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

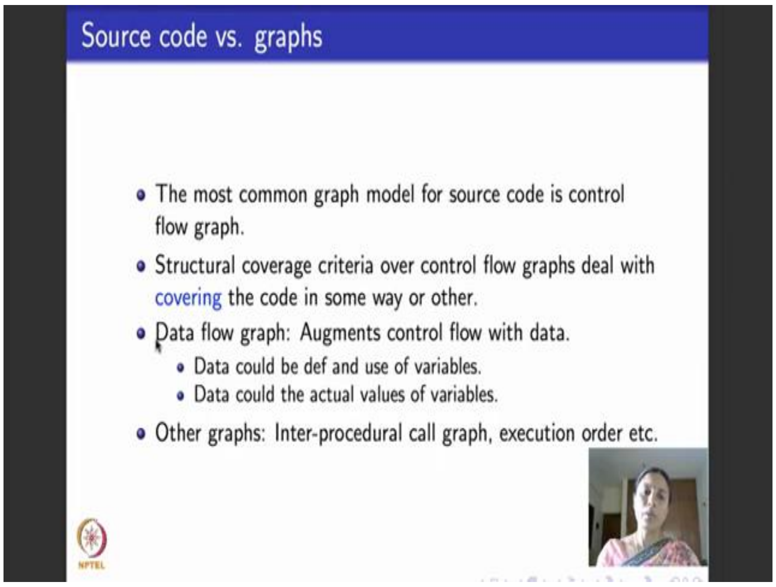
Lecture - 14
Testing Source Code: Classical Coverage Criteria

Hello again. We are going to look at the last lecture of week 3 today. So, the focus of this week has been two things: one is data flow coverage criteria on graphs and then what we did is we went back to structural coverage criteria. So, how to model source code as control flow graphs and how the various structural coverage criteria applied over them.

Today what we will do is that we will learn several graph coverage criteria--- structural and those that deal with data. And independently if you are familiar with little bit of software testing you would have heard several classical terms in testing branch coverage, cyclomatic complexity and so on. What we will do in this module is to see what are the terms that we have learnt till now, how are all of them related to classical terms in testing? In this process I will also point you to a couple of other good books in testing that you could use for general reading, part of the material of these courses is derived from those books.

So, for today's module we will not use the book by Paul Amman and Jeff Offutt, instead we will see testing as it is existed for the past four decades or so. What are the classical terms and how what we have done till now relates to these classical terms.

(Refer Slide Time: 01:26)



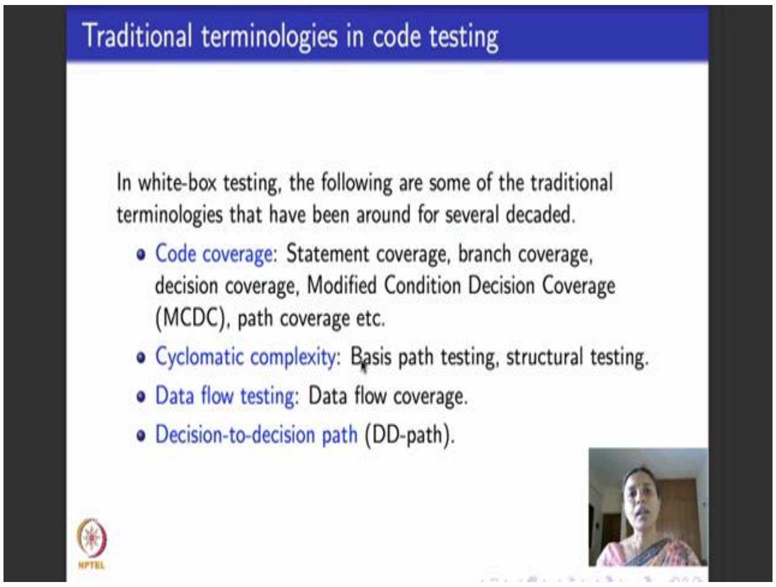
Source code vs. graphs

- The most common graph model for source code is control flow graph.
- Structural coverage criteria over control flow graphs deal with covering the code in some way or other.
- Data flow graph: Augments control flow with data.
  - Data could be def and use of variables.
  - Data could the actual values of variables.
- Other graphs: Inter-procedural call graph, execution order etc.

We know that the source code model can be written as a control flow graph as a graph model, what are the other models of the graph? The other common model of the graph is the data flow graph. The kind in data flow graph that we saw in this course took the control flow graph, augmented with definitions in uses. There is another kind of data flow graph that people use in program analysis that is, each node in the data flow graph is actually a triple representing the actual values of the various variables.

So, we will not really look at a data flow graphs for the scope of this lecture. There are several other graphs inter procedural call graphs execution order as I told you in the last lecture. What would we be doing today in this lecture is to go back in focus on the control flow graph model of a given piece of code and then understand what are the testing terminologies that have traditionally existed. Traditionally means you know somewhere from the mid seventies people have looked at these testing terminologies that we will be seeing today. So, for about four decades they have been around, very popular very well used, and will also understand how structural coverage criteria from this control flow graph relates to these classical terminologies that are always been used in test.

(Refer Slide Time: 02:43)



Traditional terminologies in code testing

In white-box testing, the following are some of the traditional terminologies that have been around for several decaded.

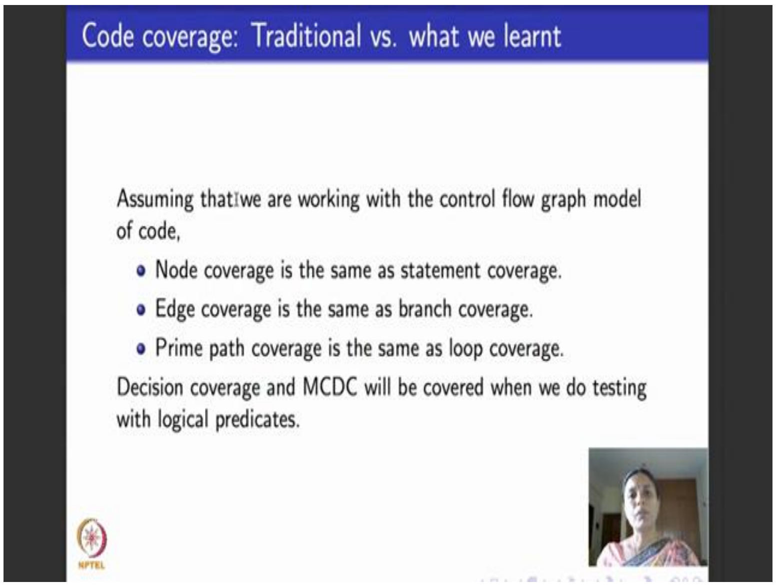
- Code coverage: Statement coverage, branch coverage, decision coverage, Modified Condition Decision Coverage (MCDC), path coverage etc.
- Cyclomatic complexity: Basis path testing, structural testing.
- Data flow testing: Data flow coverage.
- Decision-to-decision path (DD-path).

So, what are some of the classical terminologies that we will be looking at and relate them to graph coverage criteria? You might have heard some of these terms when it comes to white box testing. Ya, that is another thing, because we are looking at code and we are looking at structural coverage of code we are given white box testing right? So, in white box testing some of the popular terminologies that exist are what is called code coverage. Code coverage means it says you cover the code some way or the other in that you could do statement coverage; which says that you write test cases that will execute every statement in the code. You could do branch coverage which says you write test cases that we exercise each branch in the code. You could do decision coverage which will exercise test cases which will exercise each decision the various decisions could be those that reside in if statements as conditions in loops and so on.

Or you could do what is called modified condition decision coverage (MCDC) or you could do what is called path coverage. Things like decision coverage, MCDC and all are very popular terms that are used to test what are called safety critical software. So, we will see what these are and how they relate to the coverage criteria that we have looked at till now. Another popular term that you would have encountered while did testing was what is called Cyclomatic complexity right? Related to cyclomatic complexity people look at what is called basis path testing or structural testing.

Another popular term in testing is data flow testing that directly deals with data flow coverage mostly as we have dealt with in the course. And then, another popular term is what is called decision to decision abbreviated as DD-path testing. So, what we will do is that we will take each of these one after the other except for data flow testing. I will handle data flow testing separately as the first module of next week. So, we look at the other things which are coverage, cyclomatic complexity and DD-paths and see what they mean as far as their relationship to the kind of structural coverage criteria that we have seen till now.

(Refer Slide Time: 05:00)



Code coverage: Traditional vs. what we learnt

Assuming that we are working with the control flow graph model of code,

- Node coverage is the same as statement coverage.
- Edge coverage is the same as branch coverage.
- Prime path coverage is the same as loop coverage.

Decision coverage and MCDC will be covered when we do testing with logical predicates.

As I told you we assume that we have a piece of code and we have derived the control flow graph of that code.

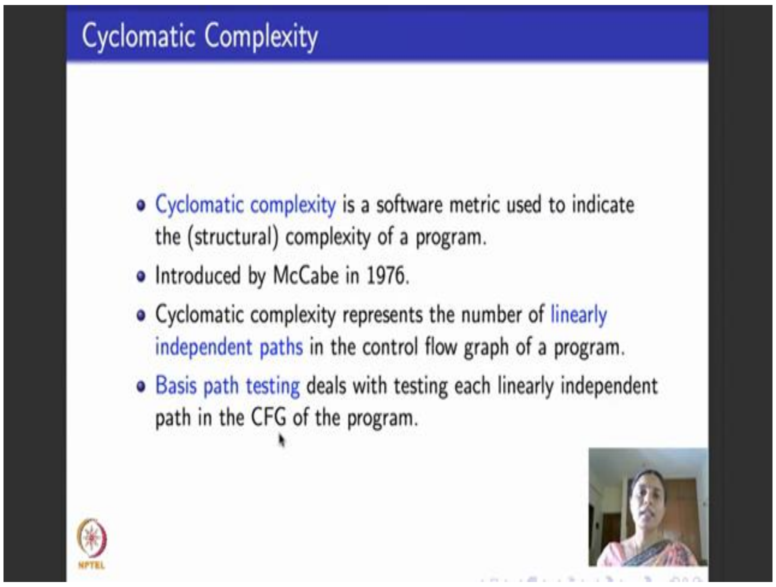
In the last module I took the statistics program example and showed you how to write the control flow graph piece by piece. In that module before that we saw pattern matching example and we saw the control flow graph and the data flow graph corresponding to the pattern matching example right. If you are working with control flow graph then what we saw as a node coverage criteria over graphs is the same as the statement coverage. Why is that so? Because if you see through examples the control flow graph, what is each node or vertex in the control flow graph correspond to? It corresponds to one statement or a set of statements that occur in sequence without branching in between them.

So, when I say my test requirement is node coverage, what I mean is execute every statement in the graph. So, what is popularly called as statement coverage is basically what we refer to as node coverage. Similarly edge coverage is the same as branch coverage. So, branch coverage says you take every branch in the graph right. So, ewhere there in branches come from? They come from edge statements, they come from loops, they come from switch case statements and they result in various edges that go out of the graph. So, when my test requirement is edge coverage then basically I am looking an branch coverage of the corresponding source code.

And then finally, loop coverage is another term. In white box testing they say you write test cases that will cover every loop it will skip the loop, it will execute the loop, once it will execute the loop for a few iterations within the maximum number of iterations allowed by the them. We all know that we saw prime path coverage. So, prime path coverage as a TR is basically the same as loop coverage in the underlying piece of code right. Now if you go back to the previous slide in the various coverage criteria we understood statement coverage, branch coverage, loop coverage as an extra thing, path coverage could be thought of as partly doing loop coverage, we said we won't explicitly look at complete path coverage right because several times it turns out to be an infeasible requirement. So, we did prime paths.

Today I will tell you few other kinds of path coverage criteria. What is left? Decision coverage and MCDC. Decision coverage and MCDC we will look at then we to testing with logic predicates as our models. So, when we finish graphs right in a few weeks from now, the next module that we will be beginning is to assume software artifacts as various kinds of logical predicates and we will design test cases based on those logical predicates. So, when we do that the coverage criteria that we see there would correspond to decision coverage and MCDC.

(Refer Slide Time: 08:00)



Cyclomatic Complexity

- Cyclomatic complexity is a software metric used to indicate the (structural) complexity of a program.
- Introduced by McCabe in 1976.
- Cyclomatic complexity represents the number of linearly independent paths in the control flow graph of a program.
- Basis path testing deals with testing each linearly independent path in the CFG of the program.

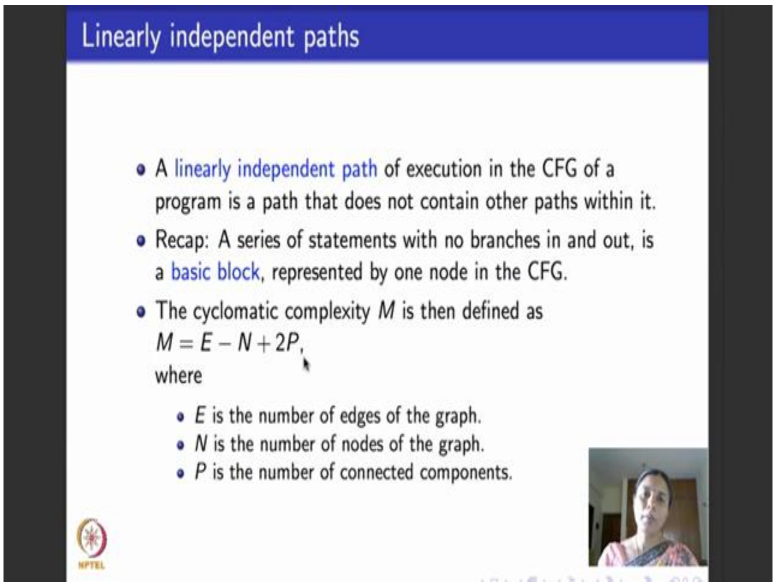
Moving on, the next popular testing term that you would have heard is what is called cyclomatic complexity. It is a pretty old term actually, it is more than 4 decades old, it was introduced by this person called McCabe in the year 1976.

So, what is cyclomatic complexity? It is a software metric; what is this software metric? Software metric is some kind of a measure about the piece of code that we have in hand. The most popular software metric is what is called lines of code and usually for fairly big software that are deployable in/come for commercial purposes, lines so there are several thousand lines of codes. So, this is a kilo lines of code a KLOC for every thousand lines of code. That is the most popular software metric. So, we say a large software has several KLOCs means several thousand lines of code.

Another popular metric that is used to measure how complex a software is, is what is called cyclomatic complexity. Here complexity is not measured in terms of the number of lines of code; it is measured in terms of the number of various branches that can occur in a software. So, what does cyclomatic complexity represent? It represents the number of linearly independent paths in the control flow graph for program. A testing that deals with cyclomatic complexity is what is called basis path testing. Basis path testing basically enumerates the number of different linearly independent paths and then tests with reference to the cyclomatic complexity of the program.

So, what can spend the next few minutes on is trying to understand how exactly to compute cyclomatic complexity given a control flow graph of a graph of a code, and what does basis path testing as a class of testing, a category of testing correspond to testing it with reference to the cyclomatic complexity measure.

(Refer Slide Time: 09:59)



Linearly independent paths

- A linearly independent path of execution in the CFG of a program is a path that does not contain other paths within it.
- Recap: A series of statements with no branches in and out, is a basic block, represented by one node in the CFG.
- The cyclomatic complexity M is then defined as $M = E - N + 2P$, where
  - E is the number of edges of the graph.
  - N is the number of nodes of the graph.
  - P is the number of connected components.

So, with cyclomatic complexity as I told you deals with a number of linearly independent paths. So, we first need to understand what is a linearly independent path. What is a linearly independent path? It is a path that does not come as a sub path of any other path.

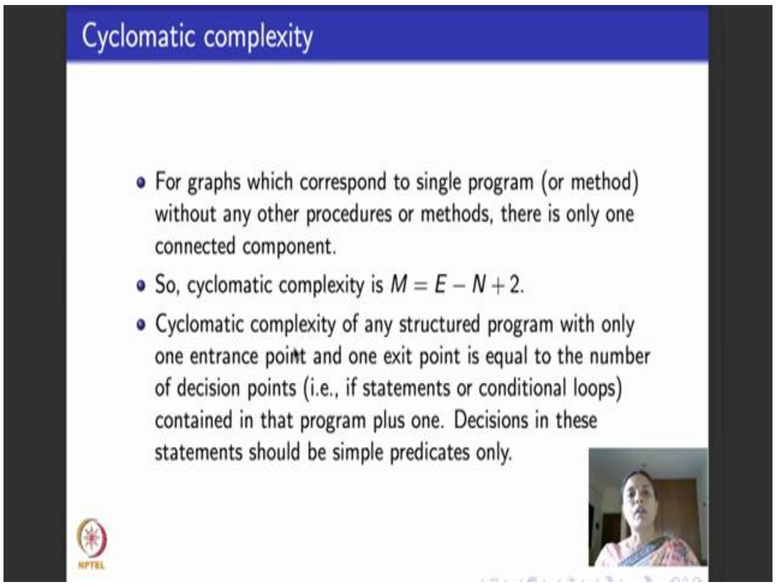
So, you take the control flow graph or the CFG of a program. There can be several paths in the graph. It is linearly independent path is similar to prime path. If you see prime path is a path that do not come as a sub path of any other simple path in the program. Linearly independent path is a path that do not come as a sub path of any other path in the program. So, except for the fact to the word simple is not there linearly independent paths are the same as prime paths, right. So, what we do is, how do we compute the cyclomatic complexity? We compute the cyclomatic complexity denoted as M, by using this formula.

So, it says M = E - N + 2P. What is E? E is the number of edges in the control flow graph, N is the number of nodes in the control flow graph what is P? P is the number of connected components in the control flow graph. How do I obtain the control flow graph? When I calculate E N and P an make one important assumption about the control flow graph. You remember when I told you that when you have a series a statements, let us say assignment statements or print statements one after the other you have the choice to keep one vertex or one node for each statement or you have the choice to collapse these series of statements as one vertex ,one individual vertex assuming that there is no branching in between them.

So such a series a statements is what we called a basic block; and when we do control flow graphs of several code fragments in the last module, I assumed one vertex for every basic block. For computing cyclomatic complexity that assumption turns to be very important, this formula will not be correct if you do not assign one vertex for one basic block. If you take the other approach where I assign one vertex for every individual statement, then this will not be the formula for cyclomatic complexity.

So, cyclomatic complexity assumes that the control flow graph is drawn in such a way that each node represents a basic block of statements, and different nodes exist only when there is a need to branch out of that node, only when there is a decision statement out of that node. Under that assumption that is how we built most control flow graphs throughout our lecture. So, for all those control flow graphs the formula to compute cyclomatic complexity is this, very easy to remember it is E - N + 2 P; where E is the number of edges N is the number of nodes, P is the number connected components.

(Refer Slide Time: 12:55)



Cyclomatic complexity

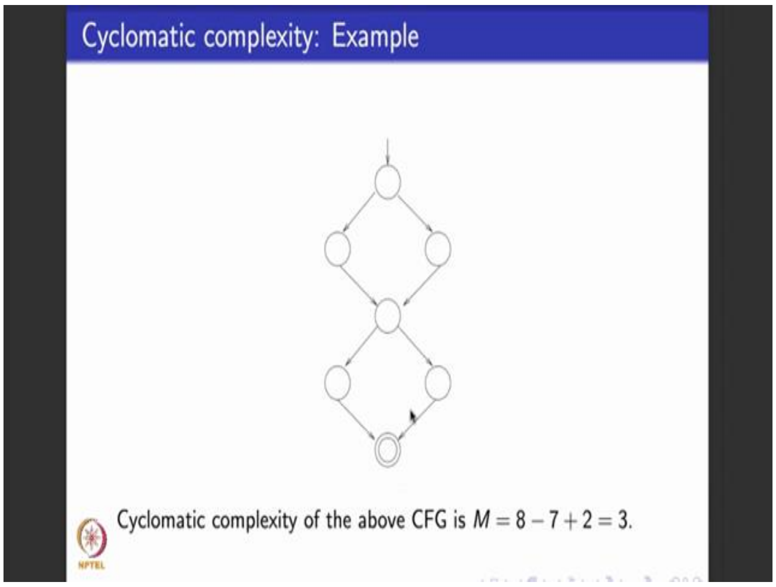
- For graphs which correspond to single program (or method) without any other procedures or methods, there is only one connected component.
- So, cyclomatic complexity is $M = E - N + 2$.
- Cyclomatic complexity of any structured program with only one entrance point and one exit point is equal to the number of decision points (i.e., if statements or conditional loops) contained in that program plus one. Decisions in these statements should be simple predicates only.

For graphs that correspond to a single program a single procedure or a single method assuming that there are no inter procedural calls that I model in my graph, then if you remember the control flow graph looks like one large connected component right. It does not looks several bits and pieces of disconnected components. If I consider that then this last term P is the number of connected components is basically one for a graph control flow graph that corresponds to one method right. So, the formula for cyclomatic complexity become simpler it is E - N + 2.

So, it so turns out that the cyclomatic complexity of a program with only one entry and one exit means with only one initial vertex and one final vertex, is the same as the number of decision points. What are decision points? Points where is branching out if switch and so on. So, number of decision points contained in that program plus 1. So, what it says is that how do linearly independent paths come, I go through a series a statements at some point I branch out. When I branch out I have a choice I take the left branch or I take the right branch assuming that it is the branch of an if statement with an else part.

So, each look at rise to one linearly independent path, and what happens is the cyclomatic complexity of a program is basically the number of such branches plus 1, and it exactly gives the number of linearly independent paths in a program. And they say that usually a good indicator of good software that is easy to maintain and easy to handle should have cyclomatic complexity somewhere between 1 and 10. Cyclomatic complexity of 1 means the software has no branching. So, it may not be very useful to look at such software. So, cyclomatic complexity should be typically between 2 and 10. Any number less than 10 is considered to be a good cyclomatic complexity for a piece of software when it comes to maintaining and handling this software.

(Refer Slide Time: 14:55)



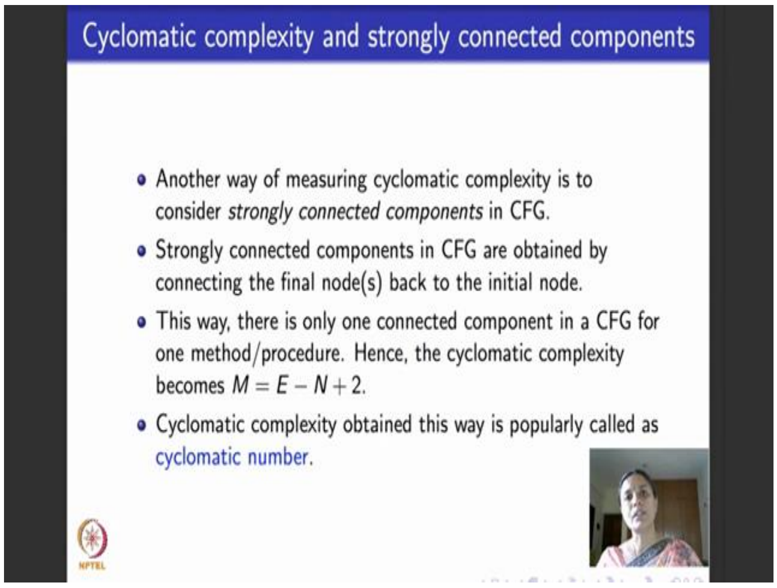
Cyclomatic complexity: Example

Cyclomatic complexity of the above CFG is $M = 8 - 7 + 2 = 3$.

So, here is a small example. So, this software, this is the control flow graph corresponding to some piece of code. It has exactly one initial vertex marked here, it has one final vertex marked here and then it has two if statements. There is one branching here which branches out to this node and this node, there is one more branching here which branches out to this node and this node. So, there are two if statements; if you count the number of edges in this graph there will be eight edges, and there be seven vertices and this whole graph is like one connected component.

So, the formula, cyclomatic complexity turns out to be 3. If you see how many branches are there in this graph? There are two decision points one here and one here. So, what did we discuss here, we said the cyclomatic complexity is the number of decision points plus 1. That is true for this example; it is the number of decision points plus 1 which is 3.

(Refer Slide Time: 15:50)



Cyclomatic complexity and strongly connected components

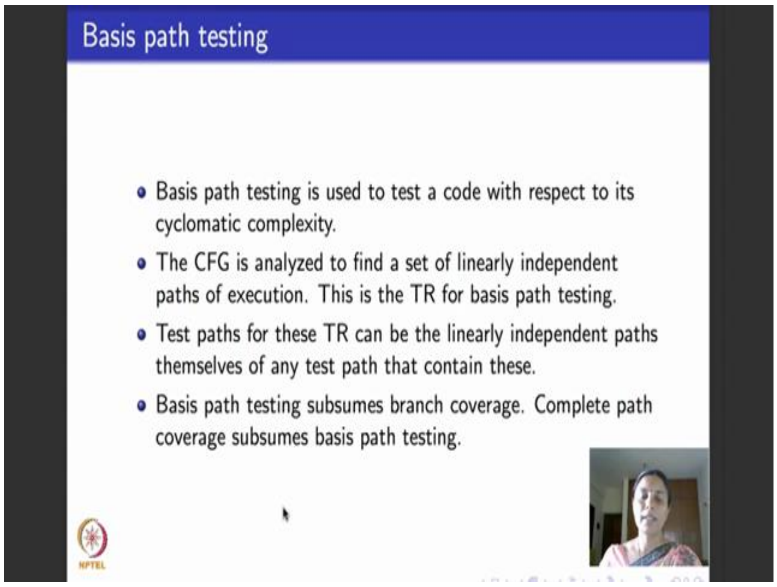
- Another way of measuring cyclomatic complexity is to consider *strongly connected components* in CFG.
- Strongly connected components in CFG are obtained by connecting the final node(s) back to the initial node.
- This way, there is only one connected component in a CFG for one method/procedure. Hence, the cyclomatic complexity becomes $M = E - N + 2$.
- Cyclomatic complexity obtained this way is popularly called as cyclomatic number.

So, that is another popular way of checking for cyclomatic complexity. What people do is the suppose this turns out to be the control flow graph of a particular program right. What they do is they put a dummy edge back from the finding vertex to it is initial vertex, like this, you trace assume that I am tracing an edge back from the final vertex, to it is initial vertex. After doing that the graph becomes one strongly connected component. You remember what strongly connected components are, we looked at them in the graph algorithms lecture. Strongly connected components are those components in which every pair a vertices are reachable from each other.

So, under that assumption cyclomatic complexity also turns out to be E - N + 2 basically because of the same reason, these exactly one connected component which actually happens to be strongly connected also. So, cyclomatic complexity becomes simpler, I do not have to compute connected components. It is usually easier this way because if you have something like this at the cost of adding a few extra edges, then you do not have to run algorithms that compute the number of strongly connected components as subroutines. So, directly count the number of edges, number of vertices, which I usually easy to do given a representation of the graph, and there you go, the cyclomatic complexity is easily measurable without calculating any strongly connected component.

So, usually this measure of cyclomatic complexity is preferred and it is popularly known as the cyclomatic number; some books and papers also call it the Betty number.

(Refer Slide Time: 17:23)



Basis path testing

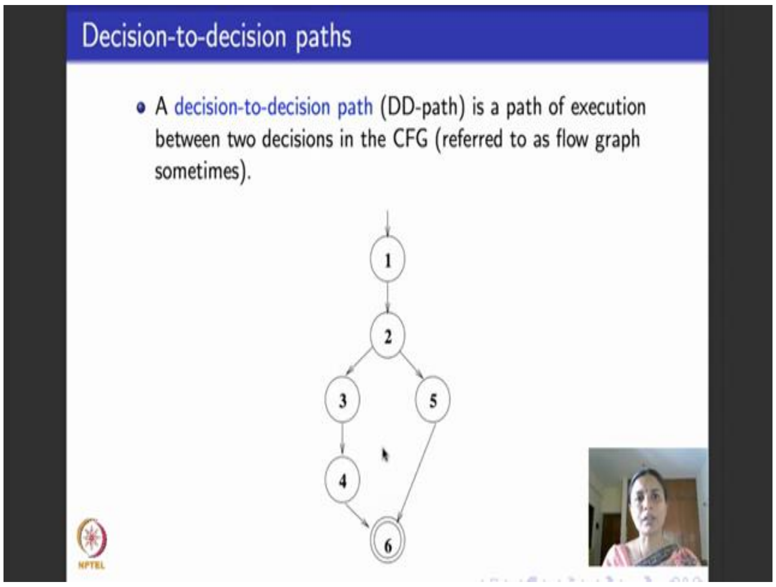
- Basis path testing is used to test a code with respect to its cyclomatic complexity.
- The CFG is analyzed to find a set of linearly independent paths of execution. This is the TR for basis path testing.
- Test paths for these TR can be the linearly independent paths themselves of any test path that contain these.
- Basis path testing subsumes branch coverage. Complete path coverage subsumes basis path testing.

So, now let us look at basis path testing. As I told you basis path testing test the cyclomatic complexity of a piece of software, what it basically does is it enumerates all the linearly independent paths in the graph or in our terms, it enumerates all the prime paths in the graph, and for each such path, this is your TR, your test requirement. And any set of test paths that satisfy this TR would hold good for basis path test. Because we looked at prime paths and enumerate in prime path and not revisiting any algorithm that will enumerate linearly independent paths, it is very similar to this. So, we consider the same algorithm that we did for prime path as working for enumerating linearly independent paths and test it.

And like we did for prime paths coverage, basis path testing subsumes branch coverage or edge coverage, complete path coverage in turn subsumes basis path testing. If you are confused what I am trying to say is that basically linearly independent paths correspond to what we saw as prime paths, the only difference is simple path versus non simple paths. So, basis path testing is more or less the same as prime paths testing. So, how will I come up with a set of test requirements for basis paths? What are basis path testing test requirements? Enumerate all the linearly independent paths; it is the same as enumerating all the prime paths.

So, I will use the same algorithm that we saw for prime paths to enumerate all the linearly independent paths, that is my test requirement, and then I come up with the test paths that satisfy the test requirement exactly in a way similar to the ones that we did for prime path test.

(Refer Slide Time: 19:05)



Decision-to-decision paths

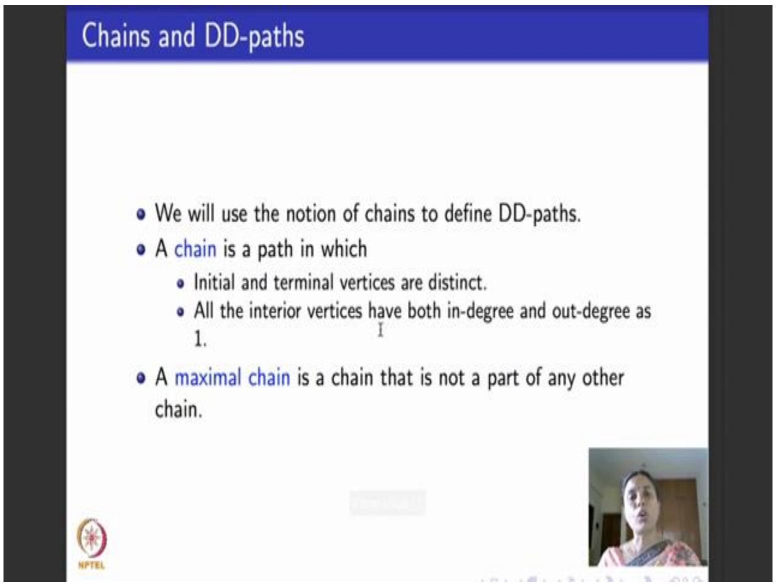
- A decision-to-decision path (DD-path) is a path of execution between two decisions in the CFG (referred to as flow graph sometimes).

So, the last one that I wanted to deal with, another popular term that is been used in testing is what is called a DD-path or the decision to decision path. This is different from basis paths or linearly independent paths or from branches or from any of the kind of paths that we have seen throughout the span of this course. So, in short what is a DD-path? A DD-path is a path of execution between two decisions in the CFG. The books that talk about DD-paths usually call the control flow graph as the flow graph or a program flow graph, but if you are confused please remember that it is the same as what we call a CFG. At the end of these modules, this slides, this module I will point you to good references where you can read more information about cyclomatic complexity and about DD-paths.

For now we will just introduce what a DD-path is and say how we could go about testing for DD-paths. But so, here is the small control flow graph, its initial vertex is 1 final vertex is 6 and where are all the branches? There is a branching here, at 2 there is a possible choice of going to 3 or going to 5, and from 3 they you can go to 4 there is no choice there and if you see 6 there are two ways that you could use to come to 6, you could have come from this 2, 3, 4 and 6 or you could have come by using 2, 5 and 6.

So, I considered 6 also as having a decision point, as representing decision point in the graph, 2 as representing another decision point in the graph. Why because for both these kinds of vertices there is a choice about either coming in or going out of the vertex the choice in more than one way to come in or more than one way to go out. So, these are what are call decision points in the graph; and DD-path says you test, your test requirement of TR is a set of all paths between two decision points in the control flow graph.

(Refer Slide Time: 21:12)



Chains and DD-paths

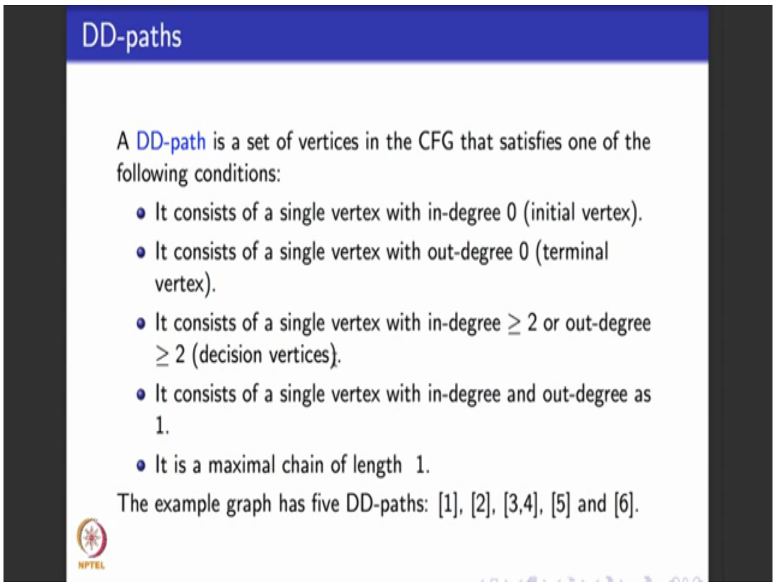
- We will use the notion of chains to define DD-paths.
- A chain is a path in which
  - Initial and terminal vertices are distinct.
  - All the interior vertices have both in-degree and out-degree as 1.
- A maximal chain is a chain that is not a part of any other chain.

So, to see what a DD-path is in detail, we will need the notion of what is called a chain in a graph. You might have heard about the term chain when you looked at what are called partial orders. You typically see partial order is in a course with discrete math let us say, chains are what are call total orders there to the sake of simplicity and redefining chains here. So, what is the chain? A chain is a path which satisfies the following two conditions. The initial vertex and the terminal vertex of this path are distinct which means it is not a loop it is not a cycle, and then all the interior vertices, what are interior vertices? Interior vertices are those that are not the initial vertex and not the final vertex, everything else in between that comes in between. All the interior vertices have both in degree and out degree as one.

So, if you try to visualize how this chain will look like, it looks like one long path in the graph from an unique initial vertex to unique final vertex. And what is a maximal chain?

A maximal chain is a chain that is not a part of any other chain. Its somewhat like prime path right, a prime path is a path that is not a sub path of any other path, the maximal chain is chain does not a sub chain of any other chain.

(Refer Slide Time: 22:27)



DD-paths

A DD-path is a set of vertices in the CFG that satisfies one of the following conditions:

- It consists of a single vertex with in-degree 0 (initial vertex).
- It consists of a single vertex with out-degree 0 (terminal vertex).
- It consists of a single vertex with in-degree ≥ 2 or out-degree ≥ 2 (decision vertices).
- It consists of a single vertex with in-degree and out-degree as 1.
- It is a maximal chain of length ≥ 1.

The example graph has five DD-paths: [1], [2], [3,4], [5] and [6].

So, what is a decision to decision path a DD-path? A DD-path is basically a path of vertices in this CFG that satisfies any of the following five conditions. If it satisfies any of these five conditions we call them all as decision to decision paths.

So, let us look at the conditions one after the other. So, it could consist of a single vertex with in degree 0. If you think about it visualizing the CFG, which is that vertex that has in degree 0; what does it mean for a vertex do not in degree 0, there is nothing, no edge coming into that vertex. The only vertex that has in degree 0 is the initial vertex then it can consist of a single vertex without degree 0. Again you visualize our model of a control flow graph which is the vertex that has out degree 0, it is our final vertex or a terminal vertex, and then decision path can also consist of a single vertex with in degree greater than or equal to 2 or out degree greater than or equal to 2.

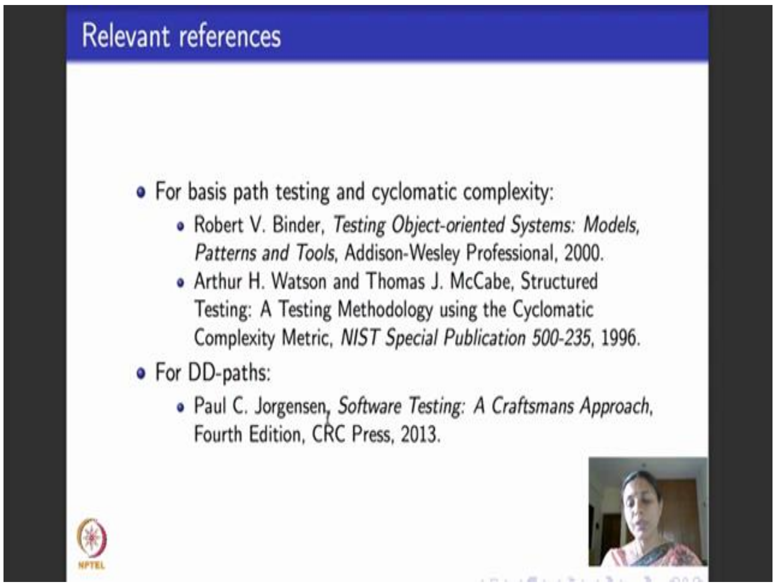
If we go back to the example graph that we had, vertex 2 is the vertex without degree as two, vertex 6 is a vertex with in degree as two. So, both vertices 2 and 6 satisfy this condition and both represent some kind of decisions that is what I was trying to explain a few minutes ago. Decision path can also consists of single vertex with in degree and out degree as both one, or a decision path could be a maximal chain of length one. So, just to clarify, in case you find it confusing, what is the decision path? Basically it is a path that takes you from one decision to another decision. As I told you when we consider decisions, in for the sake of DD-paths, decisions could be of two kinds they could be of the classical kind like this, branch out or they could be of the kind where that is decision for branching in for coming in.

So, what it says is that DD-path let us you go from one decision to one decision. So, it clubs the vertices as all these things, it says it initial vertex is one separate DD-path final vertex is another separate DD-path then, all the decision vertices which are vertices with in degree greater than or equal to 2 or out degree greater than or equal to 2 are separate decision paths, and then, in between if I have a single vertex with in degree 1 and out degree 1 then that is another decision path, and then the other kind of decision paths could be chains of length 1.

So, if I apply this definition to this example graph, how many decision paths, DD-paths will I get? The initial vertex 1is one DD-path final vertex 6 is one DD-path, the decision vertex 2 is a separate DD-path, the vertex 5 which has in degree 1 and out degree 1 is another DD-path, and 3, 4 is a chain of maximum length 1. So, this is another DD-path. So, totally for this graph there are five DD-paths: initial vertex, final vertex the decision point 2, vertex 5 which has in degree and out degree 1 and the maximum chain 3, 4 right. All of these satisfy one of these conditions that what is a written here the example graph; that means so, as 5 DD-paths which have listed here.

So, we will not really see how to enumerate DD-paths and how to test them and all because they slightly how to scope for this course.

(Refer Slide Time: 26:21)



Relevant references

- For basis path testing and cyclomatic complexity:
  - Robert V. Binder, *Testing Object-oriented Systems: Models, Patterns and Tools*, Addison-Wesley Professional, 2000.
  - Arthur H. Watson and Thomas J. McCabe, Structured Testing: A Testing Methodology using the Cyclomatic Complexity Metric, *NIST Special Publication 500-235*, 1996.
- For DD-paths:
  - Paul C. Jorgensen, *Software Testing: A Craftsmans Approach*, Fourth Edition, CRC Press, 2013.

But if you are interested in knowing more about it this is a very good book to learn more about DD-paths, it is a book called Software Testing: A Craftsman approach, by Jorgensen, and a recent addition is available. For getting to know about basis path testing and cyclomatic complexity, as I told you McCabe is the founder of Cyclomatic complexity he has an NIST report. NIST if you remember as National Institute of Standards and Technology. here is an NIST report which details how to find cyclomatic complexity, how to do basis path testing and how to use cyclomatic complexity for object oriented software for integration testing and so on it is a very good optical.

This another popular testing book the book by Robert Binder, which focuses on testing object oriented software also has an exhaustive discussion on cyclomatic complexity and basis path testing. So, if you are interested you could further read on from any of these books and feel free to ping me in the forum if you have any clarifications or questions that you would like a me to answer. But the main focus of this lecture was to help you understand that there are several classical terms that are present like the terms that we saw today, and see what they are and how they relate to whatever we have seen till now.

So, I hope you are not confused about why am I seeing these graph coverage criteria and how does it relate to cyclomatic complexity. And I hope this lecture would have helped you to understand and answer some of the questions that you had in your mind.

Thank you.