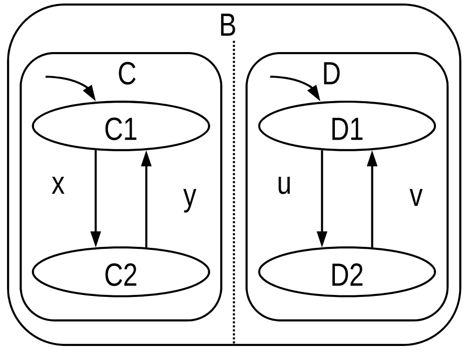
Without hierarchy



With hierarchy

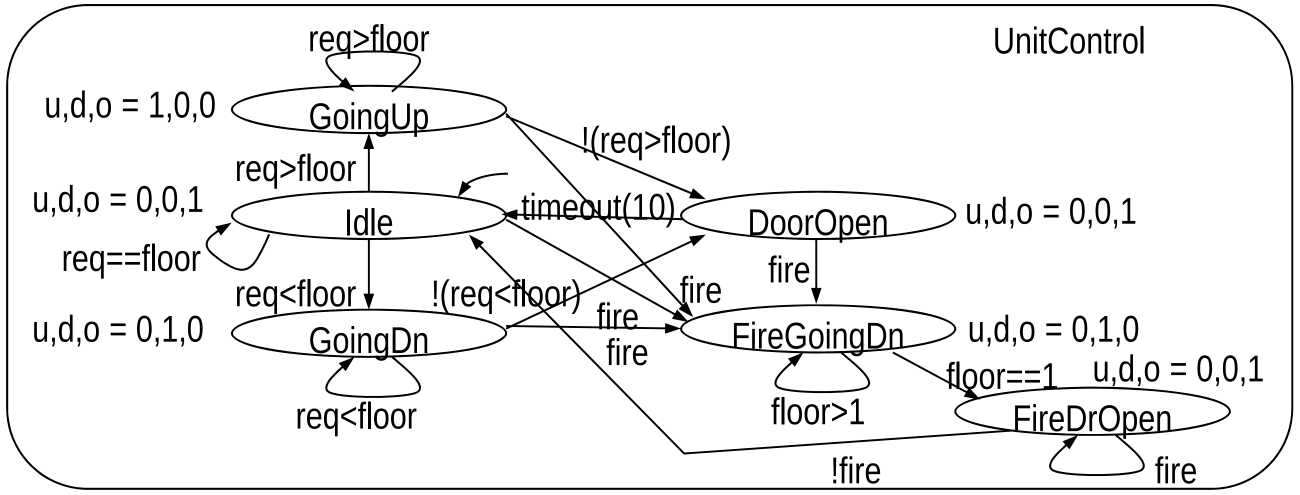


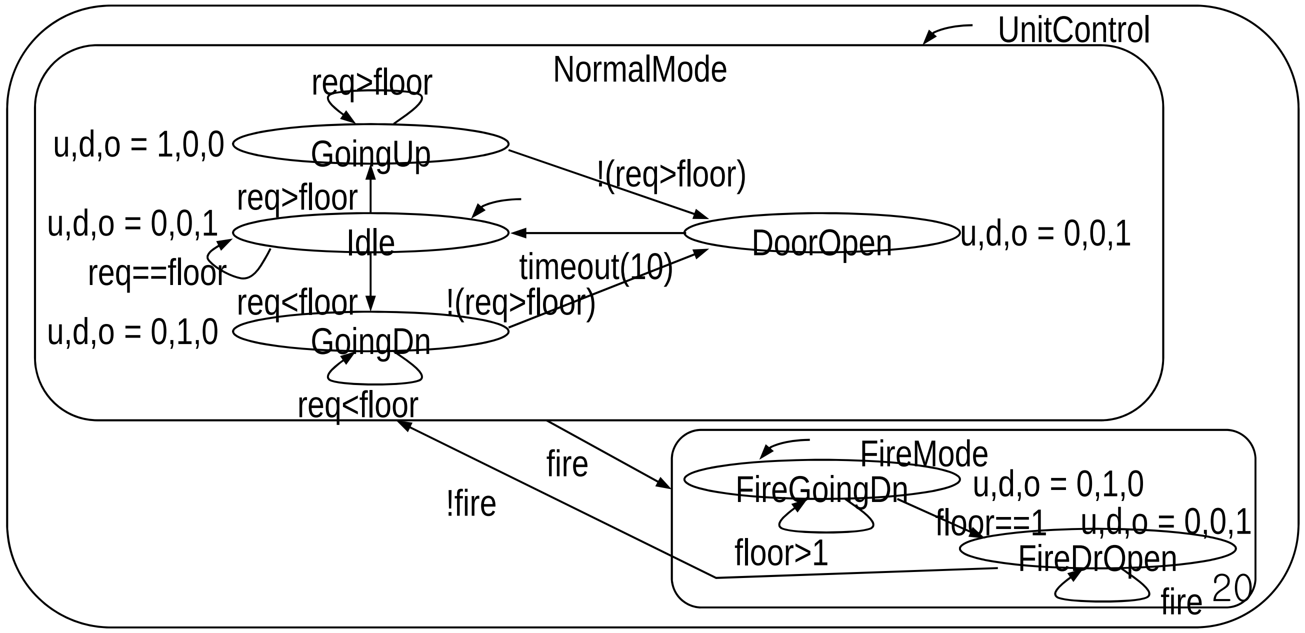
Concurrency



Hierarchy: *UnitControl* with *FireMode*

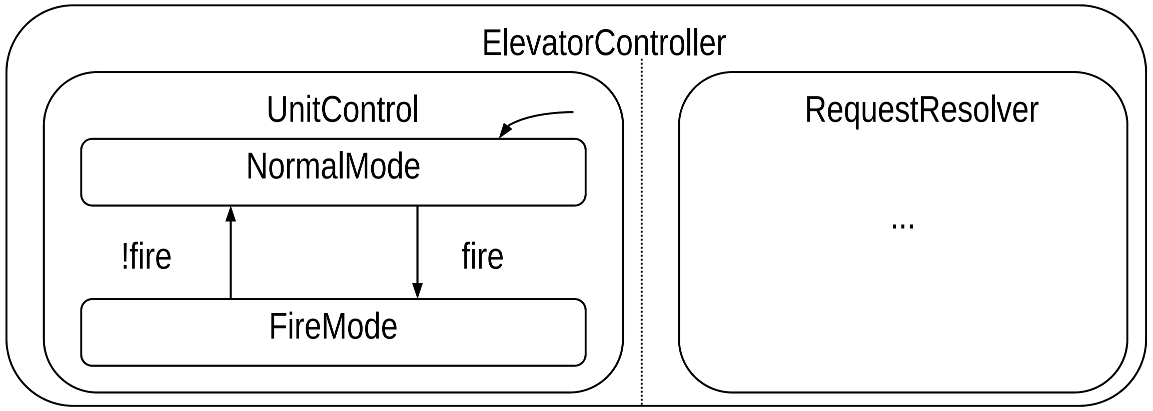
- FireMode
  - When *fire* is true, move elevator to 1st floor and open door
  - w/o hierarchy: Getting messy! →
  - w/ hierarchy: Simple! →





Concurrency: *UnitControl* with *FireMode*

- Concurrency
  - *Decompose* a state into two concurrent states
  - Conversely, *Group* two concurrent states into a new state



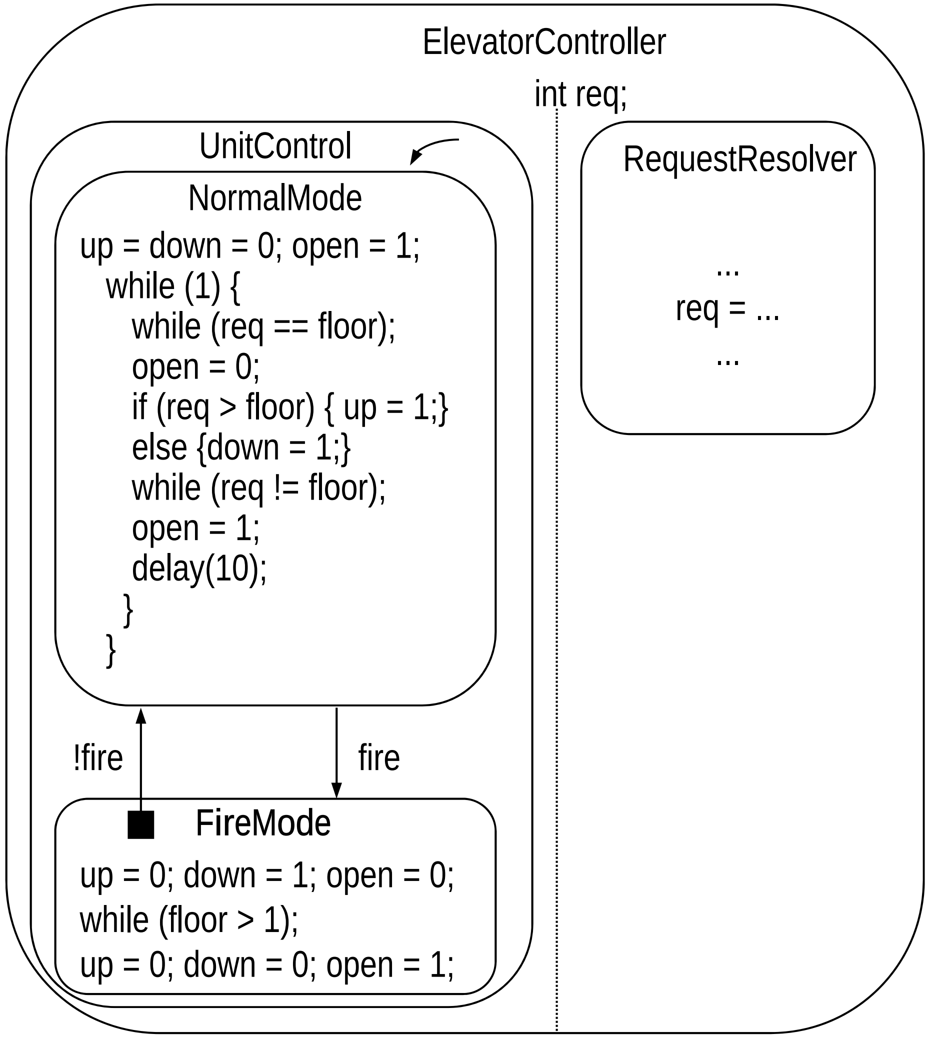
With concurrent *RequestResolver*

9.8 Program-State Machine Model (PSM)

- **Program-State Machine Model (PSM)**
  - Allow use of sequential program code to define a state's action
  - Include extensions for complex data types and variables
  - Merger: 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 of "completing"
    - 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.

PSM (II)

- PSM to describe the Elevator Controller →
  - AND-decompose: UnitControl & RequestResolver
  - OR-decompose: NormalMode & FireMode
  - *NormalMode* and *FireMode:* sequential programs
  - Black square (!*fire*): TOC (Transition-on-completion)
    - Transition from *FireMode* to *NormalMode* only after *FireMode* completed.



```
NormalMode
up = down = 0; open = 1;
  while (1) {
    while (req == floor);
    open = 0;
    if (req > floor) { up = 1;}
    else {down = 1;}
    while (req != floor);
    open = 1;
    delay(10);
  }
}
```

```
RequestResolver
...
req = ...
...
```

```
FireMode
up = 0; down = 1; open = 0;
while (floor > 1);
up = 0; down = 0; open = 1;
```

9.9 The Role of Appropriate Model and Language

- *Finding appropriate model to capture embedded system is an important step.*
  - The model shapes the way we think of the system
    - Sequential program model: Sequential program.
    - *State machine model: Think in terms of states and transitions among states*
      - When system must *react to changing inputs, state machine might be the best model.*
- *Language should capture model easily.*
  - Structured techniques can be used instead
    - E.g., Template for state machine capture in sequential program language
- We can use a different model if that model provides an advantage.
  - *Capture the model in the language using structured techniques!*

References

- [1] Frank Vahid, "Embedded system design: A unified hardware/software introduction", John Wiley & Sons, 2002.