SES

Chapter 8: Finite-State Machines

Prof. Dr.-Ing. Bernd-Christian Renner

Institute smartPORT
Hamburg University of Technology

TUHH

Contents

Introduction

- Describing embedded system’s processing behavior
  - Complexity increasing with increasing IC capacity
    - Past: washing machines, small games, etc.
      - Hundreds of lines of code
    - Today: TV decoders, autonomous cars, etc.
      - Hundreds of thousand of lines of code
  - Desired behavior often not fully understood at start
    - Many implementation bugs due to description mistakes/omissions
  - Natural language is common starting point
    - Precise description difficult to impossible

1

Models vs. Languages

Common Computation Models

- Sequential program model
  - Statements, rules for composing statements, one sequential flow
- Concurrent process model
  - Multiple sequential programs running concurrently
- State machine model
  - For control dominated systems, monitors inputs, sets outputs
- Dataflow model
  - For data dominated systems, transforms input into output streams
- Object-oriented model
  - For breaking complex software into simpler, well-defined pieces

Models vs. Languages

- Computation models describe system behavior
- Concrete languages capture models
- Several languages can capture the same model
  - E.g., Assembler, C, C++ capture sequential program model
- One language can capture variety of models
  - E.g., C++ captures sequential program model, object-oriented model, state machine model

Introductory example: Elevator Control

- Simple elevator controller
- **Request Resolver** resolves various floor requests into single requested floor
- **Unit Control** moves elevator to this requested floor

Partial English description

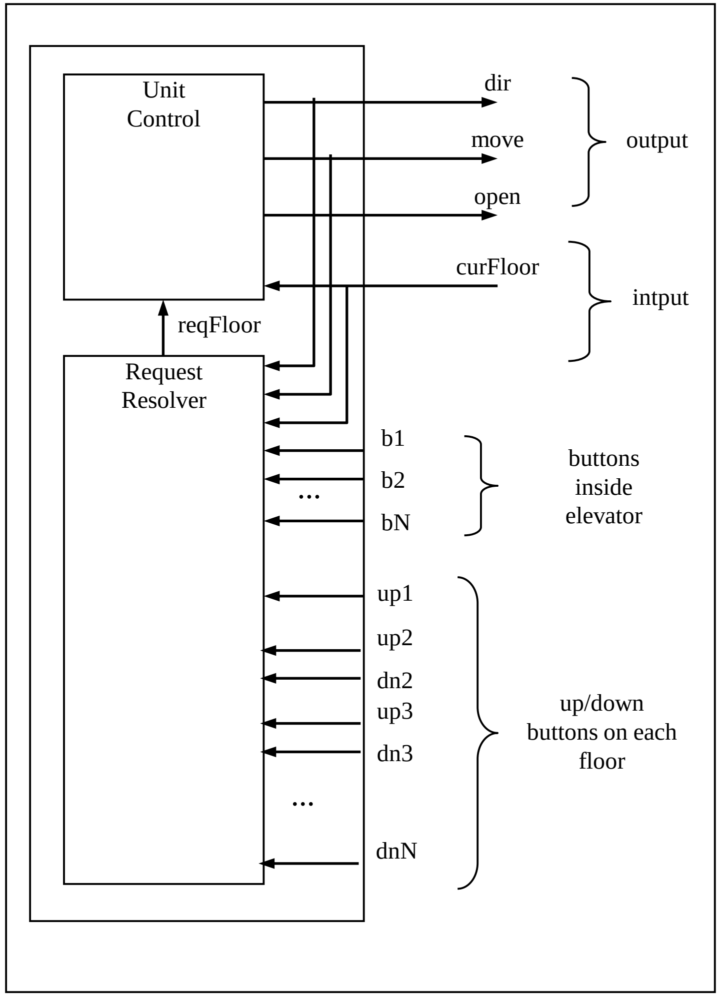
1. Move elevator either up or down to reach requested floor.

2. Once at requested floor, open door for at least 10 seconds, and keep it open until requested floor changes.

3. Ensure door is never open while moving.

4. Don't change directions unless there are no higher requests when moving up or no lower requests when moving down.

System interface



Elevator Control: Conc. Process Model

```
int curFloor; bit b1..bN, up1..upN-1, dn2..dnN;       /* Input */
bit dir, move, door;                                   /* Output */
int reqFloor;                                          /* Global variable */

void UnitControl()  {
   move = NO;
   door = OPEN;
   while (1) {
      while (reqFloor == curFloor) {}
      open = CLOSE;
      if (reqFloor > curFloor) {
         dir = UP;
      } else {
         dir = DOWN;
      }
      move = YES;
      while (reqFloor != curFloor) {}
      move = NO;
      door = OPEN;
      delay(10);
   }
}

void RequestResolver()  {
   while (1) {
      ...
      reqFloor = ...
      ...
   }
}

void main() {
   create_task(UnitControl);
   create_task(RequestResolver);
}
```

2

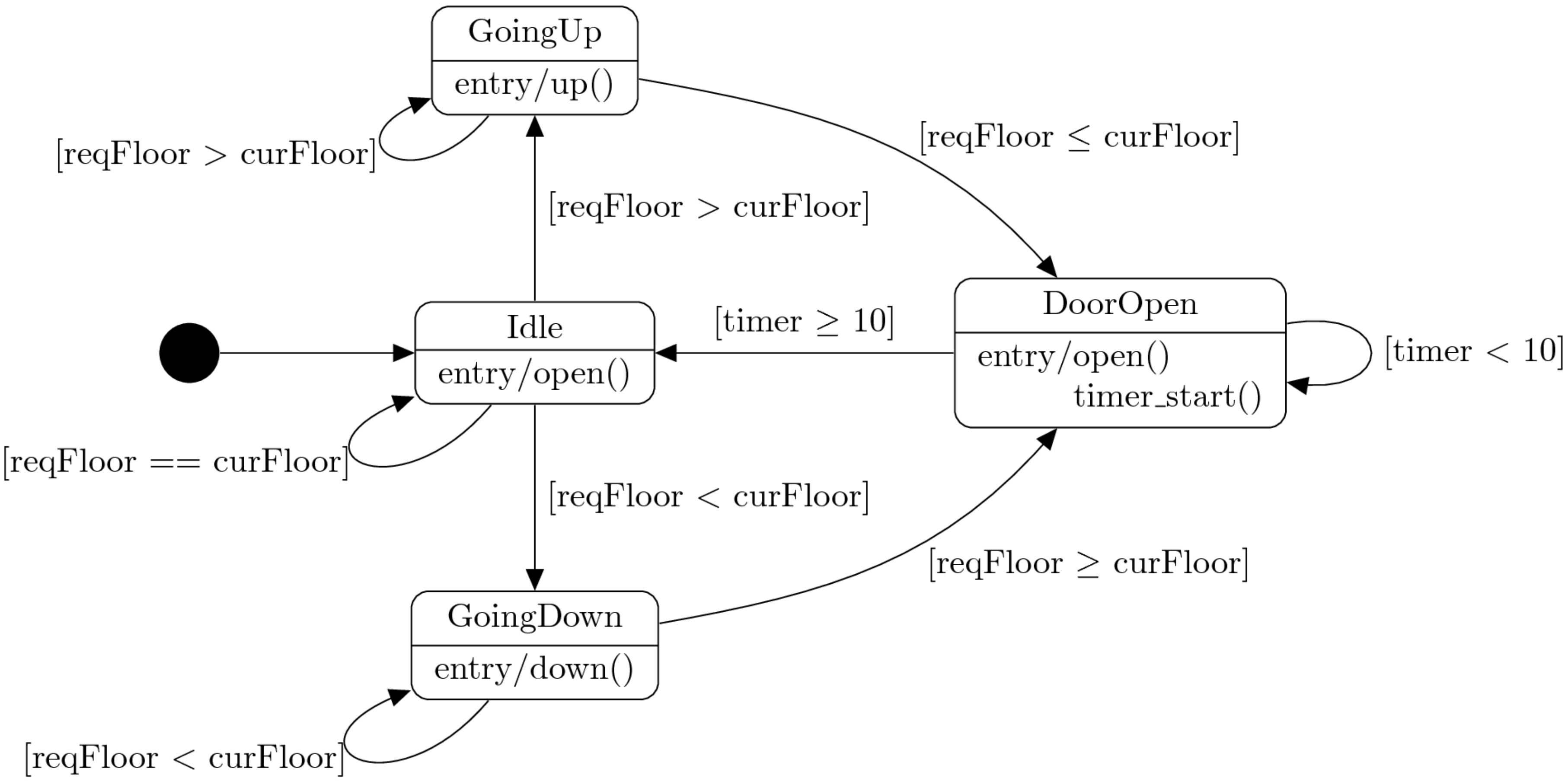
Finite-state machine (FSM) model

Finite State Machine Model

- Capturing elevator behavior as a sequential program is awkward
- Finite State Machine (FSM) model is better choice
  - Possible states
    - E.g., Idle, GoingUp, GoingDn, DoorOpen
  - Input data: reqFloor, curFloor, timer
  - Output data: dir, move, door, time
  - Possible transitions from one state to another based on input
    - E.g., reqFloor > curFloor
  - Entry actions set output, they occur when entering a state
    - E.g., In state *GoingUp:* dir=UP, door=CLOSE, move=YES

Finite State Machine (Moore-type)

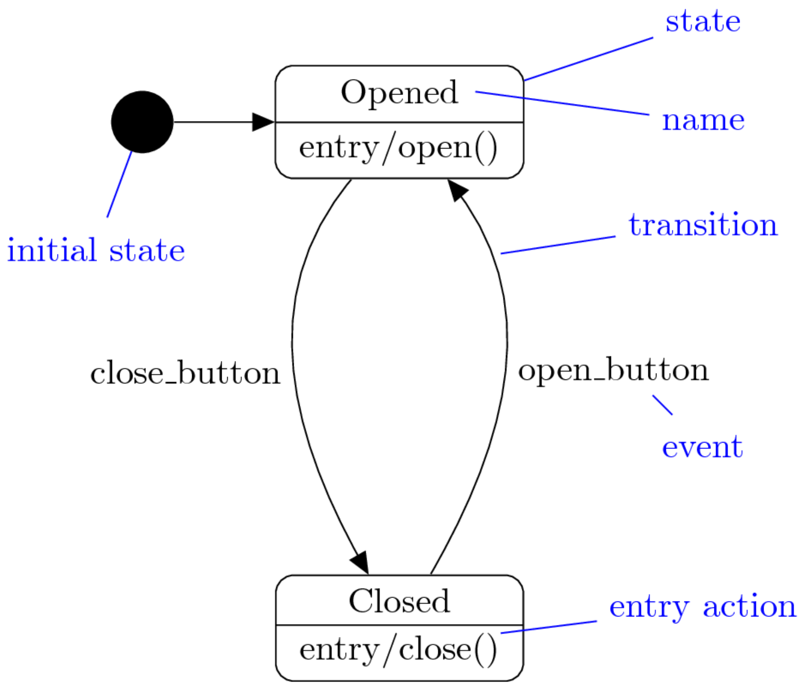
UnitControl task using a FSM



Example entry action: up() {dir=UP; door=CLOSE; move=YES;} (executed when state is entered)

Formal Definition

- An FSM is a 6-tuple <S, I, O, F, A, s_0>
  - S: set of states $\{s_0, s_1, \ldots, s_l\}$
  - I: set of Boolean inputs $\{i_0, i_1, \ldots, i_m\}$ (used in guards)
  - O: set of Boolean outputs $\{o_0, o_1, \ldots, o_n\}$ (comp. by entry act.)
  - F: next-state (transition) function (F: $S \times I \rightarrow S$)
  - A: entry action function (A: $S \rightarrow O$)
  - s_0: initial state
- Moore-type FSM
  - Action depends on state only
    → action associated with state
- Mealy-type FSM
  - Action depends on state and input
    → action associated with transition



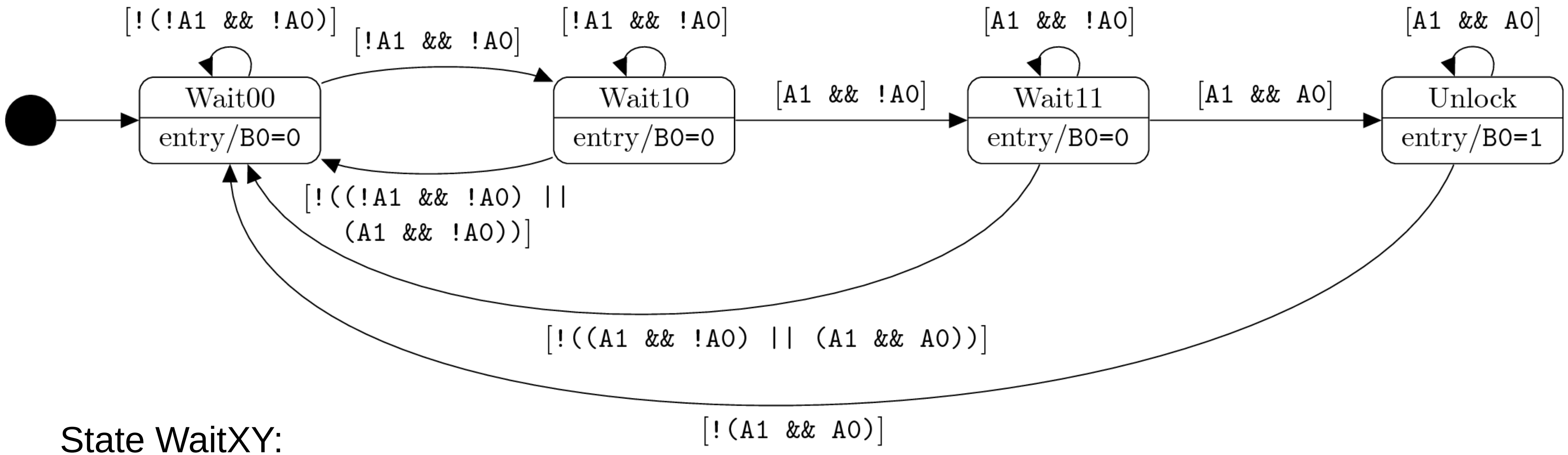
Execution of an FSM

- FSM is always in some state, called current state
  - At start: Initial state is current state and its entry action is executed
- Execution is triggered by events (i.e. reactive behavior)
  - Events carry input
- Upon each event
  - Current state's next-state function is executed, this selects a new current state and
  - entry actions of new current state are executed
  - Optional: Exit actions
- Events are assumed to occur at a rate that allows for processing (i.e. no event is missed)
- FSMs provide time-ordered behavior

Example: Electronic Lock

- Input: A0 and A1 (from switches)
- Output: B0 (B0 = 0 locks and B0 = 1 unlocks lock)
- Behavior:
  - To unlock, user first set switches such that A1A0 are 00, then 10, and 11
  - Any other sequence leading to 11 (such as 00, 01, 11) does not unlock lock, except repeating valid input is allowed
    - Example: 00, 10, 10, 11 is valid
- Time-ordered behavior is needed!
- Four states: Wait00, Wait10, Wait11, Unlock

Example: Electronic Lock



State WaitXY:
Wait for event XY to happen

Exercise: Extend FSM to sound alarm (output B1 = 1) if A1,A0=1 is reached by a sequence other than 00, 10, 11

Shortcomings of FSMs

- FSMs use only Boolean data types and operations
- FSMs can have a large state set
- Extension: FSM with Datapath Model (FSMD)
  - FSMDs use data types and variables for storing data
  - FSMDs allow to represent many states of a FSM with a single state with a variable

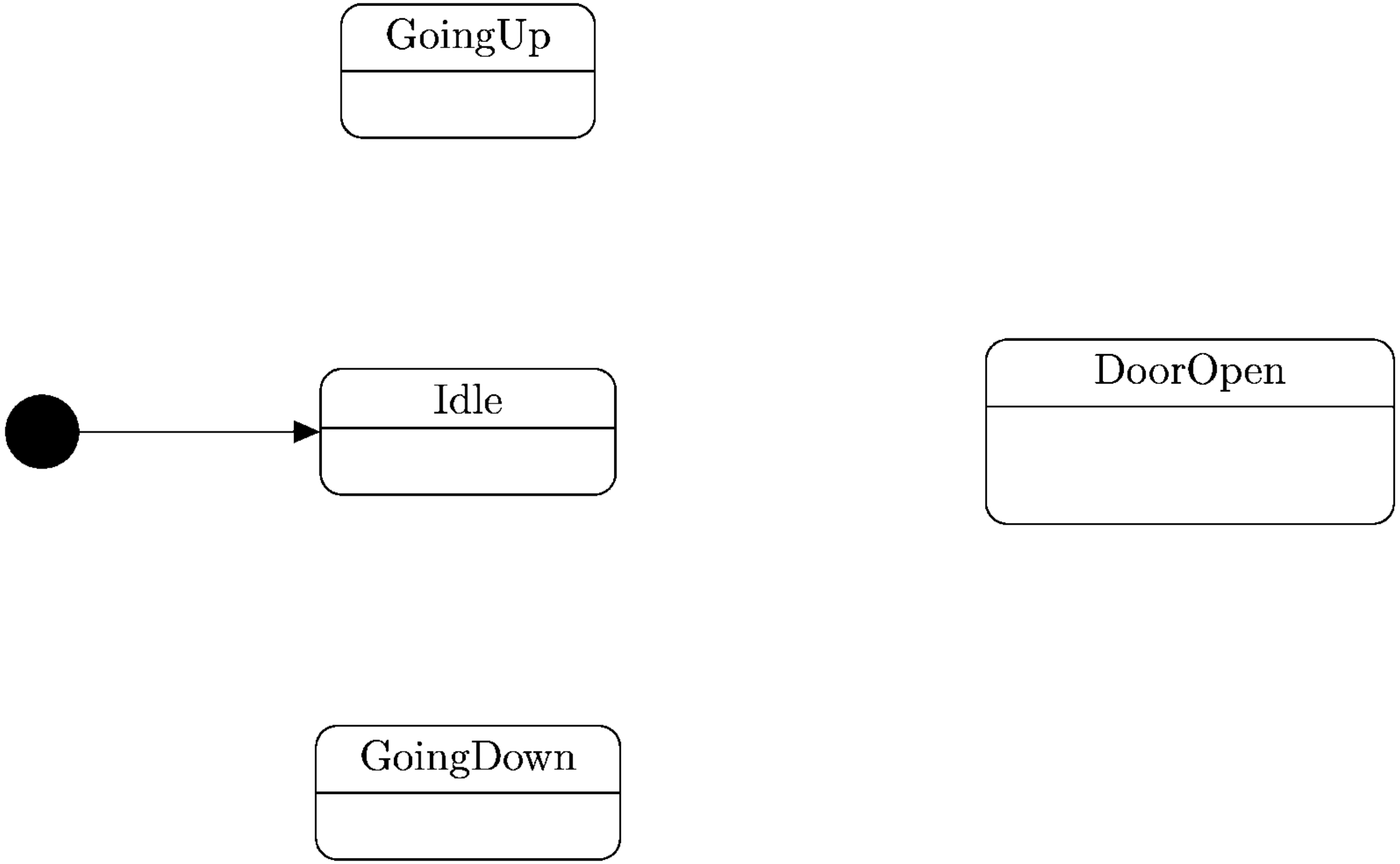
FSM with Datapath Model (FSMD)

- FSMD: 7-tuple <*S*, *I*, *O*, ***V***, *F*, A, s_0>
  - *S, I, O as before*
  - ***V* is a set of variables {v_0, v_1, …, v_n}**
  - *F* is a next-state function ($S \times I \times V \rightarrow S$)
  - *A* is an **entry action** function ($S \rightarrow O + V$)
  - s_0 is the initial state
- *I*,*O*,*V* may represent types (i.e., integers, floating point)
- *F*, A may include arithmetic operations
- *A* 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 in all states

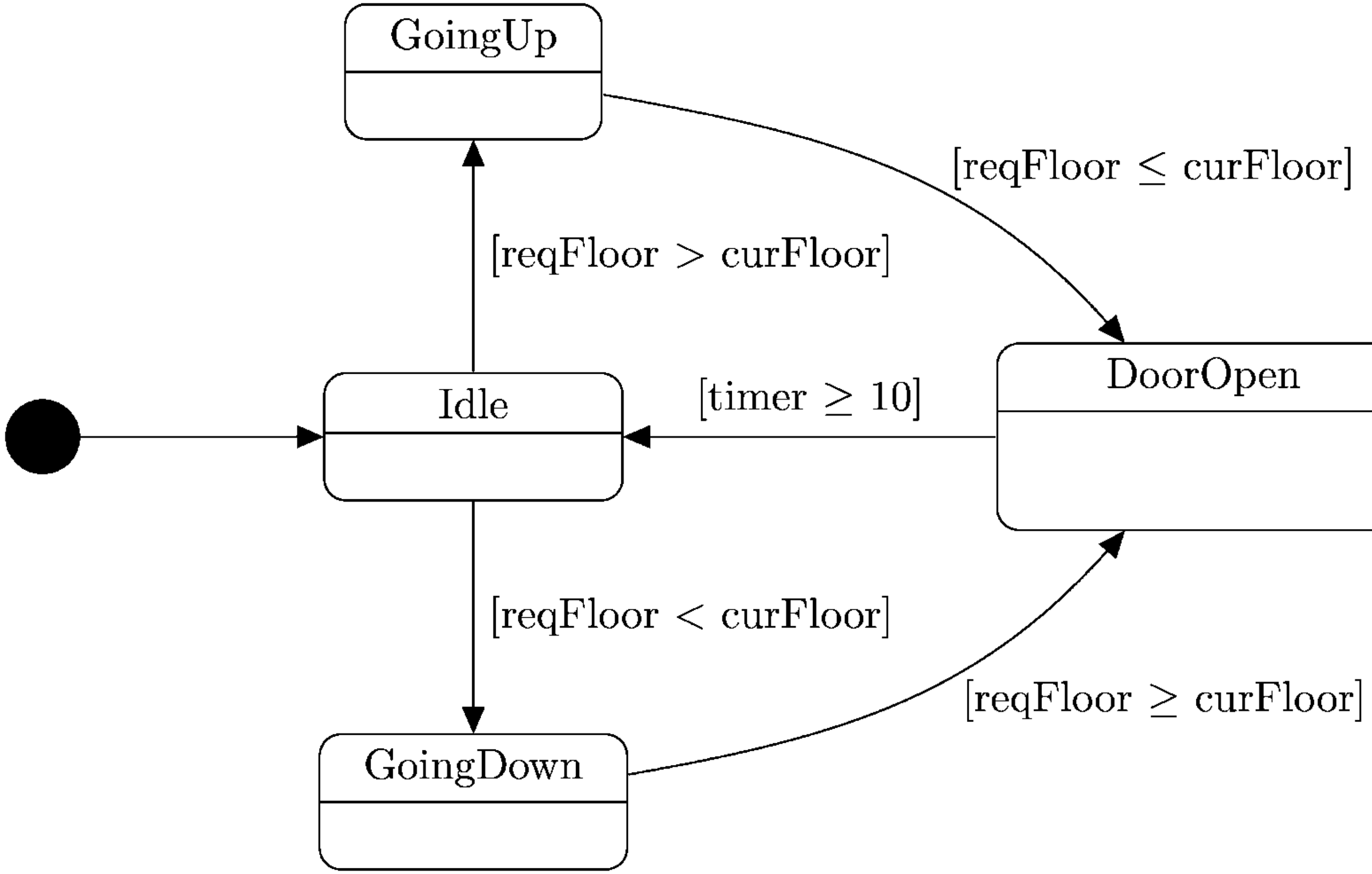
Describing a System as a FSM

1. List all possible states
2. Declare all variables
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
   - Convention: For all conditions not listed, remain in current state

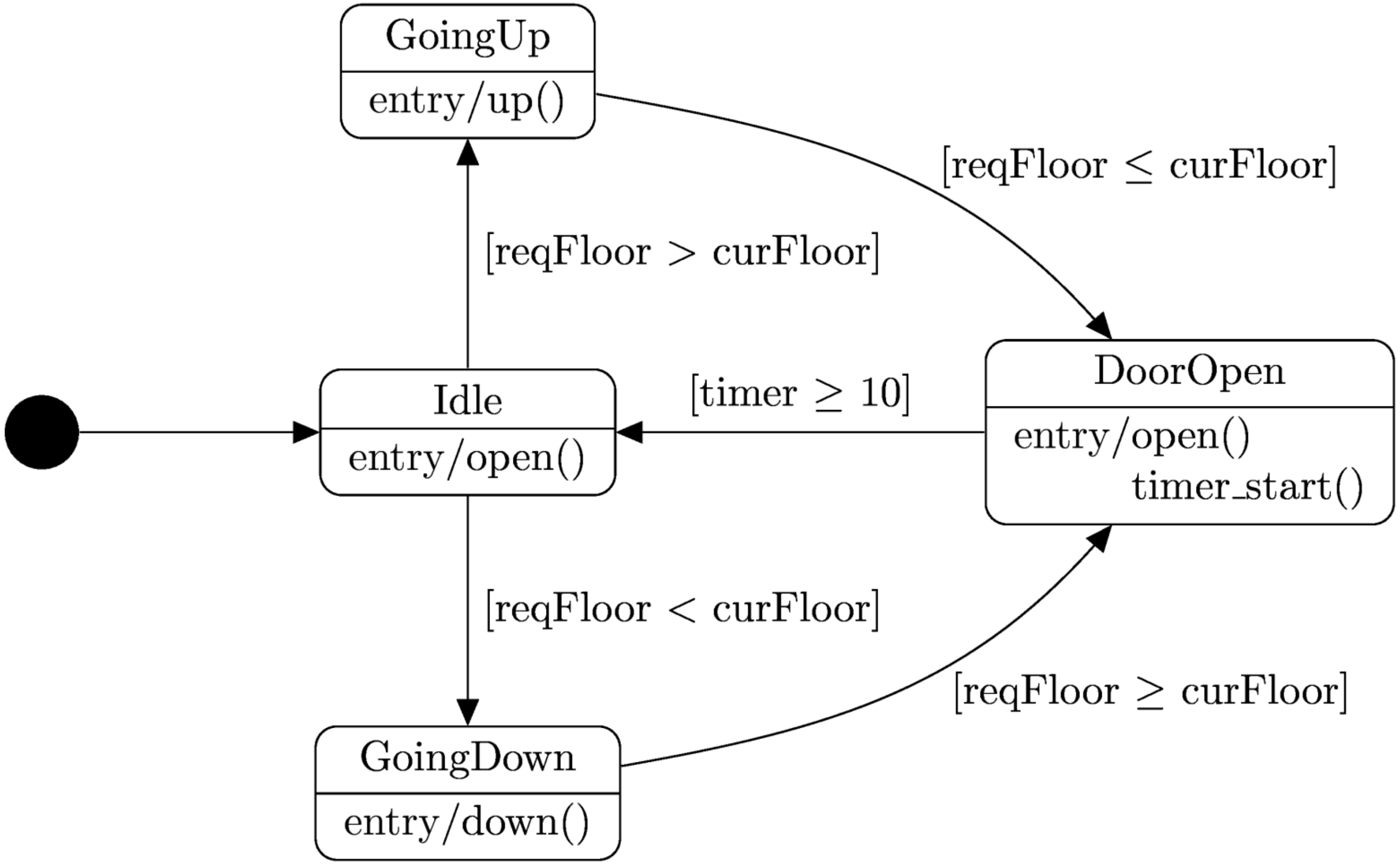
Describing a System as a FSM



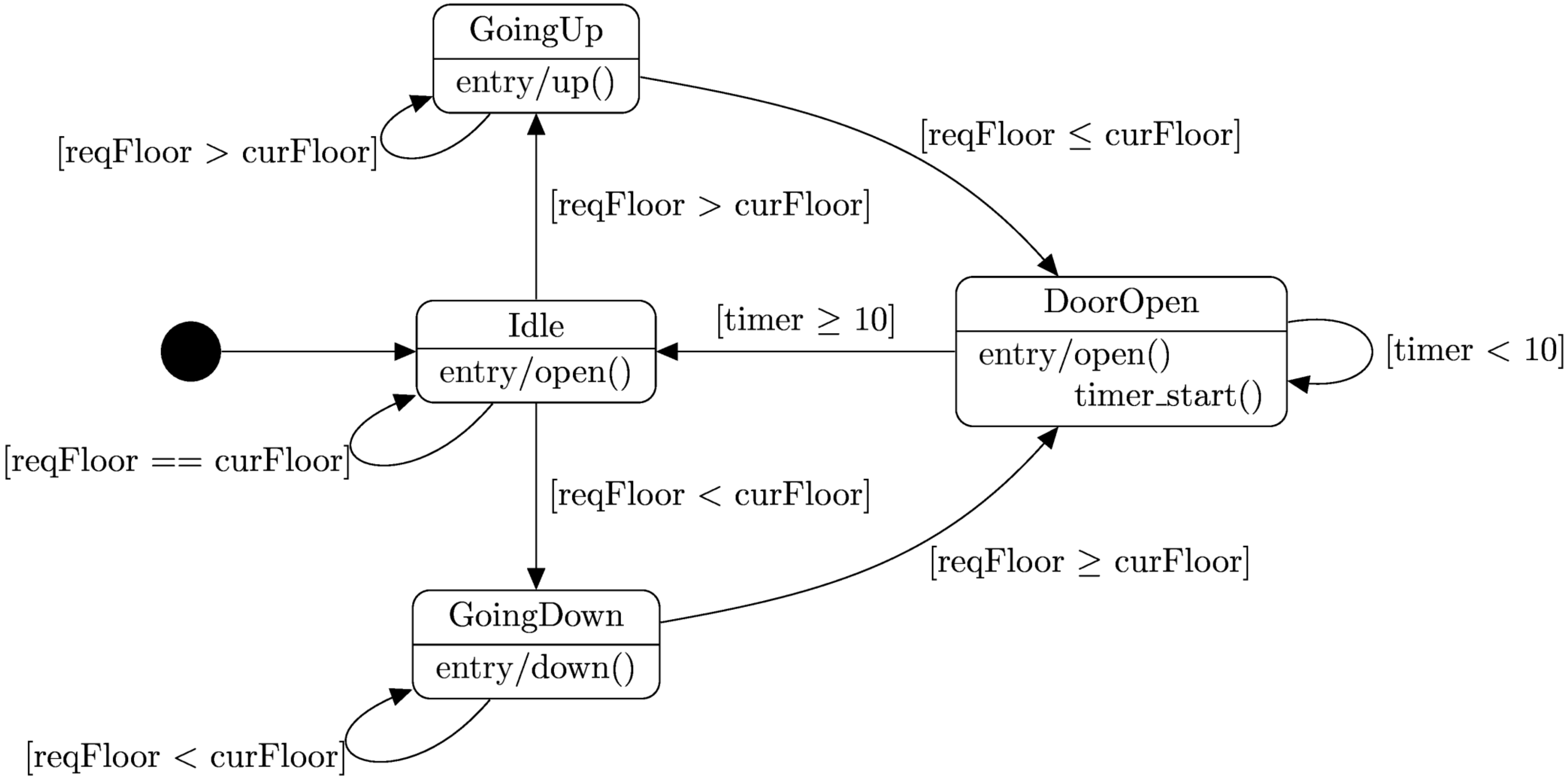
Describing a System as a FSM



Describing a System as a FSM



Describing a System as a FSM



State Machine vs. Sequent. 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 many cases
  - **Not** only due to graphical representation (state diagram)
    - Would still have benefits if textual language used (i.e., state table)
    - Sequential program model could use graphical representation (i.e., flowchart)

3

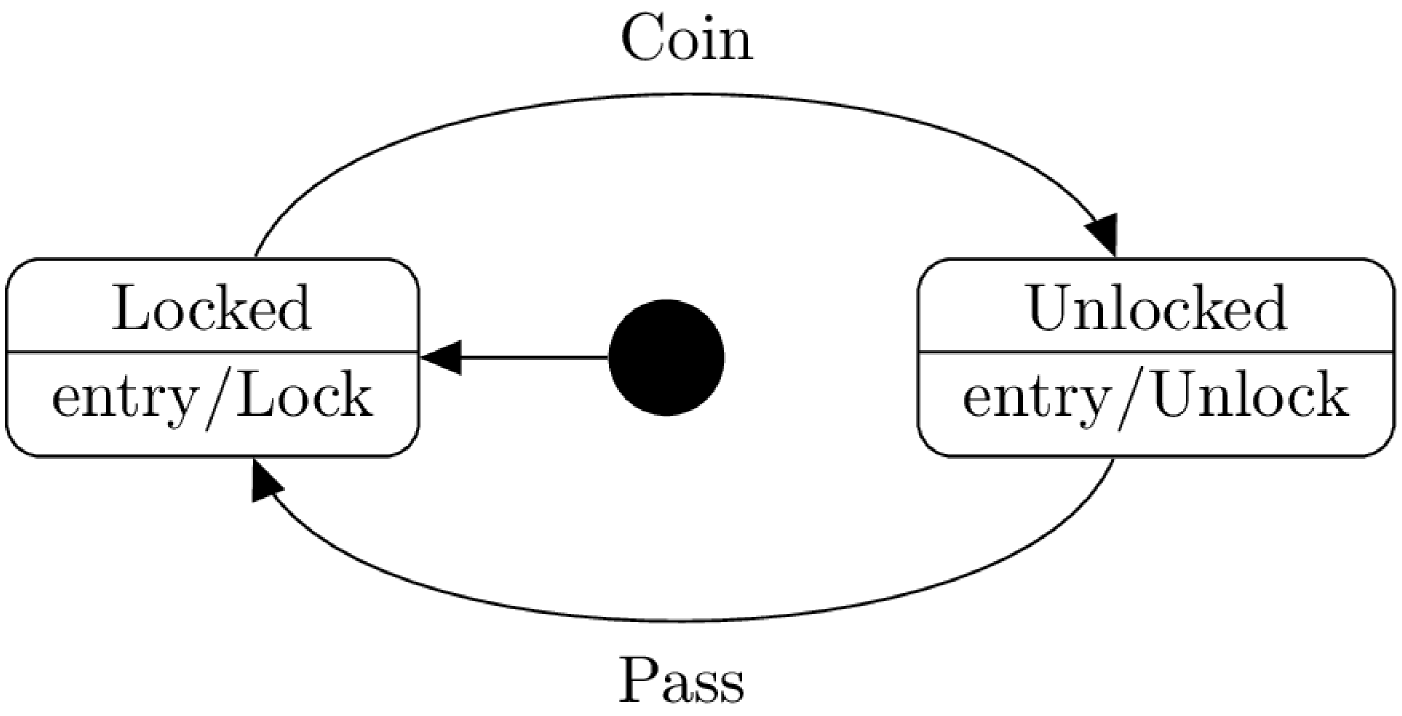
Moore- and Mealy-type FSMs

FSM Types

- Moore-type
  - Action performed when entering state
    - Associates actions with states only
    - A maps $S \rightarrow O$, (as given above)
- Mealy-type
  - Action performed depending on state and input
    - Associates actions with states and transitions
    - A maps $S \times I \rightarrow O$
    - A is no longer an entry action but a transition action

Subway Turnstile

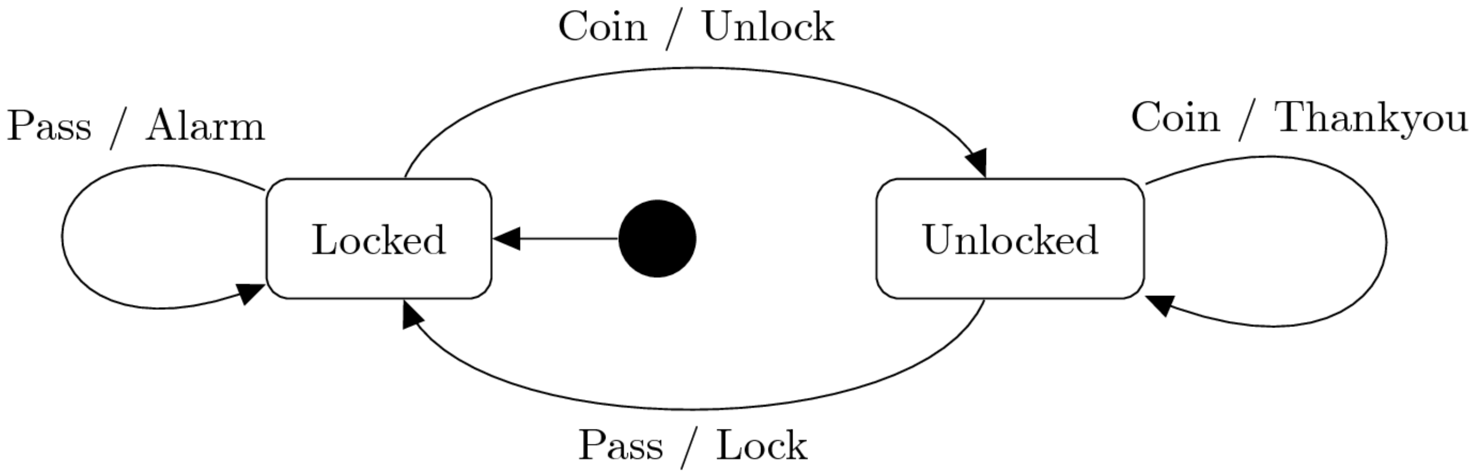
Basic functionality as Moore-type



Example by R.C. Martin

Subway Turnstile (Abnormal Conditions)

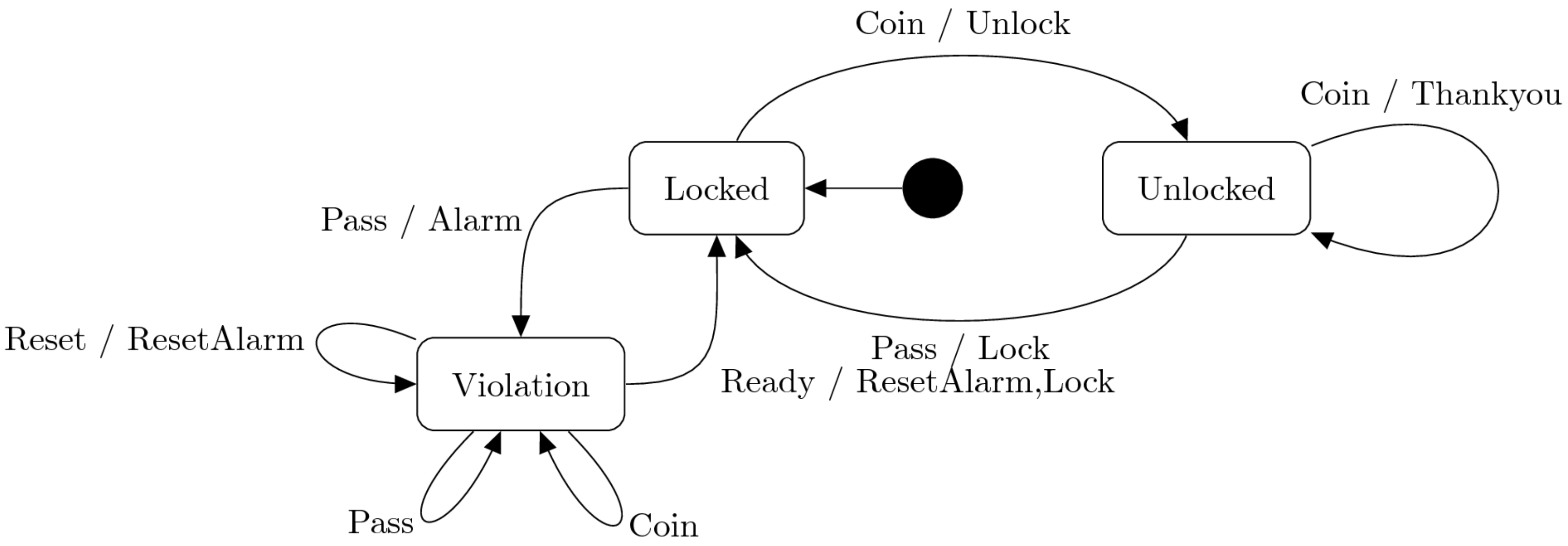
- Abnormal conditions: Turnstile is
  - in state `Locked`, but user passes through anyway
  - already unlocked and customer deposits another coin
- Mealy-type required



„Entry action“ in state `Locked` depends on previous state and input! → Mealy type FSM

Subway Turnstile (Violation State)

- Turnstile remains in state `Violation` until a repairman signals that turnstile is ready for service
- Only way out of `Violation` state is through `Ready` event
- Special event `Reset` that technician can use to turn alarm off while working on the turnstile

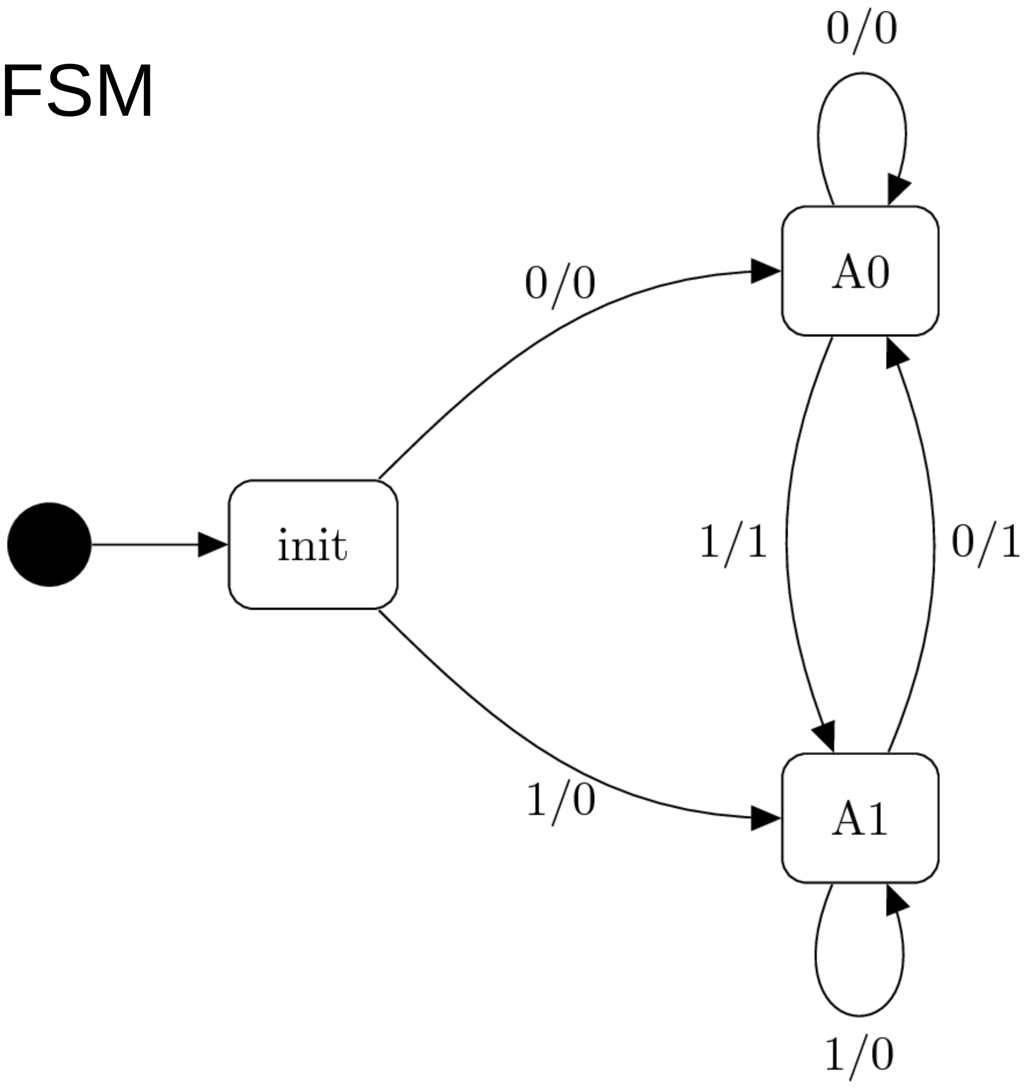


Example: Edge-Detector

- Function of an edge detector is to detect transitions between two symbols 0 and 1 in an input sequence
- Output:
  - 0 as long as most recent input symbol equals previous one
  - However, when most recent one differs from previous one, output is 1
- Convention:
  - Edge detector always outputs 0 after reading very first symbol

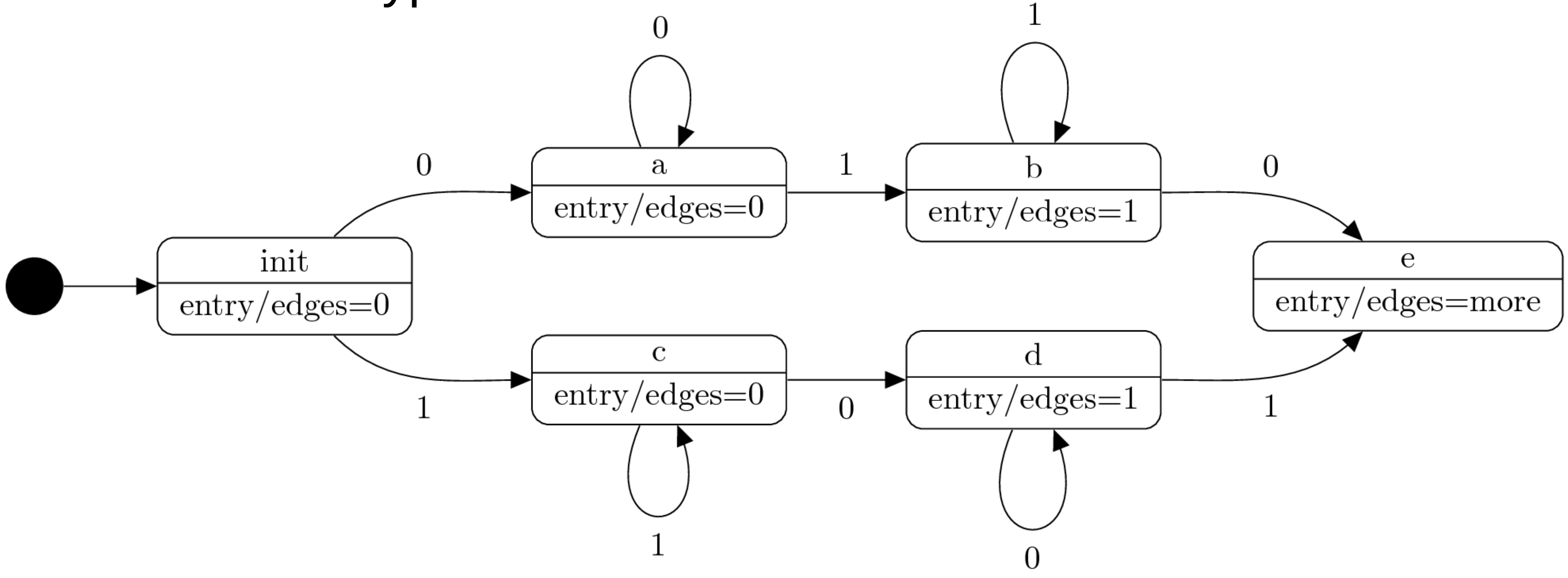
Example: Edge-Detector

- Solution: Mealy-type FSM



Example: Edge-Counter

- Extend Edge-Detector to a counter
  - Categorize input as to whether input so far contains 0, 1, or *more* than 1 edges (an edge is a 0-1,or a 1-0 transition)
  - Solution: Moore-type FSM



Example: Real Edge-Counter

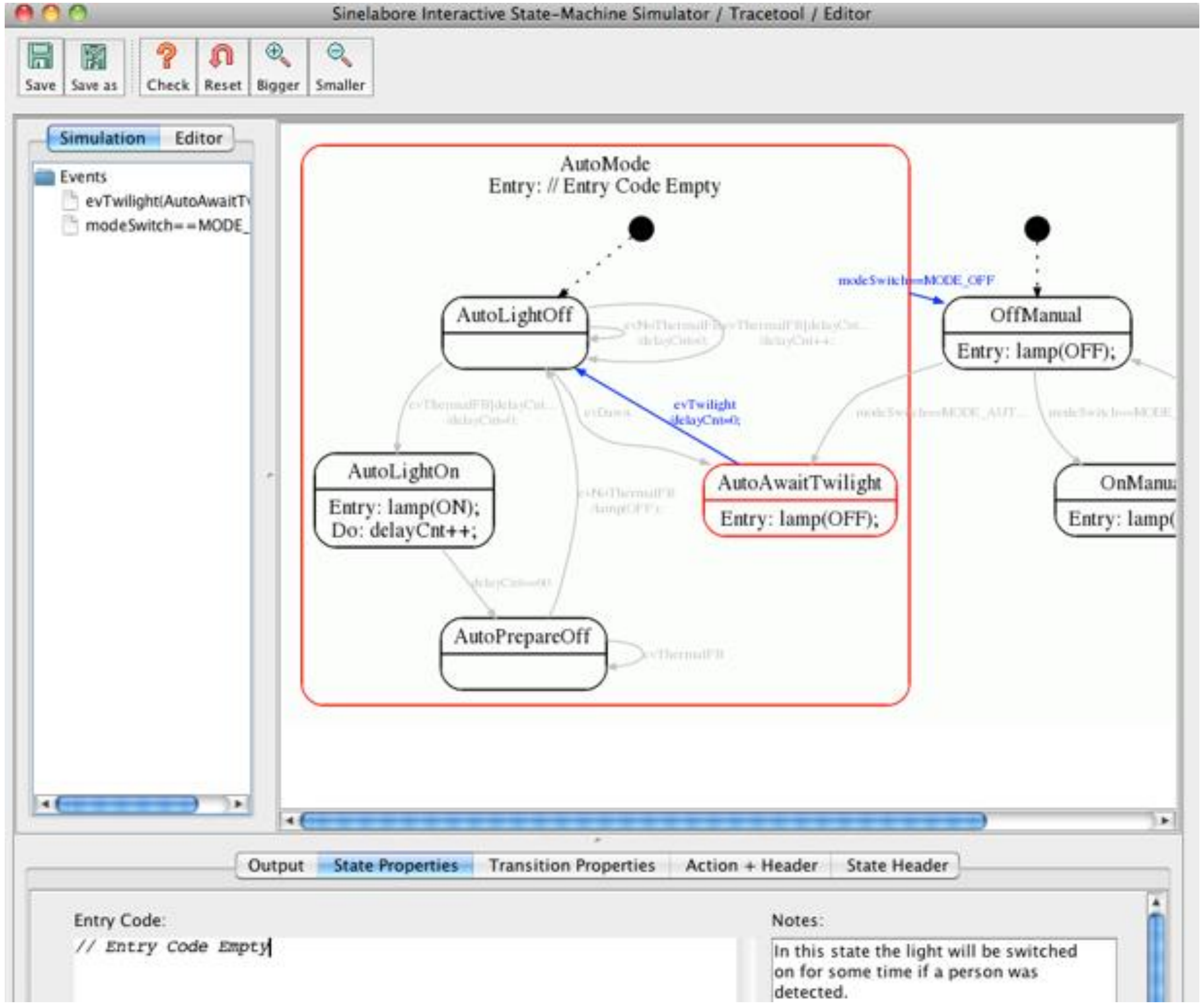
4

Implementing FSM

Implementing an FSM

- Despite benefits of FSMs, most popular development tools use sequential programming language
  - C, C++, Java, Ada, VHDL, Verilog, etc.
- Approaches to capture a FSM with sequential programming language
  - Front-end tool approach
    - Additional tool to support state machine language
      - Graphical and/or textual state machine languages
      - May support graphical simulation
      - Automatically generates code in sequential programming language
  - Direct implementation approach
    - Most common approach ...

Example: Graphical Tool



Source: http://www.sinelabore.com

FSM Switch Implementation

- Template to transform FSMs into equivalent sequential language programs
- Used with software (e.g.,C) and hardware languages (e.g.,VHDL)
- General procedure
  - Enumerate all states (enum)
  - Declare state variable initialized to initial state (e.g. IDLE)
  - Single switch statement branches to current state's case
  - Each case has actions
    - e.g. move, dir, open, timer_start
  - Each case checks transition conditions to determine next state using input variables
    - if (…) {state = …;}

```
enum states {IDLE, GOINGUP, GOINGDN, DOOROPEN};

void UnitControl() {
   enum states state = IDLE;
   while (1) {
      switch (state) {
         case IDLE: move=NO; open=YES; timer_start=0;
            if (reqFloor == curFloor) {state = IDLE;}
            if (reqFloor > curFloor)  {state = GOINGUP;}
            if (reqFloor < curFloor)  {state = GOINGDN;}
            break;
         case GOINGUP: dir=UP;open=NO; move=YES;timer_start=0;
            if (reqFloor > curFloor)     {state = GOINGUP;}
            if (! (reqFloor > curFloor)) {state = DOOROPEN;}
            break;
         case GOINGDN: dir=DOWN;open=NO; move=YES;timer_start=0;
            if (reqFloor < curFloor)     {state = GOINGDN;}
            if (! (reqFloor < curFloor)) {state = DOOROPEN;}
            break;
         case DOOROPEN: move=NO; open=YES; timer_start=1;
            if (timer < 10)    {state = DOOROPEN;}
            if (! (timer<10)) {state = IDLE;}
            break;
         default:
            assert(…);
      }
   }
}
```

UnitControl state machine in sequential programming language

FSM Switch Implementation

```
enum states {S0, S1, …, SN}

void StateMachine() {
  enum states state = S0; // or whatever is the initial state.
  while (1) {
    switch (state) {
    case S0:
      // Insert S0’s entry actions here
      // Insert transitions Ti leaving S0, can be implemented as switch statement
      // or encapsulated in function
      if( T0’s condition is true) {state = T0’s next state; /* transit. action */}
      if( T1’s condition is true) {state = T1’s next state; /* transit. action */}
      ...
      if( Tm’s condition is true) {state = Tm’s next state; /* transit. action */}
      break;
    case S1:
      // Insert S1’s entry actions here
      // Insert transitions Ti leaving S1
      break;
      ...
    case SN:
      // Insert SN’s entry actions here
      // Insert transitions Ti leaving SN
      break;
    }
  }
}
```

FSM Switch Implementation

- Switch statement method
  - Simple
  - Requires enumerating states and events
  - Has a small (RAM) memory footprint
    - One scalar variable required
  - Does not promote code reuse
  - Event dispatching time is not constant
    - Increases with the number of cases (number of comparisons)
  - Implementation can be difficult to maintain against changes in the state machine

FSM Function Pointer Implementation

- Represents concept of "state" by a function pointer
- `State` is not enumerated — it is a pointer to a state-handler function
- `Event` is basis to define more complex events
- `Fsm` stores current state in its attribute `state`
- Two "inlined" methods
  - `fsm_dispatch` dispatches events to the state machine
  - `fsm_init` triggers the initial transition
- Each function takes pointer to structure `(Fsm *)` as first argument

FSM Function Pointer Implementation

- Code for Mealy-type FSM

```
/** fsm.h */
#ifndef FSM_H_
#define FSM_H_
typedef struct Event Event;
typedef struct Fsm Fsm;

/* a state is represented by a function pointer, called for
 * each transition emanating in this state */
typedef void (*State)(Fsm *, const Event *);

/* base type for state machine */
struct Fsm {
   State state; /* current state */
};
```

FSM Function Pointer Implementation

```
/* base type for events*/
struct Event {
   int signal;
};

/* dispatches events to state machine, called in application*/
inline static void fsm_dispatch(Fsm * fsm, const Event * event) {
   fsm->state(fsm, event);
}

/* sets and calls initial state of state machine */
inline static void fsm_init(Fsm * fsm, State init) {
   fsm->state = init;
   fsm_dispatch(fsm, NULL);
}

#endif /* FSM_H_ */
```

Example: Keyboard

- Console application
  - Driven from a keyboard with keys "^" and "6“
  - Emulate pressing and releasing shift, respectively
  - Pressing "." terminates test
- Essential elements
  - State machine initialization, and dispatching events
  - Explicit constructor call
- Note necessity of explicit casting (upcasting) to `Fsm` superstruct
  - C compiler doesn't *know* that `Keyboard` is related to (derived from) `Fsm`

Example: Keyboard

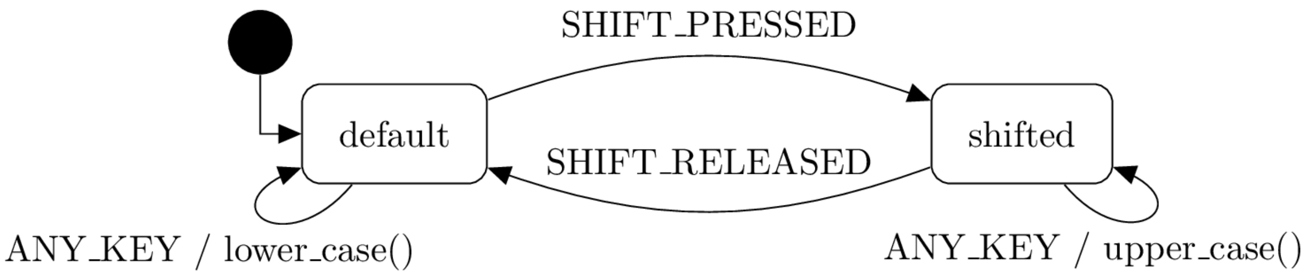
```c
/** main.c */
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include "fsm.h"

typedef struct {
    Fsm super;
} KeyboardFsm;

typedef struct {
    Event super;
    char code;
} KeyboardEvent;

/* signals used by the keyboard FSM */
enum { SHIFT_PRESSED_SIG, SHIFT_RELEASED_SIG, ANY_KEY_SIG };

void keyboard_init(Fsm * fsm, const Event * e);
void keyboard_default(Fsm * fsm, const Event * e);
void keyboard_shifted(Fsm * fsm, const Event * e);
```



Example: Keyboard

```
void keyboard_init(Fsm * fsm, const Event * e) {
  fsm->state = keyboard_default;
  printf("init\n"); // debug info
}

void keyboard_default(Fsm * fsm, const Event * e) {
  switch (e->signal) {
  case SHIFT_PRESSED_SIG:
    printf("default::SHIFT_PRESSED\n"); // debug info
    fsm->state = keyboard_shifted;
    break;
  case ANY_KEY_SIG:
    printf("default::ANY_KEY (%c) \n", (char)
           tolower(((const KeyboardEvent *) e)->code));
    break;
  }
}
```

Example: Keyboard

```
void keyboard_shifted(Fsm * fsm, const Event * e) {
  switch (e->signal) {
  case SHIFT_RELEASED_SIG:
    printf("shifted::SHIFT_RELEASED\n"); // debug info
    fsm->state = keyboard_default;
    break;
  case ANY_KEY_SIG:
    printf("shifted::ANY_KEY (%c) \n", (char)
           toupper(((const KeyboardEvent *) e)->code));
    break;
  }
}
```

Example: Test Program

```
int main(void) {
  KeyboardFsm keyboard;
  fsm_init((Fsm *) &keyboard, keyboard_init);
  while (1) {
    KeyboardEvent e;
    e.code = getc(stdin);
    getc(stdin); // discard  \n
    switch (e.code) {
    case '^': e.super.signal = SHIFT_PRESSED_SIG; break;
    case '6': e.super.signal = SHIFT_RELEASED_SIG; break;
    case '.': return 0;
    default: e.super.signal = ANY_KEY_SIG; break;
    }
    fsm_dispatch((Fsm *)&keyboard, (const Event *)&e);
  }
  return 0;
}
```

FSM State Table Implementation

- State tables containing arrays of transitions for each state
- Content of cells are transitions, represented as pairs
  - `{action, next state}`

Signals→

States→

	SIGNAL_1	SIGNAL_2	SIGNAL_3	SIGNAL_4
STATE_X				
STATE_Y				
STATE_Z	action1() STATE_X			
STATE_A				

Handling Bursts of Events

- Events are assumed to occur at a rate that allows for processing (i.e. no event is missed)
- Alternative
  - All events are stored in queue
  - State machine receives its events from queue
- Advantages
  - Events are not lost
  - Event order is preserved

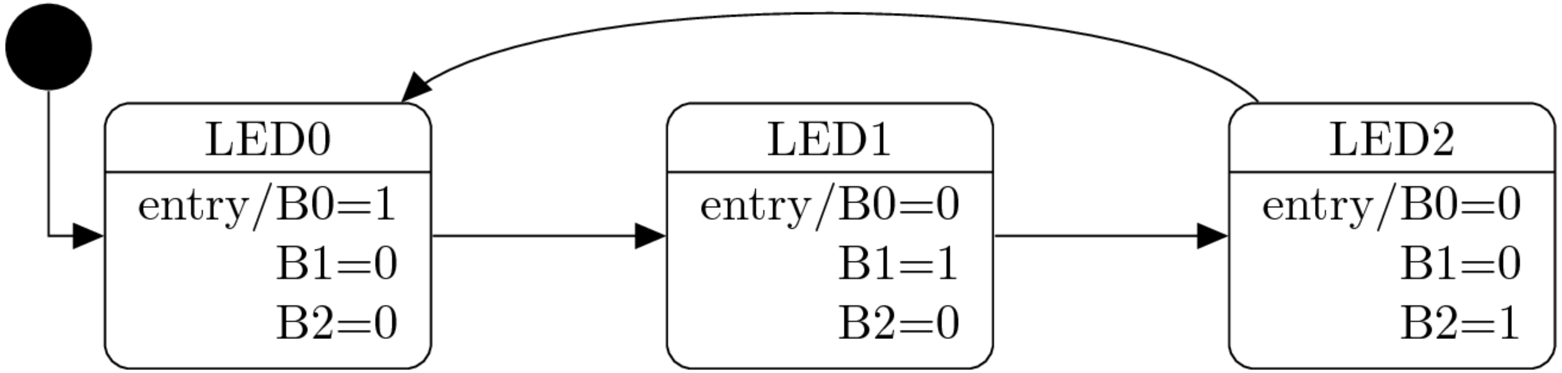
5

Synchronous FSM

Synchronous FSM

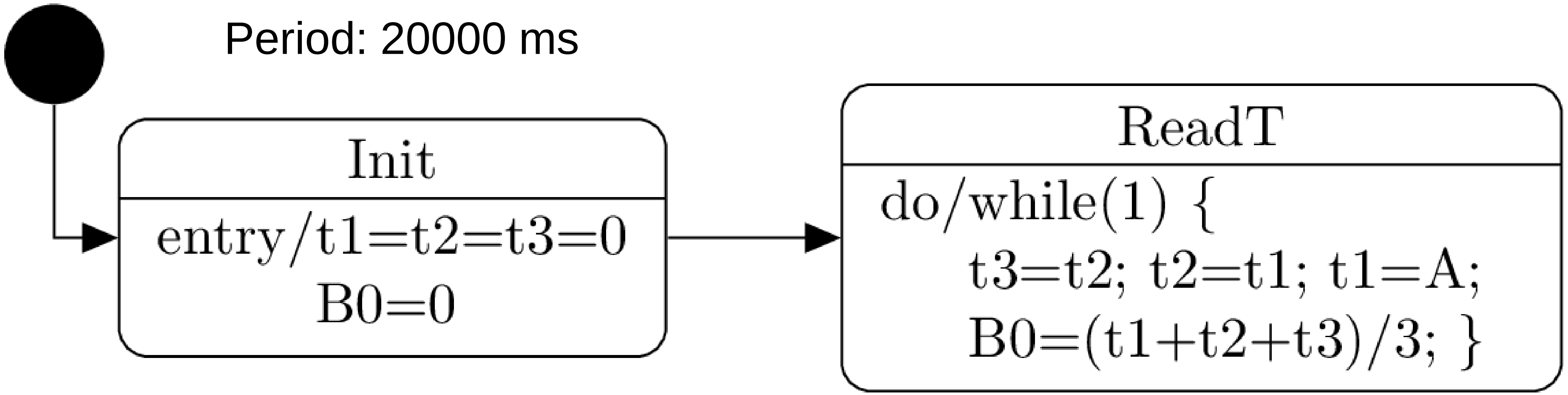
- Synchronous FSMs are time-triggered
- Transitions are performed at fixed period
- Example
  Light three LEDs connected to B0, B1, B2, one LED per second in sequence (LED appears to move)

Period: 1000 ms



Example: Averaging Values

- Output average of last 3 temperature readings
- Read temperature every 20 s from input A
- Notation do/while(1): execute periodically as long as modeled element is in state

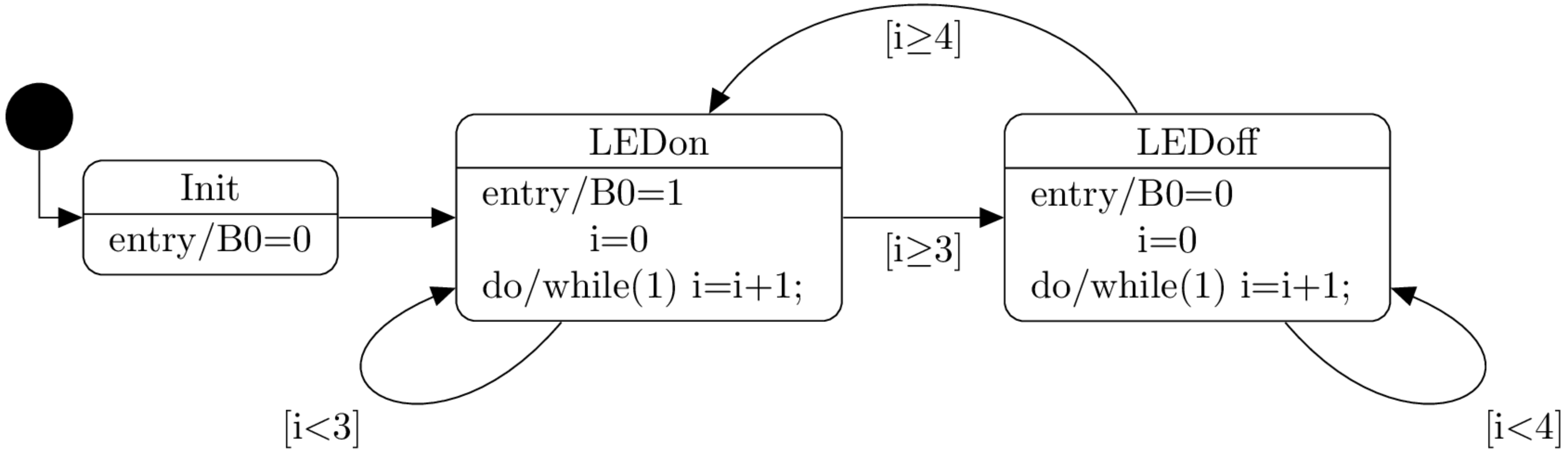


- What to do if different periods are involved?

Example: Different Periods

- A system should repeatedly blink an LED for 750 ms on and off for 1 second

Period: 250 ms



Implementation of Synchronous FSM

- Use ISR in combination with timer component
- Example:

```
volatile uint8_t timerFlag;

void timerISR() {
   timerFlag = 1;
}

void main(void) {
   timerSet(250);   // inits timer
   timerOn();       // starts timer
   ......
   while (1) {
      ......  // switch statement goes here
      while (! timerFlag) { }    // active waiting
      timerFlag = 0;
   }
}
```

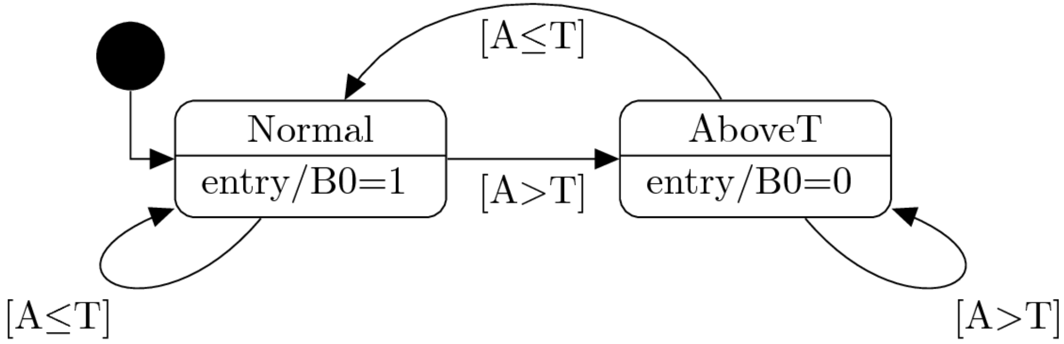
Exercise: A crosswalk system initially illuminates a *don't walk symbol* (B0=1). When a pedestrian presses button A0, the system illuminates a *walk symbol* (B1=1) for 6 s and then for 4 seconds the *don't walk symbol* blinks. Button presses of length 500 ms should be detected.

- Alternative: Use scheduler!

Example: Sampling

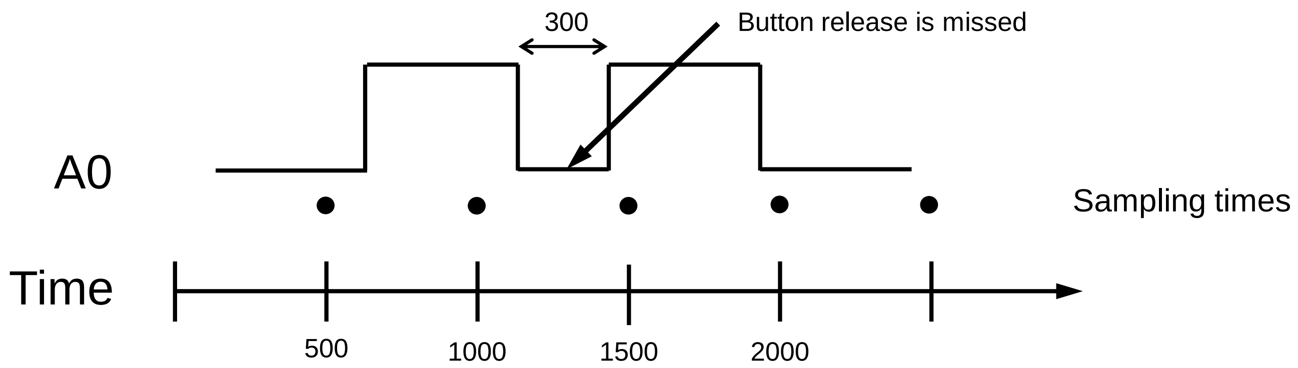
- Reading a sensor at specified period is called sampling
  - Period is called sampling rate
- Example: Audio System
  - System prevents an audio speaker from being damaged by disabling speaker (B0=0) if input level exceeds threshold T
  - Audio level is detected by an 8 Bit sensor connected to A
  - Speaker is damaged if A > T for more than 500 ms
  - Does this Synchronous FSM do the job?

Period: 500 ms



Sampling Rate

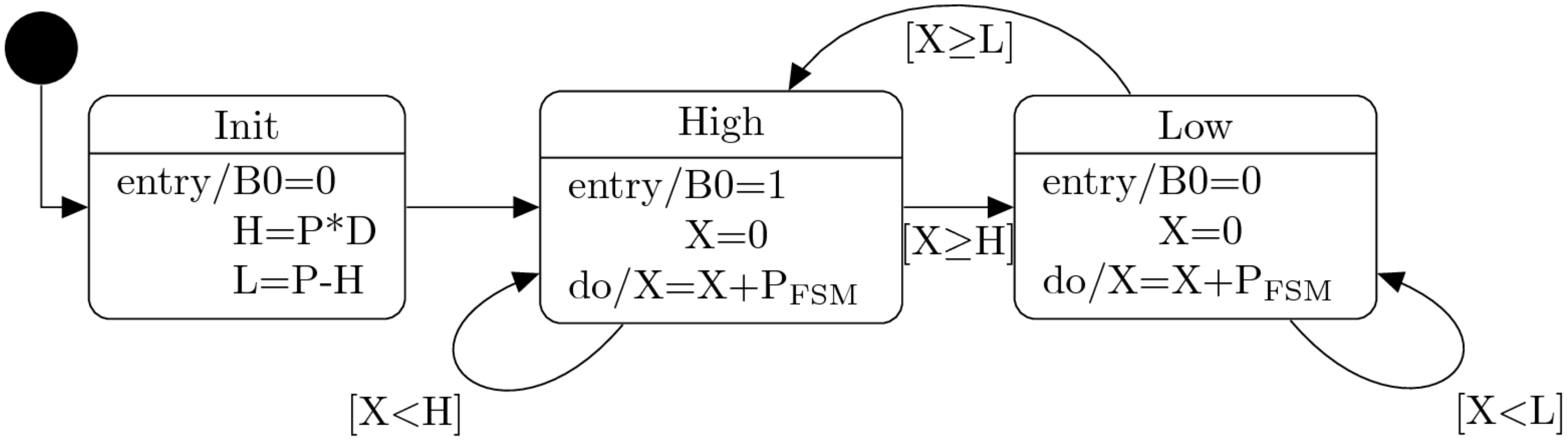
- Ideally, a system samples input as fast as possible
- If sampling rate is too high, instructions to carry out transitions may not finish, possibly resulting in erroneous execution
- Minimum event separation time (MEST):
  - smallest time between any two input events
- Period of FSM must be smaller then MEST



Pulse Width Modulation

- Period P of PWM is multiple of 1 s
- Duty cycle D is multiple of 10 %
- Output: 1 or 0

Period P_{FSM} of FSM: 100 ms



Example: P = 2000 ms, D = 0.2 → H = 400 L = 1600

6

HCFSM and Statecharts Language

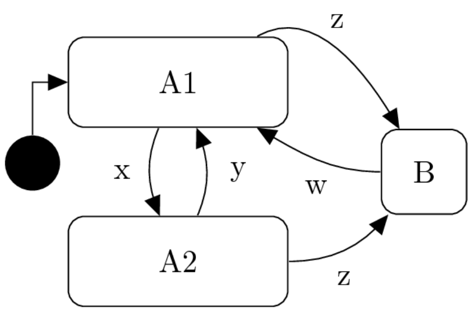
Introduction

- Traditional FSMs are good for smaller problems
- But: Tendency to become unmanageable even for moderately involved systems
- Phenomenon "state explosion“
  - Complexity of a traditional FSM tends to grow much faster than complexity of reactive system it describes
  - Reason: FSM formalism inflicts repetitions
- Solution: HCFSM
  - Hierarchical/concurrent state machine model

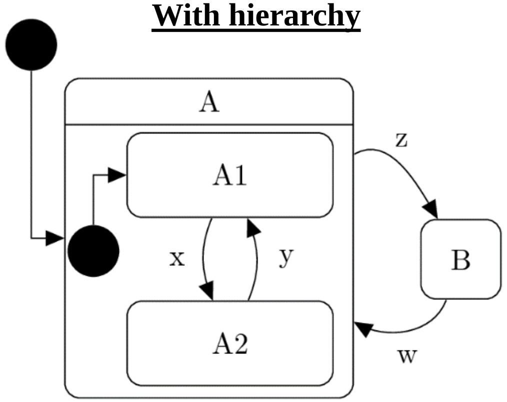
HCFSM

- HCFSM: Extension to support hierarchy and concurrency
  - If a system is in a nested state (a substate), it also (implicitly) is in surrounding state (the superstate)
  - States can be decomposed into other state machines
    - *With hierarchy* has identical functionality as *Without hierarchy*, but has less transitions
    - Known as OR-decomposition
  - States can execute concurrently
    - Known as AND-decomposition

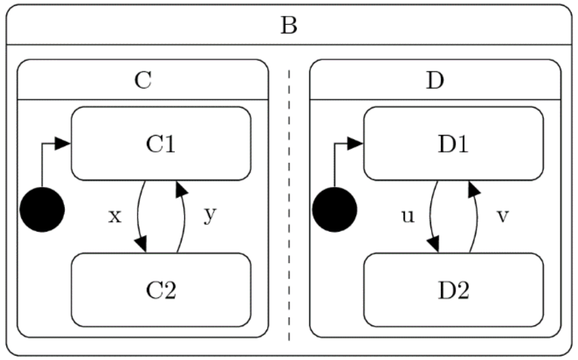
Without hierarchy



With hierarchy



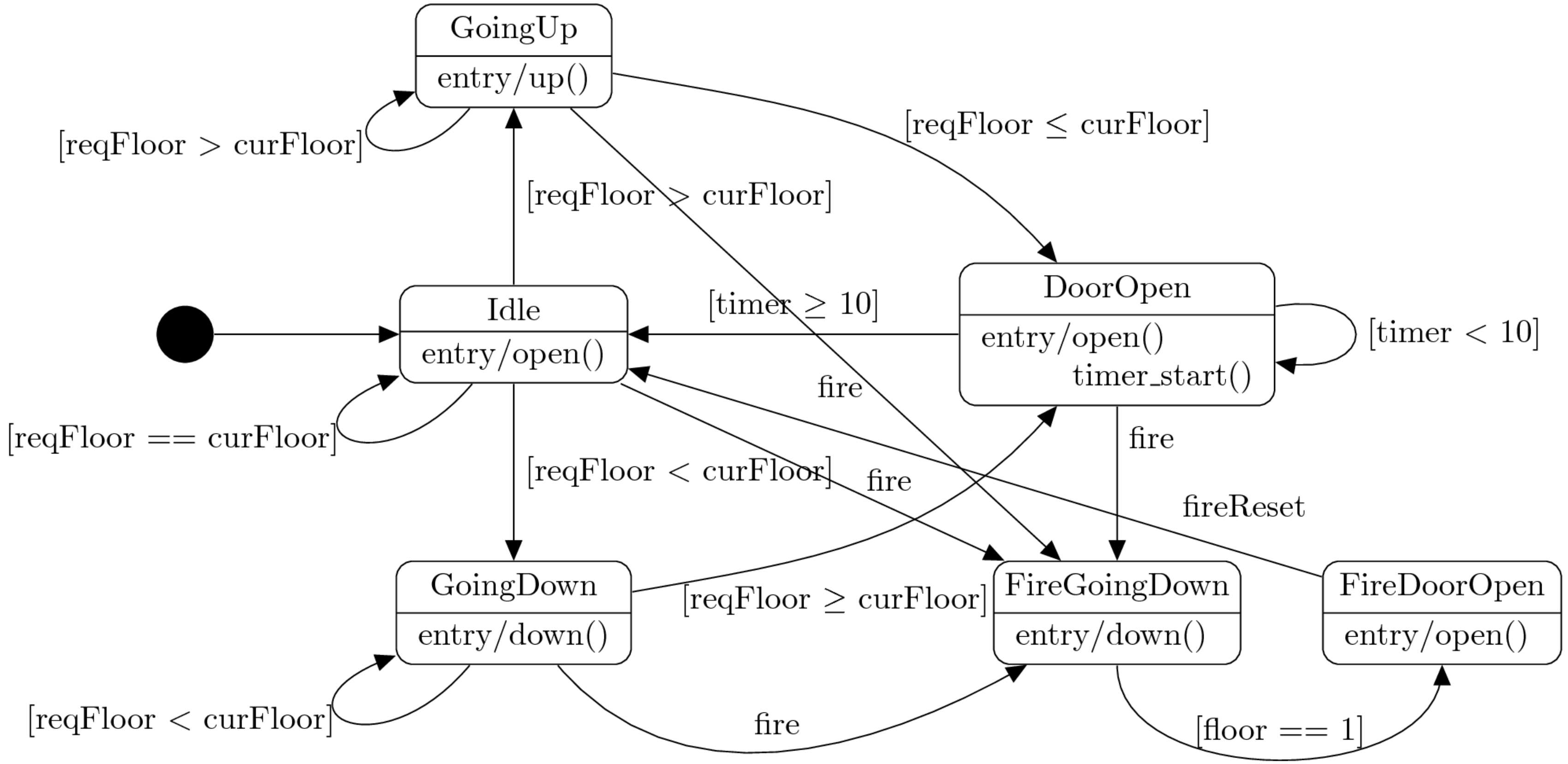
Concurrency



Example: UnitControl with FireMode

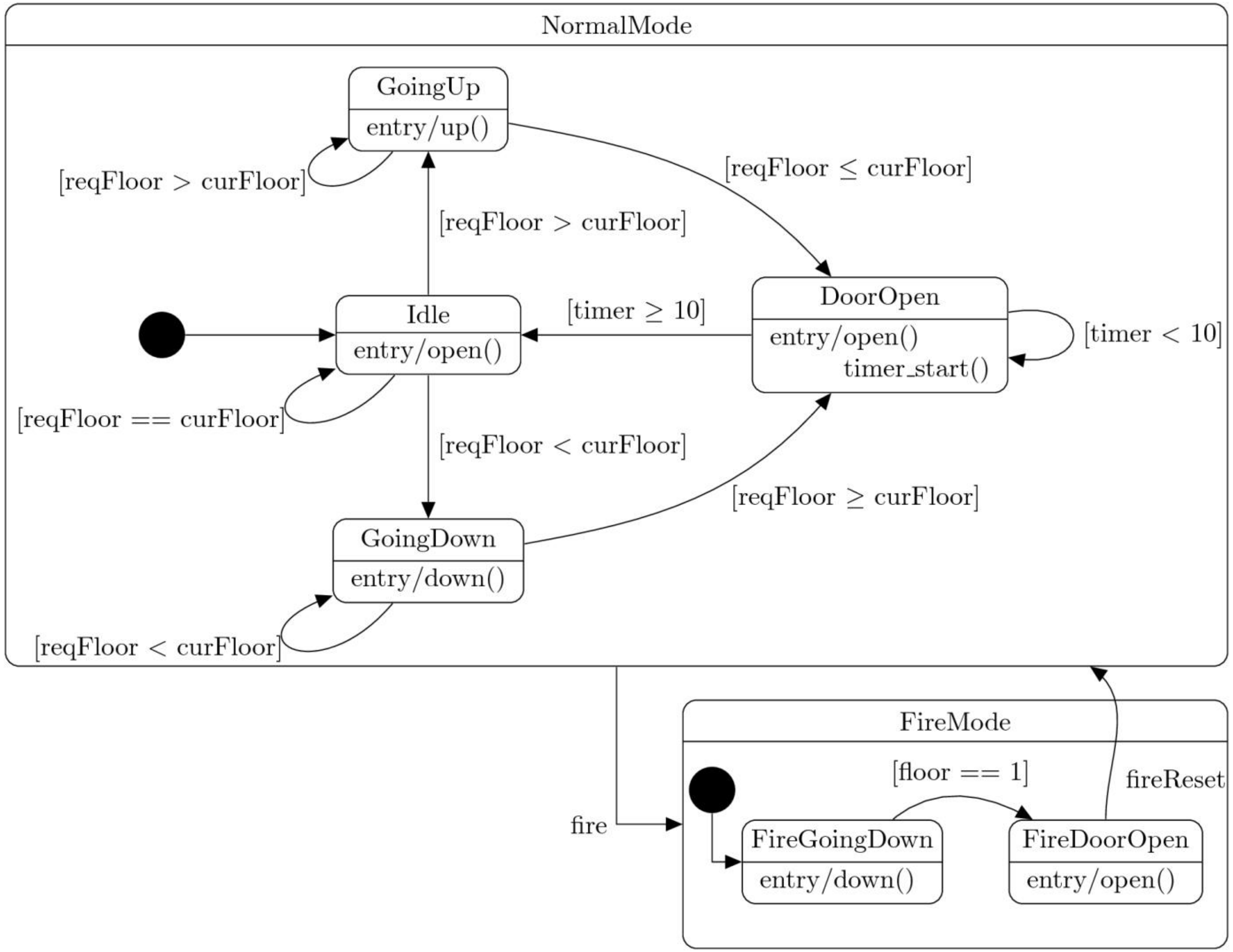
FireMode

- When *fire* is true, move elevator to 1st floor and open door



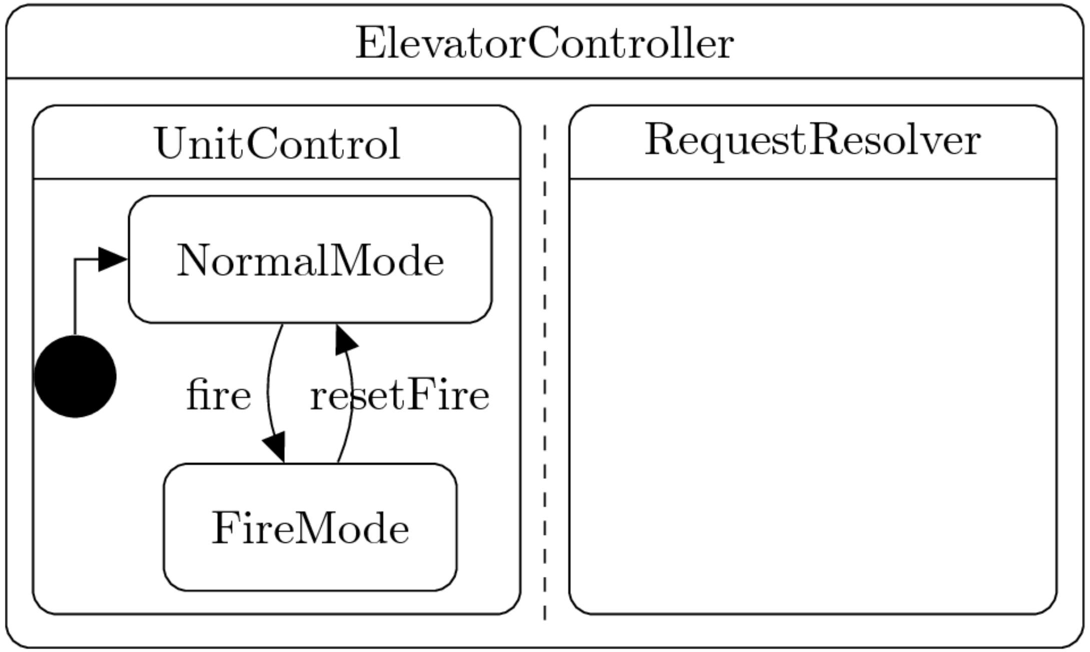
Example: UnitControl with FireMode

FireMode with hierarchy

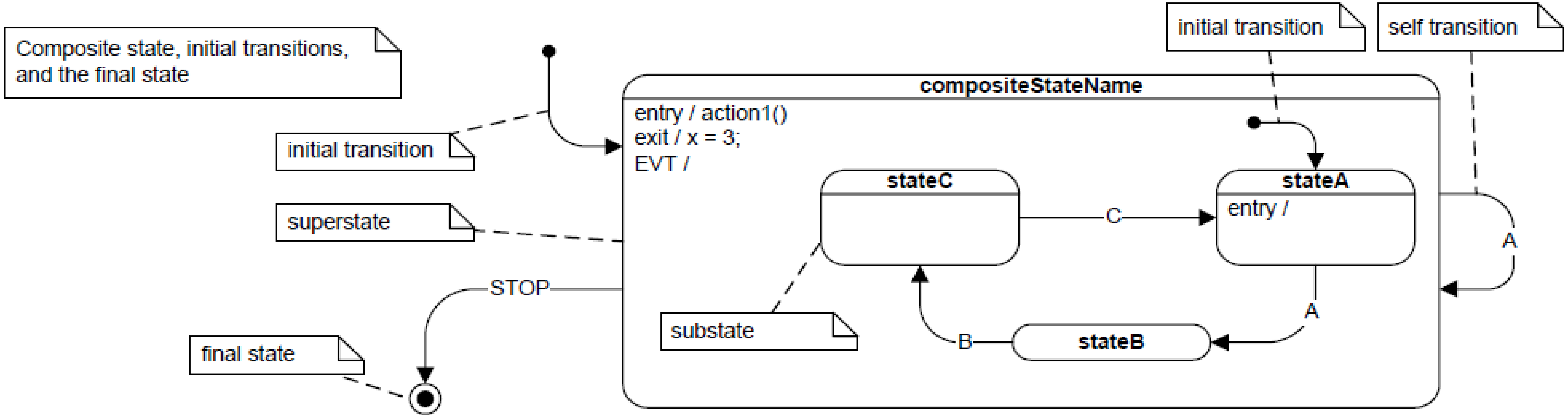


Example: UnitControl with FireMode

FireMode with concurrent RequestResolver

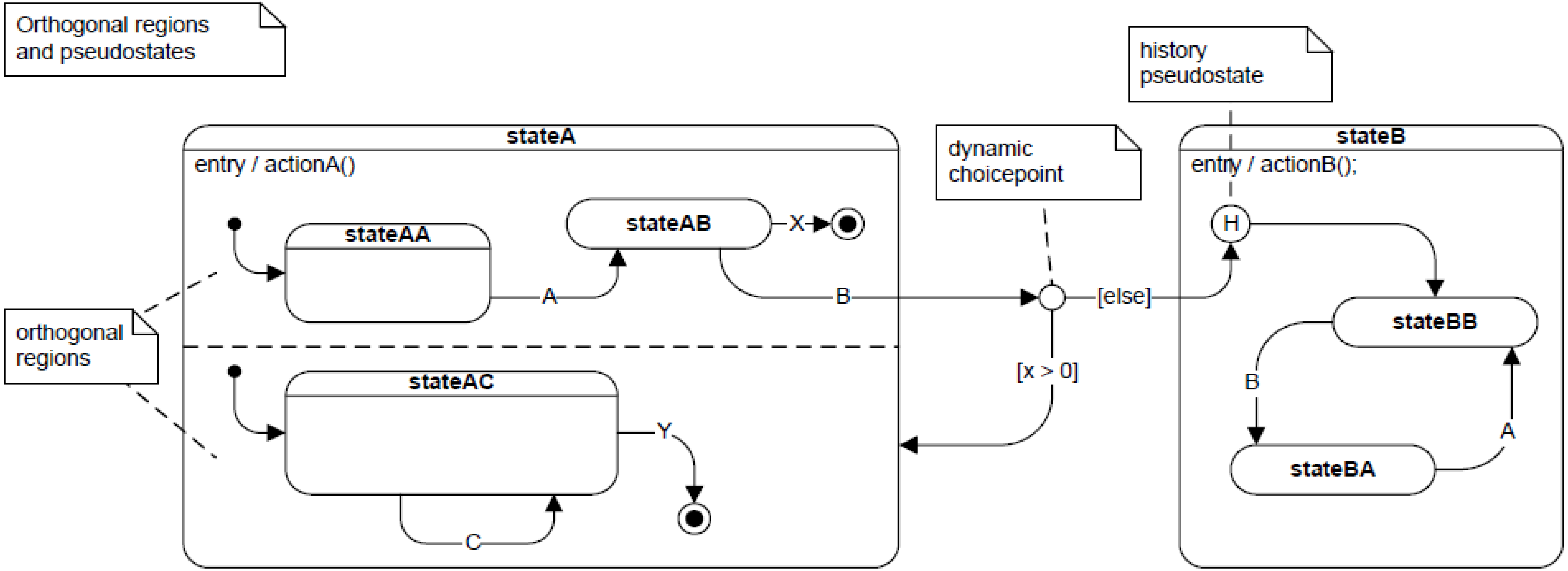


UML Notation for HCFSM



- Unified Modeling Language (UML)
  - Hierarchical states are called composite states
  - Self transitions
  - Notation for final state

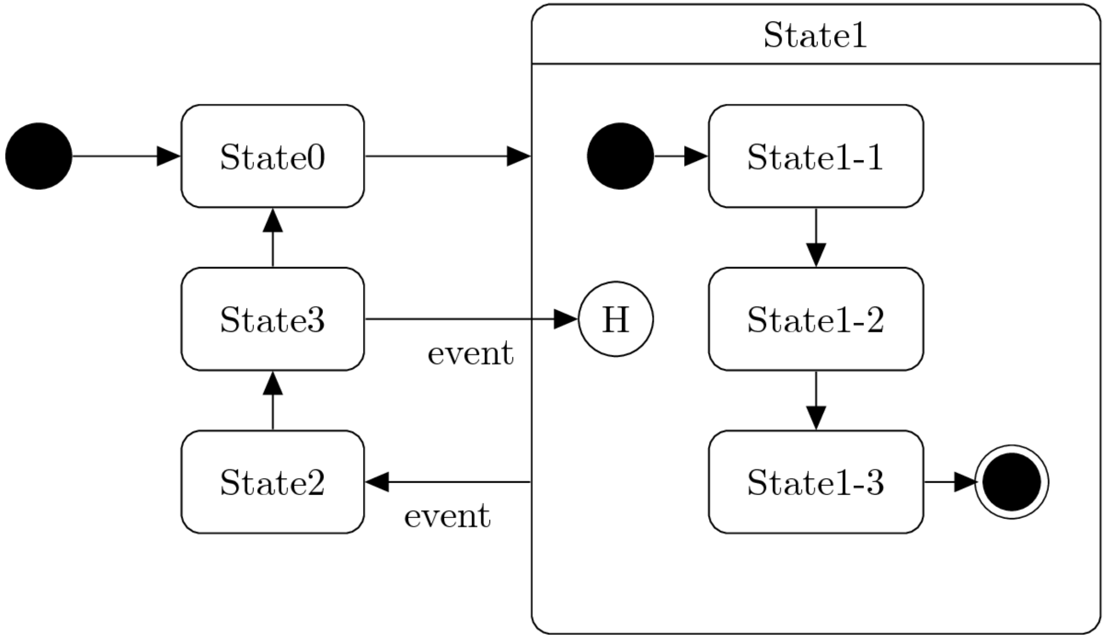
UML Notation for HCFSM



- Unified Modeling Language (UML)
  - Concurrent states are called orthogonal states
  - Dynamic choice points
  - History state
  - *timeout*: transition with time limit as condition

History pseudo state (H)

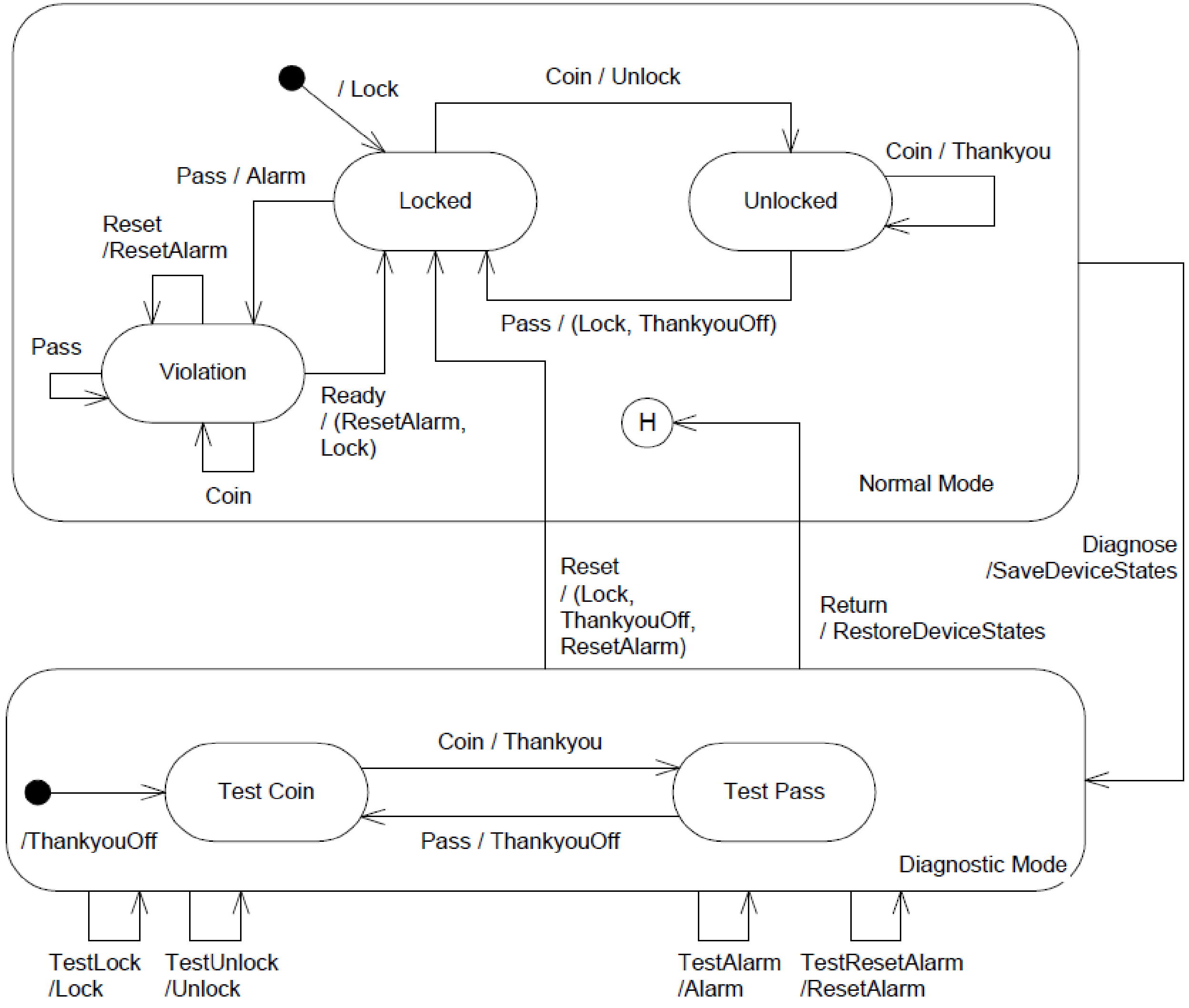
- Last substate OR-decomposed state *A* was in before leaving A
- Return to saved substate of A when returning from other state instead of initial state



Subway Turnstile

Maintenance technicians want to put turnstile into a special maintenance mode so that they can check out its functions

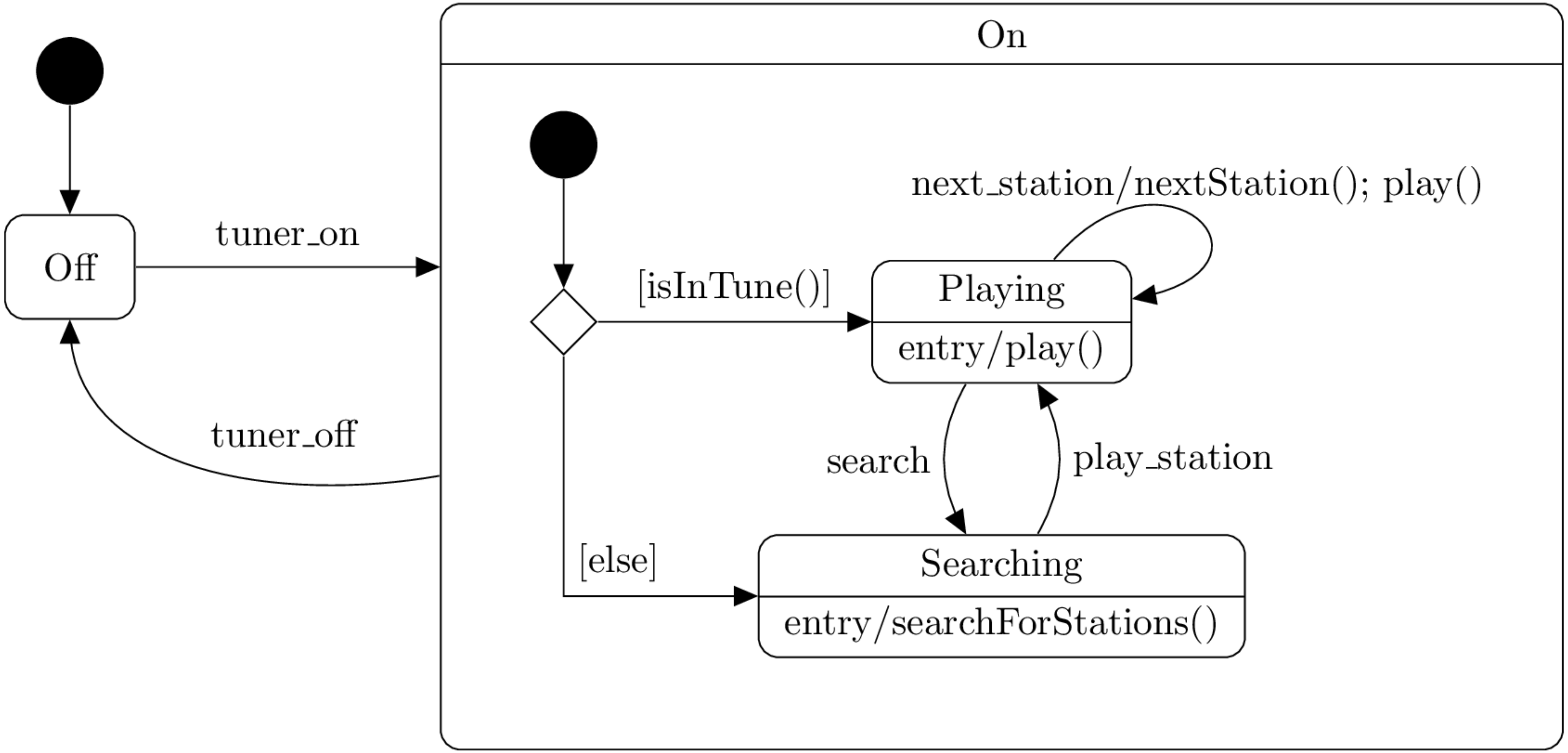
History Pseudo State (H):
It indicates that the substate within Normal Mode to be entered is the substate within Normal Mode that was last exited



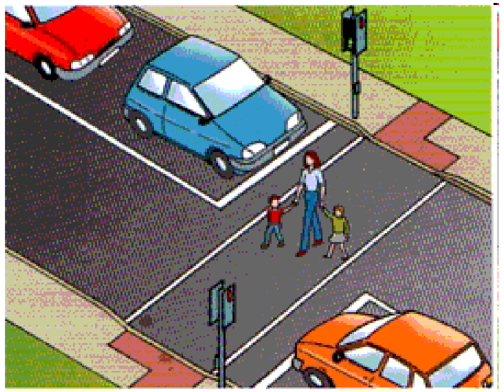
Example: Tuner

- Tuner can be in two states: *Off* or *On*
- If tuner is in state On, it is in one of the substates
  - Playing
  - Searching
- If tuner is in state *Playing*, it can be switched to
  - next station by sending event *next_station*, or to
  - state Searching
- If tuner is in state *Searching* it searches for stations and returns to state *Playing* when it receives event *play_station*
- Upon entering state *On* operation *isInTune()* is called to decide if tuner must enter state *Playing* or *Searching*

Dynamic Choice Points

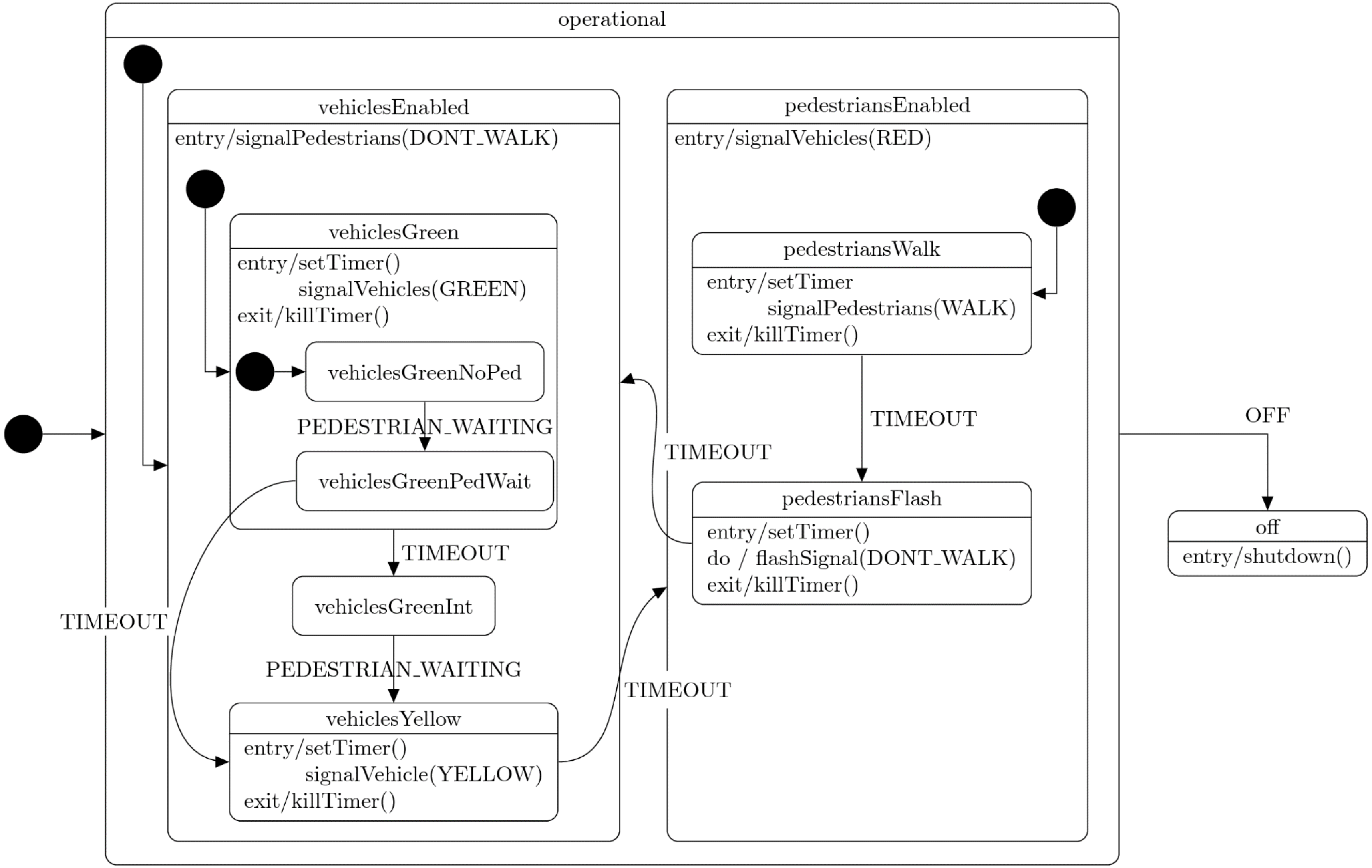


Example: Pelican Crossing



- Pelican crossing: Pedestrian crossing - traffic lights with push buttons and two colored lamps for pedestrians
- PELICAN / PELICON:
  PEdestrian LIght CONtrolled crossing
- Nominally, vehicles are enabled and pedestrians disabled
- To activate traffic light switch, a pedestrian must push a button (Event PEDESTRIAN_WAITING)
- In response, vehicles get yellow light
- After a few seconds, vehicles get a red light and pedestrians get a WALK signal, which after fixed time span changes to a flashing DON'T WALK signal
- When DON'T WALK signal stops flashing, vehicles get green light
- After this cycle, traffic lights don't respond to PEDESTRIAN_WAITING event immediately, although the button "remembers" that it has been pushed
  - Purpose: The traffic light controller always gives vehicles a specific duration of green light before repeating the cycle

Example: Pelican Crossing

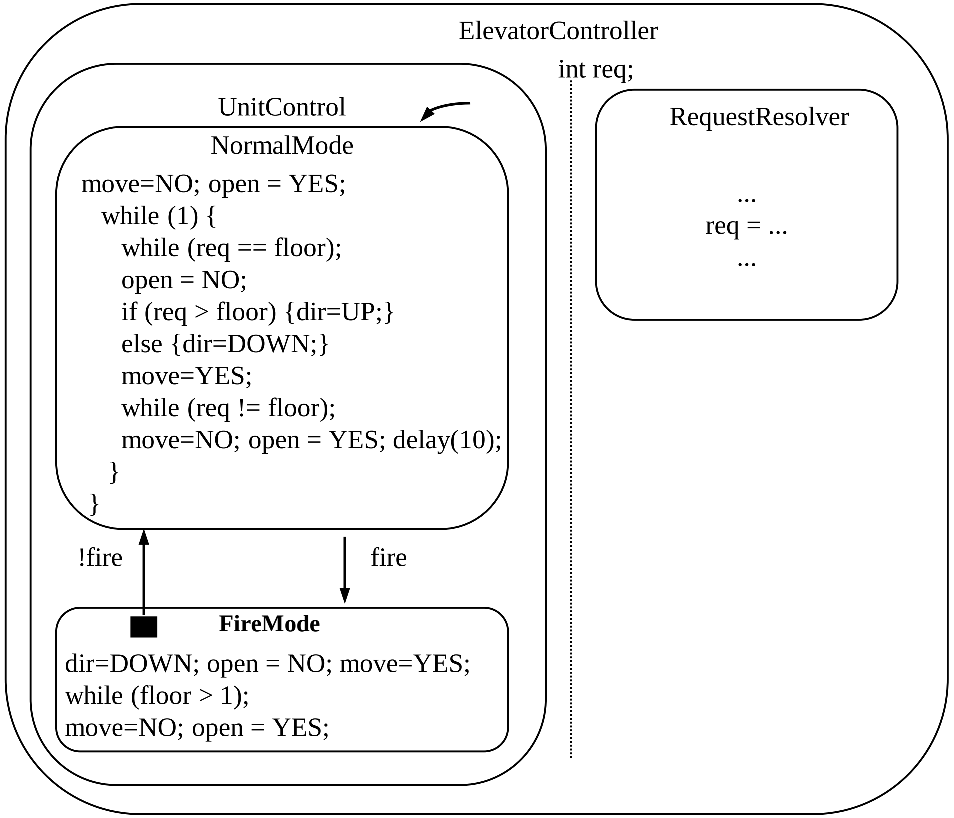


7

Program-State Machine (PSM) Model

Program-state Machine Model (PSM)

- Program-state's actions can be FSM or sequential program
- Stricter hierarchy than HCFSM
  - Transition between sibling states only, single entry
  - Program-state may "complete"
    - Reaches end of sequential program code, or
    - FSM transitions before end of code is reached
  - 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
- SpecC: C-Extension 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

Advantages of FSMs

- Useful in all development phases
  - allow finding defects already in design phase
- Allow simulation of modeled behavior
  - Execution of a FSM in a simulator allows user to generate events and observe how FSM reacts
- Allow automatic code-generation
  - Reduces coding errors
- Robustness can be automatically checked on model level
  - State names must be unique
  - States must be connected by a sequence of transitions outgoing from an initial state
  - Initial states must be defined on every state hierarchy and must have exactly one outgoing transition
  - Final states must only have incoming transitions
- Test cases can be automatically derived

8

Summary

Summary

- Computation models are distinct from languages
- Sequential program model is popular
  - Most common languages like C support it directly
- State machine models good for control
  - Extensions like HCFSM provide additional power
  - PSM combines state machines and sequential programs

Exercise I: Wrist Watch

- Design a FSM for a wrist-watch with the following behavior
  - The watch has a chronograph feature and is controlled by three buttons, *A*, *B*, and *C*
  - It has three display modes:
    - time of day
    - chronograph time (see http://en.wikipedia.org/wiki/Chronograph)
    - "split" time, a saved version of the chronograph time
  - Assume that in initial state, the watch displays time of day
  - If button *C* is pressed, it displays chronograph time
  - If *C* is pressed again, it returns to displaying time of day
  - When watch is displaying chronograph time or split time
    - Pressing *A* starts or stops the chronograph
    - Pressing B when chronograph is running causes chronograph time to be recorded as split time and displayed
    - Pressing B again switches to displaying chronograph
    - Pressing B when chronograph is stopped resets chronograph time to 0

Exercise II: Vending Machine

- A vending machine vends soft drinks that cost \$0.40
- Machine accepts coins in denominations of \$0.05, \$0.10, and \$0.25
- When sufficient coins have been deposited, the machine enables a drink to be selected and returns appropriate change
- Considering each coin deposit and the depression of the drink button to be inputs, construct a FSM
- The outputs will be signals to vend a drink and return coins in selected denominations
- Assume that once machine has received enough coins to vend a drink, but the vend button has still not been depressed, that any additional coins will just be returned

Institute smartPORT
Hamburg University of Technology

TUHH