

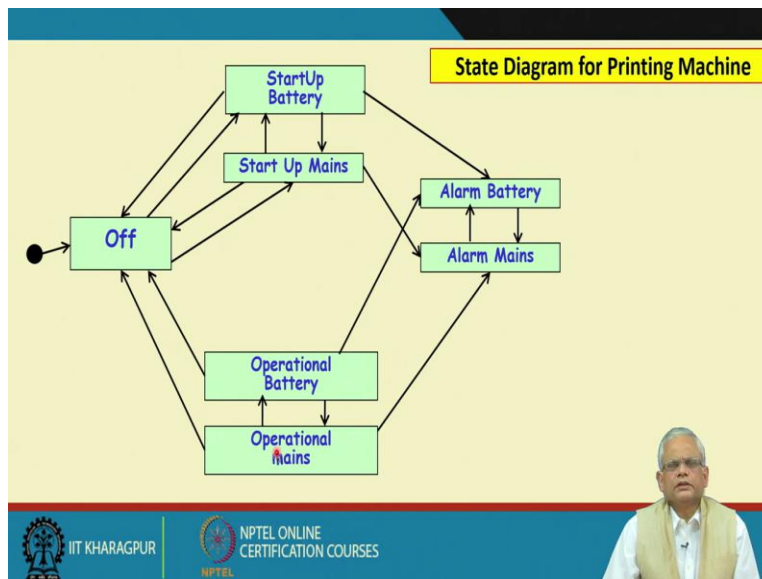
Object – Oriented System Development Using UML, JAVA and Patterns
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 23
Encoding a State Machine - I

Welcome to this lecture.

In the last lecture, we had given you a problem to work out. We had given a state machine diagram and that appeared complicated and we wanted to transform that diagram into one with composite states. I hope you have tried. Let see how to go about.

Let's revisit the problem.

(Refer Slide Time: 0:48)



The problem was about a printing machine. To start with, the machine is in the off state and if the main supply is available it starts up from mains. Otherwise it starts up from battery and there can be power failure. In that case it goes back to a startup battery, power comes back startup mains and at any point, if there is a problem, there is an alarm from battery. Even when there is a startup from main there is an alarm. And then, after the startup is complete, we knot on the transition, it goes to operational state and even in the operational state, if there is a problem, then it goes to an alarm state.

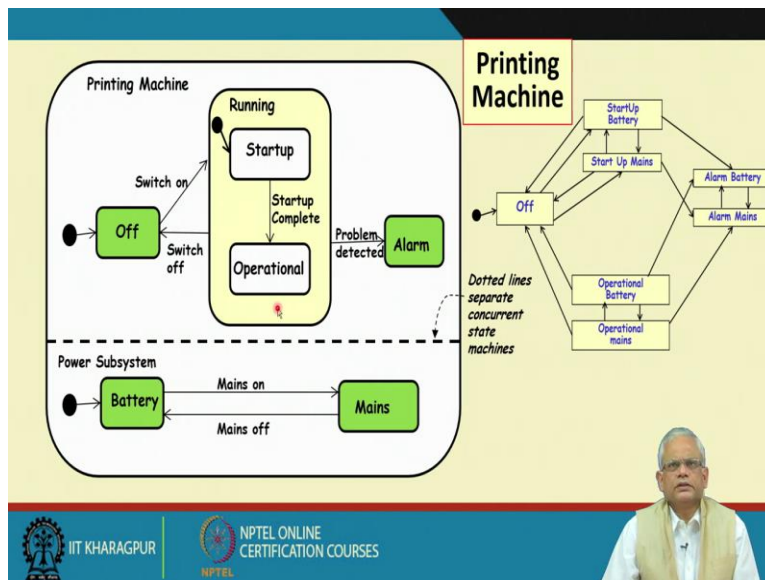
Now, one thing to notice is that, we have several transitions coming here which are similar transitions here (in the above slide). We can see four transition coming to alarm state. So, all four

are actually the same event causing transition to this alarm state. And similarly, when there is a switch off, it goes to the off state and even from startup, if there is a switch off, it goes to the off state.

So, that gives us a hint that this becomes a composite state (StartUp and Operational). So, that a transition from the composite state to the alarm state captures all the four transitions and similarly, transition to the off state captures all the four transitions. Now, the other thing that we can think of is that the battery and the main, there is a transition if there is a failure and these are two different aspects of the plant, one is about the power supply which can be from battery or main and the other is about the plant which is in the startup or in operation.

That is there are two subsystems of the printing machine; one is a machine itself, which is in the startup and operational and the power supply transits between battery and main supply and that indicates that it will be AND state of the two components of the printing machine because these are the states of two different parts of the printing machine. Now, let's redraw this.

(Refer Slide Time: 4:15)



So, this is the finite state machine (in the above slide), some of the transitions are not shown here to avoid clutter, and we can redraw that in this form. There are two parts here: Printing Machine and the Power Subsystem. These are two AND state it may be battery and one of these are the main and there is a OR composite state or a nested state.

Here once the switch on event occurs, it goes to the startup state and as the startup completes it goes to the operational state, at any time when a switch off event occurs it goes to the off state and from any of these, if there is a problem detected, then it goes to the alarm state. This is a more preferable notation it avoids clutter, shows the different parts of the system very clearly. The AND state and also the composite OR state or the nested state is helped to represent more elegantly than a simple state machine.

(Refer Slide Time: 5:51)

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

Now, let's do a few exercises.

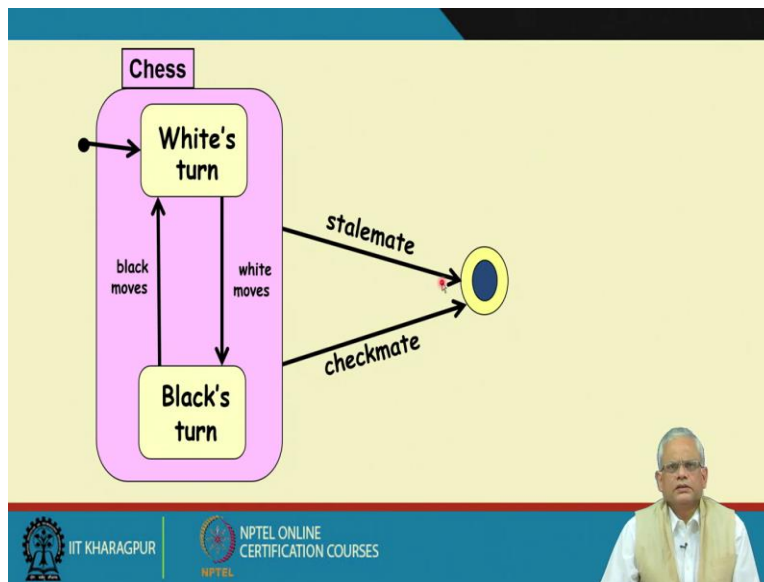
The first is to develop the state machine model of a chess game. We all are familiar with chess game. There are two sides here playing, one is black, and the other is white. The two players one play on the black side and another one the white side. They interchange after each move.

So, once white makes a move, then after that black makes a move and then white makes a move and so on. The black and white sides take turns to play and the game can end anytime when there is a checkmate that is the king has no route to escape that is in chess terminology called checkmate, or that is a stalemate that is no progress in the game neither can be won on either side and that is called as a stalemate.

So, the game ends when there is a checkmate or stalemate, and these two can occur during either black side move or the white side move and we need to use the concurrent state to simplify the state model of the game. Relatively simple problem please try here.

There are two states here: White's turn and Black's turn. One is the black's turn, for making a play, the black waits to play and finally makes the move and once the move occurs, then it becomes the white's turn to play. White may takes time, thinks about the move to make and then makes the move then it becomes black's turn and the game can end from both these states black and white, if there is a stalemate or a checkmate. It gives us the hint that when the game can end from any of the states if a checkmate or a stalemate is detected, that means we use a composite OR state.

(Refer Slide Time: 8:34)



So, this is the chess game (in the above slide), the composite OR state. White's turn and black's turn are the two states. To start with it is white's turn and then as white moves it becomes black's turn and the black plays, it becomes white's turn and so on. And at any point from any of these states if a stalemate or checkmate event occurs, then the game ends. So, this is the representation.

(Refer Slide Time: 9:13)

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.

Developing State Machine Model

IT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

The slide features a yellow background with a blue header and footer. A small video feed of a man in a white shirt and yellow vest is visible in the bottom right corner. The footer contains the logos for IIT Khargapur and NPTEL Online Certification Courses.


Now, let's just put down the things that we need to do to develop a state machine model. Typically, from a use case description, we develop the state machine model. From the description we underline the identifiable states, the actions, the conditions or the guards and the results. First, we develop a preliminary state machine model where we represent all the identified states. We draw the transitions and label them with events, actions and conditions or the guards.


If the action part is present, we represent the action. We know from the syntax how to represent the action, the guard. If these are present, we represent them otherwise we just represent the event. And then we look for hints that from different states on the same event entering into another state, that gives us a hint of a nested state or a composite OR state and also, we see when there are two different aspects of the system that are represented two or three aspects then we make that as an AND state.


(Refer Slide Time: 11:09)


- 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.

Exercise : Simple Elevator System









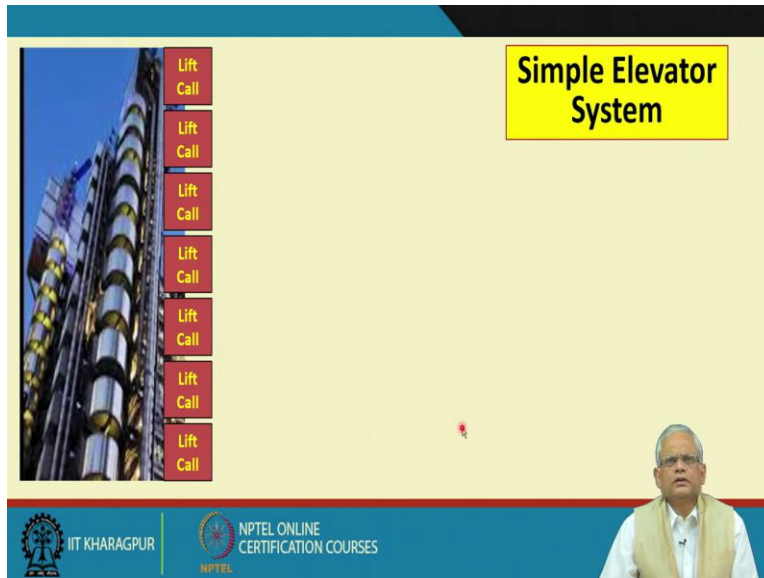
Now, let's try another problem. A simple elevator system. It is a very simple system. Here the elevator is by default at the ground floor and we know that to start with you represent to the pseudo state and the pseudo transition by default it is at the ground floor. When any upper floor lift button is pressed it goes to up state. So, there are this is a multi storey building (in the above slide image) where there is a lift system and in the lift each floor has a lift call button.

And if at a upper floor a lift call button is pressed, then lift moves and when the requesting floor is reached, it stops and it remains in the idle state and when the lift is in idle and the lower floor lift call button is pressed it moves down. And if while idle, if a higher floor button is pressed, it moves up and halts when the requesting floor is reached.

And if it is inactive for a floor for more than ten minutes it moves down to the ground floor. So, it is a very simple lift system. One thing to notice is that only when it is idle and the button at a lower floor or an upper floor is pressed, it moves. When in motion, when it is moving, no call is entertained. And based on that we can identify the states.

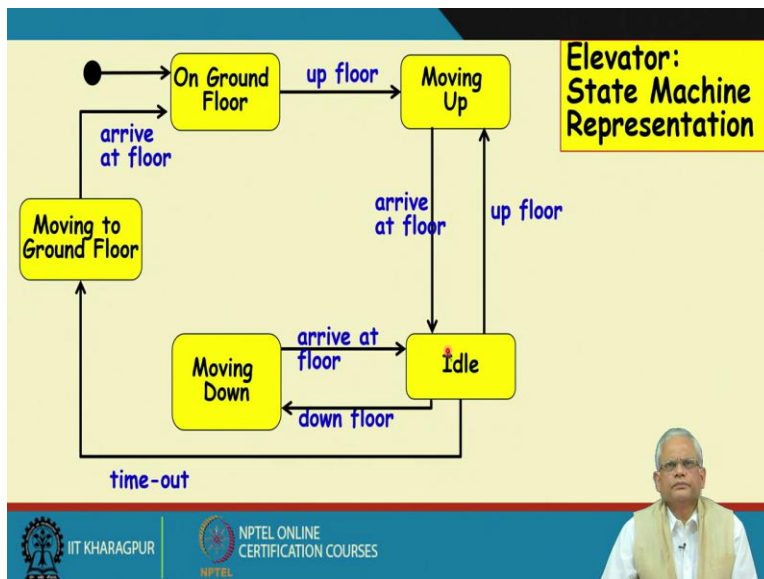
If it is inactive for more than 10 minutes, we will represent that with the help of a timeout event and then the lift is idle that is another state and the event can be lift, lift call button at a upper floor is pressed or a lift called button at a lower floor is pressed.

(Refer Slide Time: 14:16)



Based on this analysis if we try to develop the diagram, then we will get an diagram like below slide.

(Refer Slide Time: 14:27)

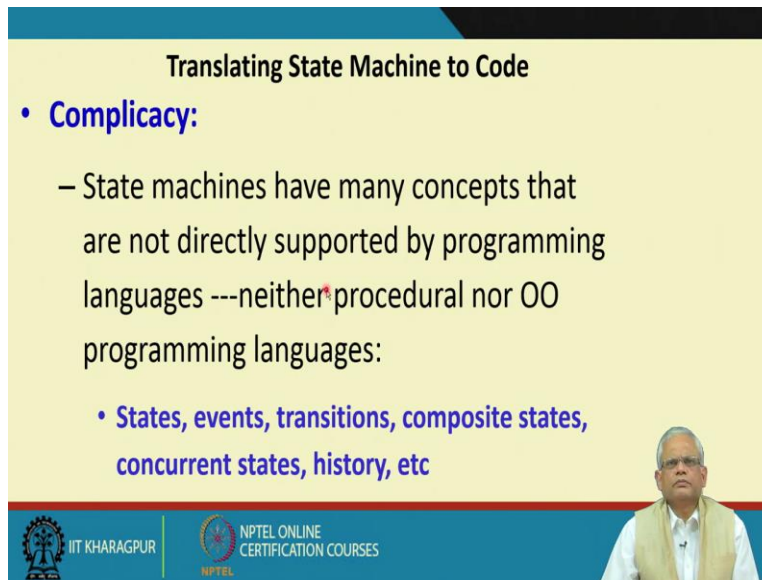


Now, if we represent the simple behavior to start with it is on the ground floor. A up floor button is pressed, it moves up and when it arrives at the requested floor becomes idle. From the idle, if the call button at the up floor is pressed it moves up and when it arrives at the floor, it becomes idle. Similarly, from the idle if a call button at a down floor is pressed it moves down and when it arrives at the floor it becomes idle.

And when idle for more than 10 minutes that we represent with the help of a timeout event; it goes to, it moves to the ground floor and arrives at the ground floor.

From the simple description that we have for the elevator system, we have drawn the above diagram.

(Refer Slide Time: 15:54)



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

The slide features a yellow background with a blue header and footer. A small video feed of a man in a white shirt and yellow vest is visible in the bottom right corner of the slide area. The footer contains the logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES.

Now, let's examine the issue that if we have developed a state machine diagram how do we encode that in either C, Java or C++ language. The complicacy that arises here is that in the languages that are existing, the languages do not support the various concepts associated with state machines to be represented automatically.

For example, for representing inheritance, we had a keyword extends in Java. If we had keywords corresponding to a state machine representation, it would have been straightforward. But here we need to understand the state machine and will give a very simple method to generate code for a state machine.

The current languages do not support any construct using which you can directly represent states, events, transitions, composite states, concurrent states, history states, etc. But then fortunately, it's not very hard. We will just describe a simple methodology using which given a state machine you can easily write the code almost mechanically and many case tools they can generate C, Java, C++ code based on the state machine representation.

(Refer Slide Time: 17:50)

How to Encode an FSM?

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

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

There are essentially three main approaches by which you can translate a FSM into a program. The first one is very frequently used is called as doubly nested switch inside a loop. The second is a state table-based method, the third is a state design pattern. The state design pattern, we will see as we proceed in this course, and discuss the design patterns. At this moment, we look at the doubly nested switch and loop and the state table.

(Refer Slide Time: 18:41)

Three Principal Ways

- **Doubly nested switch in loop:**
 - Scalar variables 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

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

These are the three main approaches to generate code from a state machine representation, the doubly nested switch inside a loop, state table, which is basically a straightforward table lookup

and the design pattern, which we said we will look at it later in the course, when we discuss about the design patterns.

In the doubly nested switch inside a loop, we use some variables to represent the states maybe a int variable. If $S = 1$, then the state is S1, $S = 2$, state is S2, if S gets value 3 state is S3 and so on.

So, a variable will represent the current state and we use this state and switch based on the current state that is the state variable, which forms the first level switch and based on the first level switch, we switch into the appropriate state and inside the appropriate state we have another nested switch setup, which event has occurred for that state and then we do the corresponding action.

Of course, it's very straightforward here as you will see the code next, but then slightly harder to handle concurrent states, composite states, history, etc a lot of ingenuity in programming is necessary we will not look into this in this course. The design pattern as you will see later it represents states by classes, each state becomes a class and transitions are the methods in these classes. We will look at this approach later, it has certain advantages. But right now, let's look at the doubly nested switch in a loop and the state table.

(Refer Slide Time: 21:07)

```
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){...
    }
}
```

First, let's look at the doubly nested switch approach, very simple approach (in the above slide). If we have a state machine, we first set up a loop here, while(TRUE) and the state is encoded by

a scalar variable state and we switch on state and based on the current state, let say is state1 is the current state, it switches into this, because the state = state1 and then we switch on the event that occurs. This basically encodes that how does the system behave or respond when a specific event occurs in a state.

For many events, a state may not produce any transition. For some events in a state, it produces a transition, let say event1 in the state1 causes a transition to state2, then we write here as the event1 occurs in state1, the system transmits to state2 (state = state2). Similarly, let us say on event2 in state1, it transits to back to the state0, then we write state = state0 and so on. So, here we first switch on the state and once we have this appropriate state or the current state that the system is in, and then you switch an event and based on the event that occur, we take the corresponding action.




The first is about a transition to another state and the other is if any actions are produced. It is easy to handle guards also, because we can have the guard encoded appropriately if certain condition then only the transition occurs. So, this is a loop in which there is a switch, this is the first level switch is the state and then the second level switch is the event inside every state. So, this is a doubly nested switch inside a loop approach.

(Refer Slide Time: 24:02)

State Table Approach

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

Present state	Enable input	Next state	Actions
BTN_off	e1	BTN_off	none
	e2	BTN_on	set red LED flashing
BTN_on	e1	BTN_on	none
	e2	BTN_off	reset red LED flashing

Now, let's look at the state table approach (in the above slide). Here based on the state machine diagram, we write down the states that are present on the state of machine. And the events that

caused transition from a state we write down here. Let us say e1, e2 caused transition from the button off state (BTN_off). And then on e1 it remains in button off, and then e2 it goes into button on (BTN_on) and if there is an action occurring, we just also mention that here on action part of the table.

Similarly, for the other states, we write the events to which these transitions occur from that state to the next state and the action that occurs. Here as the event occurs, we write the program. The state is present in a scalar variable in the table. We index into the corresponding state and then we index into the appropriate event, identify the next state and the transition occurs that is we set the state variable to the next state. And then if any action happened, we call a method or a function to have the action or maybe it can be a very simple action like setting a state variable.

We have so far looked at how to encode a state machine into either C, Java or C++ code. We just looked at the overall approach. The first one is suitable for C programming, doubly nested switch inside a loop. And the second one is a state table approach. We will just take one or two examples and see that given a state machine diagram, how do we generate code.

We are almost at the end of this lecture and we will stop here and continue from this point in the next lecture.

Thank you.