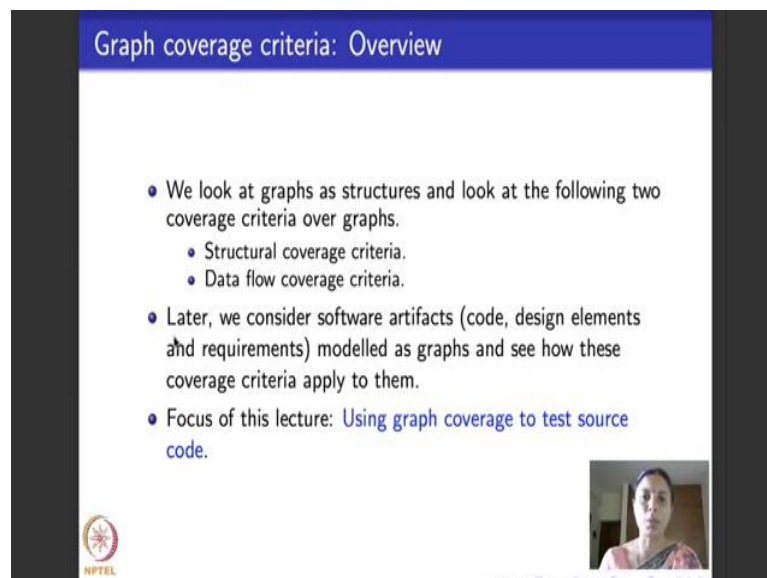**Software Testing**
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 13
Graph coverage criteria: Applied to test code

Hello again, we are in week three. What I will be doing today? We will actually begin to do real testing algorithms today. So, what we saw till now? We saw various coverage criteria over graphs structural or control flow coverage criteria, and then we saw data flow coverage criteria. While doing that we did two things--- we considered graphs as just models and defined the coverage criteria, saw what they mean and also discussed their subsumption relations. And while doing that I gave you a few examples, but I did not really tell you how to use these graph based coverage criteria to actually test software artifacts.

So, what we will do from today's lecture onwards is to consider software artifacts one after the other and then see how they can be modeled as graphs, and how we can use the various graph coverage criteria that we have learnt to be able to actually test the software artifacts. The first software artifact that I will begin with is code - source code because that is the most commonly available source software artifact, in fact it is the most exhaustive software artifact apart from testing.
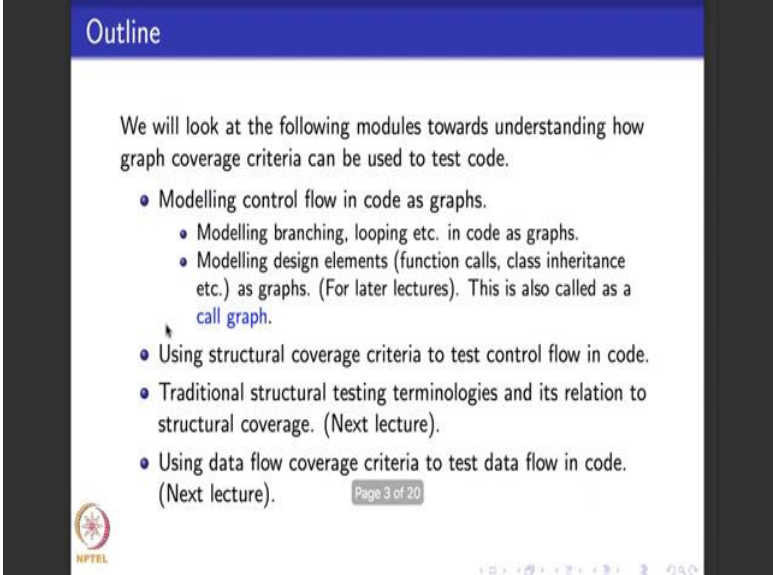
(Refer Slide Time: 01:12)

So, we will take source code we will see how to model it as graphs, and then we will look at one after the other, the various coverage criteria that we have learnt and see how we are going to apply them to actually test the source code.

(Refer Slide Time: 01:41)

So, when it comes to source code, how are we going to model source code as graphs? The most common model of source code for graphs is the control flow graph right. So, control flow graph typically models branching, looping, you know sequential calls of statements and so on and so forth. It also models things like function calls, class inheritance and so on. So, these are called call graphs or inter procedural call graphs some times and when you club them together along with the control flow graph you could call it as an inter procedural control flow graph.

What we will see today is to take one, assume that the code consists of one method, one procedure or one function not several different function calls, functions calling each other. We just take this we will try to model it, first see how to model it is a control flow graph, and then see how to use that a structural coverage criteria that we have learnt to be able to test various aspects of this code. In subsequent lectures, I will teach you how to augment this control flow graph with information about data, in particular definition and uses, and how to use the data flow coverage criteria to test these graphs.

In between, you might be wondering, I know testing a little bit, and I know several common terminology is in testing. So, what we will do is we will spend one lecture

looking at what are the common terminologies that I used when it comes to testing source code, and source code with graphs and see how what we are doing relates to what is commonly understood.

(Refer Slide Time: 03:22)

So, what will we be doing today? We will be doing following. We will take source code the code that we are going to test; I will assume without loss of generality that the code is contained within one method or one procedure does not have too many procedure calls. We will look at that when we look at design later on. And I will also assume when I do examples that you are familiar with programming languages like C or Java right, the basic how to read code. So, I will tell you how to take this code model this code as a control flow graph and then how to look at structural coverage criteria that we defined over this control flow graph abbreviated as CFG.

So, graph as we know has nodes and edges. So, what are nodes in the control flow graph modeling a code, nodes are basically statements or sequences of statements. They are commonly known as basic blocks, in the sense that is one continuous sequence of statement such that there is no branching in between, there is no if, there is no while, there is no for loops. It just may be series of assignments, a series of statements like assignments and printfs, series of asserts something like that, it just a continuous sequence of statements commonly known as basic blocks.

And then what are edges, edges basically dictate transfer of control. So, we see it through examples in detail. So, control flow graphs are many times annotated with what are called branch predicates which tell you which is the predicate on which you are branching. It comes as labels of the edges, they are also annotated with defs and uses of variable as we saw in the example. For the purposes of today's lecture we will look at examples, we look at annotations were branch predicates come. This defs and uses which deal with augmenting control flow graph with data, I will deal with it separately in another lecture.

(Refer Slide Time: 05:13)
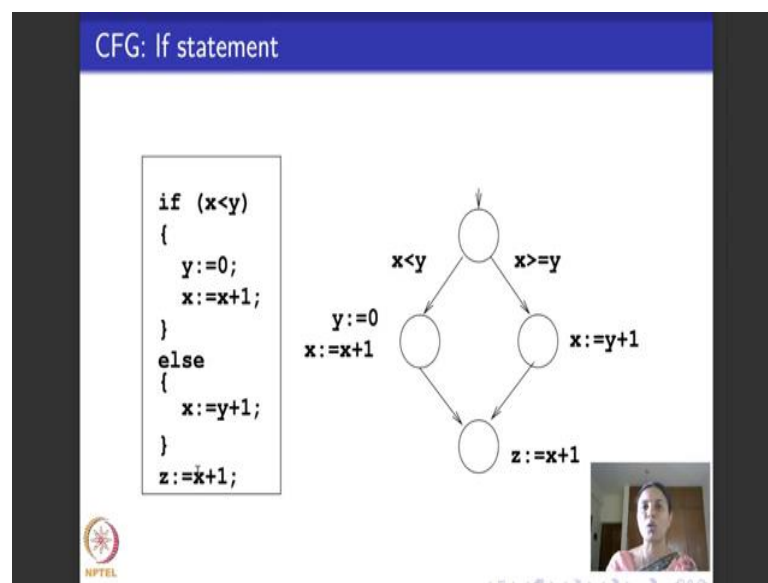
So, what we will do from now on for the next few slides is I will take each kind of software structure and tell you how with control flow graph corresponding to that software structure looks like. So, here is how control flow graph corresponding to a if statement will look like. So, let us say we have this code snippet. I told you when I show you one example like this, I am just showing you a fragment of the code, this is not a complete piece of code, definitely not compliable or executable as a standalone entity. We just look at a fragment because our focus is to understand for the concerned fragment how its control flow graph will look like.

Later we will take the full piece of code, a large example that contains all the statements and we will put together the CFGs that we learnt to draw for each of this fragments and see how the CFG for the whole code will look like right. So, what I will do is I will look

at various codes structures like branching, looping, exception handling and so on and tell you how the CFGs for each of the those code structures will look like. So, we begin with if statement because that is the simplest. So, let us say we had a code fragment that look like this. So, there is a statement which says if x is less than y, then you do these two statements--- assign 0 to y and make x as x plus 1 else which means if x is greater than or equal to y, then you say x is y plus 1. And irrespective what you do when you come out of this if statement, you say z is x plus 1.

So, how does a CFG for this if statement look like. CFG looks like this. So, this node which is the initial node, what does it say. It corresponds to this if statement. So, you checks for this condition is x less than y. Suppose x is less than y then it take this branch or this edge where it executes these two statements which correspond to these two statements on the code. And suppose not, suppose x is not less than y which means x is greater than or equal to y then it executes this statement, it says x is equal to y plus 1. Irrespective of whichever statement it executes when it comes out of the if loop control flow structure it executes this assignments statement given here in the graph z is x plus 1. So, here is how a control flow graph or a CFG corresponding to an if statement looks like.

So, there is a node that corresponds to this check or the if statement, if that check is positive let us say if x is less than y, then it takes one branch depicted by an edge and executes this statements corresponding to that branch. If this condition is false then it takes other different branch which represents the negation of that condition; in our case it is a x greater than or equal to y, and it goes to a block where it executes the statements corresponding to the else branch. Whatever it is, it has to come back and merge for this kind of an example and executes the statement that is just outside if statement; in our case it is z is equal to x plus 1. So, I hope this is clear.
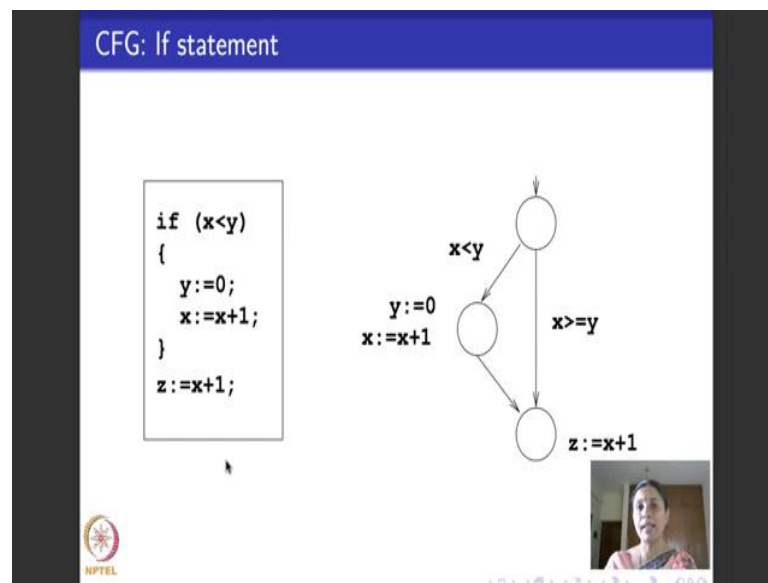
What we will do is suppose we had just if statement without an else part. So, in this example, we had a then part and an else part explicitly written out. Suppose, you just had an if statement which had a then part right I just remove the else from the previous example. How does its CFG look like? So, as you show let begin at node where I check for the truth or falsity of predicate x less than y. If x is less than y then I do this otherwise I just exit directly because there is no else part specified, I directly exit and come out to the node that represents the execution of statement that immediately follows the if statement.

So, now suppose I had an if statement with, but with this specially marked statement called return. So, let us look at this code fragment here. What it says is that if x is less than y then you return; and when you return you print the value of x because that is the statement that is immediately after this; otherwise you just return. This is some silly piece of code do not worry about what it is meant for and all. Our purpose is to understand how its control flow graph looks like. So, as always I start with one node representing this if statement where I check for truth or falsity of this predicate. Let us say it turns out to be true, I will come here, take this branch and then I do a return statement. Suppose, it is false then I come out, do a print x and then do a return statement. One thing to be noted is that please note that this return is different from this return. So, this is put by distinguishing two nodes corresponding to the two different returns that are there, one inside the if statement and one outside the if statement.

(Refer Slide Time: 09:58)
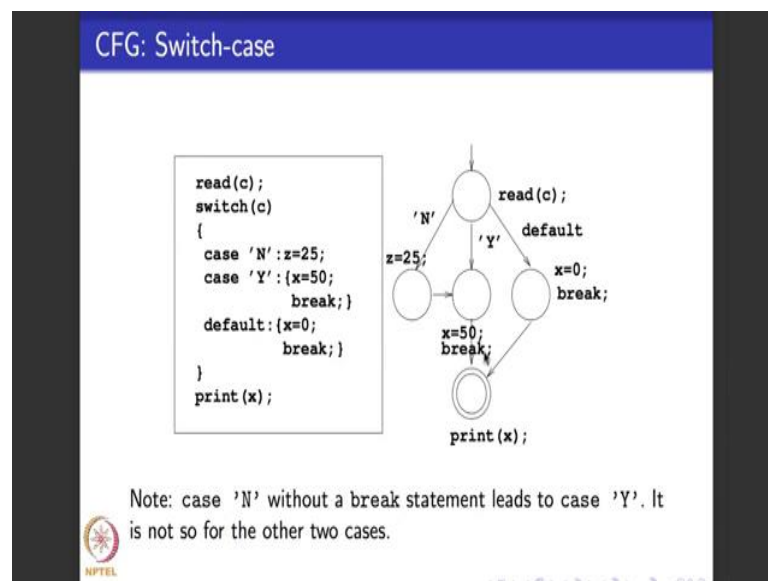
So, the next kind of branching that we will be looking at is what is switch case statement, it is another very common branching statement. So, let us look at the code fragment. What it says is that you read a value into a variable called c, c looks like a string variable and then you switch to this different cases based on the value of c. If c happens to be n the string n then you execute the statement z is equal to 25; if c happens to be the string y, then you execute these two statements--- x is 50 and break. Break means break out of the switch case statement. If it is none of these cases which means if it is the default case then you execute these two statements. One of the statements again is a break which tells

you to break out of this switch case statement. Whatever you do when you come out, let say you have a print x statement.

How does the CFG for this statement look like. So, I begin with a node which corresponds to this--- read c. And then I merge it with the same node where the switch c is also taken. I can keep another node which is serially connected to the node corresponding to read c there will be no big difference this CFGs are more or less the same. But in this case I chose to keep one node for both the statements. So, and then the next thing represents the three branches corresponding to the three cases of the switch statement. If case is N, I take this branch and do this statement z is 25; if case is Y, I take this branch and do these two statements x is 50 and break. Please read these labels corresponding to this node, they same to overlap with this edge, but they are labels that annotated this particular vertex. And if it x, if c is not N, if c is not Y then I am in the default case in which case I come out and do these two statements. And when I break, I go back here which is the print x statement that is present in this code.

Now, if you look at this CFG carefully there is one extra edge here. What is that extra edge represent? That extra edge says that the case for N if you see here does not have a break statement. If it does not have a break statement then as per the semantics, the typical semantics switch statement, I go and evaluate the next cases. So, what it says is that when it is the case for N you go ahead and lead it to the case for Y and continue from there on. If I do not want this edge then I explicitly put a break along with z is equal to 25 for the case for N. Because such a break is not there the case for N leads to the case for Y as per the semantics switch statement, and that is the reason why this particular edge exists in the control flow graph.

(Refer Slide Time: 12:48)
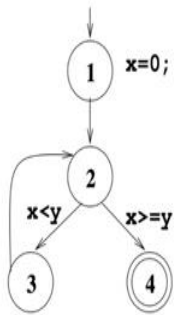
Now, what we will do is we look at loops. When it comes to loops we all know that there are several kinds of loops, there are while loops, there are for loops, there are do-while loops and so on. So, to model loops as CFGs will have to typically add a few extra nodes in the CFG. We will see through examples for each kind of loops how those extra nodes look like, and where are they added, and what do they represent as far as the loops semantics in execution is concerned.

(Refer Slide Time: 13:19)

So, we begin with while loop. Let us say you had a simple while loop that looks like this. You initialize x to 0 and then you say as long as x is less than y, you execute these two statements you call a function y. You call a function f with the parameters as x and y and you assign the value that f returns back to y then you do this simple assignment statement which is x is equal to x plus 1. How does the CFG for this code fragment containing a while loop look like? So, initially there is this node which begins for this assignment statement x is equal to 0, then I come and I have to do this node. So, one way of interpreting it is to consider this as a dummy node. Another way of dummy node or an extra node, another way of interpreting it is to consider node 2 as representing this while.

So, while this is true. So, while this predicate is true I go to node 3, where I execute the statements that occur inside the while loop. And I go back to checking for the condition in node 2 which means I go back to checking for the truth or falsity of this predicate. When this predicate becomes false that is when x is greater than equal to y, I exit out of this loop and I stop. And in this particular code, I have not really given you what we are doing when it exits the while loop. So, this does nothing annotating vertex four. And vertex four is marked as a final state because I stop there. So, is it clear please? How do while statement looks like? They will always have this kind of branching structure that is represented by nodes 2, 3 and 4. One branch we will represent the loop condition of predicate being true and it will keep looping as long as the predicate is true that is this branch between two and three. Another branch will represent exiting the loop that is when the loop predicate becomes false.

CFG: For loop

```
for (x=0; x<y; x++)
{
  y=f(x,y);
}
```

Note: Node 1 implicitly initializes the loop, and node 4 implicitly increments the loop.
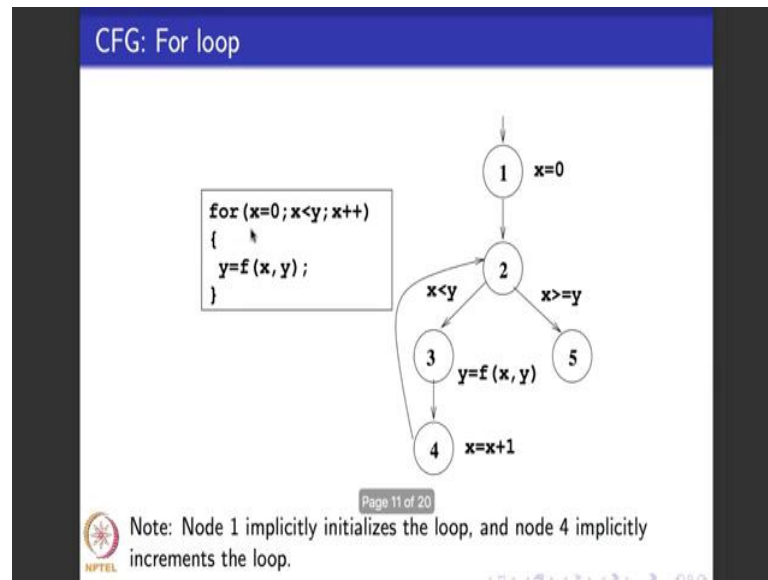
So, in the next example we look at for loops. So, here is very simple for loop.It says for x is equal to 0 as long as x is less than y, x++, you do this condition. So, next how does this look like. So, initially x is equal to 0 as labeled with an initial node. And then here comes this. So, node 1 is does not have explicit status in this code fragment if you notice. So, you can think of node one is a dummy node that implicitly initializes the loop; node 2 is the actual check for the predicate x less than y being true. So, if it is true, it goes through node 3 where the statement y is equal to f(x, y) is executed.

Now, after this what happens in the for loop, it is to go ahead and increment x using this x++. So, I add another node 4 which is actually a dummy node which implicitly represents x++ or x is equal to x + 1, it goes back after incrementing x checks whether x is less than y is true; if it is true comes back executes this increments x goes back and so on. So, this is how the for loop goes on between nodes 2, 3 and 4 and the cycle between 2, 3, 4 and 2. So, what happens is the predicate is checked, the statements are executed the variable is incremented predicate is checked again that is keeps repeating and when the predicate becomes false, that is when x is less than or equal to y, the CFG exists the for loop and takes this branch and comes to node 5. So, is it clear please how this we achieve for the for loop looks like.
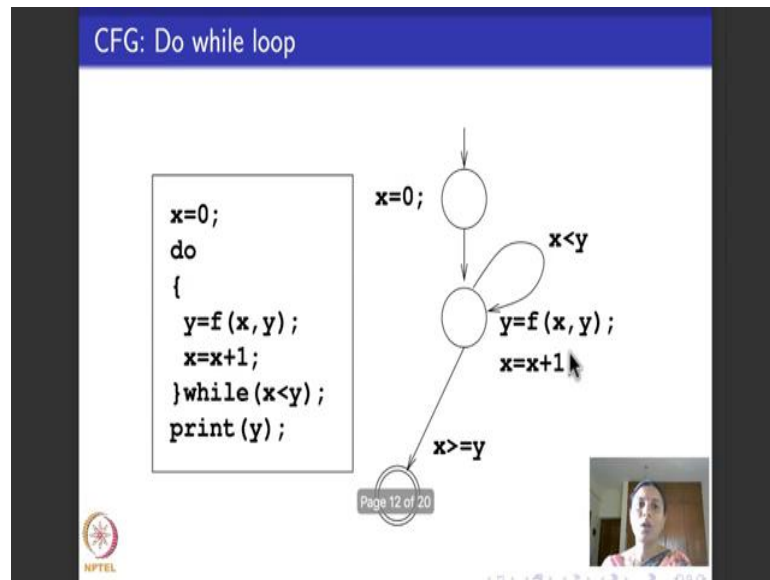
So, now we will move on and look at the control flow graph for a do while loop. So, how does the do while loop work? Unlike a while loop or a while do loop, do while loop will execute the statement inside the loop definitely at least once. It first execute the statements that come inside the loop; after that it checks for truth or falsity of the predicate that labels the loop that is what is depicted in this CFG also. So, here is a code fragment that has a do while loop. I begin with initializing x to 0, then I do the following as long as at the predicate x less than y is true. So, what do I do? I do y is equal to f(x,y) then I do x is equal to x plus 1. And when I come out of this do while loop I do print y right

So, as per the semantics first the statements are executed then this condition is checked, so that is exactly replicated on the CFG. I begin with the x is equal to 0, which corresponds to this statement then I come to a node where these two statements are executed, y = f(x,y), x = x +1 that is these two statements. And then I check for this condition x less than y that is represented as a self loop in this node because as long as this condition predicate x less than y is true. I continue to stay in this node where I execute these statements. And when this condition becomes false, I exit and go to a node where I have to actually do print y; I should have labeled this node as print y right because that is where I go to.

Please note the difference in the control flow structure for a do while loop and for a do, for loop and for a while loop. Say it for a for loop and while loop we had this kind of a branching, where I first check and branch for truth and falsity of a predicate. I go back in a loop whenever the predicate is true. And I branch out whenever the predicate is false same exists for while loop also. I check for the condition in a node I go back to the node as long as predicate is true after executing the statements that label the loop and I branch out and exit the loop when the condition is false. Whereas, for a do while loop, I stay at a particular node as long as the condition is true, and I keep executing this statements of that node as long as the conditions is true and I branch out when the condition is false. So, the loops sort of in this example shrunk to a self loop at one node.

(Refer Slide Time: 19:29)
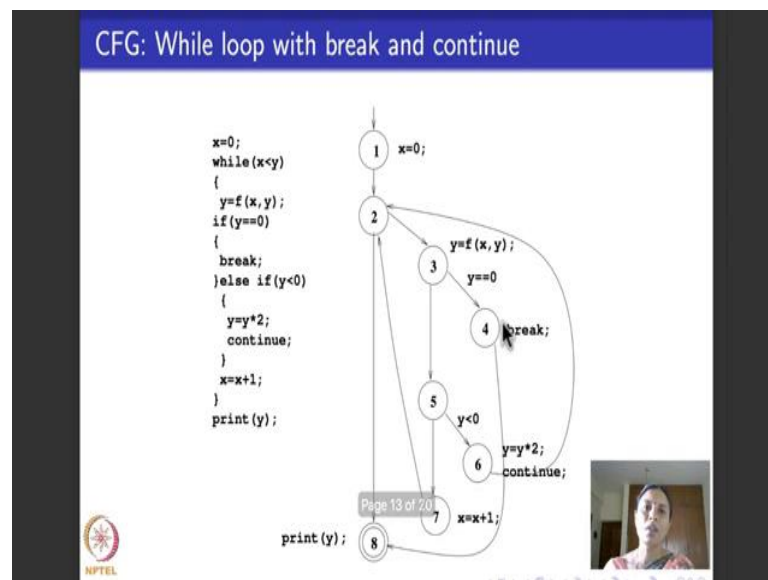
Now, I will show you slightly bigger example. Here again it is a while loop, but it has two special kinds of statements, statements like break and continue which make loop semantics little more interesting. So, here is a small code segment. So, it says start by initializing x to 0 as long as x is less than y, you execute this large while loop it begins here ends here, the last but one line. What happens inside this while loop? Inside this while loop I do several things I first do this y is equal to f(x, y) by calling the function f and then I check for two nested if statements. So, as a if y is equal to 0, then you break.
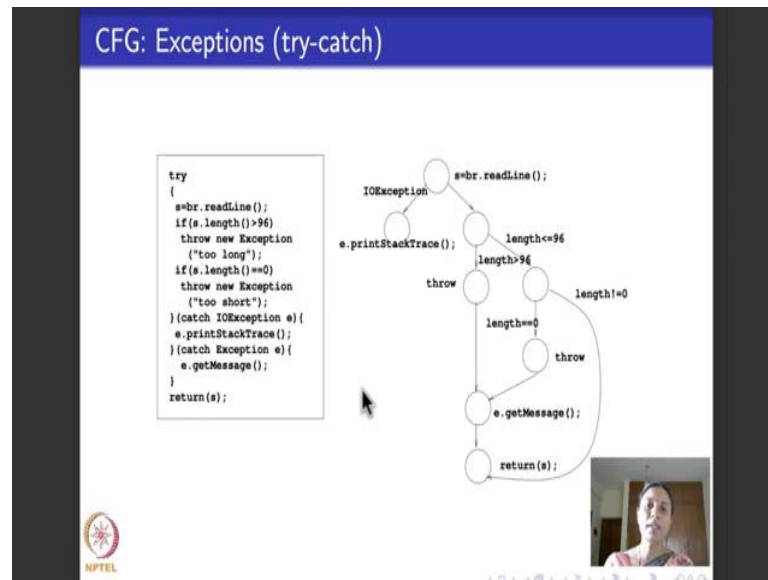
So, what is the semantics of this break? This break means where you will come, you will come here, you come to the print y statement. And then suppose this is not 0 then you

check if y is less than 0, if it is less than 0, and then you do something y = y+2 and then you continue. And when you continue implicitly what are you saying when you continue means continue into the y loop, go back and check for this condition that is what it is begin said. So, how does the control flow graph for such a code fragment look like? So, there is an initial node where x is equal to 0 is given, node 2 checks for the predicate x less than y, I haven't labeled the edges here to reduce the clutter in the control flow graph.

But suppose x is less than y is true, then I take this branch to 3, where I execute the statement y = f (x, y); after that I will check if y is equal to 0; if y is equal to 0. Then I break because at break I come out and go to the print f statement, which is that node 8. Suppose, y is non zero then I go here to the else part; in the else part the first thing that I do is to check if y is less than 0; if y is less than 0 then my code says you do these two statements y is equal to y into 2 and then you do continue.

Continue as I told you means that I go back and check the condition of the while loop. Suppose, y was not less than 0, y was greater than 0 then I come out of the second if statement and execute this statement x = x+1. But please remember even here I am within the while loop, so I have to go back through this edge from 7 to 2 to the main condition of the while loop node were the condition is checked and I repeat the sequence of executions. So, I hope this makes it clear how this semantics of while combined with if, break and continue works and how the control flow graph corresponding to such statements looks like.
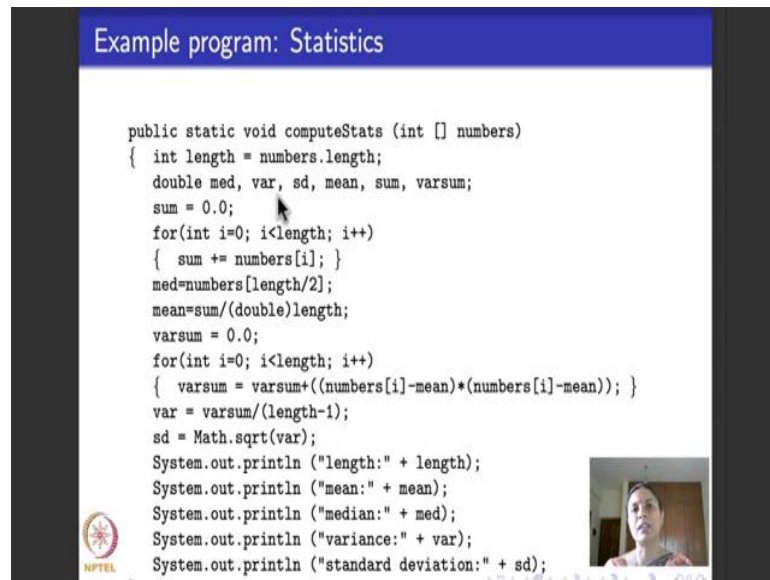
Now, here is another example of a control flow graph. You might be familiar with this exception handling through using try and catch in Java. So, how does the CFG if such try catch exception handling look like? So here is a again the small code fragment which involves the try catch piece of exception handling. So, it says there is a try here, there are two catches here. So, what it says is that you read the line, assign it to x, and you will see the length of the line that you have the read. If length is greater than 96 then you throw an exception saying is too long.

If length is less than 96 then you check if the length is actually 0; if length is actually 0, then you throw another exception saying too short right. And if it is too short then you go back and get the message; if it is too long then you go and print, if it is not too long sorry then you go and print this stack trace. So, the control flow graph of such a code fragment looks like this. I begin with reading line and assigning into s. Then what I do is I come here and check whether what is the length of the statement line that I have just read. If length is greater than 96, then I throw an exception and go out right to get another message. Suppose length was less than or equal to 96 then I now check if the length 0 or not. If the length is 0 then I throw another exception then I go back to get a message; if a length is non zero then I go and return s. So, this is the how to CFG corresponding to try catch statement looks like.

(Refer Slide Time: 23:47)

So, now, what we have done through all this slides where we have looked till now, is that I have shown you how to through examples how to draw the control flow graph corresponding to several different code constructs. Control flow graphs corresponding to branching which involve if with then and else, without else, if with return statements, control flow graphs corresponding to switch case statement, control flow graph corresponding into different kinds of loops - while loop, for loop, do while loop and CFGs that include while loop with branching and breaks and continue-s. And finally, in this slide we saw an example of a control flow graph for exception handling through a statement like try catch.

What now we move on we will see is I will show you of full piece of code, piece of code that does something, we will draw its full control flow graph and we will see how to use the various data structural coverage criteria that we have learnt to be able to test that source code. So, the code that I want to look at is an example of a program that computes the typical basic entities that deal with statistics like median, variance, standard deviation, mean and so on.

So, what it takes is that it takes an array of numbers and it returns what are the various parameters. So, it returns the length the array, it returns the mean value of the array, it returns the median, it returns the variance and it also returns the standard deviation. So, those are all these println statements. And what is the code do the code basically has to

for loops, one for loop right here and another for loop right here. The first for loop is use to compute median and mean, the second for loop is use to compute the variance and the standard deviation. So, I am sorry I put the code in smaller font to make it squeezed into one slide this curling closing brackets is sort of gone down, but this represents the full piece of code. It takes an array of numbers the code is called computeStats. And it outputs length, mean, median, variance and standard deviation.

How does it go about doing it, it says the length is an integer variable and the rest of the numbers are all declared is double, it initializes the sum to be 0. Then it goes into a for loop where it repeatedly adds the number in the array to itself, and computes the sum of all the numbers in this array, in this for loop. Then it says median is this, right, numbers of length by 2 which is the midpoint and it says mean is sum/double(length).

So, now this for loop computes the variance sum. So, initializes the varsum to 0 and then it gets into this. And what it does is the there it computes the varsum as this difference which is standard way of doing at in statistics. When it comes out it computes the variance is varsum divided by length of the array minus 1 and standard deviation is the square root of this variance. And then it has all these print statements that print the various values. Now, what is our goal, a goal is to take this piece of code and model it as a control flow graph and use the various structural graph coverage criteria that we have learnt to be able to actually test this code.

So, how am I going to model this code as a control flow graph? So, initially when it comes to CFG, please remember we ignore these entities, we ignore these statements. So, I begin with a node that initializes the sum that is what is given here as node 1. Then the next thing that is there is this for loop; it initializes i to 0 checks for this truth of this predicate and there is a node that increments i, whenever this predicate is true it assigns sum to be this value.
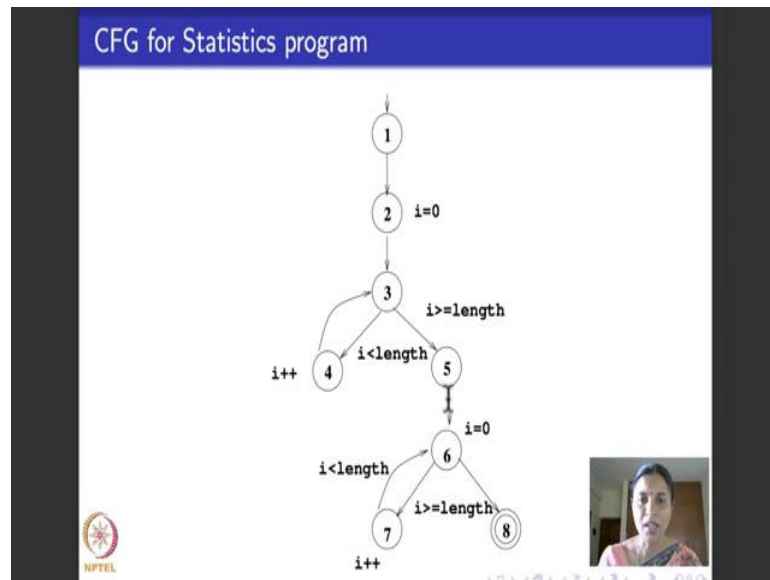
So, that is represented in this fragment of the CFG. So, it initializes--- nodes 2, 3 4 and 5 represent the first for loop. So, it initializes i to zero node 3 is where it checks whether i is still less than length or i is greater than or equal to length. If i is still less than length it increments i++ and adds the thing to sum. If i is greater than length it goes out here and then enters the next for loop which is this loop. So, here again I check for the same this thing and I do this. Just for simplicity and to reduce clutter I have not labeled this control flow graph fully with all these statements. Like for example, I have not used these two statements median its numbers.length by 2 and mean is sum of double(length), varsum is 0. Where should they all come they should all come here with labels of this node, but I wanted to focus only on structural graph coverage criteria.

So, I have just reduced myself to looking at the main CFG corresponding to this one. I have removed all the labels that we are use to annotating the CFG with. I have retained sum just to tell you that this little fragment here is for the first for loop, this is for the second for loop. Ideally speaking I should have put all other labels, but then the CFG would have become to cluttered and the focus on understanding this example for structural coverage criteria will be lost. So, I have just retained as minimal labels as possible, this CFG by no means is complete when it comes to a label annotations. So, now, if taken this code modeled it as a graph.

(Refer Slide Time: 29:57)
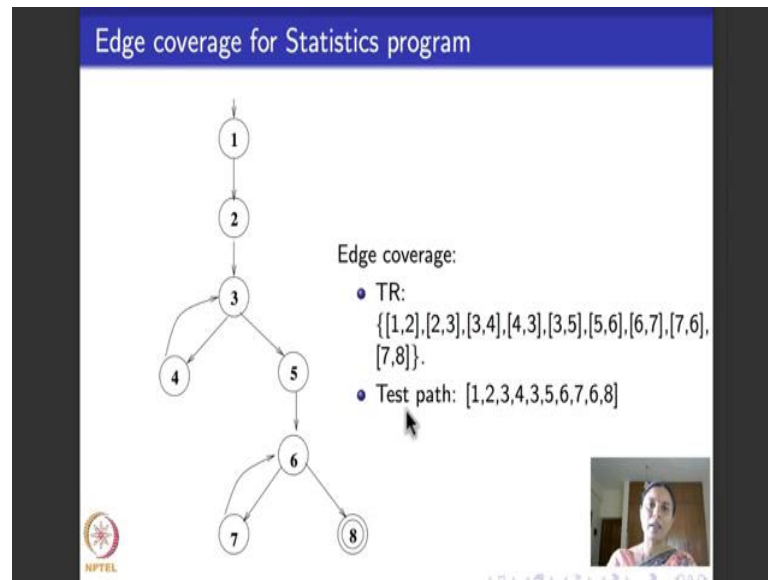
Our focus now purely on this graph, right? We want to now look at this graph go back recap all the structural coverage criteria that we have learnt till now and see how we can use some of them to test this graph. So, the first coverage criteria that I would like to, you can apply any of the coverage criteria that we have learnt till now. We will apply a small sampled two or three different coverage criteria for the purposes of understanding.

So, suppose I want to do edge coverage criteria for this graph. If you see you have taken CFG that was here and further removed all the labels retained only the core graph. Understanding is that this is the graph that corresponds to that java code that corresponds to this statistics program suppose I want to achieve edge coverage for this graph. What am I test requirements test requirement of TR says cover every edge. So, this is a set that contains the list of all the edges of this graph if you see as put all the edges in this set (1, 2), (2, 3), (3, 4), (4, 3) and so on.

And what is a test paths that will need this test requirement, it is fairly simple. I start from one I do (2, 3), (4, 3), I have covered these four edges. Then I take this branch, now 5 and then I do 6 and now I do not want to do 8 then I would have lost the edges 7, (6, 7) and (7, 6). So, I from 6, I got to 7, 7 to 6, 6 to 6 that is the path that I have to traced out here. I will just read out once again for clarity when I go from 1 to 2 to 3 and then move on to 4, come back to 3, come to 5, move to 6, come to 7, come back to 6 and then branch out to 8.

So, using one long test path that visits from the node 1, which is the initial node to the node 8, I have managed to achieve the test requirement of edge coverage. So, this is like a minimalistic test case, minimalistic because this is only one test path that is enough to test the coverage criteria of edge coverage for this graph. Of course, there nothing, no harm in you might say that this why one test path, I would write four test paths, I would write two different test paths like for example, I would write 1, 2, 3, 4, 3, 5, 6, 8 as one test path which are the edges that are not covered there these two edges.

And when you say fine I will write on other test path that cover those two edges, I could do 1, 2, 3, 5, 6, 7, 6, 8. So, my test case requirement of edge coverage is met through two test paths. That is also fine, anything is fine, but idea is the number of test paths we want to achieve for any kind of coverage criteria implicitly we also desired them to be as minimal possible. In this case, I could achieve it with just one test path, so I just wrote that, but there is nothing that says this is only option available we could do it with test paths that have two test paths, three test paths as long as you cover all the edges we are doing fine.

(Refer Slide Time: 32:52)
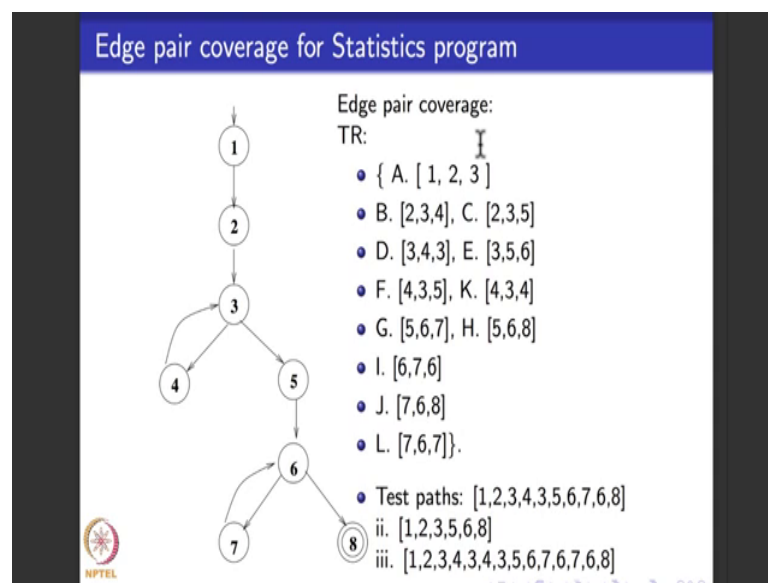
Now, let us say we want to do edge pair coverage for the same statistic program CFG. What is the test requirement for edge pair coverage, the TR for edge coverage is this set. I have written it such that I have given a label A, B, C, D, E F and so on for each test case. I will be using these labels in a few later lectures and I have also grouped them to

indicate the edge pair right like for example, 1, 2, 3 is this pair of edges (2, 3), I could do 4 or I could do 5. So, I have listed them together similarly from three I could do 3, 4, 3 or 3, 5, 6, so I have listed them together because both begin at 3. Similarly, the ones that begin at 4, I have listed them together, these two together and then the rest of them. So, this is my TR or test requirement for edge pair coverage which lists all paths of length two because it subsumes at coverage node coverage I am not listed those paths again and for this I will not be able to do it with just one test path.

So, this one test path is that long test path that we saw do before which is 1, 2, 3, 4, 5, 6, 7, 6, 8. So, this test path is needed. What does it miss out if you notice,which are the edge pairs that it misses out? Does it include for a example 5 6 8, it does not right because it does not come directly. If you see here, 5, 6, 7 comes 7, 6, 8 comes, 5, 6, 8 does not come and for now assume that I do not want to do side trips and detours. So, I have put another path that includes 5, 6, 8. So, I do 1, 2, 3, 5, 6, 8, so that is my second test path. And now we go back here and see what else as this missed. This has missed 4, 3, 4 also because if you see it is not there here it is defiantly not there here, because the paths does not if an come here. It is also missed 7, 6, 7 that is not in both these paths. So, I have another test path that includes both of them.

So, test paths always have to begin at 1 which is an initial node end at 8 which is a final node. So, it is fairly long path, because it has to go through the loop once again. So, I do 1, 2, 3, 4, 3, 4. That way I include this 4, 3, 4 and similarly for this one also I do 5, 6, 7, 6, 7, so that I can have 7, 6, 7 and then end it with 8. So, this is the test requirement or TR for edge pair coverage; and I need minimally three test paths to be able to achieve edge pair coverage for this graph.
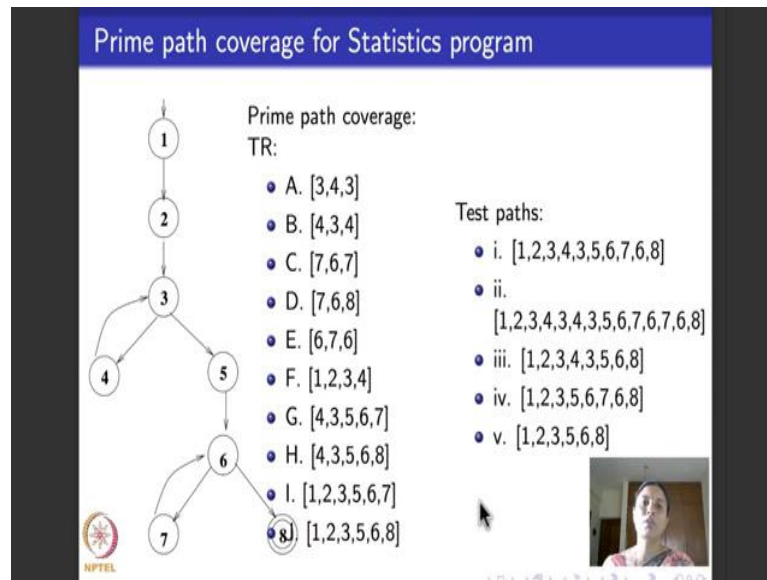
So, we look at prime paths coverage. You remember I told you what are the algorithms to enumerate the test requirements for prime paths, so I run that algorithm on this graph, and this is the set of prime paths that I get. If you remember and go back to the slides that we had done then, you would see that I would have taken same example graph and we would have worked out the prime paths as these. So, I am just taking it. It so happens that this is the graph that comes the CFG for the statistics program, but these are the set of prime paths, so that is my TR and for meeting this TR, I need five different test paths right. So, this is how I do prime pair coverage. If you see the example, these five different test paths they will neatly execute the loop there are two loops in the graph, one for loop here and one for loop here, between these they will neatly skip the loop and execute the loop each in turn for both for loops. So, I have achieve loop coverage simplicity prime path coverage for this program.

So, what was what it we do in this module? We looked at how to draw control flow graphs corresponding to source code, and how to apply elementarily structural graph coverage criteria to test the control flow graph in turn the source code by using this criteria. In the next lecture, we will continue with source code. I will tell you what are the classical notions of testing related to source code, and how to they relate to the structural coverage criteria that we have seen so far.

Thank you.