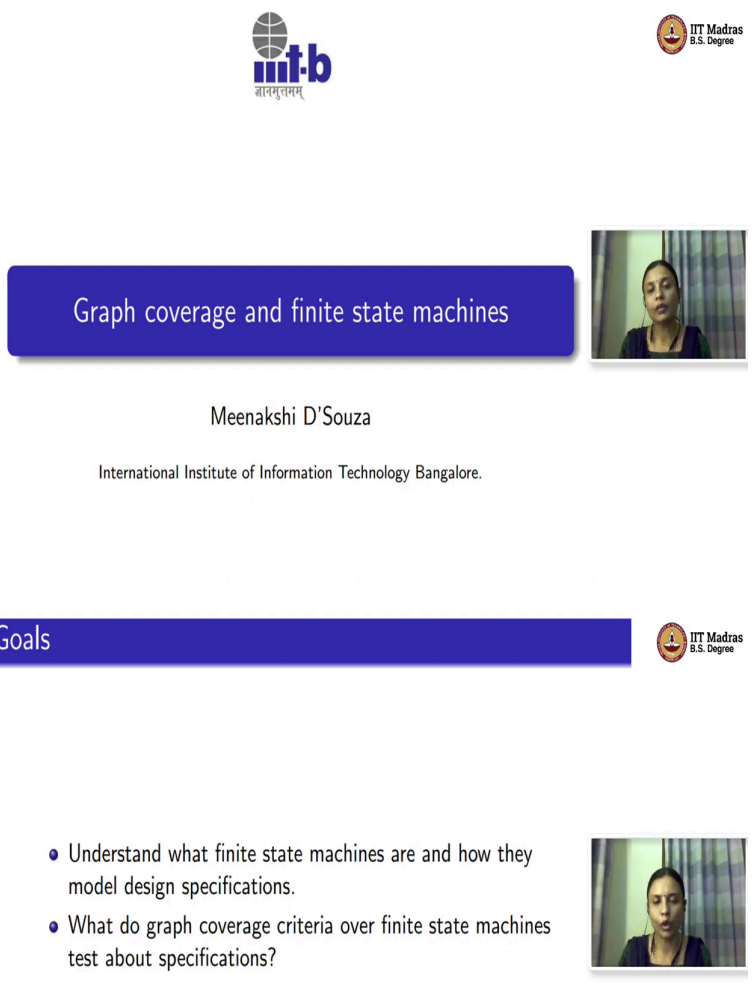
Software Testing
Professor Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore
Graph Coverage and Finite state machines

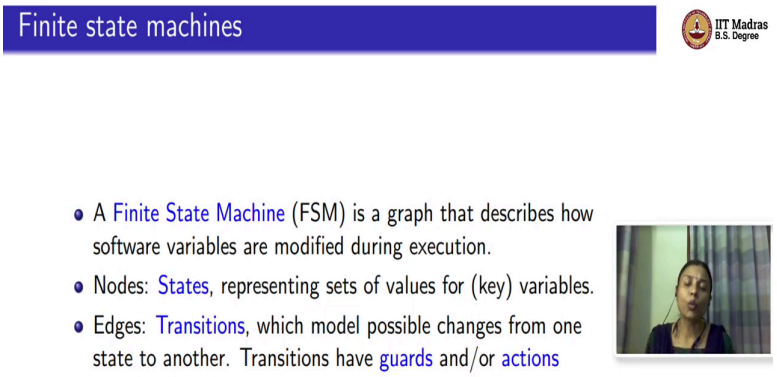
(Refer Slide Time: 0:15)

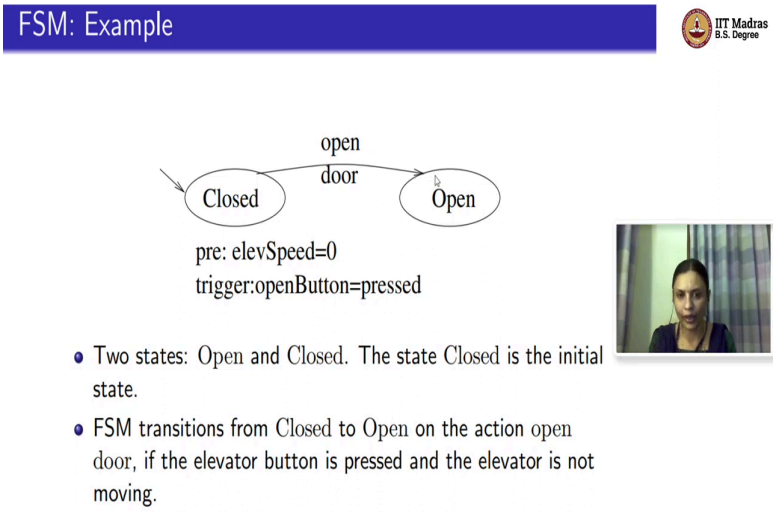


Welcome back. This is the last but one lecture of week 4, where we are still looking at graphs and testing based on graphs we saw graphs for control flow, data flow, call graphs, graphs over sequencing constraints. This is one last model that we will see for now before we move on. Graphs which are of a special kind called finite state automata or finite state machines which are very useful entities in computer science. So, our goal is to understand what a finite

state machine is very briefly as we want them for testing and how they model various kinds of design specifications what the graph criteria over these kinds of special graph like entities look like and how why are they useful for modeling specifications.

(Refer Slide Time: 1:08)





So, finite state machine abbreviated as FSM is typically taught as a part of a automata theory or a computability course. It is an entity that as it is name says it has a finite number of states as many as much memory as it has, in it is states. The states are the vertices or the nodes of the graph and edges are basically transitions. So, finite state machine is a special kind of graph that typically describes how maybe a certain variables that are of interest to you in a program behave and the nodes of the finite state machines have a special name they are called states they represent values for variables, sometimes only a subset of the variables the ones

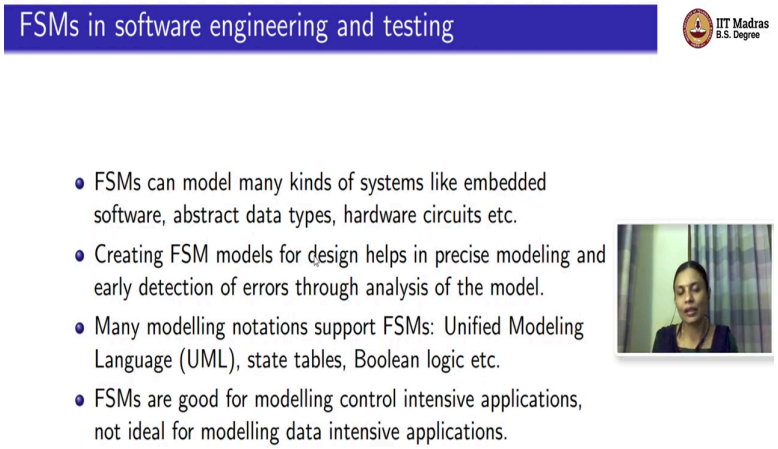
that are important for us edges of these graphs variabling finite state machines are called transitions.

They model how a state changes from one state to the other how the finite state machine changes from one state to the other transitions as edges can have several parameters associated to it, it can have guards which tell you whether a particular transition can be enabled or not when you are at a particular node or a state if the guard is not true then the particular transition or the edge cannot be taken to go to the next state. State I mean transitions in finite state machines also have actions which tell you what happens when that transition or the edge is taken.

So, here is a very small toy example it is a two state finite state machine, one state per node it has two nodes. This incoming arrow is the initial state it represents let us say the state of an elevator the elevators are these lifts that we use and the state it represents the states of the door of the elevator. So, it says the door can be closed and if somebody presses open door, the door can go to open state that is this edge or transition that transitions from the closed state to open state but for this transition to happen on this event of opening the door.

There are some conditions that must be satisfied a preconditional condition is that the elevator speed is 0 that is the elevator is not moving and the trigger is somebody is standing at the entry to the elevator and pressing the open button open door button. So, this is a small two-state finite state machine one state is closed which is also the initial state the other state is open which is the second state an action of pressing a button for opening the door results in the action open door and results in change of state by taking this transition as long as elevator speed is 0 that is it is not moving.

(Refer Slide Time: 4:05)



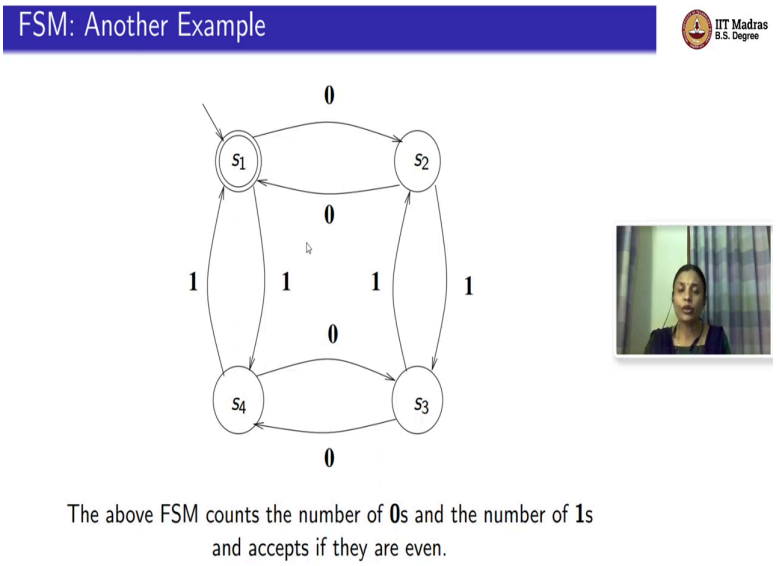
So, this is a very small example we look at more interesting examples. So, finite state machines as I told you are very useful models in computer science they can model several kinds of systems embedded software like for example, autopilot control logic can be modeled as a finite state system in a plane elevator states can be modeled as a finite state system, it can model several abstract data types we will see one example later in this lecture, it is very useful for modeling hardware circuits with gates Boolean logic and it gates it is useful for parity checking, it is useful for model or division several other simple tasks can be done by finite state machines.

If I create a finite state model for the task that I have in mind as a design or as a specification then it helps me to precisely model what we have in mind because these machines have a precise semantics and by doing it early in the life cycle by the design stage itself I can detect errors early by analyzing the finite state machine model typically there are several modeling languages in software engineering that support finite state machine the most popular of them is unified modeling language UML which has state machines and state machines which are richer with hierarchy and concurrency called state charts these are diagrams that are available as a part of the UML language.

Then you can write state machines as simple state tables and as I told you hardware circuits are Boolean logic or also state machines that are popularly used for modeling they are typically considered very good for modeling control application something happens then there is something as a result, something happens and there is something as a result but they

are not good for modeling applications that deal with lot of data like graphics applications and stuff like that.

(Refer Slide Time: 6:06)



Here is another abstract example of our finite state machine looks like by now all of you should be comfortable with reading this graph. There are four nodes in this graph when we call the graph as a finite state machine we do not call them nodes we call them states. So, the four states are S1, S2, S3 and S4 as the incoming arrow indicates S1 is both the initial state and as the double circle indicates it is also the final state and all these transitions or edges tell you how S1 state S1 changes to state S2.

So, input to this finite state machine is usually a binary string a string of zeros and ones and it takes the binary string and this finite state machine does the simple job of counting the number of zeros it sees in the string and counting the number of ones it sees in the string and if this number of zeros and number of ones both of them happen to be even numbers, then the finite state machine accepts it is input binary string, otherwise it rejects it, when it accepts an input binary string it will accept it by a path ending with the final state S1 if the path or the run of the finite state machine on an input binary string ends in other states like S2, S3 or S4 then you say the finite state machine rejects the string it will do so because the string will have had either odd number of zeros or odd number of ones.

So, how does this finite state machine work let us take a simple input let us say 0 0 1 0 as an input 0 0 1 0. So, if you see this string has three zeros and one 1 both are order number. So, this input string 0 0 1 0 should be rejected by the finite state machine how does the finite state

machine do that it starts from the state S1 first letter it reads is 0 it takes this transition goes to state S2 second letter it reads is also 0. So, it takes the lower transition from S2 back to S1 0 0 1 0 was our input. So, the third letter that it encounters in the binary string is 1.

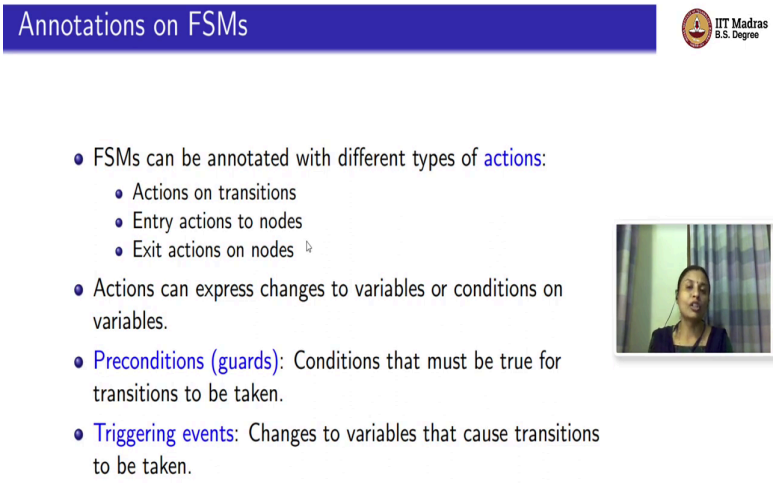
So, it is in state S1 goes down to S4 and the last letter that it sees in the binary string is 0. So, it uses this transition from S4 to S3 and ends up at S3 note that S3 is not a final state and because the finite state machine ends up there the path of the finite state machine as it comes while reading the string ends up in a non-final state this particular string is not accepted rightly so because the goal of this finite state machine is to accept strings with even number of zeros and even number of ones similarly suppose you give it a string like 1 0 1 0 will the finite state machine accept it yes it should because it has two ones and two zeros both are even numbers let us see how it does it always starts with the initial state S1 first string it is reading is one.

So, it uses this edge or transition goes to S4 it remember input is 1 0 1 0 then it reads the next second zero transitions from S4 to S5, S3 the third string letter that it sees is one. So, it uses this transition or edge from S3 back to S2 goes to S2 the last letter it sees in the binary string is back 0. So, it uses S2 back to S1. So, 1 go down 0 go right 1 go up again and 0 go left. So, S1 to S4 to S3 to S2 back to S1 because this input string took it back to S1 and it happens to have an even number of zeros and ones the logic or the design of the finite state machine is correct and it ends up accepting the string.

In general, if you look at it you know what is this how is this finite state machine designed if you see it has four states it uses these states to count keep a count at the parity of the number of zeros and ones. So, it says okay whenever I see an odd number of zero go out, if the odd number becomes even number of the same digit come back. So, that is what it does for state S1 if it sees first 0 goes out second 0 comes back similarly if it sees the first S1 then it goes from S1 to S4 second one comes back and so this is the logic that is used.

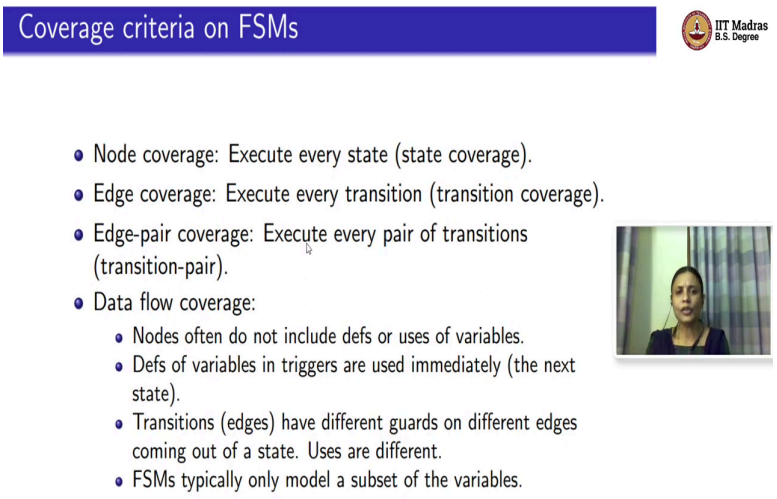
So, if it is able to correctly come back to where it started then it would have definitely counted even number of zeros and even number of ones it does not matter which order they occur because all possible combinations are taken care of here. So, this is the finite state machine that does simple counting and accepts even number of zeros and even number of ones. So, like this you can design lots of elementary finite state machines which are very useful for us our goal is to consider them as graphs and see what their role they play in software testing.

(Refer Slide Time: 11:32)



So, as I told you this was a very small FSM this was even smaller the open door closed elevator door they can in general model very rich control logic they can be annotated with different types of actions, actions that happen on transitions, actions that happen when you enter states, actions that happen when you exit the states and these actions can represent what changes happen to the variables then as we saw in the elevator door example it can have preconditions associated with states to ensure that a transition is enabled can have guards associated with the transition itself to check whether the transition is enabled when the guard holds it can have events that trigger a particular change of state to another stage. So, several different things are possible.

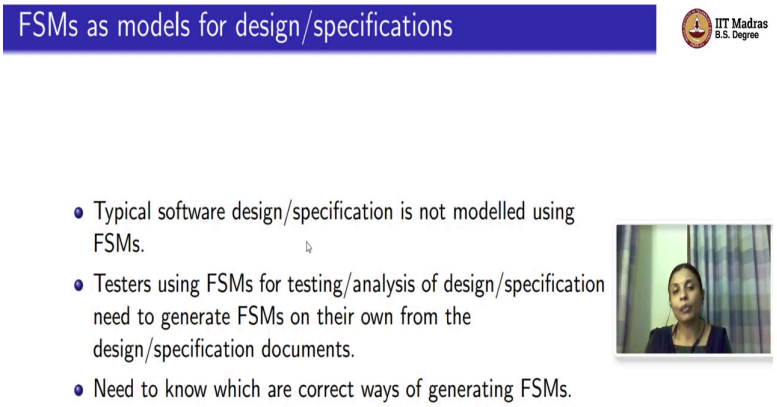
(Refer Slide Time: 12:22)



Now for us a finite state machine is a graph. So, I can it has nodes for states it has transitions or edges. So, I can think of the same coverage criteria structural coverage criteria I can think of nodes which is execute every state it is also the same as saying state coverage reach every state. This next one is edge coverage which is execute every transition. So, because edges are called transitions in an FSM you can say this is transition coverage you could also say execute every pair of transitions or edges that is also fine prime paths I do not know whether it makes sense here but data flow makes sense because during machine I mean finite state automata are going to talk about a lot of data flow.

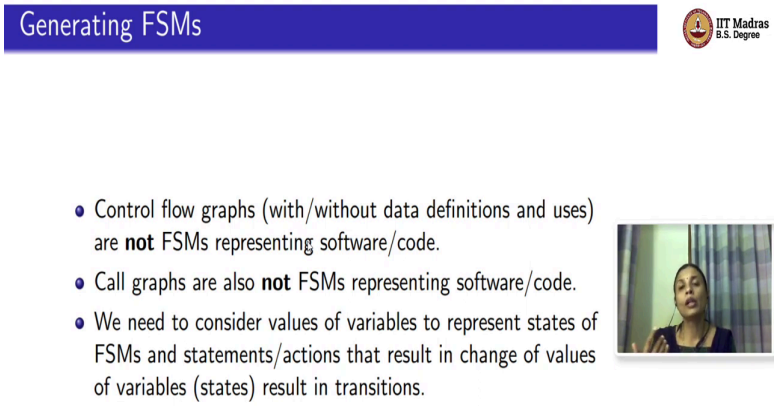
So, if you see the nodes in the finite state automata may not carry too many definitions because they do not represent statements of program but the definitions of the variables act as triggers that will help or stay a state machine move from one state to the other through a transition, transitions themselves can have guards. So, they exhibit uses of a particular state and based on my convenience I can take a subset of the variables and model it as a state machine.

(Refer Slide Time: 13:38)



So, we have seen it enough in abstraction. So, how do how do you model state machines. So, this is a tricky problem typically cannot be taught within the scope of one lecture or two lectures like this typically we do not model a full code or specification using finite state machine we abstract the features that we want to model check for ourselves whether we can model it as a finite state machine then go about modeling the finite state machine this needs some practice on how to generate correct finite state machines and you can run simulations to check if they are correct or not.

(Refer Slide Time: 14:13)

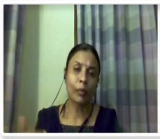


For example, it is wrong to take a control flow graph with or without definitions and uses of data variables and call them as finite state machines they are not finite state machines representing the behavior of code or a method or any piece of software. Similarly call graphs or data flow graphs that is control flow graphs plus definitions and uses of variables they are also not finite state machines representing behavior of code or any piece of software. So, they are control flow graphs, data flow graphs, call graphs are structural entities finite state machine represent the behavior of software.

So, we need to consider for that the change in the values of all the variables that will represent the states of the finite state machine and actions that will result in the change in the values of the variables which result in transitions. So, this is done by trial and error when you gain an experience but to convey the point about what I am trying to tell you I will walk you through an example of how to consider a particular piece of code or maybe a sequence of method calls and how to model a behavior of a code that uses this as a finite state machine.

(Refer Slide Time: 15:32)

FSM: Queue example



- We consider a class Queue with operations enqueue, dequeue.
- The queue is managed in the usual circular fashion.
- Methods of the class:
  - enqueue(): Adding an element,
  - dequeue(): Removing an element, and
  - isEmpty(): Checking if the queue is empty and return true if it is empty.
- For simplicity, we assume that the length of the queue is 2.

Queue

```
public class Queue
{
   // A typical Queue:[], [o1], or [o1,o2], where o1 and o2 are not null.
   private Object[] elements;
   private int size, front, back;
   private static final int capacity = 2;
   public Queue ()
   {
      elements = new Object [capacity];
      size = 0; front = 0; back = 0;
   }

x
```

Queue, contd.

```
public void enqueue (Object o)
throws NullPointerException, IllegalStateException
{
// Effects:  If argument is null throw NullPointerException
// else if queue is full, throw IllegalStateException,
// else make o the newest element.
if (o == null)
   throw new NullPointerException ("Queue.enqueue");
else if (size == capacity)
      throw new IllegalStateException ("Queue.enqueue");
   else
      size++;
      elements[back] = o;
      back = (back+1) % capacity;
   }
}
```

Queue, contd.

```
public Object dequeue () throws IllegalStateException
{
// Effects:  If queue is empty, throw IllegalStateException,
// else remove and return oldest element of this
if (size == 0)
   throw new IllegalStateException ("Queue.dequeue");
else
{
   size--;
   Object o = elements [(front % capacity)];
   elements[front] = null;
   front = (front+1) % capacity;
   return o;}
}
```

So, this is an example that we will walk through. So, let us consider that there is a class called queue which has operations like enqueue which is adding to the queue and dequeue which is removing from the queue it is a queue. So, it comes as first in first out order and the queue is managed in the usual circular fashion. So, enqueue is used for adding method enqueue a particular element dequeue is used for removing the element and let us say there is one more method in the class queue which checks whether the queue is empty or not.

So, suppose the queue is empty that is this is empty method returns true then you may not be able to dequeue it. So, it is useful method to have. So, the queue can be of any finite length but for the purpose of illustration assume that the length of the queue is true. So, let us start looking at the code. So, there is a class queue. So, let us let us say the for now just for our

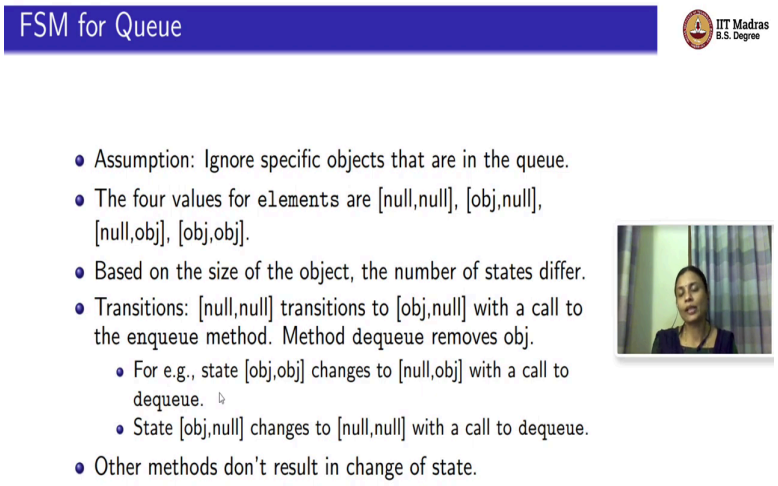
assumption assume that length of the queue is two that is it can have zero elements one element or at max two elements.

So, a typical queue can be empty as you see here it can have only one element or it can have two elements. So, this part declares the queue and it says capacity is two then it has the size of the queue front and back elements and then it initializes them and this is the code for the method enqueue. So, your idea is to enqueue an object o into the queue. So, this enqueue can throw null pointer exception if the queue is argument is null it can throw illegal state exception if the queue is already reached it is capacity and you are trying to add one more object using the enqueue method that is not possible otherwise it will correctly add this element o and make it available at the right place in the queue.

So, this code does that if o is null then you throw null pointer exception if the size of the queue is already reached the capacity of the queue then you throw illegal state expression otherwise increase the size by one size plus plus and then you push the element in the right place and increase the capacity.

Similarly, the method dequeue it if the queue is empty then I can not dequeue. So, it will throw an illegal state expression otherwise it will remove and return the oldest element of the queue. So, it says if size is the zero then throw an illegal expression otherwise decrease the size and take the element in the front reduce the capacity and return the standard. So, there is a queue class it has three methods enqueuer, dequeue and a check method that checks if the queue is empty and without loss of generality assume for now that the maximum capacity of the queue is very small it is only two.

(Refer Slide Time: 18:12)



So, a finite state machine for the queue will have things like this assume that we are not worried about what kind of specific objects go into the queue. So, there could be four values of the kind of things that go into, the queue the queue could be null fully empty or the queue could have objects at one end and null following it or the queue could have null at one end with object following it or it could be full with two objects those are the four values of this elements based on the size of the object the number of states can differ and we do not know what this object is.

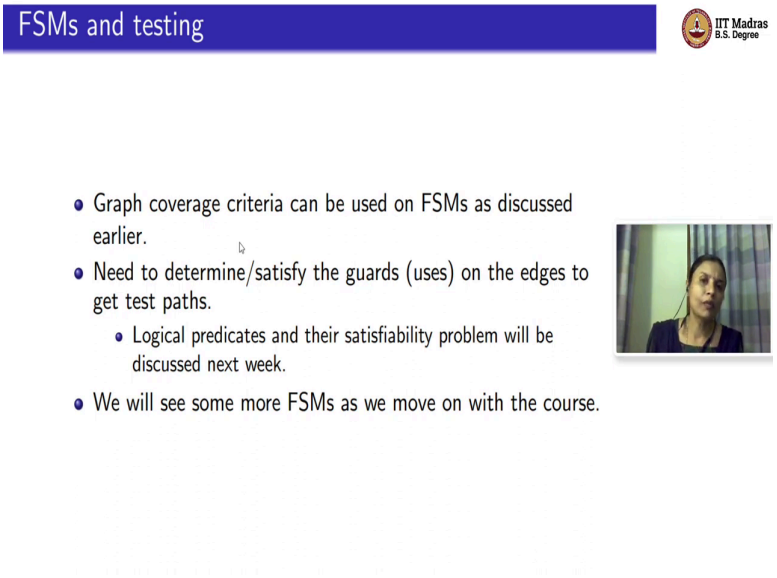
So, what would your transition look like suppose an enqueue method is called then null null will get replaced with object null because an object of ty type o has been added to the queue. Similarly, if a method dequeue will remove the object for example, if I have full queue object object after dequeueing I will get null object suppose I had object null then after dequeueing this object will get removed and I will get null null ideally I should have drawn this finite state machine but it would have a lot of options and become a cumbersome drawing.

So, I am just describing the states and transitions verbally is empty is another method in the class queue that is just a check it does not change the state of the queue. So, that does not come as a first class entity in the finite state machine mode. So, intuitively for a class queue like this that has all these three methods once a capacity is fixed what is a finite state machine model it models the possible values that the queue can hold at any point in time it says those values are all different states.

The queue can be fully empty it can hold one object at this end or that end or the queue can be full with both objects and calling of an enqueue or dequeue assuming that it works

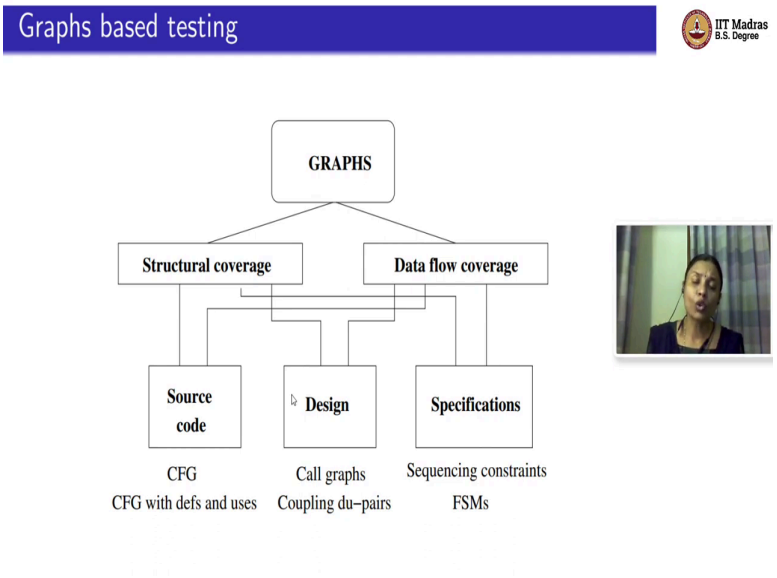
correctly and does not throw an exception results in change in the state of the queue and these are represented by transitions. So, this is a simple example of how I take something and model is a finite state machine I basically take understand what are the states by assigning capacities and values of variables and then I figure out what operations result in change of state from one to the other what operations result in transitions that is what I am trying to figure out.

(Refer Slide Time: 20:37)



So, as I told you graph criteria that we saw can be used on FSM I can do state coverage transition coverage covering pairs of transitions data flow coverage criteria everything the only thing is state machines can have guards or conditions on transitions or preconditions on the edges. So, you cannot simply do no coverage or rate coverage you might have to check whether the guard is true and if it is true then the edge can be taken. So, guards are basically expressions that involve some conditions relational operators, logical operators and stuff like that. So, how are we going to deal with it that is the next module that we will be starting with starting from week 5. So, and later in the course we will have the opportunity to see more such finite state machine like graphs.

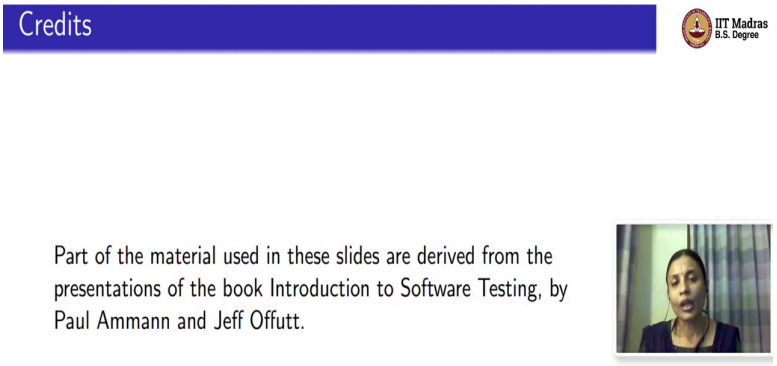
(Refer Slide Time: 21:29)



So, this is about the last lecture for now on graph based testing. So, we started with graphs some basic graph algorithms we broadly covered structural coverage and data flow coverage over source code, plain within a method within a function or procedure which was represented by control flow graph or control flow graph with data definitions and uses we covered design or design integration testing where control flow graphs became call graphs and instead of considering all the definitions and uses of data we considered coupling variables and the last definitions and first uses.

We also saw that graphs can be modeled specifications can be modeled as graphs particularly specifications like sequencing constraints and behaviors of certain entities can be abstracted out and modeled as FSMs. So, this last line below describes the kind of graphs that we saw kind of variables that we do and the top parts describe the software artifacts that we dealt with as graphs. When we later in the course do object oriented testing we will revisit this but for different kinds of graphs.

(Refer Slide Time: 22:38)



So, I will stop here for now and in the last lecture of week 5 a week 4 I will tell you about general popular coverage criteria on source code. Thank you.