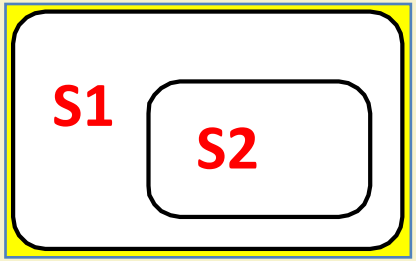
Some Important Features of State charts

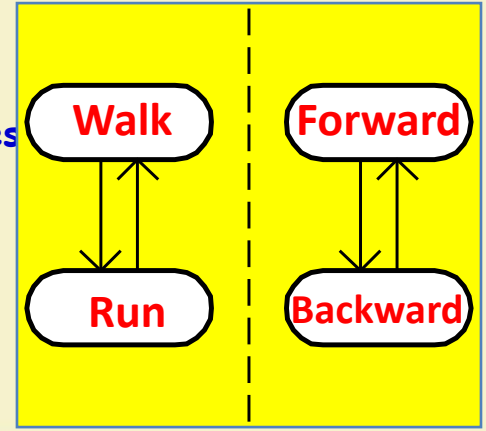
- Two major features:
  - **Nested states**
  - **Concurrent states**

Nested State



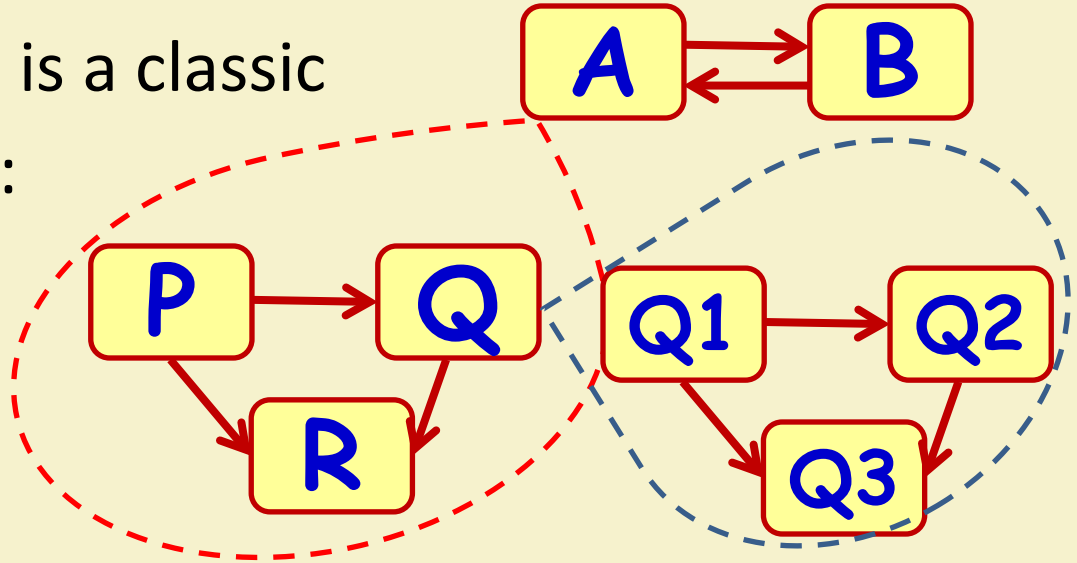
- Several other features have also been added:
  - History state
  - Broadcast messages
  - Actions on state entry and exit

Concurrent States

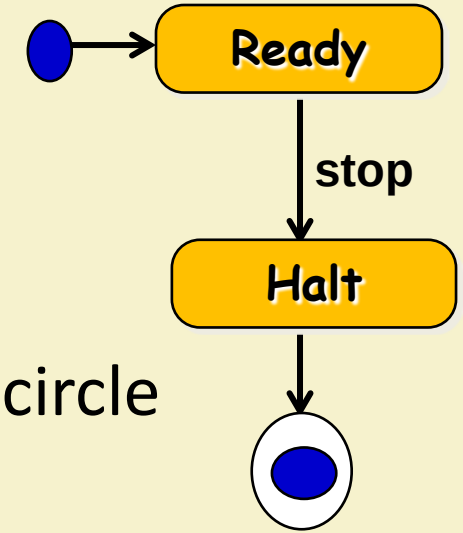


Nested States

- Hierarchical organization is a classic way to tackle complexity:
  - **superstates**
  - **substates**



State Chart Diagram



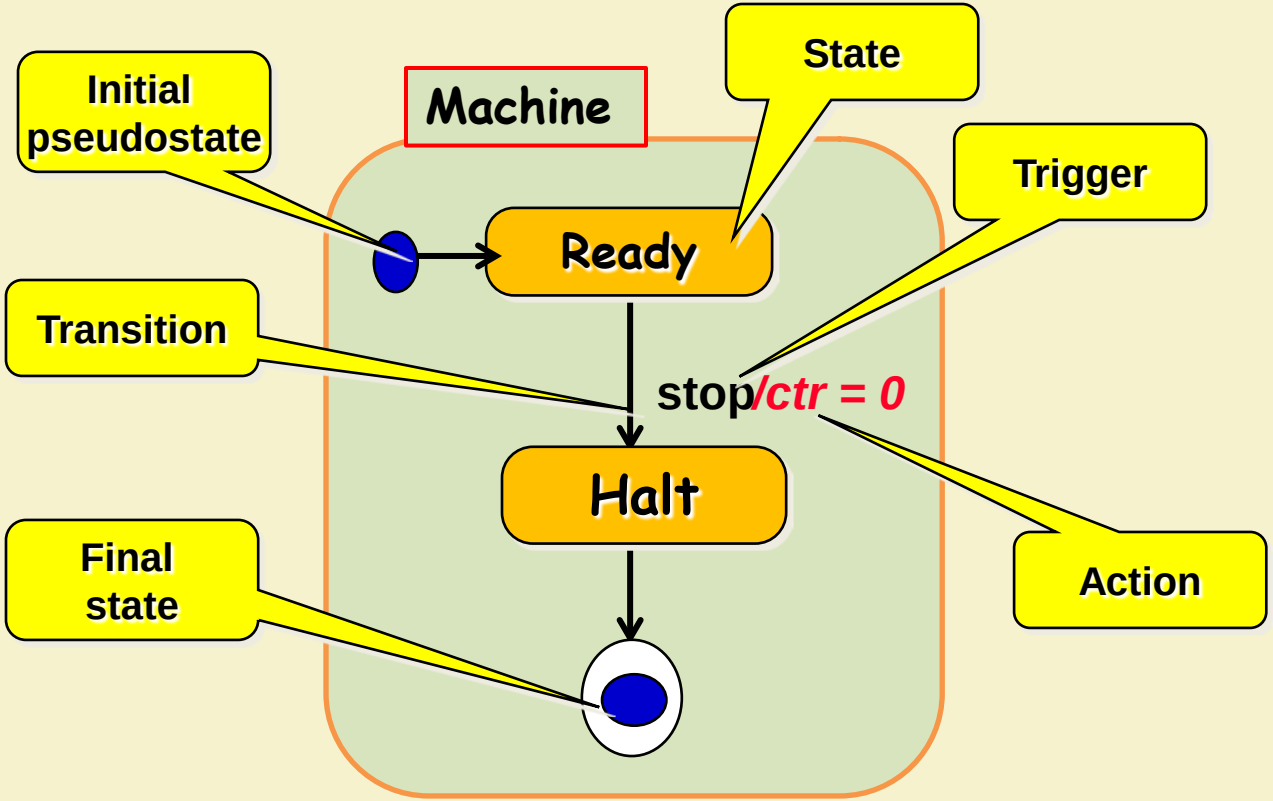
- Elements of state chart diagram:
  - **Initial State:** A filled circle
  - **Final State:** A filled circle inside a larger circle
  - **State:** Rectangle with rounded corners
  - **Transitions:** Arrow between states, also boolean logic condition (guard)

- A state is drawn with a round box:
  - Labelled with name of the state.
- A transition is drawn with a labeled arrow,
  - Event
  - Guard
  - Action

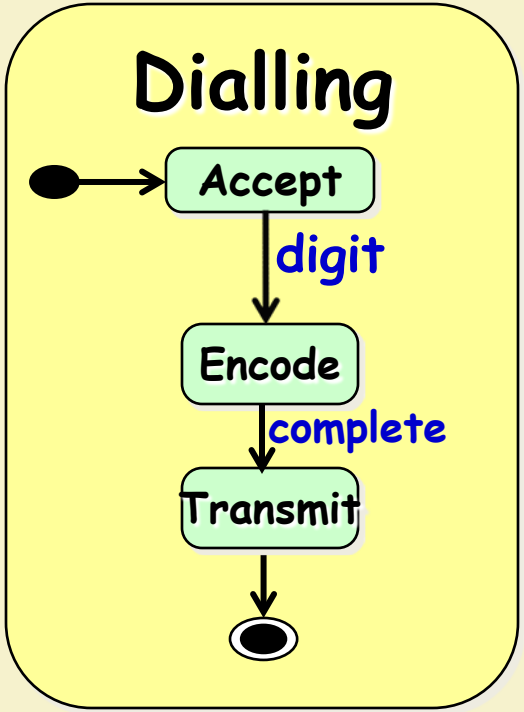
UML State Machine Diagram Syntax

Lights ON

Basic UML State Model Syntax

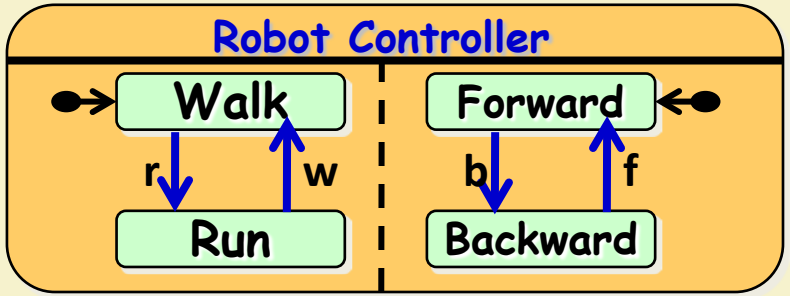


Composite State: OR state

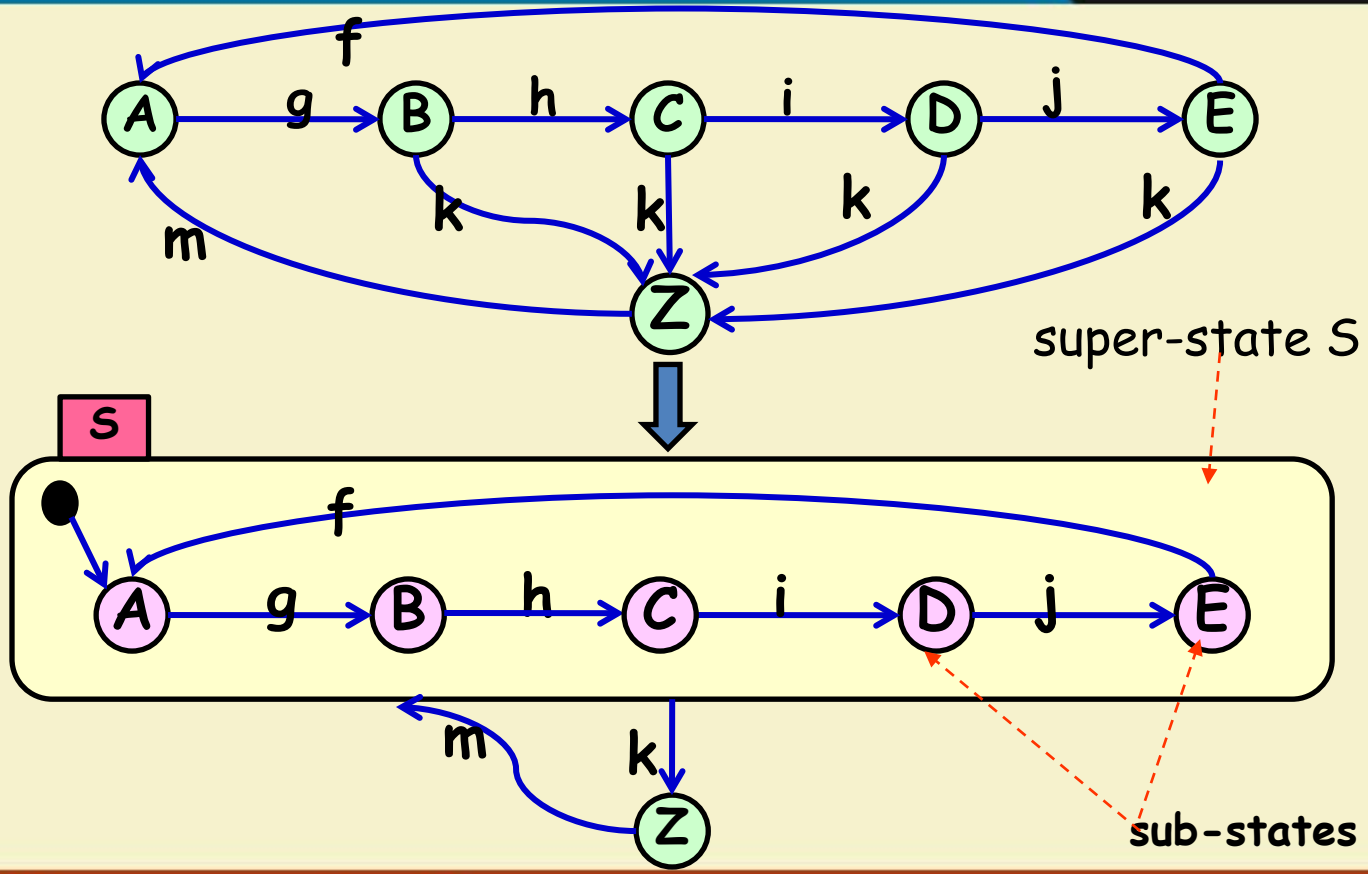


Composite States: AND/OR States

- Composite states can have:
  - **concurrent substates -and- relationship**
    - substate separated from others by dotted line
  - **disjoint substates -or- relationship**
    - transitions between substates



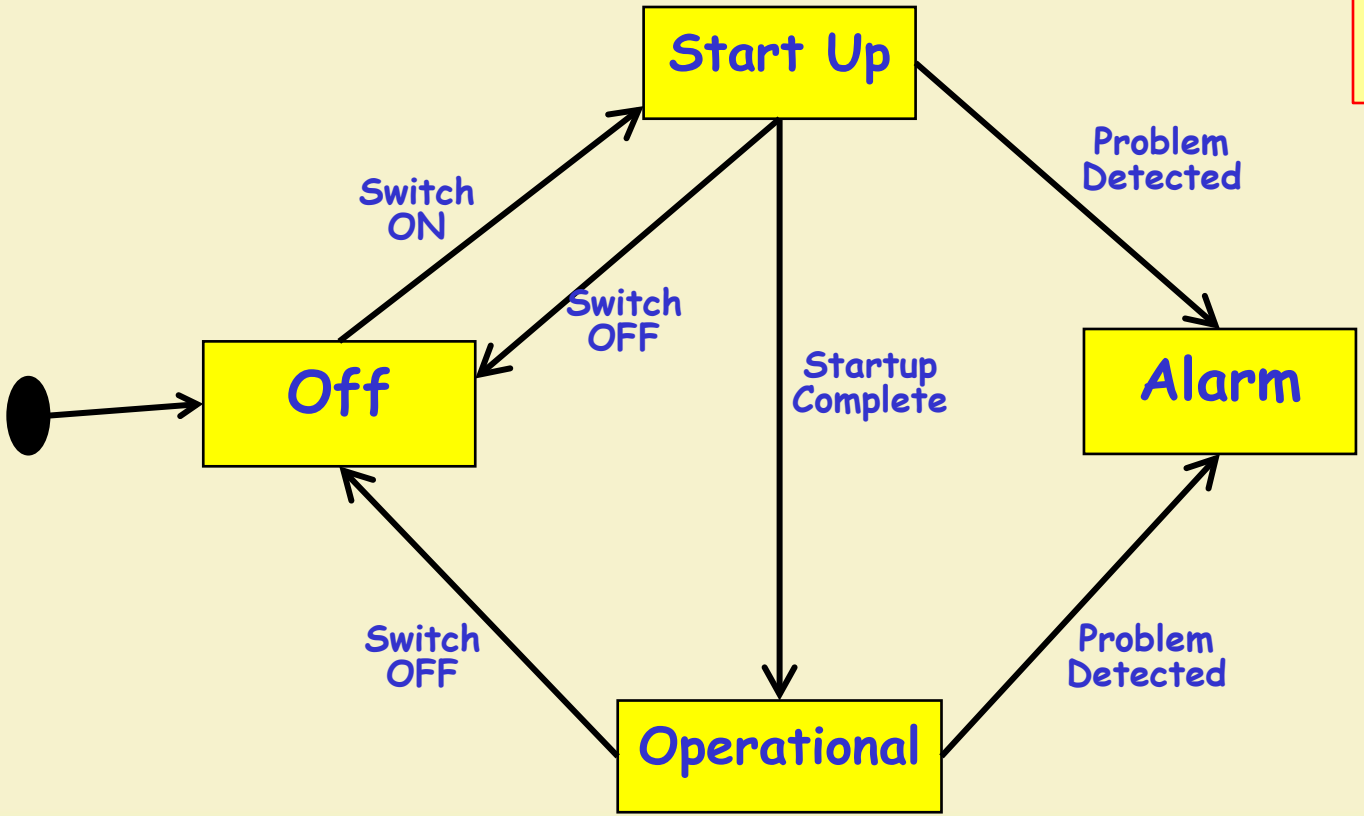
Introducing Hierarchy



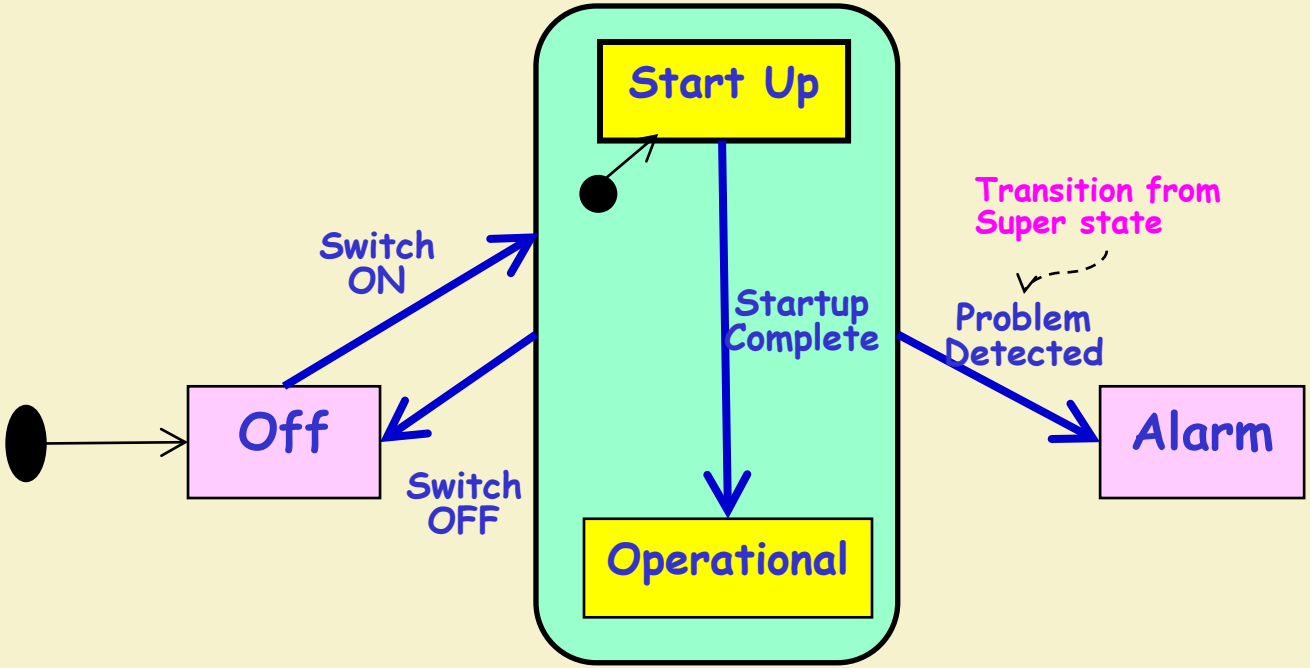
Exercise

- A printing machine is in the power off state to start with.
- When powered on, it goes into a startup state.
- After completion of the startup state, it enters operational state.
- Whenever, a problem is detected, it enters an alarm state.
- Whenever it is switched off, it goes into a power off state.

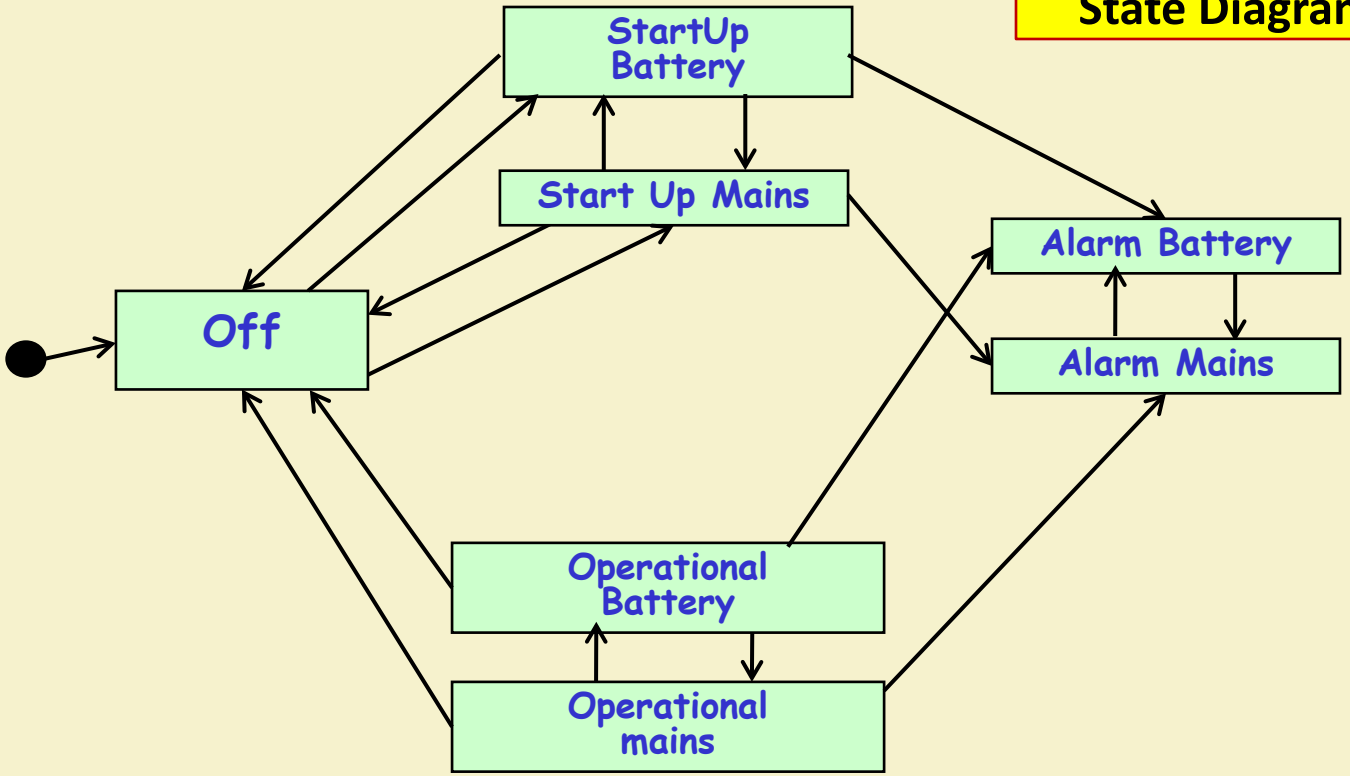
Printing Machine



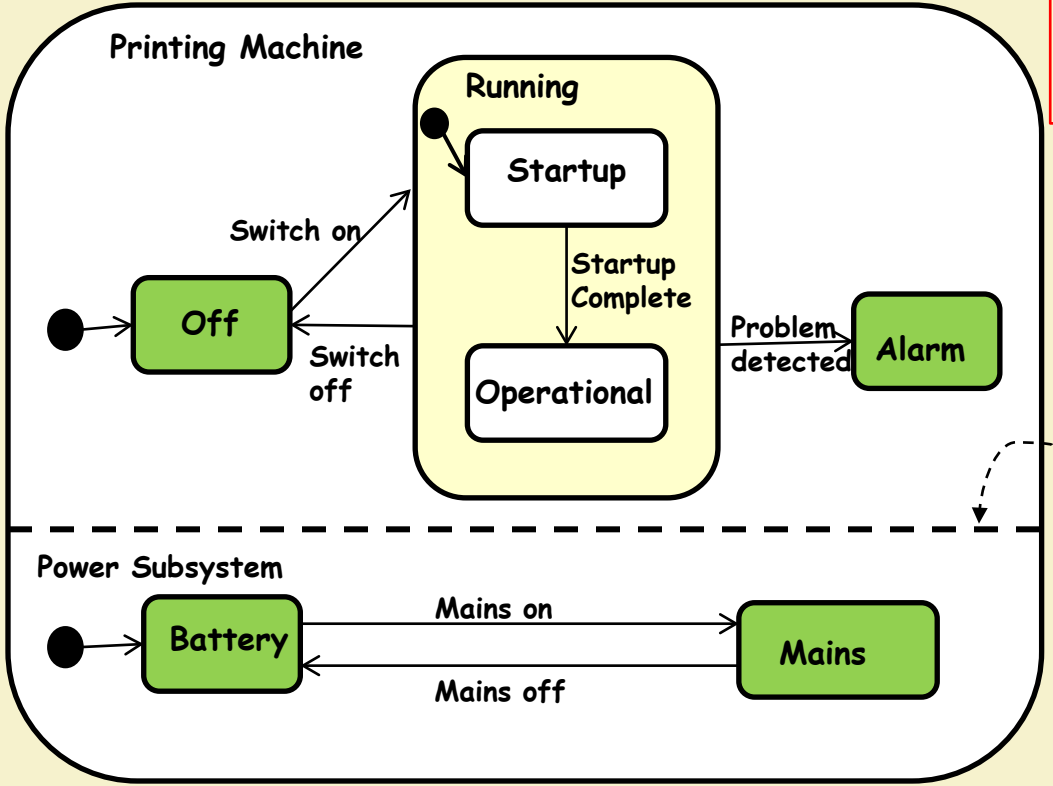
Refined Printing Machine



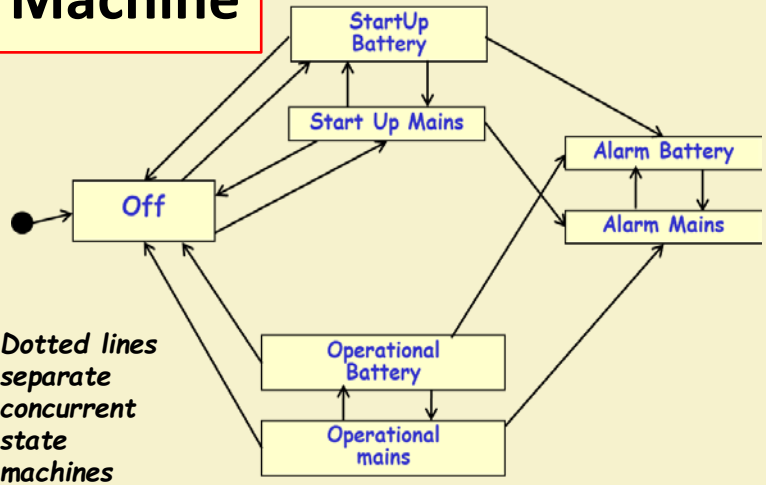
State Diagram for Printing Machine



Quiz: Convert to State Machine Model with Composite states…



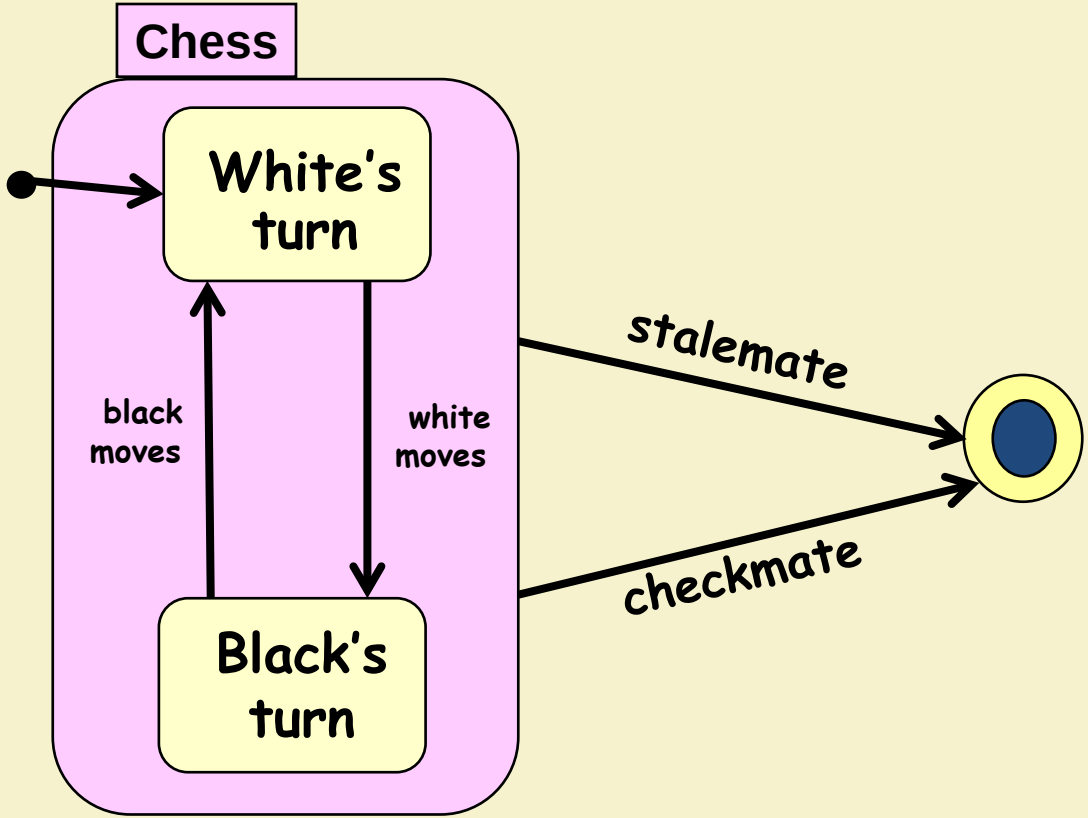
Printing Machine



Dotted lines separate concurrent state machines

Exercise 1: Develop State Machine Model

- In a chess game:
  - Black and white sides take turn to play.
- The game can end anytime when:
  - Either there is a checkmate, or
  - There is a stalemate.
- Use concurrent state

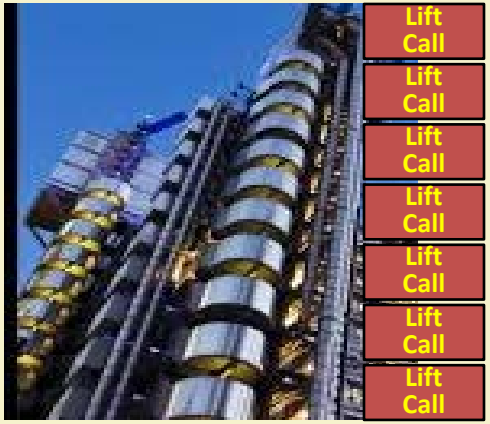

Chess
White's turn
Black's turn
black moves
white moves
stalemate
checkmate

Developing State Machine Model

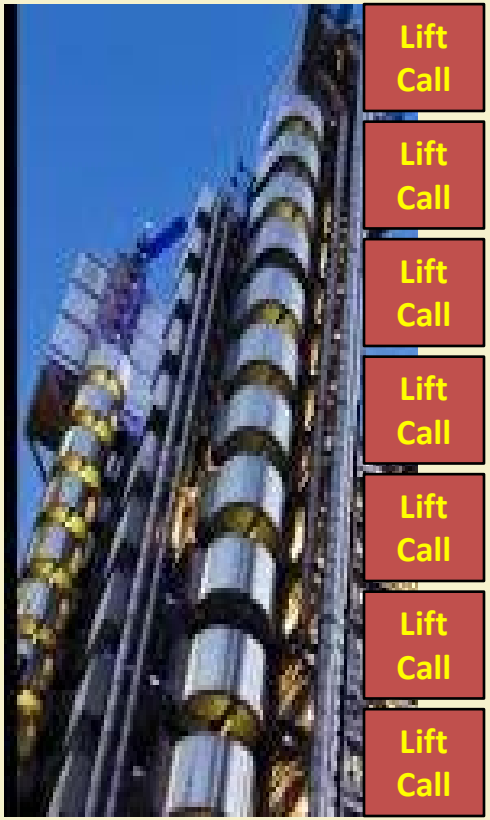
1. In the use case description, underline the states, actions, conditions and results.
2. Develop a preliminary state machine with the identified states and label transitions with events, actions and conditions as the case may be.
3. Refine into a hierarchical and concurrent state machine.

Exercise : Simple Elevator System

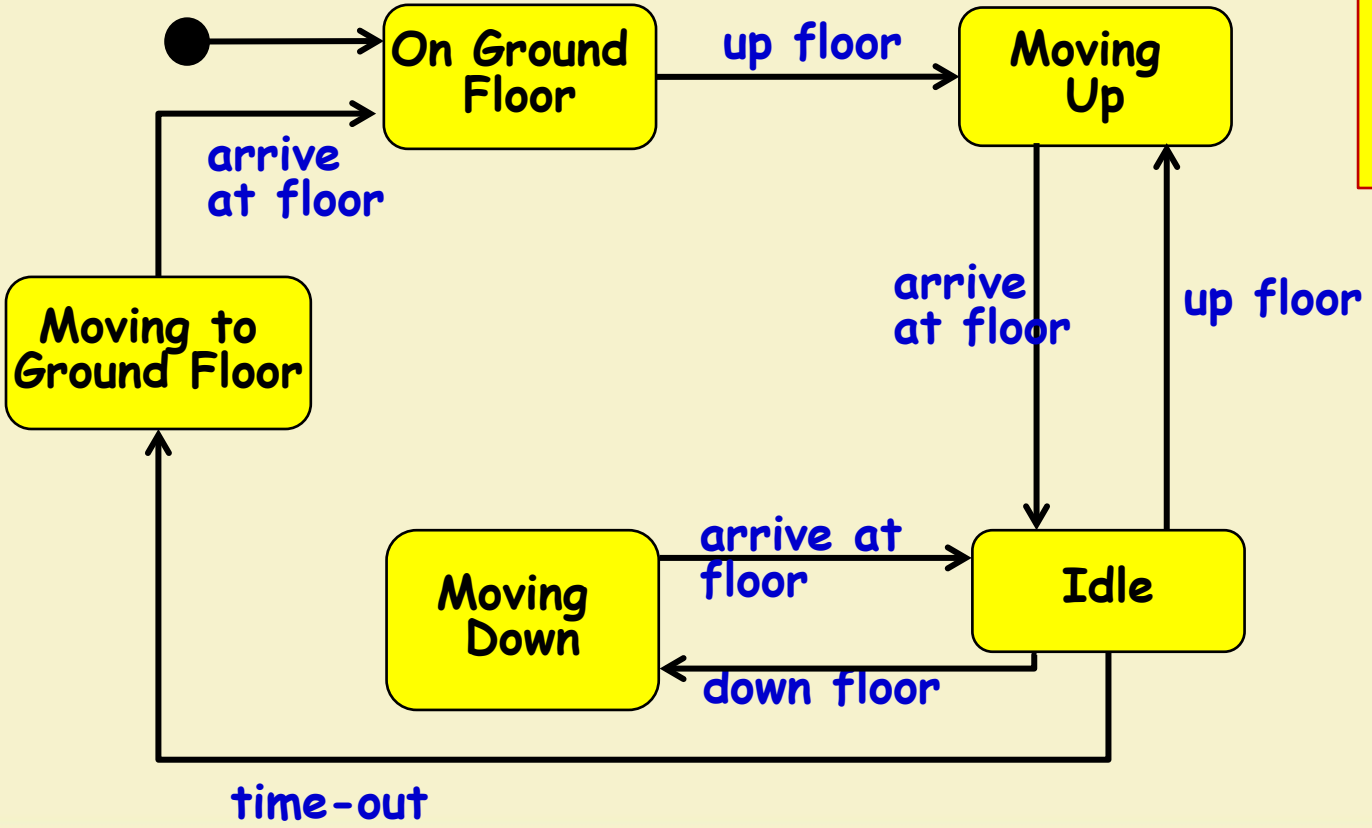
- The elevator is by default at the ground floor.
  - It moves up when button at any upper floor is pressed and halts when the requesting floor is reached.
- When idle and a button at a lower floor is pressed:
  - It moves down. Halts when requesting floor is reached.
- When idle and a button at a higher floor is pressed:
  - It moves up. Halts when requesting floor is reached.
- When it is inactive at a floor for more than 10 minutes:
  - It moves down to the ground floor.



Simple Elevator System



Elevator: State Machine Representation



Translating State Machine to Code

- **Complicacy:**
  - State machines have many concepts that are not directly supported by programming languages ---neither procedural nor OO programming languages:
    - **States, events, transitions, composite states, concurrent states, history, etc**

How to Encode an FSM?

- Three main approaches:
  - **Doubly nested switch in loop**
  - **State table**
  - **State Design Pattern**

Three Principal Ways

- **Doubly nested switch in loop:**
  - Use scalar variables to store state --- Used as switch discriminator for first level switch
  - Event type is argument to second nested switch
  - Harder to handle concurrent states, composite state, history, etc.
- **State table:** Straightforward table lookup
- **Design Pattern:**
  - States are represented by classes
  - Transitions represented as methods in classes

Doubly Nested Switch Approach

```
int state, event; /* state and event are variables */
while(TRUE){
    switch (state){        /* Wait for event */
      Case state1: switch(event){
                     case event1:  state=state2; … break;
                     case event2:…
                     default:
                     }
      Case state2: switch(event){…
                                  }
                 }
    }
```

State Table Approach

- From the state machine, we can set up a **state transition table…**

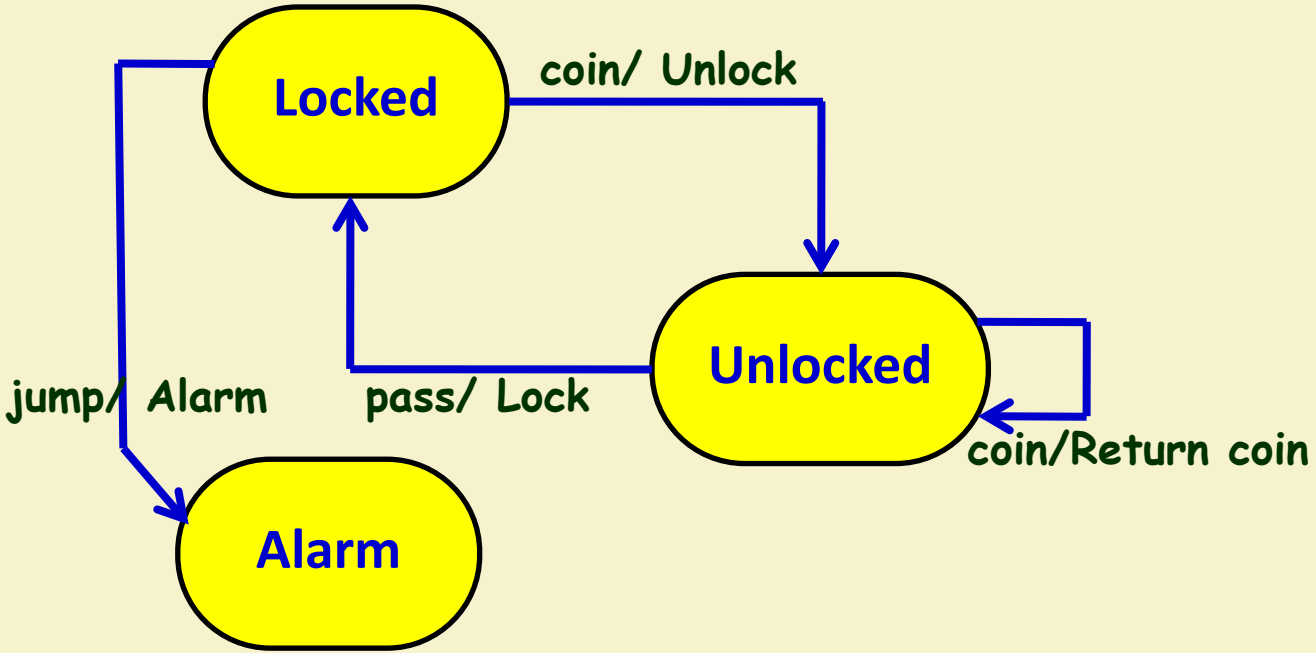
Present state	Event	Next state	Actions
S1	e1	S3	Open Valve
	e2	S2	set red LED flashing
S2	e1	S1	Close Valve
	e2	S4	reset red LED flashing

Exercise: Turnstile

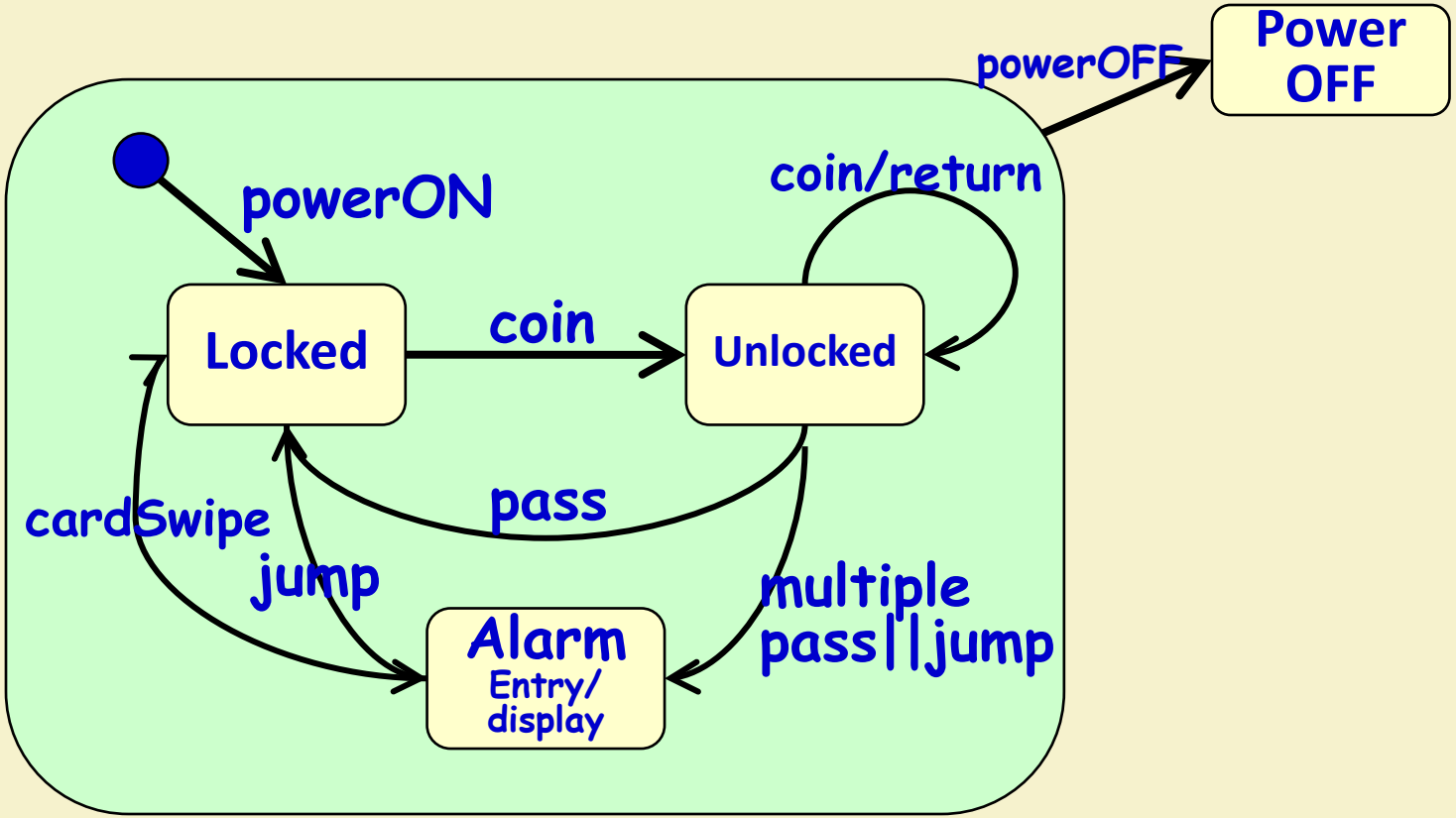
- A turnstile check gate would be installed at a railway station.
- When the turnstile is powered on:
  - It starts in the locked state.
- When a commutator drops a one rupee coin:
  - The gate gets unlocked and a message “Gate Open” appears.
  - Exactly one person is allowed to pass through before the gate gets locked.
  - If more than one person try to pass through, an alarm is *sounded and gate gets locked.*
- Also when any one tries to jump over the gate:
  - An alarm is started and gate is locked.

- When the alarm is ON:
  - A message **"Please wait: Gate temporarily blocked"** is displayed.
  - If any one still inserts a coin, it return the coin without unlocking the gate.
  - The alarm is reset when an attendant swipes a card and the gate starts at the locked state.
- When any one inserts a coin when the gate is already open:
  - The coin is returned.
- If there is a power failure any time:
  - The gate gets locked and **"Power fail: inoperative"** message is displayed and coins are not accepted.

Turnstile: First-Cut State Model

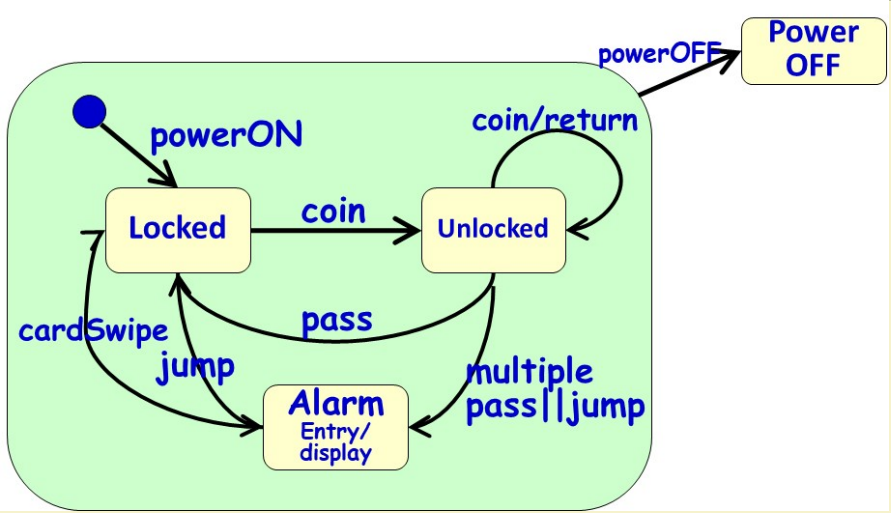


Turnstile: Refined State Model

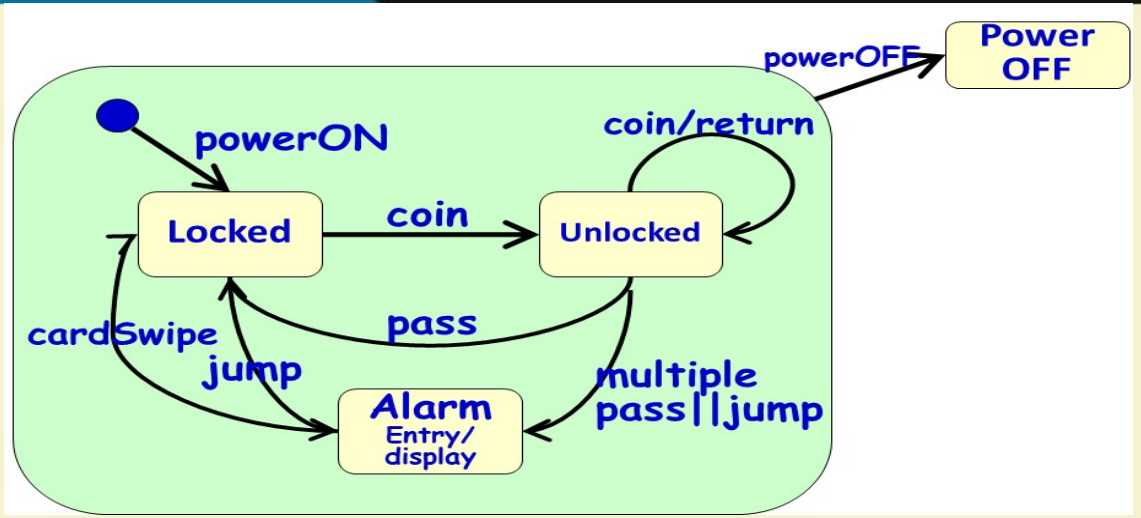


```
enum State {Locked, Unlocked, Alarm, PowerOFF};
enum Event {Pass, Coin, multiplePass, jump, cardSwipe};
static State s = Locked;
void Transition(Event e){
        switch(s){
              case Locked:
                        switch(e){
                             case Coin:
                                   s = Unlocked;
                                   Unlock();
                                    break;
                             case Jump:
                                     s=Alarm;
                                    Alarm();
                                     break;
                                   }
               break;                                  cont…
```

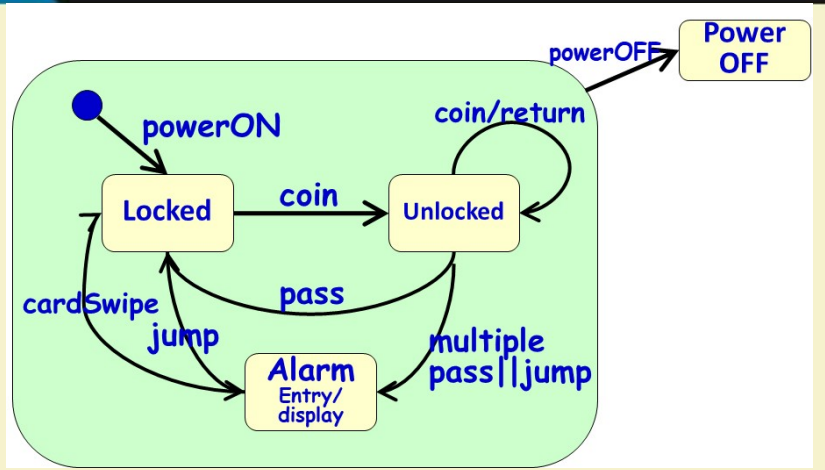
C Code



```
case Unlocked:
  switch(e){
    case Coin: returnCoin();
               break;
    case Pass: s = Locked;
               Lock();
               break;
    case jump:
   case multiplePass: s=Alarm;
               Alarm();
               break;}
    break;
  case Alarm: …. … ….
```



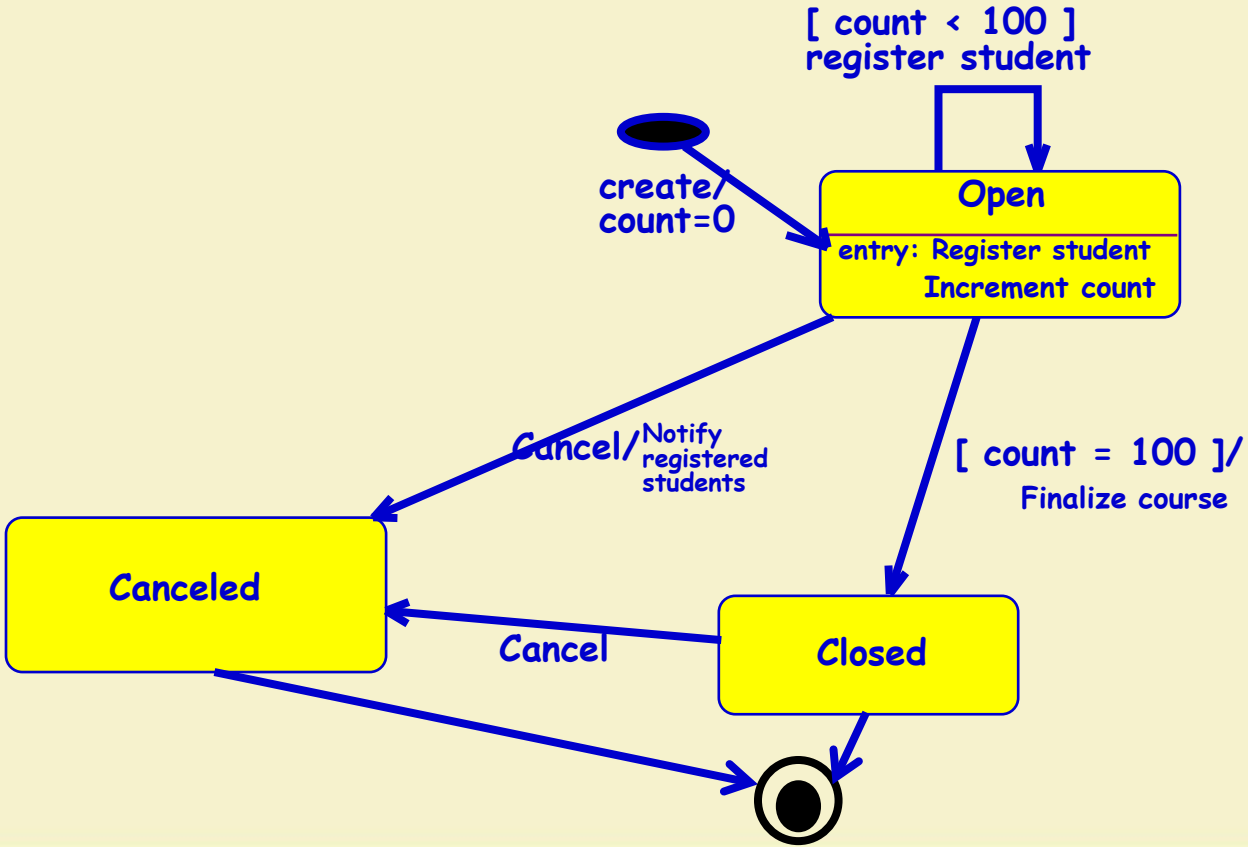
```
public class Turnstile{
enum state{ Locked, Unlocked, Alarm}
public Turnstile(){ state=Locked;}
public pass(){
    if(state== Unlocked) state=Locked;}
public coin(){
        switch(state){
                case Locked: state=Unlocked; break;
                case  Unlocked: returnCoin(); break;
                }}
public jump(){…}
…}
```



Java Code

Assignment: Course Registration Software

- When a course is created, it is initialized to open, and student registration made 0.
- If the course is open and a student registers, the count incremented.
- A course can be cancelled anytime: the registered students are notified and their registration cancelled.
- If registered student number reaches 100, the course is closed for registration, and it is allocated a room and a time slot.



State Transition Diagram: Course Registration Software

Interaction Diagrams

Interaction Diagram

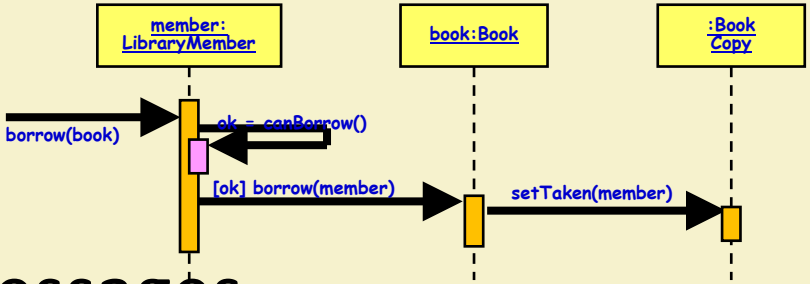
- **Can model the way a group of objects collaborate to realize some behaviour.**
- How many interaction diagrams to develop for a problem?
  - Typically each interaction diagram realizes behaviour of a single use case
  - **Draw one sequence diagram for each use case.**

Interaction Diagrams

- Three kinds:
  - **Interaction overview diagram**
  - **Sequence** and **Collaboration** (now communication in UML 2.0) diagrams.
- Sequence and Collaboration diagrams are equivalent:
  - However, portray different perspectives
- As we shall see:
  - **These diagrams play a very important role in the design process.**

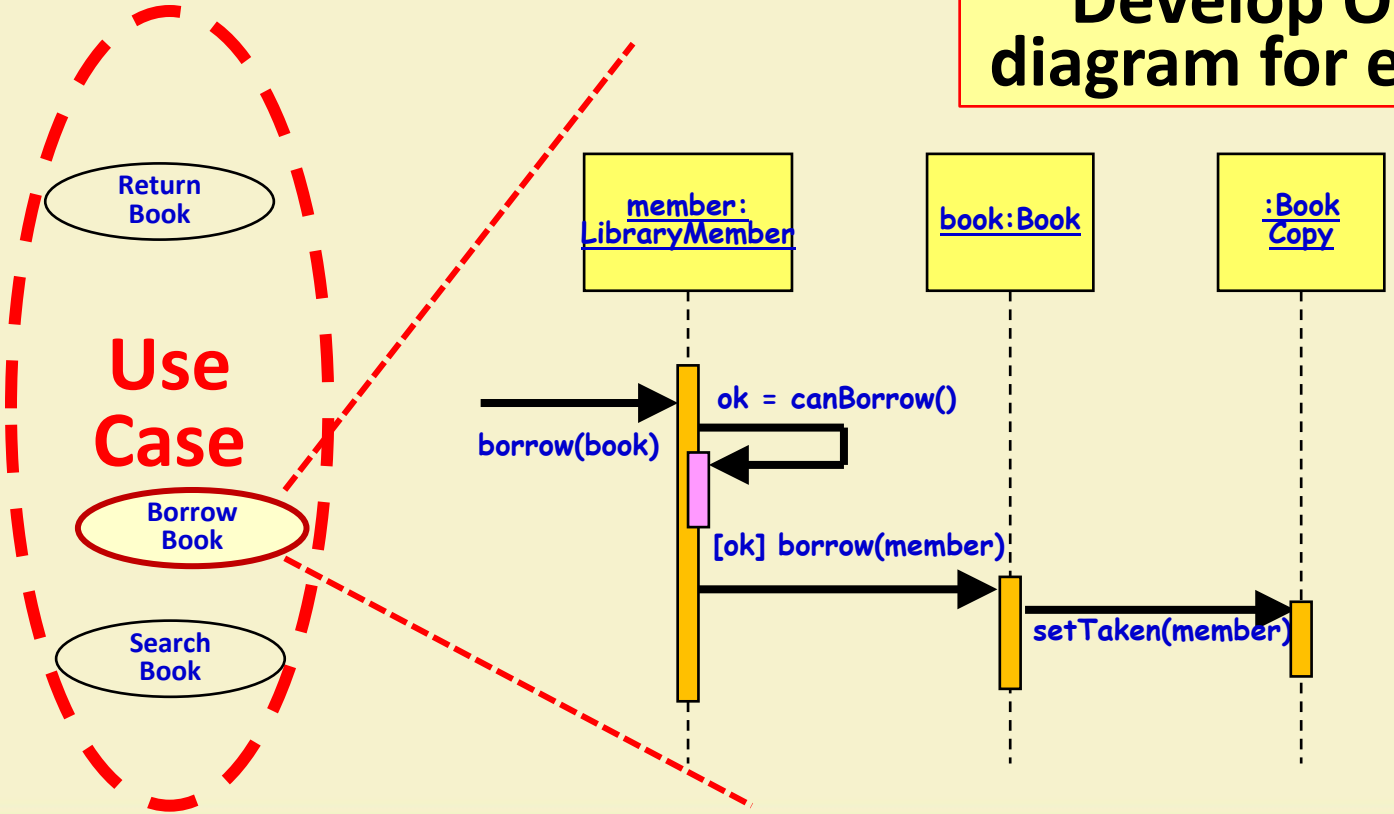
A First Look at Sequence Diagrams

- Captures how objects interact with each other:
  - To realize some behavior.



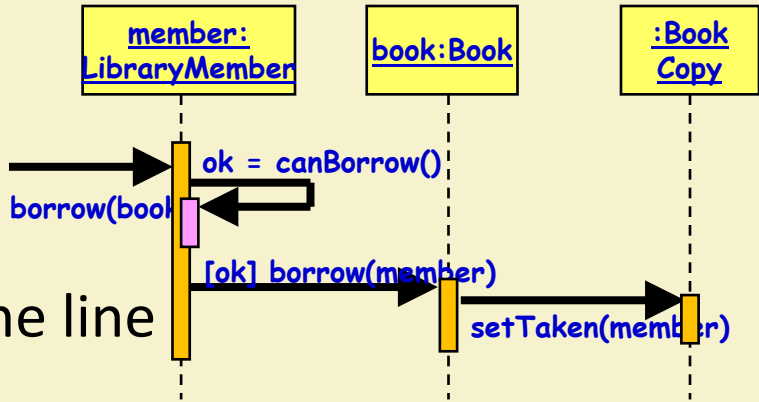
- Emphasizes time ordering of messages.
- Can model:
  - Simple sequential flow, branching, iteration, recursion, and concurrency.

Develop One Sequence diagram for every use case…



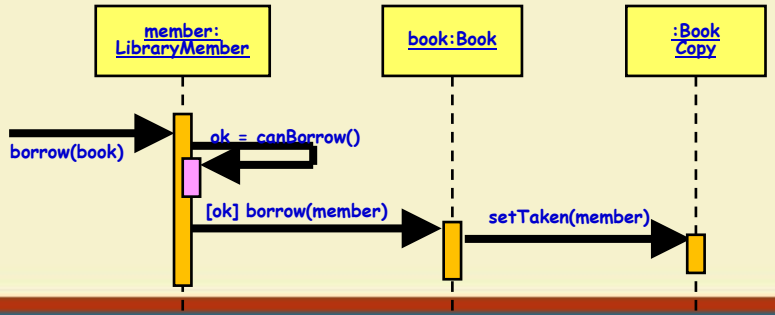
Sequence Diagram

- Shows interaction among objects as a two-dimensional chart
- Objects are shown as boxes at top
- If object created during execution:
  - Then shown at appropriate place in time line
- Object existence shown as dashed lines (lifeline)
- Object activeness shown as rectangle on lifeline



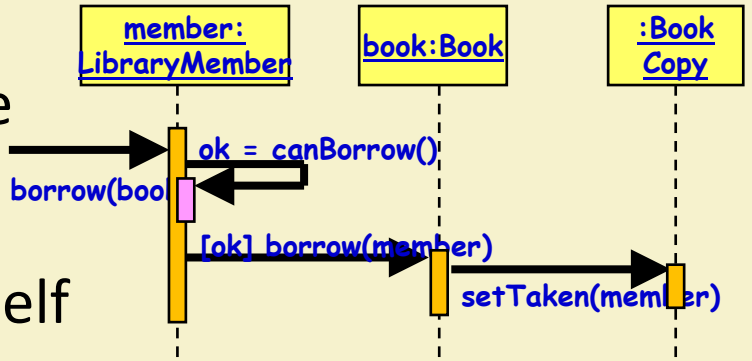
Sequence Diagram

- Messages are shown as arrows.
- Each message labelled with corresponding message name.
- Each message can be labelled with some control information.
- Two types of control information:
  - condition ([])
  - iteration (*)



Gist of Syntax

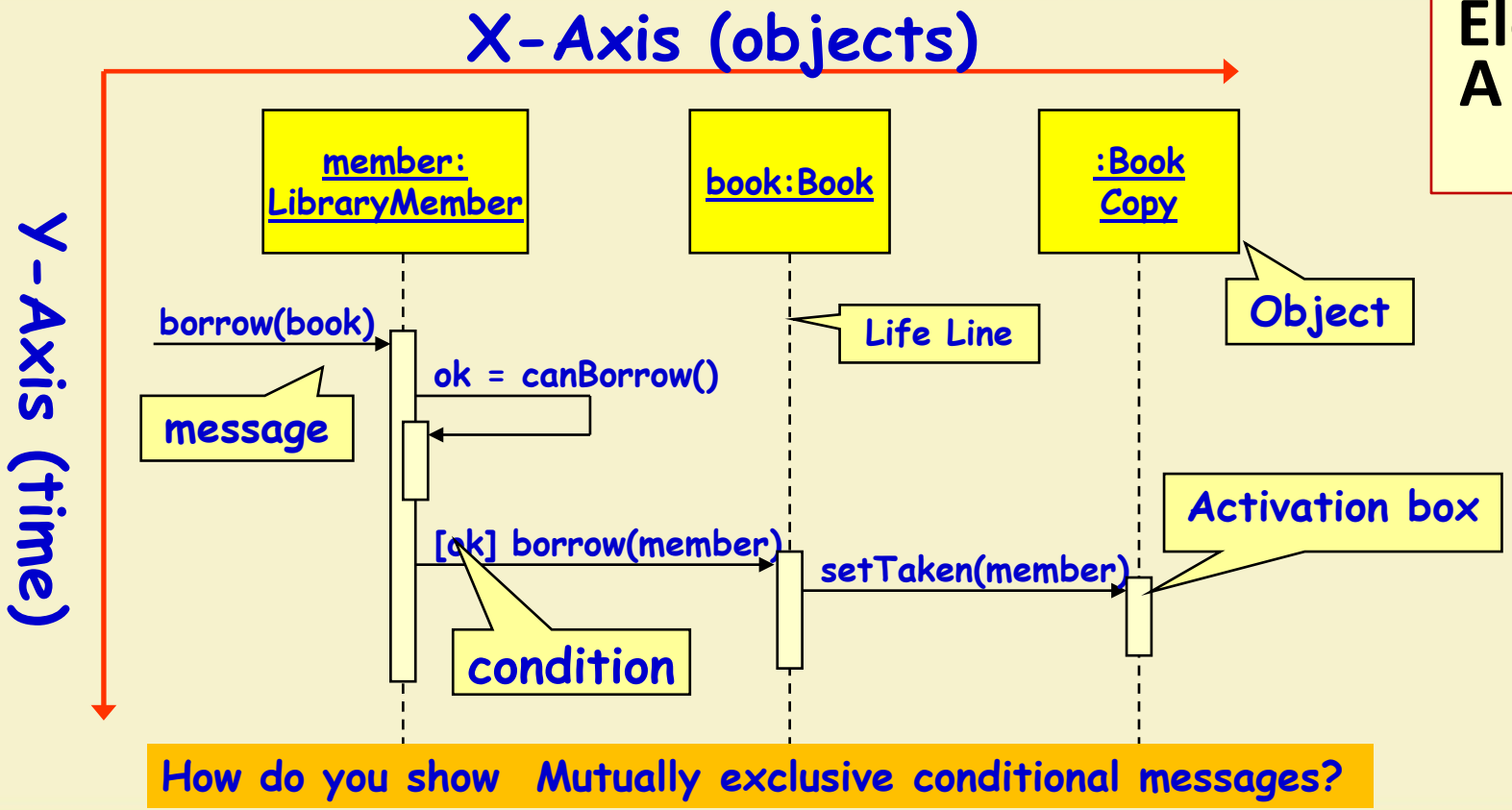
- **iteration marker** *[for all objects]
- **[condition]**
  - Message sent only if condition is true
- **self-delegation**
  - a message that an object sends to itself
- **Loops and conditionals:**
  - **UML2 uses a new notation called interaction frames to support these**



Control logic in Interaction Diagrams

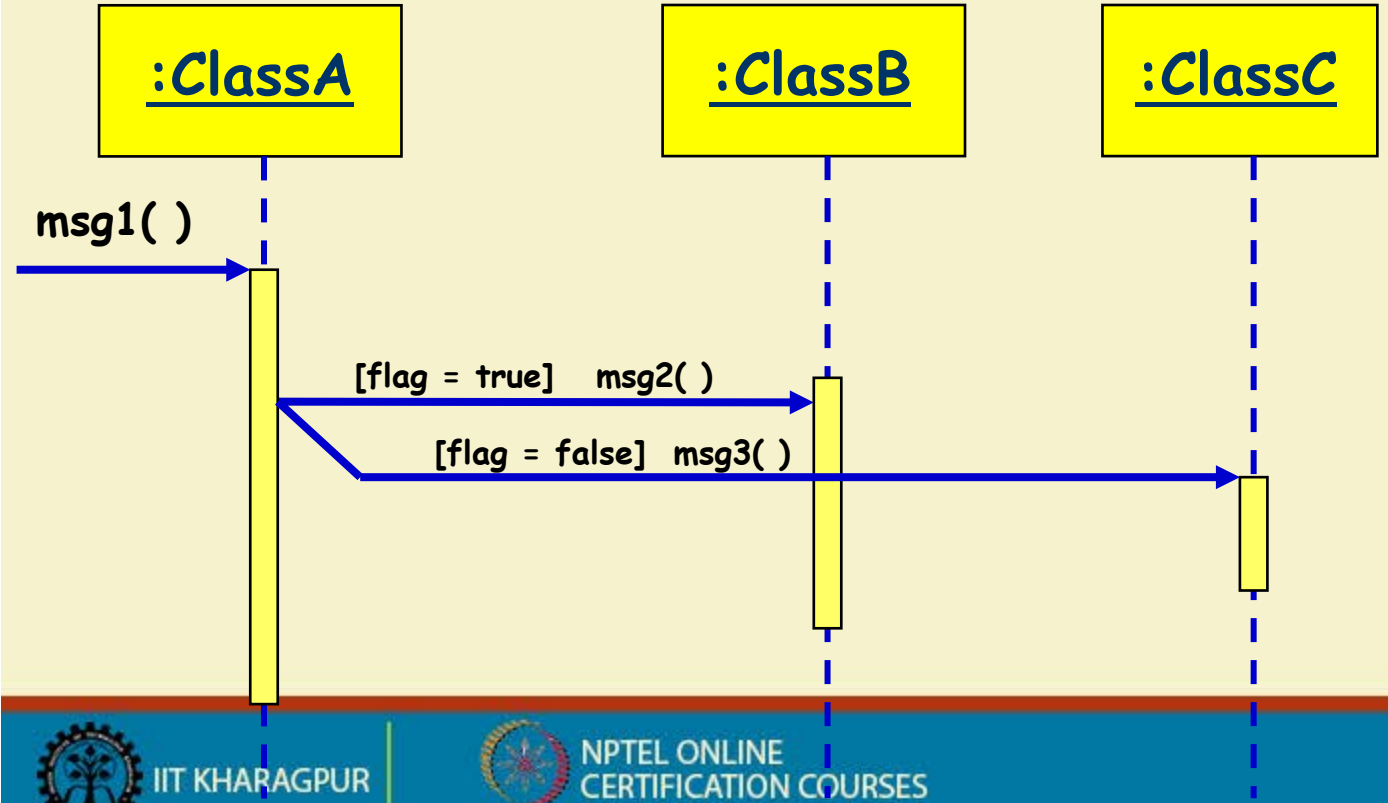
- **Conditional Message**
  - [variable = value] message()
  - Message is sent only if clause evaluates to *true*
- **Iteration (Looping)**
  - * [i := 1..N] message()
  - "*" is required; [...] clause is optional
  - **The message is sent many times to possibly multiple receiver objects.**

Elements of A Sequence Diagram



How do you show Mutually exclusive conditional messages?

How to represent Mutually exclusive conditional messages? Eg. ClassA on receipt of mesage1 sends either msg2 to ClassB or msg3 to classC,



Sequence Diagrams: Conditional Messages