

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

Lecture - 17
Design Integration Testing and Graph Coverage

Hello again. This is the third lecture of fourth week. We are still looking at graph coverage criteria. What we did in the last lecture is we finally, moved out of looking at code and testing of code using graphs.

We started with design in the last lecture; I gave you the basics of what is the kind of testing which is done with design. The kind of testing that is done with design is called integration testing. We saw the classical view of integration testing, the various approaches to integration testing, what is tested and what is not. What we will do today is we will see how graphs can be used to do the kind of integration testing that we saw earlier. Specifically, we will see what are the coverage criteria that we have already seen on graphs that can be used for integration testing and what new coverage criteria could be defined over graphs when we look at integration testing.

(Refer Slide Time: 01:04)

Goals

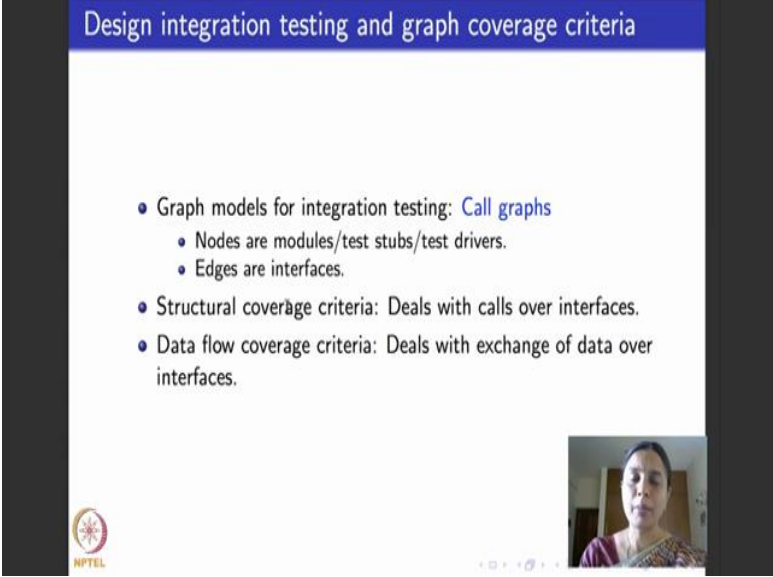
- Applying graph coverage criteria (structural and data flow) to design elements.
- Phase of testing where this will apply— **Integration testing**.

NPTEL

So, what are the goals for today's lecture? We want to be able to apply the graph coverage criteria that we have already learnt which is structural coverage criteria and

data flow coverage criteria, to be able to do design testing that is to be able to do integration testing that involves software design.

(Refer Slide Time: 01:23)



The slide is titled "Design integration testing and graph coverage criteria". It contains the following bullet points:

- Graph models for integration testing: **Call graphs**
 - Nodes are modules/test stubs/test drivers.
 - Edges are interfaces.
- Structural coverage criteria: Deals with calls over interfaces.
- Data flow coverage criteria: Deals with exchange of data over interfaces.

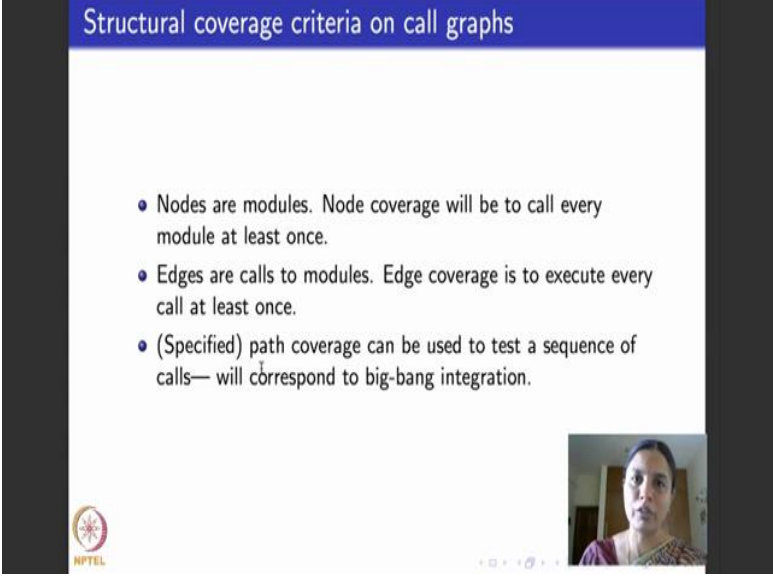
In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is visible in the bottom left corner of the slide.

And while doing so, what will be the graph models that we will consider? The most popular graph models that we will consider are what are called call graphs as we saw last time. What is the call graph well there are several modules or components the software is broken up into these modules? And modules talk to each other through interfaces and when modules call each other or talk to each other through interfaces, the graph that models this is what is called call graph. The vertices or nodes of these call graphs are the various modules. Sometimes when I am doing bottom up or top down testing we saw that some modules could be replaced with test stubs or test drivers. So, the vertices of this call graph could be either the modules or the tests stubs or the test drivers.

The edges are basically the interfaces, the various calls. What do structural graph coverage criteria over such call graphs deals with it? So, happens that structural coverage criteria basically deal with calls over interfaces. You are basically checking if one module is calling every other module that it is supposed to. And the called module is it working fine right and so on. And then what you data flow coverage criteria over such call graphs deal with? They deal with how data is exchanged between the interfaces that connects these 2 modules.

Suppose it is a normal call interface where one module is calling another module. So, the data is passed as parameters the called module does its work and returns some data right. So, data flow coverage criteria deals with this passing and return of data across the interfaces.

(Refer Slide Time: 03:09)



The slide is titled "Structural coverage criteria on call graphs" in a blue header. It contains three bullet points: "Nodes are modules. Node coverage will be to call every module at least once.", "Edges are calls to modules. Edge coverage is to execute every call at least once.", and "(Specified) path coverage can be used to test a sequence of calls— will correspond to big-bang integration." The NPTEL logo is in the bottom left corner, and a small video feed of a person is in the bottom right corner.

- Nodes are modules. Node coverage will be to call every module at least once.
- Edges are calls to modules. Edge coverage is to execute every call at least once.
- (Specified) path coverage can be used to test a sequence of calls— will correspond to big-bang integration.

So, we will first look at structural coverage criteria then we will move on to data flow coverage criteria over call graphs. We are not looking at graphs that correspond to control flow right now. We are only looking at graphs that correspond to call graphs. Sometimes while modelling call graphs you might also have to model parts of the CFG, we will see through examples; how that is done.

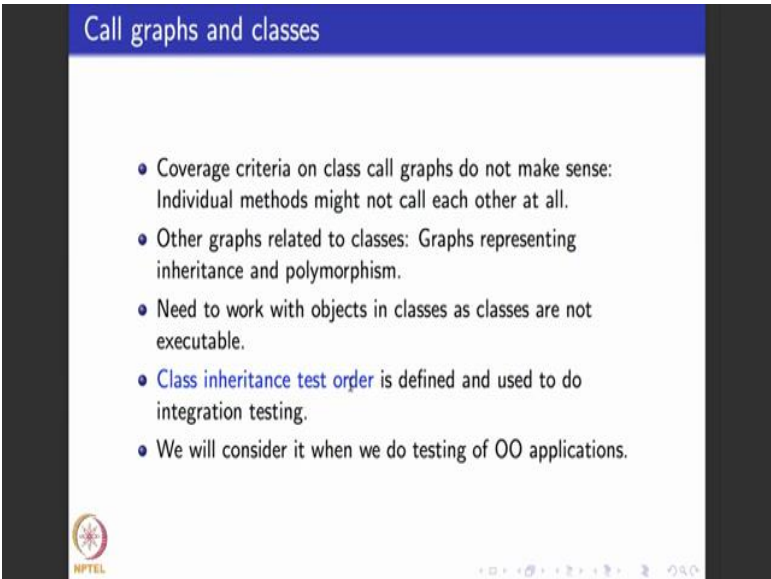
So, now what do structural coverage criteria over call graphs deal with? If you remember what are the various structural coverage criteria that we dealt with. We dealt with node coverage edge, edge pair coverage then complete path coverage which was infeasible then we looked at specified path coverage, prime path coverage. So, what is node coverage deal with here? What are nodes in our kinds of graphs when we do design integration testing? Nodes are basically software components or modules. So, node coverage would mean call every module that is supposed to be around right. So, suppose I have a procedure, that is a standalone procedure that is not being called by any other procedure or method, then it might as well not be there right. So, node coverage will

make sure that every component that is there in the modularized software is indeed useful and being called by some component.

What is edge coverage deal with, what are edges here ? As we saw edges for call graphs are interfaces. So, edge coverage deals with executing every call at least once. So, make sure that each call is executed at least one. So, it basically tests for coverage to make sure that there are no unnecessary calls unneeded, if there are calls that are never executed once then edge coverage will not be met and maybe it means that those modules can be removed, they are not needed at all, right.

So, edge coverage is a useful coverage criteria. Then it so turns out that other edge pair prime path coverage may not be very useful because if you remember the sole purpose of prime path coverage was to be able to handle loops in graphs. In call graphs we do not have loops and other things that we worry about. So, we do not consider prime paths coverage, but if you remember the various kinds of testing that we looked at--- top down, bottom up, big bang and so on in the last lecture. So, when I am doing big bang integration that is when I am putting together all the modules that I have developed so far and testing the integrated way of working together, then I can use what is called specified path coverage as structural coverage criteria to be able to do big bang coverage. So, this is one kind of use of structural coverage criteria.

(Refer Slide Time: 05:55)



Call graphs and classes

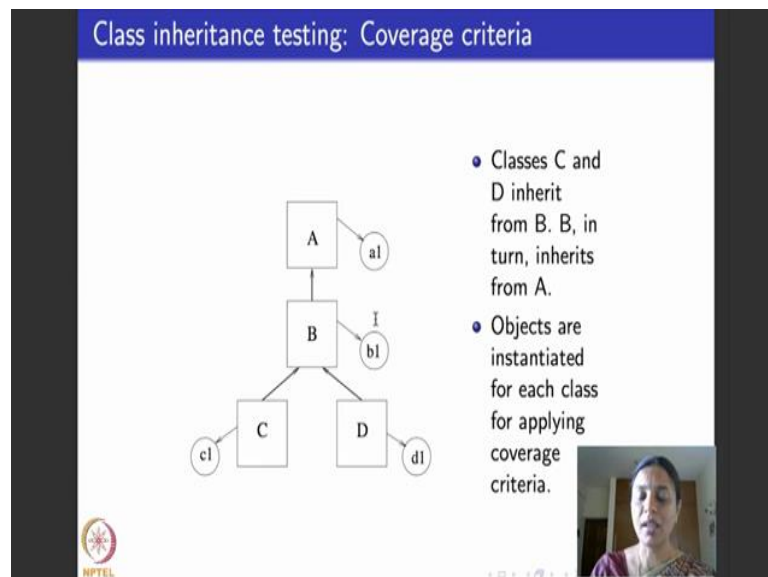
- Coverage criteria on class call graphs do not make sense: Individual methods might not call each other at all.
- Other graphs related to classes: Graphs representing inheritance and polymorphism.
- Need to work with objects in classes as classes are not executable.
- Class inheritance test order is defined and used to do integration testing.
- We will consider it when we do testing of OO applications.

NPTEL

What other kinds of call graphs could be there in software. In object oriented software there are what are called class call graphs. And now we can say do we look at coverage criteria over class call graphs? But I would say that coverage criteria over class call graphs does not make sense at all. Why? Because individual methods residing within the class may not call each other at all. Classes are not executable objects. So, what could be other kinds of graphs that are related to classes? Typical other kinds of graphs that come related to classes are graphs that deals with class inheritance and iso and polymorphism and so on.

So, what we will do later in subsequent weeks after we look at all the various coverage criteria? We will spend one week where we will exclusively look at testing object oriented applications. In that week we will revisit this class call graphs and I will tell you lot in detail about how to test the large piece of object oriented software for functionalities like inheritance, polymorphism and so on. So, there is this theory called class inheritance test order that people define to be able to do integration testing over classes, but we will not look at it today. We will look at it after a few weeks then we dedicatedly look at object oriented testing concepts.

(Refer Slide Time: 07:22)

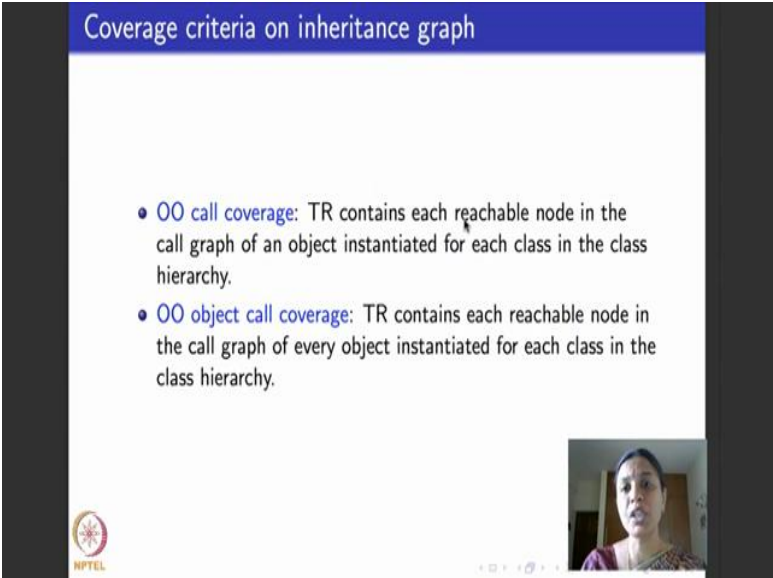


Right moving on when I do class inheritance testing in terms of coverage criteria, I could do the following. Consider this picture, if you look at this picture there are 4 classes here A, B, C and D. And read this connection arrow as inheritance. So, classes C and D

inherit from B, B in turn inherits from class A. As I told you classes are not executable. So, I really cannot do in terms of testing anything for this inheritance. Instead what I do is I consider objects that are instantiated for each of these classes before applying coverage criteria. So, in this picture I have depicted an object a1 that is instantiated for class capital A object b1 that is instantiated for class capital B object d1 for class D object c1 for class C.

So, this if this is the picture depicting an inheritance a hierarchy over class graph, how do I apply structural coverage criteria over such an inheritance hierarchy? So, it does not make sense to consider plane classes and apply inheritance hierarchy, I look at objects instantiated for each class and then apply inheritance hierarchy based testing.

(Refer Slide Time: 08:44)



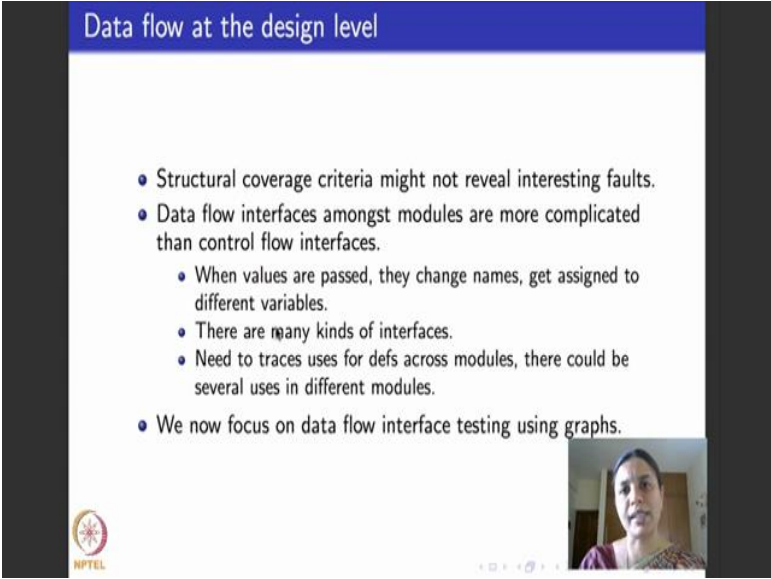
The slide is titled "Coverage criteria on inheritance graph" in a blue header. It contains two bullet points:

- **OO call coverage:** TR contains each reachable node in the call graph of an object instantiated for each class in the class hierarchy.
- **OO object call coverage:** TR contains each reachable node in the call graph of every object instantiated for each class in the class hierarchy.

In the bottom right corner, there is a small video inset showing a woman speaking. In the bottom left corner, there is a logo for NPTEL.

So, when I do that 2 kinds of coverage criteria makes sense the first kind of coverage criteria is what is called object oriented call coverage. So, we will see what it says, it says the test requirement or TR for OO call coverage contains each reachable node in the call graph of an object instantiated for each class in the class hierarchy.

(Refer Slide Time: 09:20)



Data flow at the design level

- Structural coverage criteria might not reveal interesting faults.
- Data flow interfaces amongst modules are more complicated than control flow interfaces.
 - When values are passed, they change names, get assigned to different variables.
 - There are many kinds of interfaces.
 - Need to trace uses for defs across modules, there could be several uses in different modules.
- We now focus on data flow interface testing using graphs.

NPTEL

So, I go back, I will look at this class hierarchy and then what do I say I say my test requirement is each reachable node in the call graph of an object instantiated for each class in the class hierarchy right. So, each reachable node should be covered, and the second one is object oriented of OO object call coverage that says the TR contains each reachable node in the call graph for every object instantiated for each class in the class hierarchy. So, every, so they here this picture depicts only one object instantiated per class, but that need not be the case there could be several of them. If that is the case then I do what is called OO object call coverage.

So, basically in summary what we are saying is that I look at class inheritance graph that looks like this do not consider classes plain because they are not executable. Instead you look at objects that are instantiated for each class. When I test the calls relate or when I test the calls related to objects in the call graph, one object per class, then it is called OO call coverage. When I test the calls related to objects instantiated in the class graph for each object in the class then it is called object call coverage. We will revisit as I told you all these concepts later in a week when we dedicatedly do object oriented testing applications.

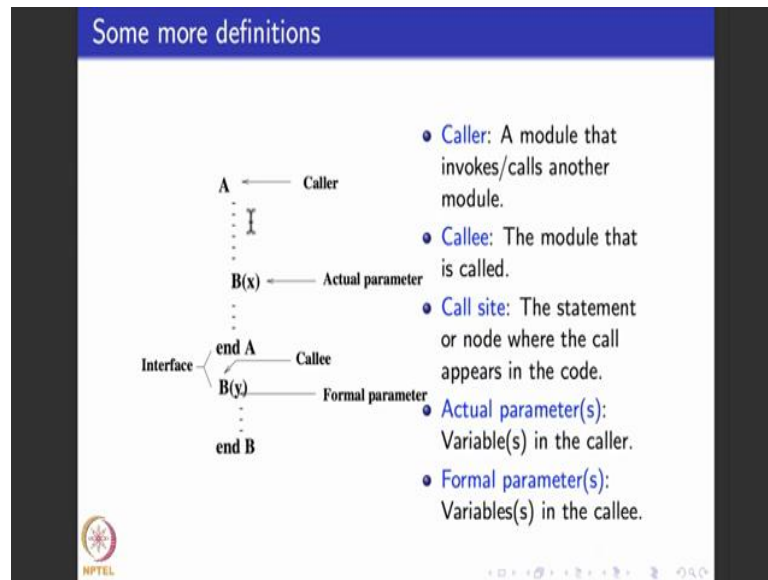
Now, they done with structural coverage criteria. I would like to move on to data flow coverage criteria over design. As I told you right data flow coverage criteria is very interesting when I look at inheritance testing because it actually deals with the

parameters that are passed and returned. Maybe that is a richer source of error instead of just checking for structural coverage criteria, we just check is a module called, is a module interface called and so on. It just checks whether it is called or not of course, that is also useful, but what is more useful is actually reasoning and testing for the values that are called and returned in each of these calls and returns. That is what data flow coverage criteria deals with.

So, data flow interfaces are more complicated than control flow interfaces, why? Because values are passed they; obviously, change names we will see it through examples I might pass a certain value in the variable name x, it gets passed to the callee as a variable let us say y. So, they change names they get assigned to different variables and as I, as we discussed in the last module there are several different kinds of interfaces, I could communicate by explicitly passing parameters and returning values. I could communicate by reading from and writing on to a global set of variables, I could communicate by sending and receiving messages across dedicated buffers. So, data flow testing has to be done across all these kinds of interfaces. And the other important thing is remember now definitions and uses or variables as we deal with in data flow criteria will have to run across modules. So, a particular variable could be defined in one module and used later in another module.

So, we will see through examples what each of these mean. So, the focus for the rest of today's lecture, this lecture would be to look at data flow coverage criteria, but specifically looking at data flow over call graphs as we use it first.

(Refer Slide Time: 12:26)



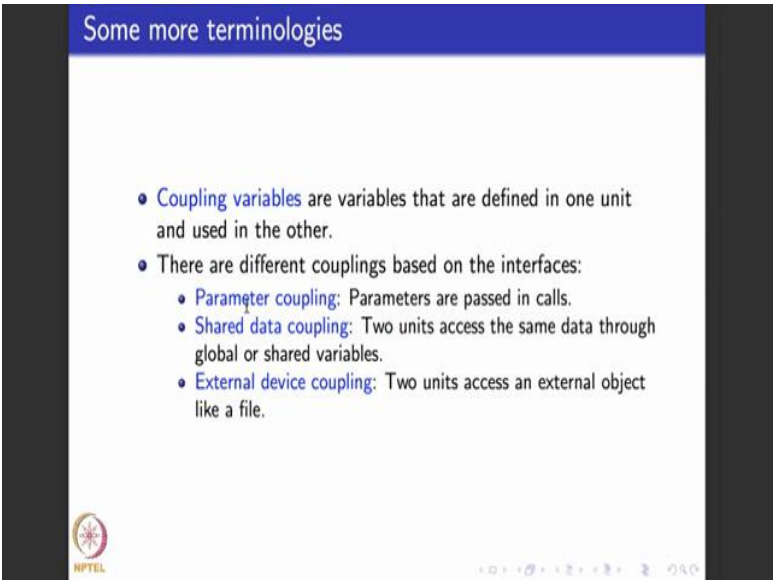
So, here is an example. Let us say if you look at the picture on the left hand side let us say there is a method A, a method or procedure A. A has some codes I am not really written what the code is read these dot, dot dots (...) that you see here is some code that corresponds to A as a part of the code that is written for A, A also calls another method or a procedure B with a parameter x.

When A executes and when this line of A comes into picture what happens, control gets transferred to the method or a procedure B, which is right here down there. And remember A calls B with actual parameter x and it is passed to B as y, which is the formal parameter. Strictly speaking x and y are the same value, but they take different variable names. I need to be able to remember the fact that x and y are the same values, but they take different variable names; x takes x in A and the same x is called y in B.

So, A runs, at some point in time A calls B with parameter x then x is passed to be as y then B executes. Read these dot, dot, dot (...) as some code that corresponds to B. I have not really written the code for A and B because the focus now is to test for the interfaces test for A calling B and B returning something. So, B runs something B finishes end B comes when B finishes the control is passed back to A and the rest of A executes and finishes. So, here are some terminologies that we will use for the rest of the couple of lectures.

So, A is called the caller or a module that calls another module in this case it is the module that it calls is B. B is called the callee module which is the module that is being called by A the call site is this statement the statement in a where the procedure call happens. Actual parameter is x, this x that is passed as a parameter to be, formal parameter is y the parameter that callee B takes from the caller A. Interface is the place where control is transferred from A to B and after B finishes running, control is transferred back to A. So, what are the terminologies that we looked at till now, this caller, this callee then there is call site which is the place where the call happens or the interface is connected. Then there are actual parameters then there are formal parameters.

(Refer Slide Time: 15:02)



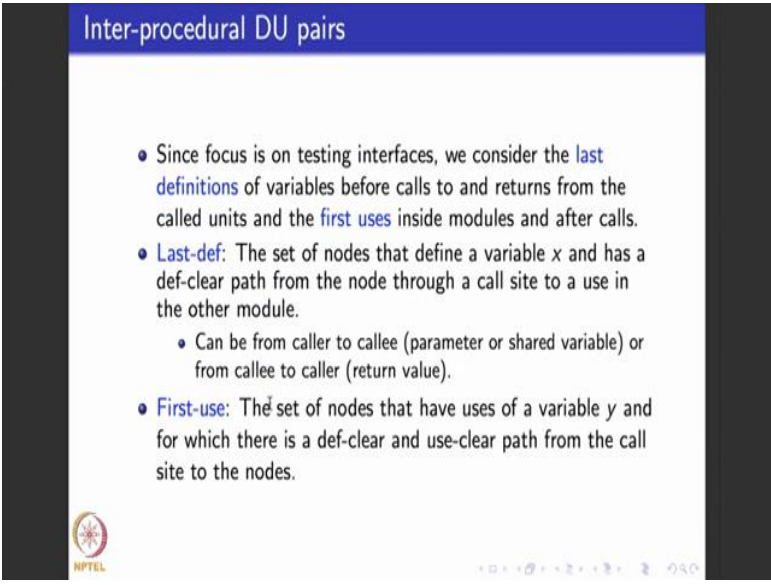
The slide is titled "Some more terminologies" in a blue header. It contains a bulleted list of definitions for coupling variables. The first bullet point defines coupling variables. The second bullet point states that there are different couplings based on interfaces, followed by three sub-bullets: parameter coupling, shared data coupling, and external device coupling. The NPTEL logo is in the bottom left corner, and navigation icons are in the bottom right corner.

- **Coupling variables** are variables that are defined in one unit and used in the other.
- There are different couplings based on the interfaces:
 - **Parameter coupling:** Parameters are passed in calls.
 - **Shared data coupling:** Two units access the same data through global or shared variables.
 - **External device coupling:** Two units access an external object like a file.

So, now if one more terminology before we move on if you go back to this figure, look at the actual parameter x. It is passed as x and B. What B take x as, B takes x as y. So, we say x and y are what are called coupling variables. So, coupling variables are variables that are defined in one unit and used in the other. And as we know based on the kind of interfaces that we are looking at there could be several different kinds coupling. There could be parameter coupling, example of parameter coupling is what we saw here. A calling B with x as a parameter and B is returning it back to a with some at something else. Coupling could be in terms of share data when modules communicate by reading from and writing to a set of global variables or shared variables or coupling could be in terms of an external device.

You could have a database that resides in a particular database server and then you say I read data from that database server and that could also deal with explicit coupling variables. But when we look at this particular module, I will assume that parameter coupling is one coupling that we are looking at and the abstract def use pairs that we will look at will deal mainly with parameter coupling and in other cases also it should go through at a same level of abstraction.

(Refer Slide Time: 16:23)



The slide is titled "Inter-procedural DU pairs" in a blue header. It contains three bullet points. The first bullet point states that since the focus is on testing interfaces, we consider the last definitions of variables before calls to and returns from the called units and the first uses inside modules and after calls. The second bullet point, "Last-def", defines it as the set of nodes that define a variable x and has a def-clear path from the node through a call site to a use in the other module, with a sub-bullet noting it can be from caller to callee (parameter or shared variable) or from callee to caller (return value). The third bullet point, "First-use", defines it as the set of nodes that have uses of a variable y and for which there is a def-clear and use-clear path from the call site to the nodes. The NPTEL logo is in the bottom left corner, and navigation icons are in the bottom right.

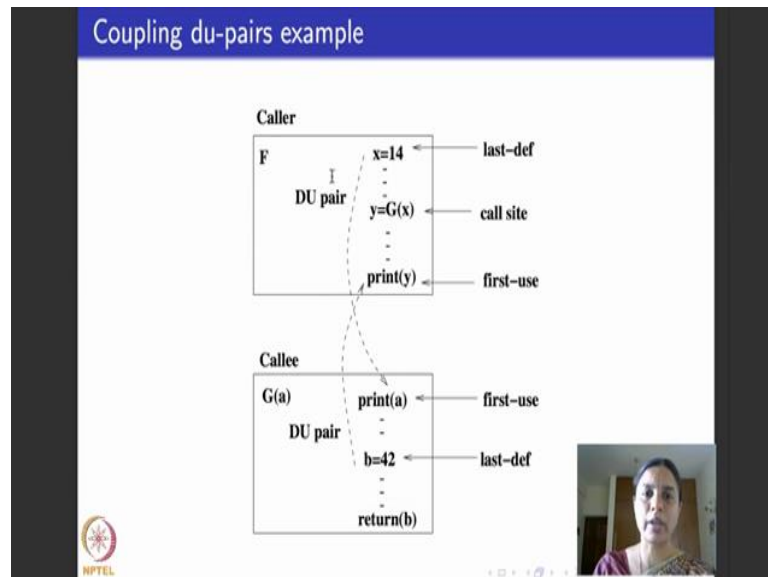
- Since focus is on testing interfaces, we consider the **last definitions** of variables before calls to and returns from the called units and the **first uses** inside modules and after calls.
- **Last-def:** The set of nodes that define a variable x and has a def-clear path from the node through a call site to a use in the other module.
 - Can be from caller to callee (parameter or shared variable) or from callee to caller (return value).
- **First-use:** The set of nodes that have uses of a variable y and for which there is a def-clear and use-clear path from the call site to the nodes.

Some more definitions before we move on and look at an example. Remember what is the focus, the focus is to be able to understand and test how data is passed across interfaces right. So, the focus is on interfaces. So, if I go here, if I consider data flow testing for A and data flow testing for B, there could be a whole set of variables that reside any A, whole set of other variables reside in B. Which I will do when I do normal data flow testing for source code, I will test it for that, but the goal for today is to be able to test how A calls B and how return B returns A.

So, what I do is I focus on the parameters that are passed and returned and the values that are defined and used around the parameters that are passed and returned. So, that gives us the definition of what are, what is called the last definition of variables which occur just before calls to the another module, and what is called first uses of variables which occur soon after the module is called, soon after the module is called inside the called module.

So, what is last def? Last definition is a set of nodes in the control flow graph of the caller module that define variable x and it has a def clear path from the node through the call site to a use in another module.

(Refer Slide Time: 17:53)



So, to better understand this let us look at an example. So, here is a caller, method or procedure F calls another method or procedure G . So, G is the callee, let us say F has some code that goes like this again, please I have read these dot dot dots (...) as some code that is there, but I am interested I have put the statements corresponding to only fragments of the code that I want to focus on. So, at some point in it is working F has the statement, where it says x is 14. And after that it is def clear for x , x is not defined in these places where I am pointing to, in these dot dots after x is equal to 14. And then F calls G of x with this value of x and then control gets passed to G .

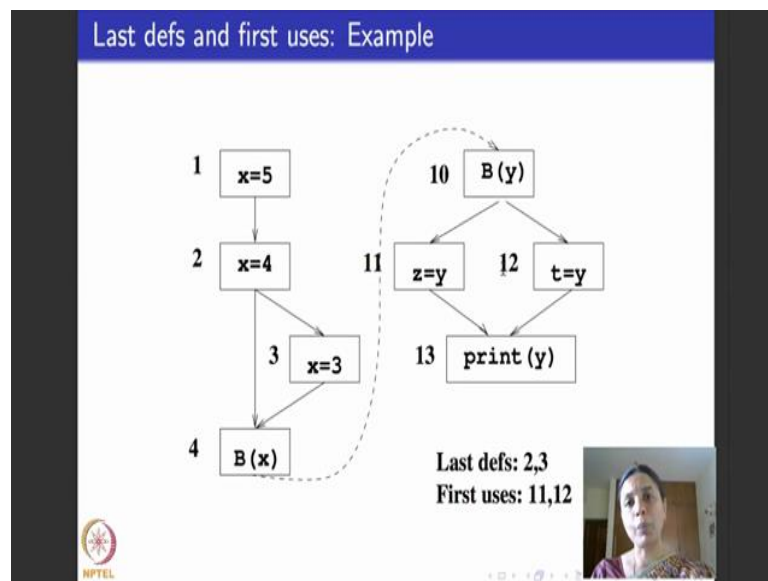
What is G do? G takes x as a , it runs as $G(a)$. So, this x actual parameter is passed as a just a formal parameter to G the first thing that G does is to print the value that it obtains. So, the last def of x was here and the first use of x passed as a is in this printf statement. Is that clear? Please note that I do not consider the parameter passing itself as a use. So, you could argue that this could be thought of as the last use of x in F and this could be thought of as the first use of x as a , coupled with a in G . But typically when we do integration testing the point of parameter passing at the call site is not considered a use.

What we consider is after it is passed by the called method or procedure, inside the called method or procedure where it is first used.

So, this print a of x being passed as A is considered the first use of the variable x that is coupled with a and now G after prints a it executes a few more statements. At some point it defines b as something like this b is equal to 42 then it executes something this bit after the definition of b is def clear for b and then it returns b. When it returns b, what is b passed on as it is passed on here into the variable y because F call G of x and keeps it as y after some point F decides to print y right. So, the last def of b that was here gets first use as print y when b is coupled with y here again I do not consider this return statement as the first use of b. So, going back to the definition what is last def say? Last def says that a variable x has a def clear path from the node through the call site to a use in another module.

Now, this can be of two different kinds. It can be from a caller to a callee or it can be from a callee back to the caller. We saw examples of both these cases right. So, here if you see, this is from the caller to the callee, this last def x is equal to 14 is from the caller to the callee. This last def for b being 42 is from the callee back to the caller. So, two kinds of last defs can be there. Similarly, what is first use? First use, the set of nodes that have uses of variable y, for which there is a def clear and a use clear path from the call site to the nodes. So, there is a def clear and a use clear. So, if you see here I have marked the first uses also, because once this last def happens this is the place where it is first used and in between we assume that there are no further definitions or use. One thing to note is that this calling, and return at the call site does not constitute a part of the definition of last def or first use.

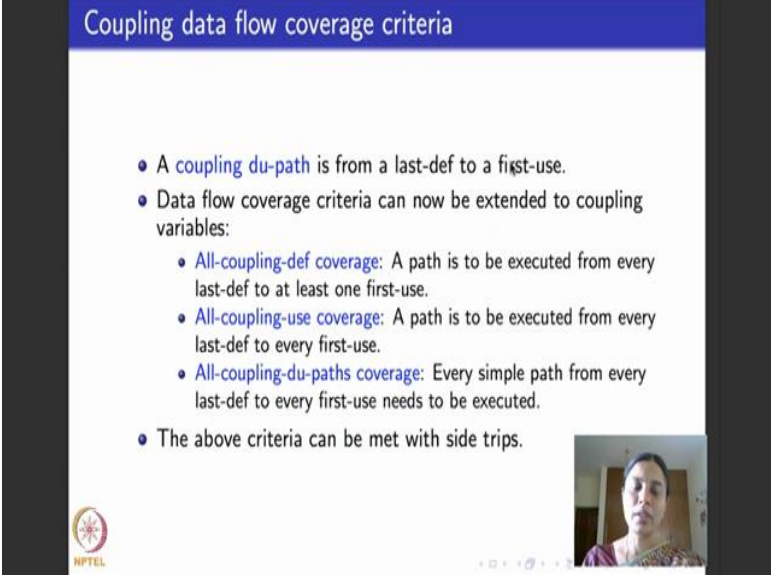
(Refer Slide Time: 22:11)



So, just a few more examples before we move on. So, here is another piece of code. I have depicted it differently this time. So, what we have shown is not as boxes like this containing the caller and the callee modules. What I have shown is I have shown it directly along with the control flow graph of the caller module and the callee module. So, the one that you see on the left hand side here; x is equal the which and vertices labelled with 1, 2, 3 and 4 this is part of the caller module. B is the caller module right and in this module is a caller module the callee module is B, the variables that are coupled are x and y B runs, maybe there is an F statement here one of these happens and then it prints y.

So, what are the last definitions? Last definitions for x could be either at node 4, I mean node 2 or node 3 based on what the decision of this statement is. And what are the first uses, not at the call site, please remember that, it is either at 11 or 12 based on what this condition is. So, is this clear?

(Refer Slide Time: 23:25)



The slide is titled "Coupling data flow coverage criteria" in a blue header. It contains a list of bullet points:

- A coupling du-path is from a last-def to a first-use.
- Data flow coverage criteria can now be extended to coupling variables:
 - All-coupling-def coverage: A path is to be executed from every last-def to at least one first-use.
 - All-coupling-use coverage: A path is to be executed from every last-def to every first-use.
 - All-coupling-du-paths coverage: Every simple path from every last-def to every first-use needs to be executed.
- The above criteria can be met with side trips.

In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is visible in the bottom left corner of the slide.

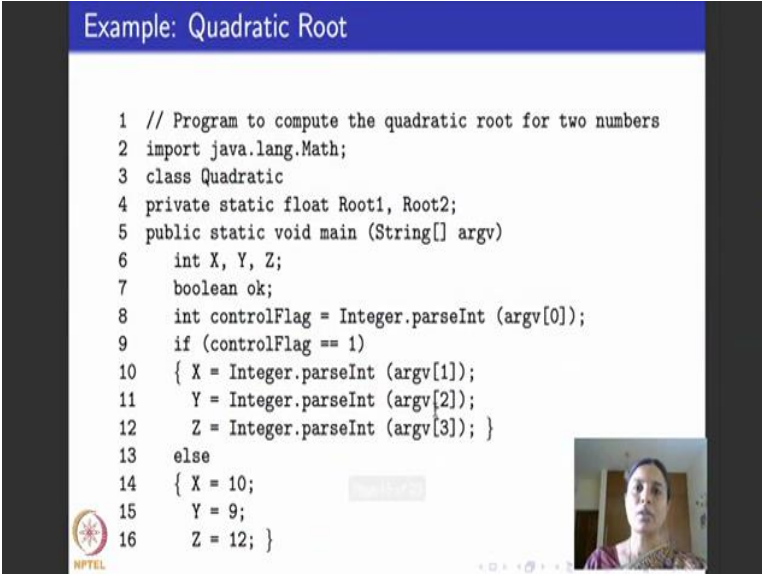
So, moving on, we had three data flow coverage criteria that we define for call graphs. What are the three data flow coverage criteria? If you remember there were all defs coverage, cover every definition to at least one use then it was all uses coverage, cover every definition to every use. And then the third was all du-paths coverage, from every definition take every kind of paths to every use.

So, I retain the same definitions instead I look at coupling variables. So, what is a coupling du-path, a coupling du-path is a path from not any F def to any use it is from the last def to a first use. So, I am in a situation where my code is fragmented into 2 modules. One module calls another module. So, in the caller module there is a place where last definition of a variable happens. And in the callee module there is a place where the first definition of a variable happens. Then this pair is what is called a coupling pair. Here x and y is a coupling pair. So, what I do is, I take coupling du-path as a path from last def to first use once I have coupling du-paths, I take the same definitions of data flow coverage criteria that we looked at earlier, and consider the same definition, but instead of considering it overall du-paths I consider it from last def to first use.

So, all coupling def coverage is a path to be executed from every last def to at least one first use. All coupling use coverage is a path to be executed from every last def to every first use. All coupling du-paths coverage says that execute every simple path from every last def to every first use. As always you do best effort touring. So, whenever I have to

meet this data flow coverage criteria, if I have to do a side trip or a detour I can do that to be able to make these test requirements satisfiable. Of course, the one requirement that I have about side trips, to repeat, is that side trips have to be definition clear right.

(Refer Slide Time: 25:37)



Example: Quadratic Root


```
1 // Program to compute the quadratic root for two numbers
2 import java.lang.Math;
3 class Quadratic
4 private static float Root1, Root2;
5 public static void main (String[] argv)
6     int X, Y, Z;
7     boolean ok;
8     int controlFlag = Integer.parseInt (argv[0]);
9     if (controlFlag == 1)
10    { X = Integer.parseInt (argv[1]);
11      Y = Integer.parseInt (argv[2]);
12      Z = Integer.parseInt (argv[3]); }
13    else
14    { X = 10;
15      Y = 9;
16      Z = 12; }
```

So, what we will end today's module with is, I will explain to you through one full code example how data flow coverage criteria works. So, here is the program that we will look at. So, this is a java program to compute the quadratic root of 2 numbers right. So, I assume that you have an equation of the form $Ax^2 + Bx + C$ right. And I want to be able to compute 2 values for x , the standard formula says that x is given as this formula right root 1 and root 2.

(Refer Slide Time: 26:14)

Example: Quadratic Root, contd.

```
23 // Three positive integers, finds quadratic root
24 private static boolean Root (int A, int B, int C)
25 {
26     double D;
27     boolean Result;
28     D = (double)(B*B)-(double)(4.0*A*C);
29     if (D < 0.0)
30     {
31         Result = false;
32         return (Result);
33     }
34     Root1 = (double) ((-B + Math.sqrt(D))/(2.0*A));
35     Root2 = (double) ((-B - Math.sqrt(D))/(2.0*A));
36     Result = true;
37     return (Result);
38 } // End method Root
39 } // End class Quadratic
```





So, how does the code do? The code has a class called quadratic there are 2 variables which are floating point numbers root1 and root2. What are x, y and z ? These are the coefficients of the polynomial, let us say $Ax^2 + Bx + C$, A, B and C are the coefficients. So, I pass them as x, y and z and then I have this Boolean entity called ok, which basically tells me whether this polynomial has a solution or not. And then what I do is I take these parameters. Suppose I do not pass them as normal integer coefficients then you just assign some default values in this case we have done 10, 9 and 12.

(Refer Slide Time: 26:58)

Example: Quadratic Root, contd.

```
17 ok = Root (X, Y, Z);
18 if (ok)
19     System.out.println("Quadratic roots:" + Root1, + Root2);
20 else
21     System.out.println ("No Solution");
22 }
```



And then you call this function method called root with these as parameters x, y and z.

So, root is the main method of this class quadratic that computes the quadratic root. So, root returns this value ok which says basically whether the quadratic roots have been computed or not and if they have been computed it is stored in the variables root 1 and root 2. So, if ok turns out to be true the values of root 1 and root 2 are printed. If ok turns out to be false the value is what is printed is to say that this equation has no solution. So, what is the focus here remember the focus here is to be able to look at interfaces. So, where is the interface the interface is at this statement, this is the call site where this main quadratic class calls a method root with 3 formal parameters x, y and z which are the coefficients of the quadratic equation that I am considering.

So, now we look at the code for root. What happens? It takes as input 3 positive integers, which are the coefficients and it finds the quadratic root. So, these are the three. So, x, y and z which are formal parameters here are passed as actual parameters A, B and C. So, what is coupled with what x is coupled with A, y is coupled with B and z is coupled with C right. And then it uses the standard formula that it computes for computing the quadratic equation. So, here what it computes is an intermediate value which is B squared minus 4 A C. You can look up for how to compute how to solve quadratic roots is a standard formula to solve an equation of the form $Ax^2 + Bx + C = 0$. So, one, if I compute B squared minus 4 A C then one solution is given as $\frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$, that is what is this.

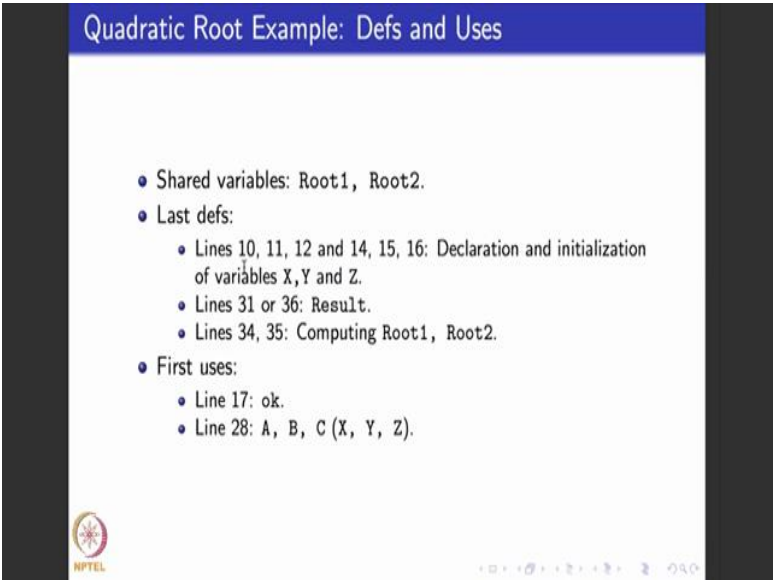
The next solution is given as $\frac{-B - \sqrt{B^2 - 4AC}}{2A}$. D computes B squared minus 4 A C. If D is 0 then there is no solution to the quadratic equation if D is less than 0, if D is greater than 0 then I can compute the 2 roots. So, what we do is we first compute B squared minus 4 A C, and check whether it is less than 0. If it is less than 0, then this procedure this method root stores the result is false and returns the result otherwise it computes this two square roots, stores it as root 1 and root 2 sets result to be true and returns the result.

So, how is the return passed here? Return is passed and assigned to this Boolean variable ok. If you remember ok was a Boolean variable. So, return is passed and assigned to this Boolean variable A. So, what if is true, if ok is true means what that I have computed root 1 and root 2 store it in these 2 variables as that is, how I return the result is true

across this statement, if ok was false then I have returned false which means I have not been able to compute this square root. So, the system correctly prints.

So, what is the goal now? The goal is there is this main class which calls is the method root. Root computes the, root basically checks if the given system of equations has a solution or not based on the value of D, and if it does not have a solution it says returns false. If it has a solution it outputs the two roots right. I want to be able to apply the data flow coverage criteria that I learnt now in this lecture to be able to test this. So, what are the things that we have to first do? We have to first identify the last defs and first uses.

(Refer Slide Time: 30:48)



Quadratic Root Example: Defs and Uses

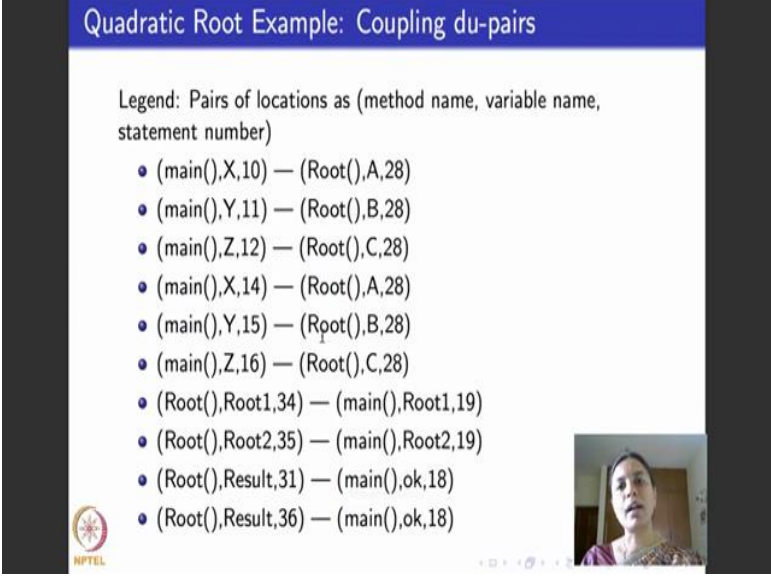
- Shared variables: Root1, Root2.
- Last defs:
 - Lines 10, 11, 12 and 14, 15, 16: Declaration and initialization of variables X, Y and Z.
 - Lines 31 or 36: Result.
 - Lines 34, 35: Computing Root1, Root2.
- First uses:
 - Line 17: ok.
 - Line 28: A, B, C (X, Y, Z).

So, which are the shared variables? If you see root 1 and root 2 are the 2 shared variables. Which is the actual parameter that is passed and returned? X, y, z are the parameters that are passed, result is the parameter that is returned. So, which are the last defs? Last defs can occur at lines 10, 11, 12 or at lines 14, 15 and 16 which basically declare and initialize the 3 variables x, y and z. So, we will go back to the code. If you see lines 10, 11 and 12 take x, y and z as input. If it is some unacceptable value that is given as input it sets its own values for x, y and z as some fixed constants in line 14, 15 and 16.

After this it is directly passed to this called method root. So, the last defs for x, y and z are at lines 10, 11, 12 or at lines 14, 15 and 16. And when they are passed x, y and z is coupled with A, B and C respectively right. And then what else is passed, result. If you

see result is computed as false at line 31 or is true at line 36. So, it is passed here and then lines 34 and 35, if result happens to be true compute root 1 and root 2, lines 34 and 35 computes square root 1 and square root 2. Which are the first uses? Oh, result is returned as in line 17 right and x, y, z are passed A, B, C right we saw this.

(Refer Slide Time: 32:33)



Quadratic Root Example: Coupling du-pairs

Legend: Pairs of locations as (method name, variable name, statement number)

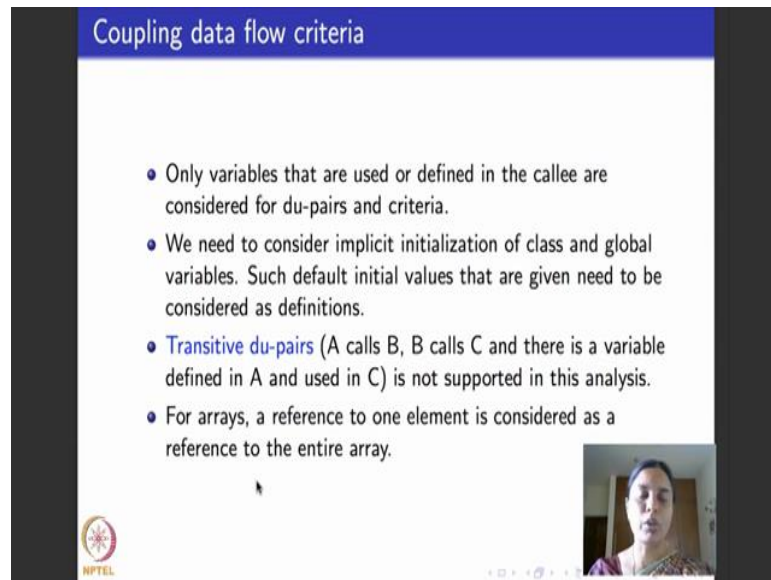
- (main(),X,10) — (Root(),A,28)
- (main(),Y,11) — (Root(),B,28)
- (main(),Z,12) — (Root(),C,28)
- (main(),X,14) — (Root(),A,28)
- (main(),Y,15) — (Root(),B,28)
- (main(),Z,16) — (Root(),C,28)
- (Root(),Root1,34) — (main(),Root1,19)
- (Root(),Root2,35) — (main(),Root2,19)
- (Root(),Result,31) — (main(),ok,18)
- (Root(),Result,36) — (main(),ok,18)

Now, what are the coupling D u pairs, as I told you X is coupled with A, Y is coupled with B, the Z is coupled with D. Main method calls the root at passes X, Y and Z, takes X, Y and Z as input in lines 10, 11, 12 or at lines 14, 15, 16. Wherever they are taken as input they are passed as A, B and C respectively at line 28. So, read this as, which is the method that is using it, which is the actual variable name, what is the statement number? That is what I given--- method, variable name and statement number. Method main calls method root at line number 10, by passing formal parameter as x, and actual parameter as where a is used at line number 28. And so on for the other things also. Method root method root returns back the value of root 1 to main which is used by main at line number 19. This must be the print statement yeah, this must be the print statement of the method main.

Similarly, for result which is coupled with the variable ok in the main variable right. Once I do this then I can normally write my data flow coverage criteria--- all defs, all coupling defs, all coupling uses, all coupling du-pairs. I have not written the TR and the test cases for data flow coverage criteria in this case because this particular example code

does not have any errors, but you can go ahead and do it as a small exercise for yourself. Write down the test requirement and give a test path that will satisfy the test requirement for coupling du-pairs coverage.

(Refer Slide Time: 34:21)



The slide is titled "Coupling data flow criteria" in a blue header. It contains a list of four bullet points:

- Only variables that are used or defined in the callee are considered for du-pairs and criteria.
- We need to consider implicit initialization of class and global variables. Such default initial values that are given need to be considered as definitions.
- **Transitive du-pairs** (A calls B, B calls C and there is a variable defined in A and used in C) is not supported in this analysis.
- For arrays, a reference to one element is considered as a reference to the entire array.

In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is visible in the bottom left corner of the slide.

Now, I hope this example would have clarified some details about how to use integration testing across parameter passing and shared variable interfaces to be able to test for data flow coverage criteria. Just a few other additional points that I would like to mention about data flow coverage criteria. One is when we look at data flow coverage criteria, even in the examples that I showed you, if you see I sort of did not give you the pieces of code that really did not matter to us. If you go back to this slide the rest of the code, I put it out as dot dot dot (...).

So why, because our focus is mainly on the values of the variables that are used or defined in the callee. We really do not look at other du-pairs when we are focusing on interface integration testing. The other thing we need to consider which this example didn't illustrate is that in languages like Java and C, there could be an implicit initialization of class or global variables. These implicit initializations will also have to be considered as definitions even if they do not come as explicit defs or statements in the code.

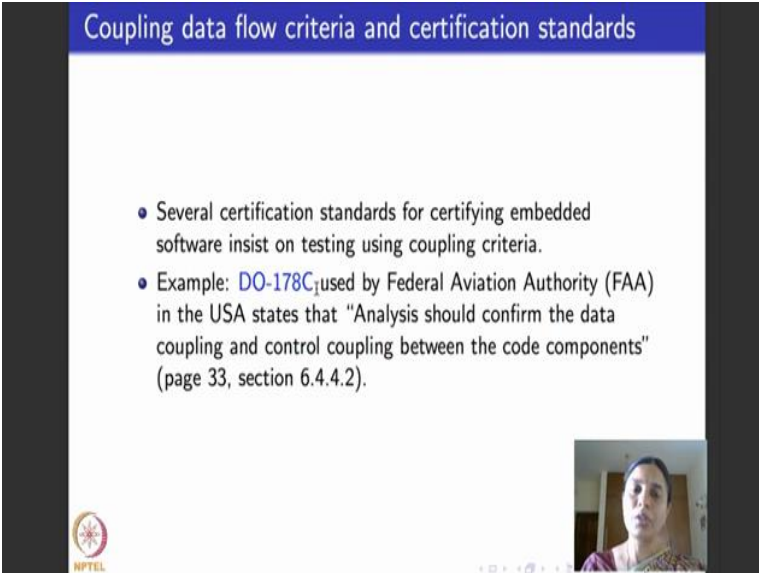
One more thing that I would like you all to note is that suppose you had this kind of a situation. A method A calls a method B, B in turn call C and let us say there is a variable

that is defined in A and used in C. So, this is what is called a transitive du-pair right. It goes down a callee hierarchy. The kind of analysis that I have explained to you in today's lecture will not suffice to do this kind of call graphs.

So, you need slightly different kinds of analysis for that and that we will not deal with in the scope of this course. So, if you have to do this kind of analysis, I would say that you change the code a little bit for to make method A use method B's variable directly coupled them with and B's and C's variable, you couple it. Avoid coupling A and C variable I will point you to some papers at the end of this lecture where you can read up some more information about this, but the kind of analysis testing that we did today will not work for transitive pairs.

And the other thing is that whenever you look at arrays, if I pass if I have to pass an array as a formal parameter or a natural parameter, how will I give ? There the interpretation that we take is a reference to one element in the array is considered as the reference to an entire array.

(Refer Slide Time: 36:49)



The slide is titled "Coupling data flow criteria and certification standards" in a blue header. It contains two bullet points: "Several certification standards for certifying embedded software insist on testing using coupling criteria." and "Example: DO-178C used by Federal Aviation Authority (FAA) in the USA states that 'Analysis should confirm the data coupling and control coupling between the code components' (page 33, section 6.4.4.2).". In the bottom right corner, there is a small video feed of a woman speaking. The NPTEL logo is in the bottom left corner.

- Several certification standards for certifying embedded software insist on testing using coupling criteria.
- Example: DO-178C used by Federal Aviation Authority (FAA) in the USA states that "Analysis should confirm the data coupling and control coupling between the code components" (page 33, section 6.4.4.2).

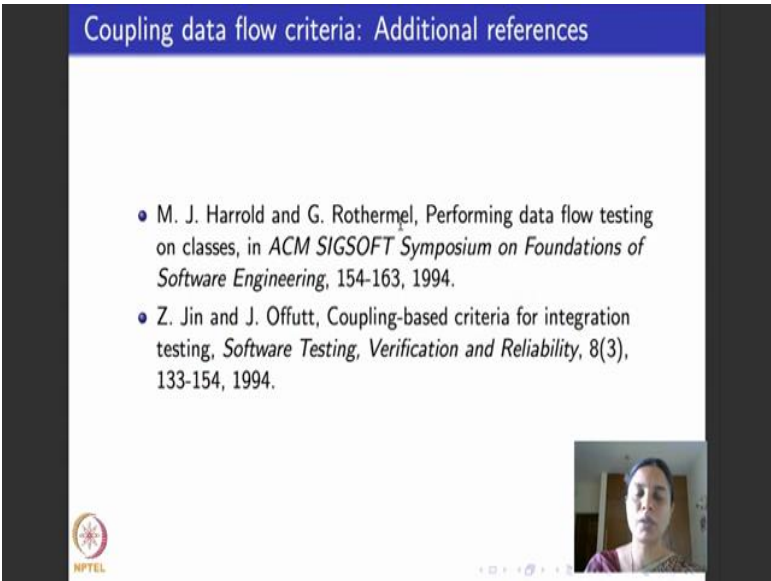
I would also like to mention one important point. Data flow kind of testing that we saw today which is coupling data flow criteria is very important when it comes to certifying testing safety critical software. If you remember right in the first week I told you that there are several standards and several standards that test for safety critical software. What are safety critical software? Software that flies our planes drives our cars controls

nuclear power plants maybe the software that runs metro trains across the metros of India, because if there is a failure in this software it can mean life threatening loss or damage.

So, these safety critical software go through certification standards which is what we discussed in the first week. One example of the certification standard that explicitly mentions coverage criteria is the standard DO-178C. It is used by federal aviation authority in the USA and in India also several of the DRDO firms used 178C as the reference for testing avionics applications. In that if you go to the standard look at page 33, this particular section it explicitly mentions 2 tests for coupling data flow analysis which is what is the kind of testing that we looked at today.

So, it says analysis should confirm the data coupling and control coupling between the core concept. So, it asks you to do testing and provide evidence of the fact that you have tested the interface for both structural coverage and data coupling.

(Refer Slide Time: 38:21)



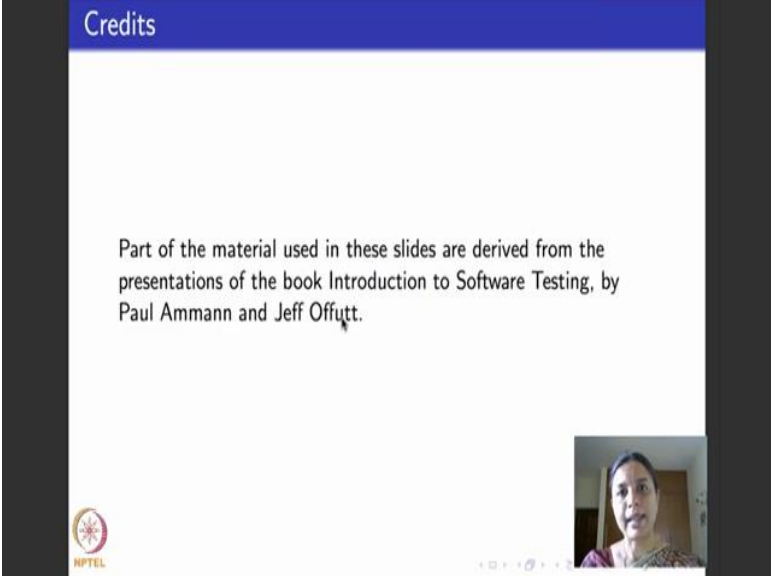
Coupling data flow criteria: Additional references

- M. J. Harrold and G. Rothermel, Performing data flow testing on classes, in *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 154-163, 1994.
- Z. Jin and J. Offutt, Coupling-based criteria for integration testing, *Software Testing, Verification and Reliability*, 8(3), 133-154, 1994.

NPTEL

So, here are some papers where you can read up for more information. This paper by Harrold and Rothermel is considered the first definition, first place where this coupling and data flow testing across coupling was introduced. And this paper by Jin and Offutt is the paper that you should refer to for explicit first time introduction of the term last def and first uses; thank you, what we will do next module is to be able to look at specifications and see how graphs are used for specifications.

(Refer Slide Time: 38:45)



Credits

Part of the material used in these slides are derived from the presentations of the book Introduction to Software Testing, by Paul Ammann and Jeff Offutt.

NPTEL

Thank you.