Software Testing

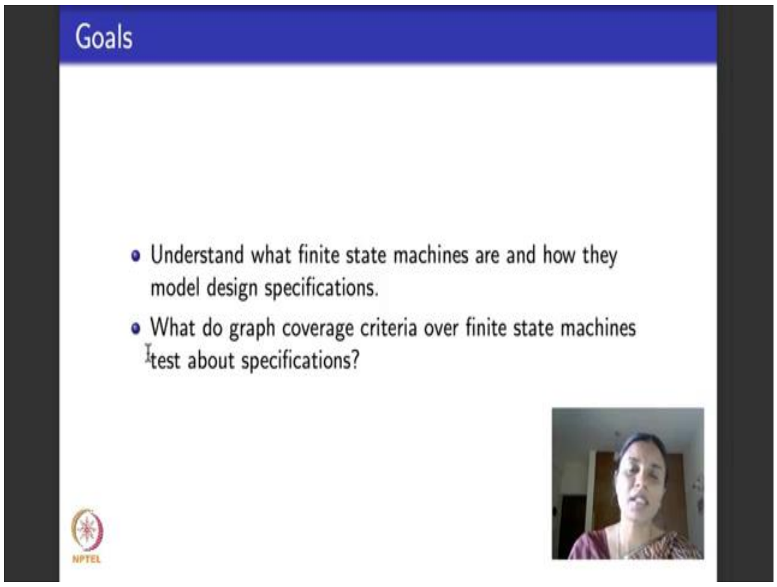
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 19
Graph coverage and finite state machines

Hello there, this is the last lecture of the fourth week, we will be winding up with looking at graphs today. This will be the last lecture where we will see testing based on coverage graph coverage criteria for now, what I will do today is introduce you to this popular model called finite state machines most of you might be familiar with it, and see what kind of graphs they are and how graph coverage criteria will be applicable to them.

So, far what did we see now? We saw graphs structural coverage criteria, data flow coverage criteria, then we saw how to apply to source code then we saw how to apply them to data flow along with source code and the CFG. Then we saw how to apply them to design elements specifically we saw how to apply it to sequencing constraints; and then I told you some amount of sequencing constraints need you to keep track of state information. So, when you need to keep track of state information, finite state machines or finite state automata come in to be very handy.

(Refer Slide Time: 01:10)



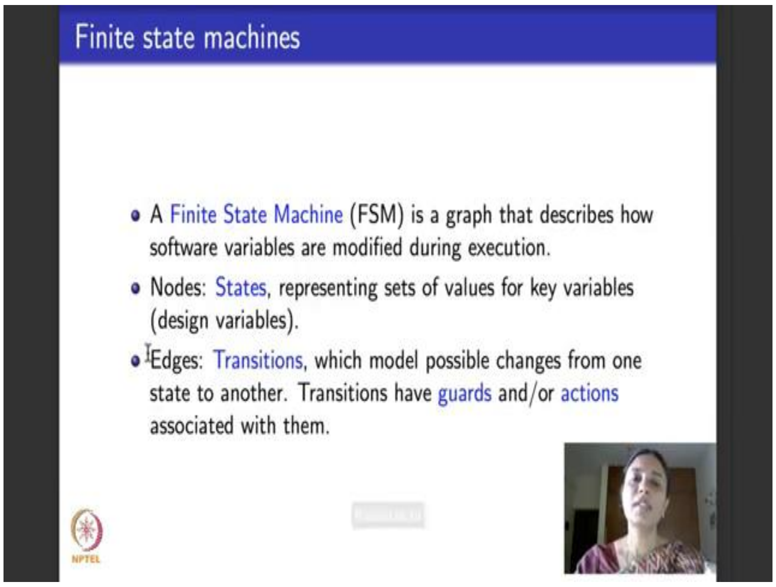
Goals

- Understand what finite state machines are and how they model design specifications.
- What do graph coverage criteria over finite state machines test about specifications?

So, today we will introduce those models and see how a graph coverage criterion is useful for them. Another popular design models are what are called UML diagrams.

UML is basically Unified Modeling Language, a collection of popular diagrams each of which is represented using both graphical and textual notation, and is used to document design. I will not really be able to give you detailed test case design for UML diagrams, but I will tell you what sort of diagrams basically correspond to finite state automata and how the coverage criteria that we saw till now can be used on them. At the end of this lecture we will revisit the entire graph based testing that we have looked at in the past three weeks, and I will briefly recap all that we did.

(Refer Slide Time: 01:59)



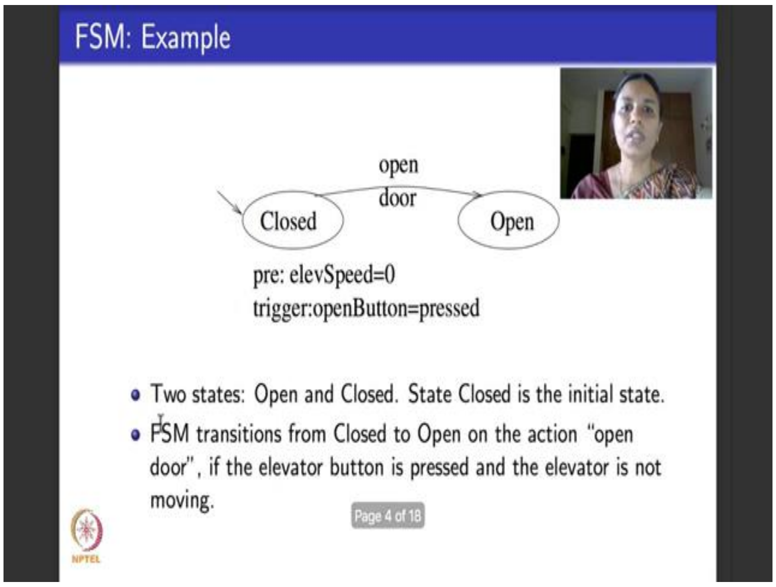
Finite state machines

- A Finite State Machine (FSM) is a graph that describes how software variables are modified during execution.
- Nodes: States, representing sets of values for key variables (design variables).
- Edges: Transitions, which model possible changes from one state to another. Transitions have guards and/or actions associated with them.

What is the finite state machine? I hope most of you will be familiar with it. If you have done computer science or IT you would have seen a course called automata theory or formal languages or theory of computation; finite state automata are one of the first models that you will learn in the course. So, a finite state machine can be thought of as a finite graph, the nodes in the graph or what are called states, and the edges in the graph are what are called transitions. So, what do the nodes describe? The nodes typically describe what happens to certain variables and software at a certain point in time. They tell you the values that these variables take at that point in time. What are transitions describe? They describe how the machine or the program changes state from one stage to the other.

So, if a stage depicts the values of certain variables at a certain point in time, transition might mean executing a particular statement in the program, which results in change of values of one or more states and this results in a new state. So, those are the edges of a finite state machine. Typically many interesting finite state machine models that I useful for software design always have guards, conditions associated with edges, triggers associated with nodes and so on. So, here is a simple example of a finite state machine, it has 2 states, 2 nodes; one called closed, one called open. States are always given these names with special identifiers which help you to identify what these states are while modeling using finite state machines.

(Refer Slide Time: 03:23)



FSM: Example

- Two states: Open and Closed. State Closed is the initial state.
- FSM transitions from Closed to Open on the action "open door", if the elevator button is pressed and the elevator is not moving.

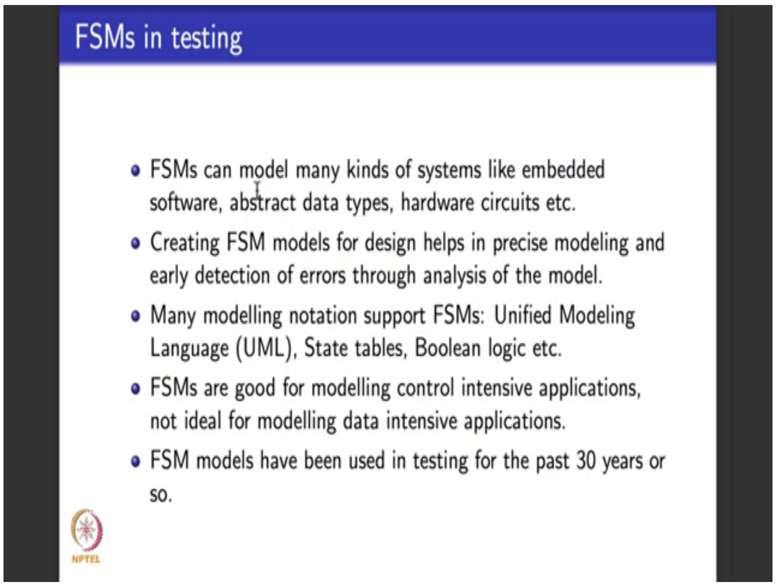
Page 4 of 18

As we have seen till now there is an incoming arrow at the state closed. So, this says that closed is an initial state. This small, toy finite state machine can be thought of as the finite state machine corresponding to an elevator. It is a very highly abstract machine, but it represents an elevator. The elevator could be, door could be the closed or it is open and it says that there is a transition or an edge from the state closed to the state open, when somebody presses this open door button.

You could think of it as there being a door button and then when you press that open door button, the state machine changes from state closed to state open. And what are the preconditions for this transition or this state change to happen? Preconditions, here it is abbreviated as pre in short for precondition, it says the elevator speed should be 0, which basically means that the elevator should not be moving, if it moves the door cannot be opened and it says that somebody should have pressed the open door button.

So, the precondition says that elevator should not be moving, its speed be 0, and the trigger is the open button command should have been pressed by somebody. If this happens and the machine transitions from closed to open. Of course, this is just a small machine, it does not really describe all the transitions or the full behavior or design of an elevator, I just took this abstract level description to explain what a finite state machine is to you.

(Refer Slide Time: 05:17)



FSMs in testing

- FSMs can model many kinds of systems like embedded software, abstract data types, hardware circuits etc.
- Creating FSM models for design helps in precise modeling and early detection of errors through analysis of the model.
- Many modelling notation support FSMs: Unified Modeling Language (UML), State tables, Boolean logic etc.
- FSMs are good for modelling control intensive applications, not ideal for modelling data intensive applications.
- FSM models have been used in testing for the past 30 years or so.

So, how are finite state machine useful for us in testing? Finite automata are very classical models, any course and automata theory or theory of computation would begin with them they are very robust models, but we would not really look at finite state automata and their theory and what they mean because our focus is to be able to purely understand it from the point of view of testing. Surprisingly, finite state machines have been around in testing for more than 30 years now, and they have always been considered is popular models that have been used for designing test cases to achieve some goal or the other with reference to testing.

So, typically finite state machines, what do they model? The model design of embedded software; software that sits in cars, planes, phones, toys etcetera. They model several abstract data types like for example, in the last lecture we saw queue with an abstract data type, right. So, finite state machines can model such abstract data types, almost all of hardware, hardware circuits can be modeled using finite state machine. Several protocols can be modeled using finite state machine.

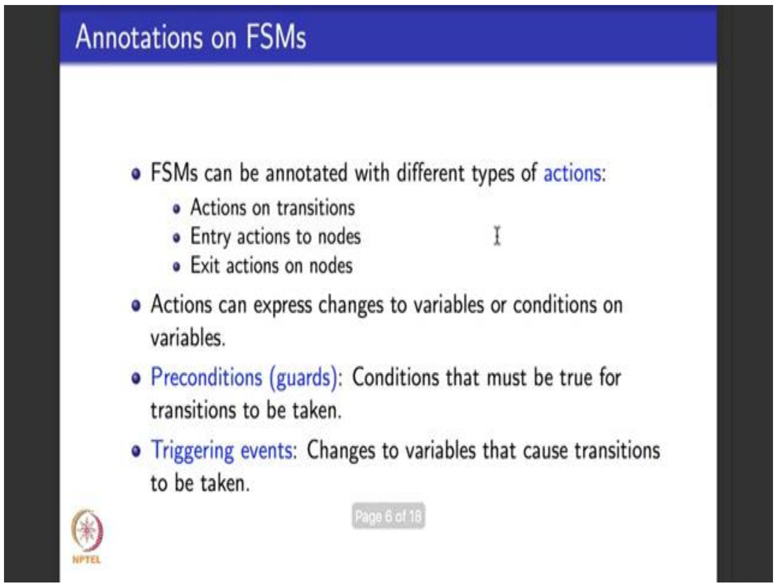
So, a large class of design and design element can be modeled using finite state machines and why does this help? Typically finite state machines model design and when I create FSM models for design, I am doing it before I write code. In fact, if my model is good enough then there are several tools that process finite state machine design models and automatically generate code.

So, when I come up with formal models of design, it is been found by past experiences in several projects, that such models help you to identify, test and detect errors early even before you begin coding, so that these did errors if detected early can save development cost and time. And the other thing is many as I told you many popular modeling notations also support finite state machines. The most popular modeling notation is that of UML. UML has a diagram one of it is diagrams is that of finite state machines, the other diagram that it has a state charts which can be thought of as finite state machines with hierarchy and concurrency specific finite state machines.

And the other thing to note is that whenever I have a control intensive applications right like elevator, something controlling something a software controlling an entity right controlling a piece of hardware or a system, finite state machines are considered to be very good models for that. But suppose I have a data intensive application like something that handles a database server, something that processes data and renders it as high speed GUI, FSMs are not considered good models for such things, we typically do not work with finite state machines for them right.

So, FSM, the summary of this slide is that finite state machines are reasonably popular. Fairly large class of software can be modeled the design of such software can be modeled using finite state machines.

(Refer Slide Time: 08:24)



Annotations on FSMs

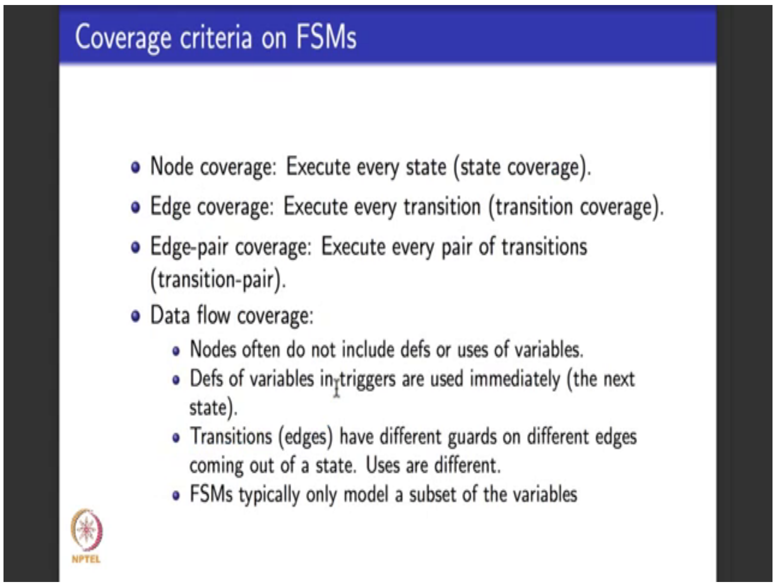
- FSMs can be annotated with different types of actions:
  - Actions on transitions
  - Entry actions to nodes
  - Exit actions on nodes
- Actions can express changes to variables or conditions on variables.
- Preconditions (guards): Conditions that must be true for transitions to be taken.
- Triggering events: Changes to variables that cause transitions to be taken.

Page 6 of 18

So, it helps to look at our graph coverage criteria on finite state machines and see what they mean. Before we move on I would like to mention to you that finite state machines are beyond just simple graphs. When we saw structural coverage criteria we saw plane graphs and when we saw data flow coverage criteria we saw graphs that were annotated with definitions and uses, finite state machines have lot more annotations than data flow graphs.

They can be annotated with different types of actions, actions could be on the transitions of the machine, actions could be on the node specifically as actions that dictate when to enter a node, actions could be on the node dictating when to exit from the node; and what do these actions represent? They represent change of values of variables, they represent change of evaluation of conditions, several different things right. As we saw in the elevator, a small example, there could be preconditions associated with nodes that tell you when to take a transition out of a node. There could be conditions or guards associated with edges that tell you when to take a particular edge, they could be triggering events and so on. So, finite state machines are graphs, but they have a lot of extra add on information to them.

(Refer Slide Time: 09:41)

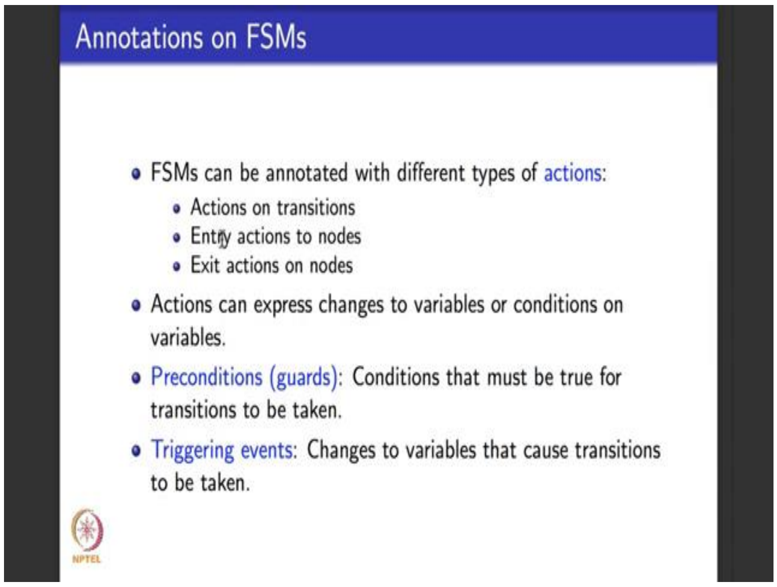


Coverage criteria on FSMs

- Node coverage: Execute every state (state coverage).
- Edge coverage: Execute every transition (transition coverage).
- Edge-pair coverage: Execute every pair of transitions (transition-pair).
- Data flow coverage:
  - Nodes often do not include defs or uses of variables.
  - Defs of variables in triggers are used immediately (the next state).
  - Transitions (edges) have different guards on different edges coming out of a state. Uses are different.
  - FSMs typically only model a subset of the variables

Now, let us look at the structural coverage criteria that we saw and see what they mean. So, simplest structural coverage criteria that we saw was node coverage. In the context of finite state machine it means execute every state. That makes a lot of sense no, when it comes to saying that if a machine models are designed, then you execute every state in that design right. The next coverage criteria is edge coverage, it means execute every transition which means execute every state change. The third coverage criteria is edge pair coverage, execute every pair of transitions. Now path coverage, prime path coverage is not very useful, because we do not really look at loops as they come from control flow graphs in finite state machines, what would be useful is specified path coverage. As we go down the course we will see some more finite state machines where this could be useful for us.

(Refer Slide Time: 10:49)

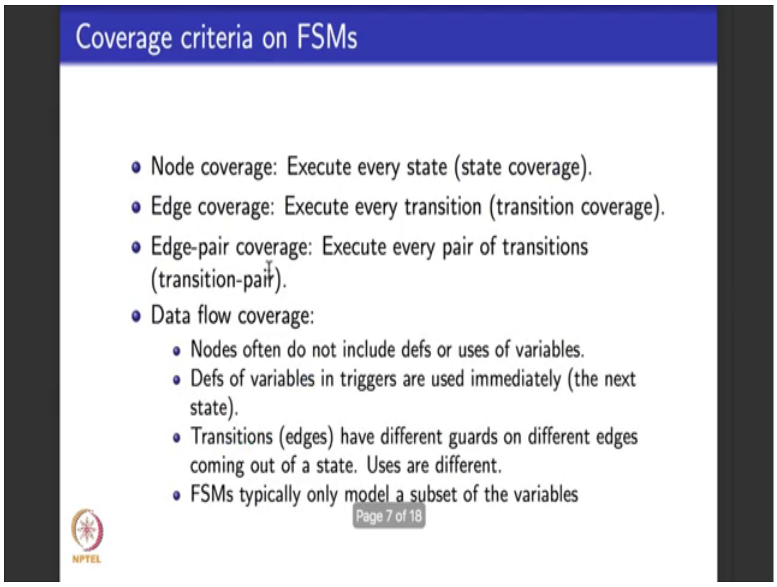


Annotations on FSMs

- FSMs can be annotated with different types of actions:
  - Actions on transitions
  - Entry actions to nodes
  - Exit actions on nodes
- Actions can express changes to variables or conditions on variables.
- Preconditions (guards): Conditions that must be true for transitions to be taken.
- Triggering events: Changes to variables that cause transitions to be taken.

Now, let us move on and look at date and flow coverage. In the data flow coverage when it comes to applying for finite state machines definitions and uses can get very complicated. If you go back to the previous slide I told you that there could be actions associated with edges or transitions and then there could be actions associated with nodes and those actions further on could be entry actions or exit actions.

(Refer Slide Time: 11:05)



Coverage criteria on FSMs

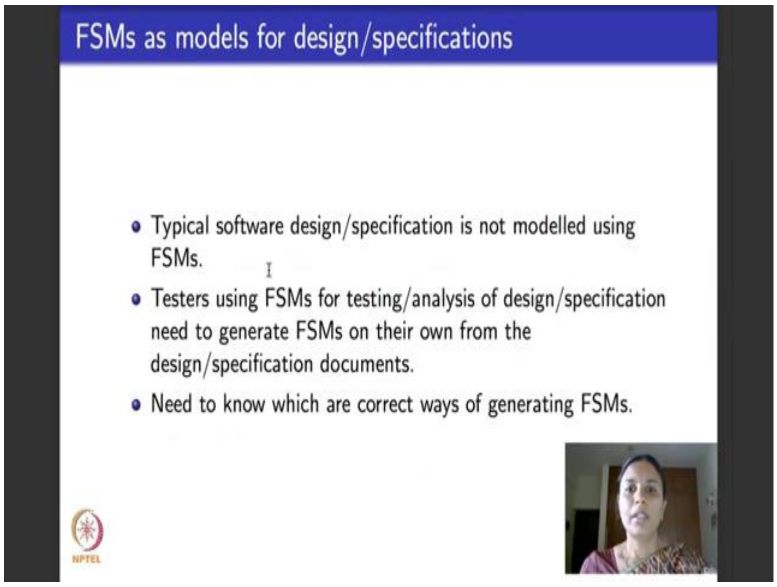
- Node coverage: Execute every state (state coverage).
- Edge coverage: Execute every transition (transition coverage).
- Edge-pair coverage: Execute every pair of transitions (transition-pair).
- Data flow coverage:
  - Nodes often do not include defs or uses of variables.
  - Defs of variables in triggers are used immediately (the next state).
  - Transitions (edges) have different guards on different edges coming out of a state. Uses are different.
  - FSMs typically only model a subset of the variables

Page 7 of 18

So, the there could be several different kinds of definitions associated with nodes and edges, and the other thing to be noted is that in when you look at control flow graph of code when we say annotations and edges uses on edges, they typically occur as predicates that correspond to branching, one will be one predicate the other one will be the negation of that predicate. In finite state machine, 2 edges coming out of the same node could have very different kinds of guards that are labeling them. Some of them could use a certain variable, some of them need not use the same variables at all they could use a completely different set of variables.

So, the users on each edge that comes out of a node could be very different from the other edge that comes out of a node. So, data flow coverage criteria gets a little cumbersome when we have to work with the term finite state machine. Instead what we will do is next week I will introduced you to logical coverage criteria; logic coverage criteria come in handy when it comes to handling data with FSM, we will revisit finite state machines at that time right.

(Refer Slide Time: 12:07)

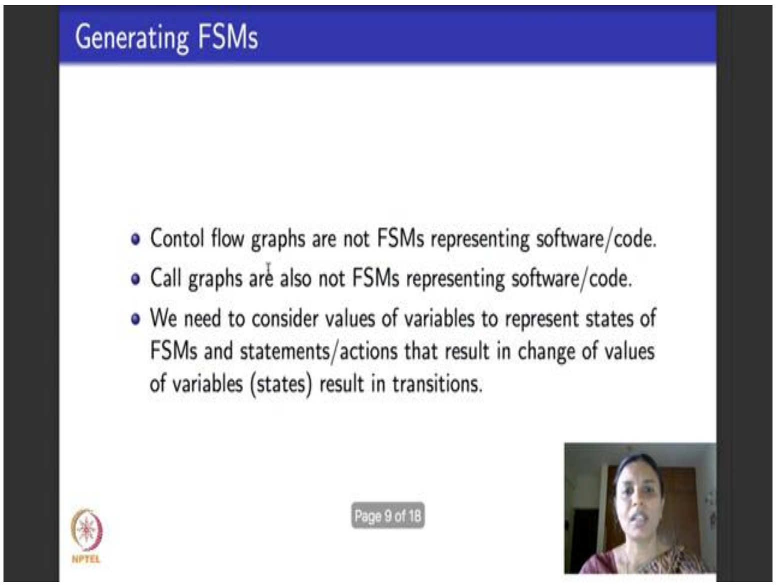


FSMs as models for design/specifications

- Typical software design/specification is not modelled using FSMs.
- Testers using FSMs for testing/analysis of design/specification need to generate FSMs on their own from the design/specification documents.
- Need to know which are correct ways of generating FSMs.

So, the other thing I told you is that finite state machines typically model software design or specification. And when you look at the large software organization none of the designers or architects typically sit down and draw or document the finite state machine for you. They typically write design or specifications as English statements and sometimes maybe some sketches and other diagrams, and whenever they are missing and if you find them to be useful for testing, as a tester the owners is on you to be able to draw finite state machines. So, it helps to get some idea of what exactly are finite state machines and what do they represent when it comes to modeling design of software.

(Refer Slide Time: 12:50)



Generating FSMs

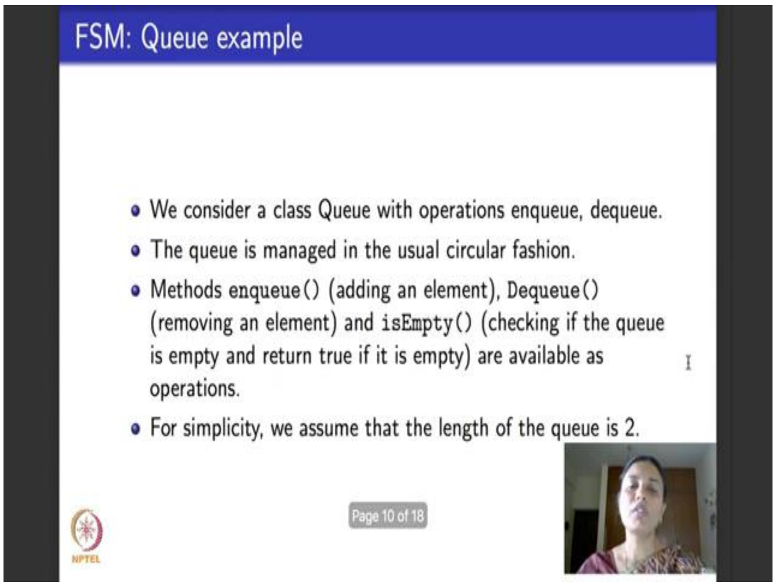
- Contol flow graphs are not FSMs representing software/code.
- Call graphs are also not FSMs representing software/code.
- We need to consider values of variables to represent states of FSMs and statements/actions that result in change of values of variables (states) result in transitions.

Page 9 of 18

So, one thing to note is that control flow graph which we learned how to draw, that is not a finite state machine corresponding to any design or specification or even code. It is not a finite state machine because control flow graph does not give you values associated with variables in states. So, it can never really be a finite state machine in the sense that we introduced it to be. Similarly call graphs which was another model that we saw when we saw design elements which talked about how one module calls another module, the call graphs are also not really finite state machines, again because they do not debit data associated with states and the finite state machines.

What we need to do when we define finite state machines is that we need to consider values of variables that are present in the code. Each state of a finite state machine can be thought of as a tuple containing the values of the designated set of variables at any specified point in time and then transitions in the finite state machine tell you how when some statement in the program executes, the values of one or more variable changes and how it affect these transitions.

(Refer Slide Time: 14:02)



FSM: Queue example

- We consider a class Queue with operations enqueue, dequeue.
- The queue is managed in the usual circular fashion.
- Methods `enqueue()` (adding an element), `Dequeue()` (removing an element) and `isEmpty()` (checking if the queue is empty and return true if it is empty) are available as operations.
- For simplicity, we assume that the length of the queue is 2.

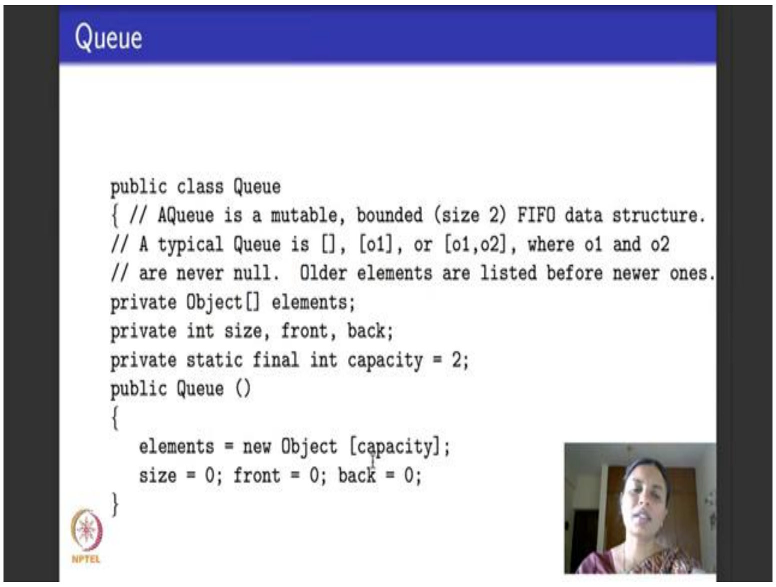
Page 10 of 18

So, what we will do is that if you remember in the last lecture I had introduced you to this example queue example and we looked at one sequence in constraint in the queue.

So, the class queue typically comes with several methods that support operations on queues, one method is that enqueue where you add an element of the queue, another method is dequeue where you remove an element from the queue, and you could also do things like methods which query and tell you is the queue empty is the queue full and so on. So, and we saw that a simple sequencing constraint like, for dequeue to happen and enqueue should have happened in the past could be described and tested, but more complex sequencing constraints saying that the number of enqueues at any point in time should be greater than or equal to the number of dequeues cannot be tested by directly writing them as sequencing constraints. We said last time that we would need finite state machines for testing that.

So, what I will do today is we will look at this queue example, and I will give you an idea about how a finite state machine for such a queue abstract data type will look like and because see when I talk about a queue, I need to implicitly assume a bound on the length of the queue. Suppose I say I can keep adding to the queue then in some sense the state space of the finite state machine will become infinite and it will no longer be finite state. So queues, we assume or of some bounded length for the sake of illustration to make things simple I have assumed that the length of the queue is just two. Of course, it does not correspond to a realistic model, to make it realistic you could assume that the queue is of fixed length n for some arbitrarily large n that you wanted to be, the same reasoning will extend for that length also.

(Refer Slide Time: 15:49)

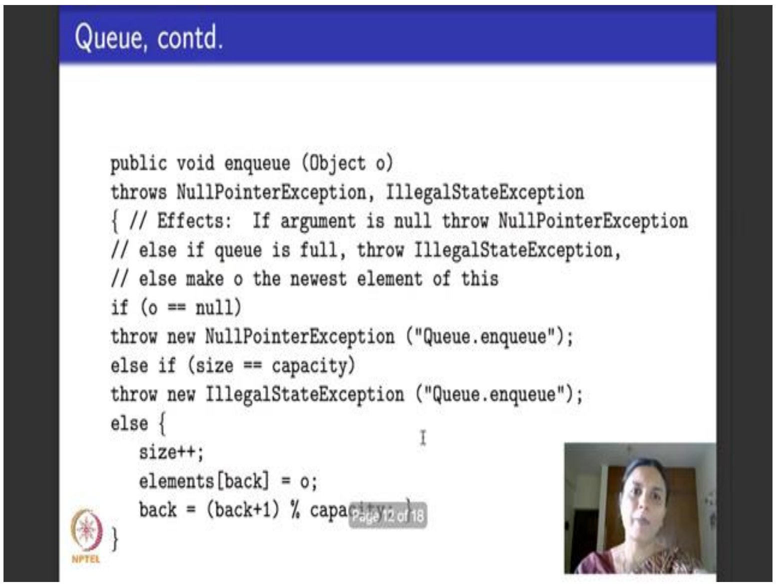


Queue

```
public class Queue
{ // AQueue is a mutable, bounded (size 2) FIFO data structure.
// A typical Queue is [], [o1], or [o1,o2], where o1 and o2
// are never null.  Older elements are listed before newer ones.
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;
}
```

So, here is a part of the code for the class queue. So, what is a queue? Queue is a mutable, bounded first in first out data structure. I have assumed the bound to be 2 for the sake of simplicity. So, what could queue look like? It could be empty it could just have one object or it could have 2 objects, we assume that these objects are not null and because it is first in first out the older elements are listed before the elements that are new in the queue. So, queue has things like size front and back, and it also it is main state element is set of an elements. We assume that the capacity of the side queue is 2. Initially, its size is 0, there is nothing in the front nothing in the back and element is something that I want to add now.

(Refer Slide Time: 16:37)



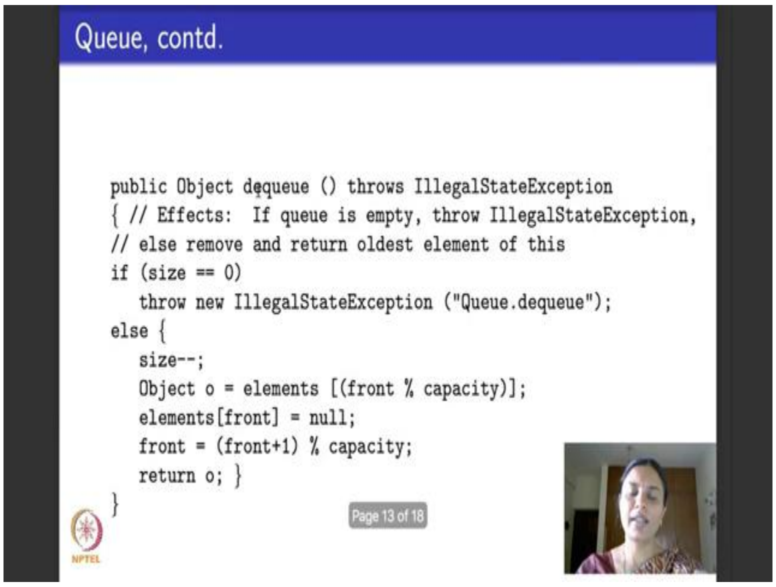
Queue, contd.

```
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 of this
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; }
}
```

Page 12 of 18

So, there are 2 methods that I have, in this slide I have presented enqueue method.

(Refer Slide Time: 16:39)



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; }
}
```

Page 13 of 18

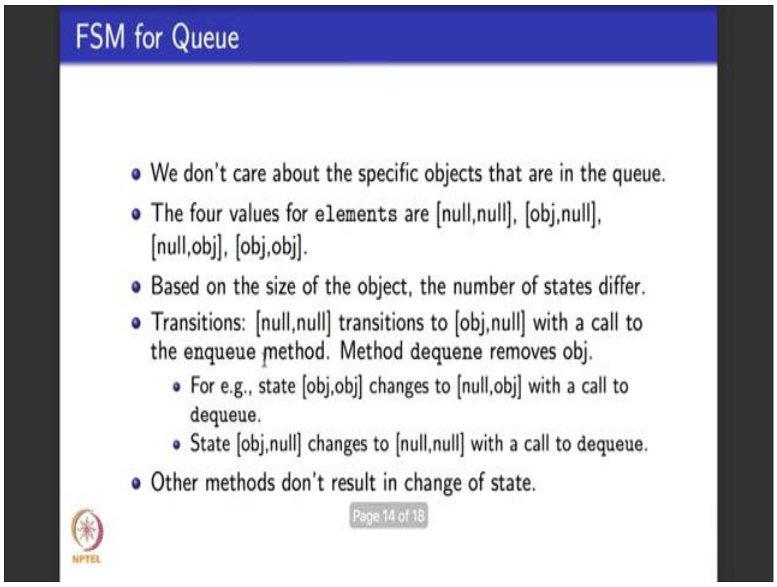
In this slide I have presented dequeue method of course, the queue class can have lot more methods like I told you, you could have a method just check if the queue is empty, you could have a method that checks if the queue is fully and so on. I have not described all those methods. For simplicity I have just given you enqueue and dequeue methods. So, we will see what enqueue does. So, enqueue inserts an object o into the queue and whenever it is not possible to insert, it could throw exceptions. It throws a null pointer exception if the argument to be inserted is null. We assume that the objects to be inserted or non-null as I told you here we assume that o 1 and o 2 are never null.

And it throws an illegal state expression if the queue is already full. So, it cannot insert any more element into the queue. So, this is the code for the method, it says if the object to be inserted is null you throw a null pointer exception or if the size is same as the capacity of the queue then you throw a illegal state exception, otherwise you increase the size and add the object over to the elements and increase the capacity, is this clear.

What is dequeue do? Dequeue tries to remove the topmost element from the queue. If the queue is empty and there is nothing to remove it will throw an exception, illegal state exception, otherwise it will remove the oldest element or the topmost element from the queue. So, if the size of the queue is 0, then you throw illegal state exception. Otherwise you decrease the size, you remove that object in the front because it is queue which FIFO first and first out and then you reset the new thing and adjust the capacity accordingly, and then the object that you removed you return that object.

So, now suppose we have to model the queue, the contents of the queue as a finite state machine, what would be a correct approach? The correct approach would be to first think about what is the state of the queue at any point in time. Remember that states in a finite state machine talk about the values of all the variables involved in a program or the design of a program and it actually depicts the existence, the state of the system as it would exist in real life. So, when I look at the queue data structure what is it is state? It is state is the content of the queue, what is the contents of the queue at any point in time.

(Refer Slide Time: 19:08)



FSM for Queue

- We don't care about the specific objects that are in the queue.
- The four values for `elements` are [null,null], [obj,null], [null,obj], [obj,obj].
- Based on the size of the object, the number of states differ.
- Transitions: [null,null] transitions to [obj,null] with a call to the enqueue method. Method dequene removes obj.
  - For e.g., state [obj,obj] changes to [null,obj] with a call to dequeue.
  - State [obj,null] changes to [null,null] with a call to dequeue.
- Other methods don't result in change of state.

Page 14 of 18

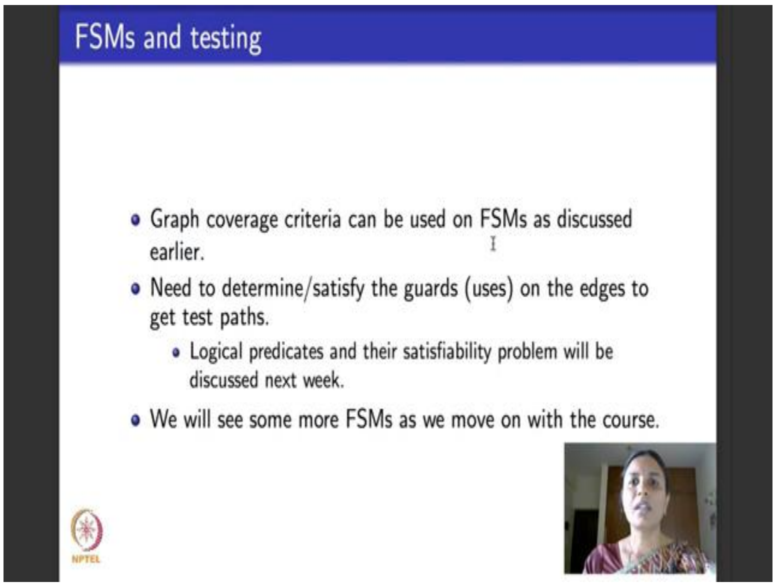
So, we do not really care about the specific objects in the queue, all that we want to know is that the; what are what is there in the queue. The queue could be have size at most 2. So, the queue could be empty there could be one non null object at the beginning of the queue, front of the queue, there could be one non null object at the back of the queue or the queue could be full, two non null objects right. So, these could be the 4 values of this variable element which describe the contents of the queue.

You go back to the queue code queue consists of elements and the queue is of size 2. So, all that I am tracking is what is the current content of the queue. Is the queue empty, does the you have one non null object, does the queue have one non null object in the front or at the back, does the queue have 2 non null objects? So, those are these values and then these are the states right. So, there could be it several different states you can compute how many states would be there, but about 6 of them would be reachable states. What would be the transitions. Like for example, suppose I have something like this [object, object] which what are the methods of the method calls that will change the state of the queue will a method call which says is the queue empty change the state? No, right?

Because what will is the queue empty do? It will basically look at the queue check whether it is empty or not if it is empty it will say true if it is non-empty it will return false. It is not going to be able to change the content of the queue. But methods like enqueue and dequeue will change the state of the queue because they alter the variable elements they alter this variable elements. When a enqueue, I add something it will the element when I dequeue I remove something from the elements. Like for example, here is the transition suppose I begin with [null null], that changes to object null when I enqueue an object and similarly when I dequeue an object, if I have a full queue and I dequeue the topmost object, that gets replaced with empty and then it becomes like this. If the queue is a capacity one which it looks like this it has 1 non null object if I dequeue again, then the queue changes to a null queue right.

So, this is how the state machine for a queue would look like. I have not really drawn the state machine pictorially I would like to leave that as a small exercise for you to do. Assume that these are the states of a state machine and draw edges to depict the change of states, and label those edges with method names like enqueue and dequeue which tell you when you do an enqueue and when you do a dequeue, how the state changes in terms of elements getting added or removed from the queue respectively? Try and draw this finite state machine as a small exercise that you could do for yourself.

(Refer Slide Time: 22:05)

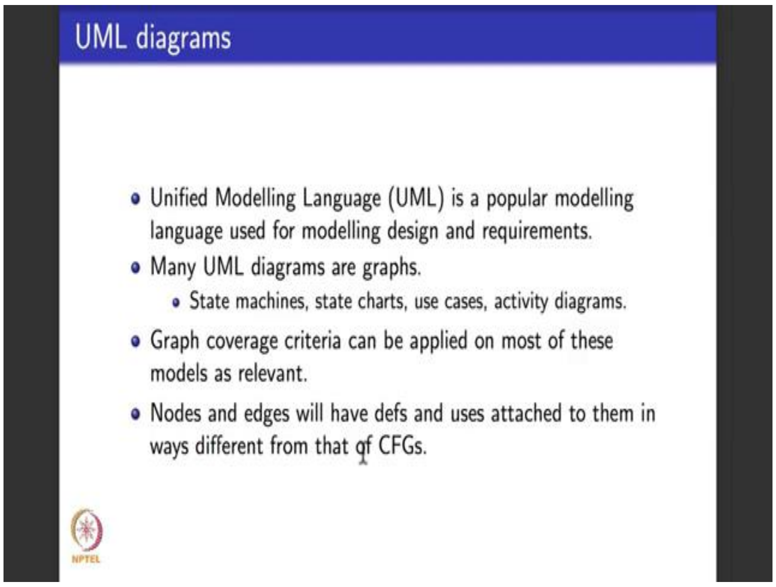


FSMs and testing

- Graph coverage criteria can be used on FSMs as discussed earlier.
- Need to determine/satisfy the guards (uses) on the edges to get test paths.
  - Logical predicates and their satisfiability problem will be discussed next week.
- We will see some more FSMs as we move on with the course.

Now, when it comes to testing, as I told you any kind of graph coverage criteria that deals with structural coverage criteria can be used except for prime paths,. We specifically use node coverage, edge coverage, edge pair coverage and specified path coverage when it comes to finite state machines. Coming up with test paths for these coverage criteria can be non trivial because you have to solve for the actions and the guards that come in state machines. We will deal with it when we look at how to solve logical predicates next week, we will also see some more examples of concrete finite state machine as we move on in the course.

(Refer Slide Time: 22:56)



UML diagrams

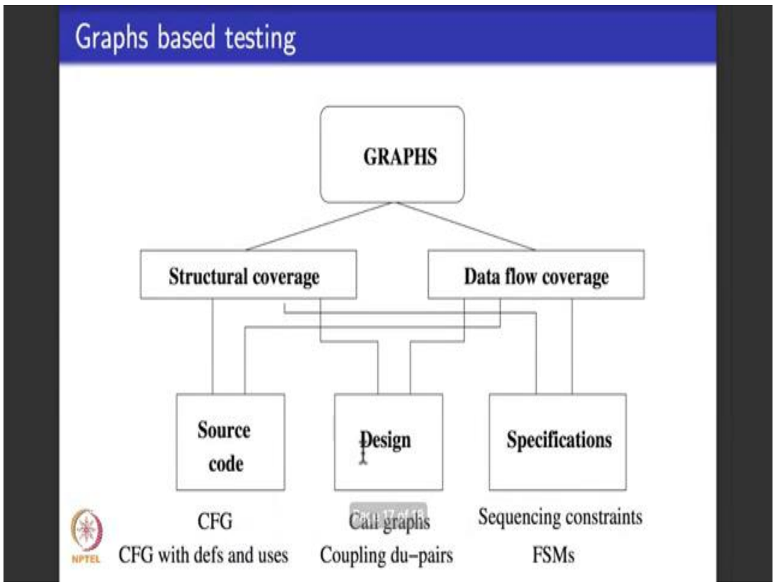
- Unified Modelling Language (UML) is a popular modelling language used for modelling design and requirements.
- Many UML diagrams are graphs.
  - State machines, state charts, use cases, activity diagrams.
- Graph coverage criteria can be applied on most of these models as relevant.
- Nodes and edges will have defs and uses attached to them in ways different from that of CFGs.

So, that is all I wanted to talk to you about finite state machines. I will briefly spend some time looking at some part of UML diagrams and tell you that the coverage criteria that we have seen till now can also be used to test UML diagrams. UML, in short for Unified Modeling Language, is a very popular modeling language used for modeling designs of systems.

There are about 14 or 17 UML diagrams, and here are the ones that look a lot like graphs: state machines, state charts, activity diagrams. They are basically some special kinds of graphs and the kind of coverage criteria that we have seen specifically path coverage criteria, specified path coverage criteria, can be used to test most of these graphs. One is to thing to be noted that each of this graph is very different from a typical control flow graph so you have to be careful when we define test cases and path coverage criteria on them.

As I told you I really would not be able to do UML now for most part of the course. If time permits towards the end of the course maybe we could look at UML diagrams and see is testing specific to UML diagrams.

(Refer Slide Time: 24:00)


Graphs based testing

We have come to an end of graph based testing. Next week onwards, I will begin logic predicate based testing. Here is a quick recap of what we have seen till now. The primary structure or model that we worked with for the past three weeks was that of graphs. The 2 main kinds of coverage criteria that we saw on graphs were structural coverage criteria and data flow coverage criteria. We defined these coverage criteria purely graph theoretically, we saw what are the various structural coverage criteria. If you want to list them, we saw node coverage, edge coverage, edge pair coverage, complete path coverage, specified path coverage, prime path coverage.

Then we saw subsumption, how each coverage criteria subsumes one or more of the other. Then we augmented graphs with data specifically with data definitions and data uses. Then we saw what are called def use paths or du-paths, and then we saw three main data flow coverage criteria: all defs coverage, all uses coverage and all du-paths coverage. Then what we did where we applied this graph coverage criteria that we learnt to source code, to design and to specification. In source code, we specifically learned how to draw a control flow graph for various code snippets, applied them to a full example and so, how the various structural coverage criteria can be used to test CFGs.

Then we augmented CFGs with defs and uses and saw how to use data flow coverage criteria on this augmented CFG. Then we moved on to design I gave you the basics of designs integration testing, then we saw call graphs and we saw how to apply structural coverage criteria on call graphs; then we saw coupling du-pairs: variables that are defined in one used in the other, right from actual parameter to formal parameter, from a caller method to a callee method, and augmented this data flow coverage criteria to coupling variables. Finally, the last couple of lectures we saw how to use graph coverage criteria on specifications. I told you how to use graph coverage criteria to model and test sequencing constraints and in today's lecture we saw finite state machines. This week's assignment will deal with this part, I will give you assignments the talk about data flow coverage criteria design and specifications.

Next week I will upload a video on how to solve that assignment. Please try to do it before you see the video once, it give you a good practice, and that will be the end of graph coverage criteria for this part of the course. We will move on to looking at logical predicates, define what are the coverage criteria on logical predicates and how to apply them to code and to specifications.

Thank you.