

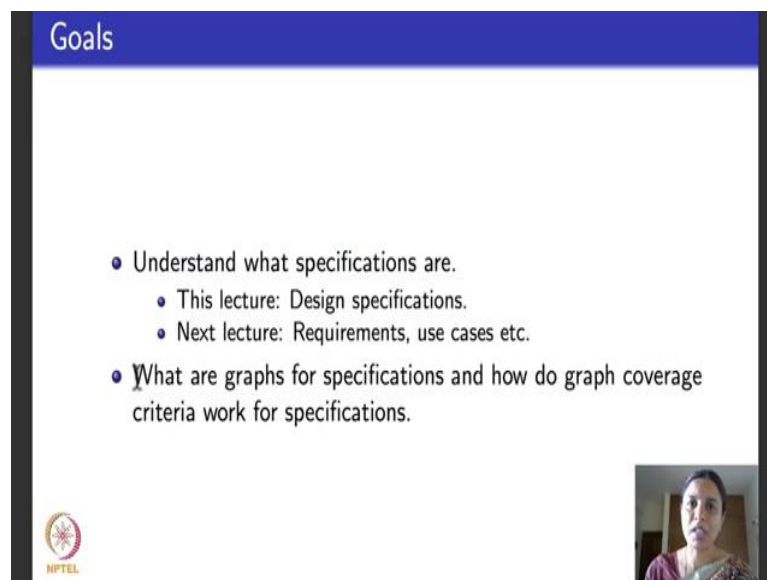
Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 18
Specification Testing and Graph Coverage

Hello again, we are now doing the fourth lecture of fourth week. As I said this will be the last week where we look at graph coverage criteria. We saw graph coverage criteria on code. Last 2 lectures I told you how to apply graph coverage criteria for integration testing that deals with testing for design that is interaction and interfaces between modules. Now we will move on to the requirements on design.

So, what we will see in this module and in the next module is somewhere between design and specifications and see how graph criteria apply to test these also. So, we will understand what specifications are. The focus of this lecture would be specifications that actually talk about design itself.

(Refer Slide Time: 00:51)

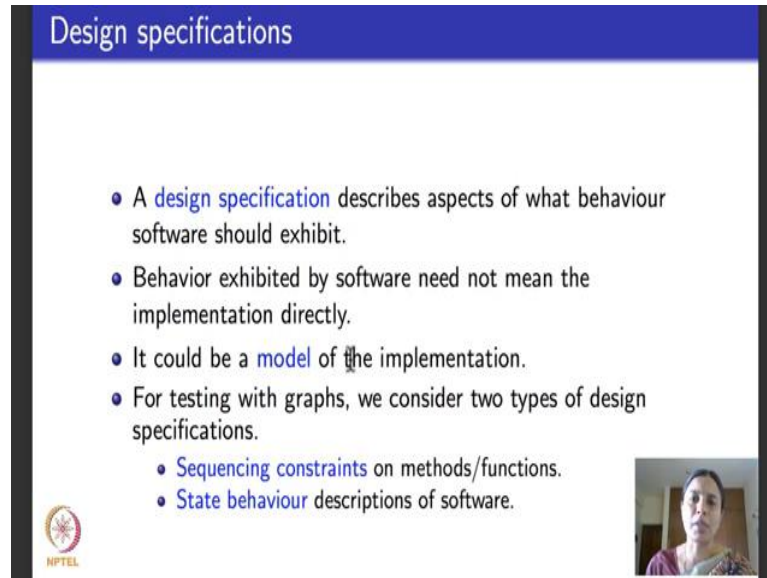


The slide has a blue header with the word "Goals" in white. Below the header, there is a white area containing a bulleted list of goals. In the bottom right corner of the slide, there is a small video inset showing a woman (Prof. Meenakshi D'Souza) speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

- Understand what specifications are.
 - This lecture: Design specifications.
 - Next lecture: Requirements, use cases etc.
- What are graphs for specifications and how do graph coverage criteria work for specifications.

So, I will explain to you through examples what it means to write specifications that talk about design because there are several specifications that we need to write along with the design to ensure the things are working correct, even though these specifications may not explicitly map to the requirements of a software.

(Refer Slide Time: 01:35)



The slide is titled "Design specifications" in a blue header. It contains a bulleted list of points. In the bottom left corner, there is a small circular logo with the text "HPTCL" below it. In the bottom right corner, there is a small video inset showing a person's face.

- A **design specification** describes aspects of what behaviour software should exhibit.
- Behavior exhibited by software need not mean the implementation directly.
- It could be a **model** of the implementation.
- For testing with graphs, we consider two types of design specifications.
 - **Sequencing constraints** on methods/functions.
 - **State behaviour** descriptions of software.




So, today we will deal with those kind of specifications and in the next lecture, I will introduce you to finite state machines, finite state automata which are a popular model to write requirements and then we will see how graph coverage criteria work on them. What is a design specification? So, as I told you design describes; what are the modules in the software, how software is broken up into its components and how they interact with each other and what happens? A design specification talks about what are the requirements or constraints that need to be satisfied by the design that is being written and base-lined to make the design correct.

Not all design specifications are derived directly from the requirements of software. Some of them are extra specifications written to ensure that the design is indeed correct. So, design specification can be thought of as the path of the model, design model corresponding to the implementation. While we do graph based testing we will consider 2 types of design specifications. What I will do in today's lecture would be what is called sequencing constraints which are constraints on methods and functions that come in the design or modules that come in a design. Not all sequencing constraints can be used to write all kinds of design specifications. So, certain things that cannot be used we will look at finite state machines that describe state behavior that we will do in the next week.

(Refer Slide Time: 02:52)

Sequencing constraints

- Sequencing constraints are rules that impose constraints on the order in which methods may be called.
- They can be encoded as preconditions or other specifications.
- They may or may not be given as a part of the specification or design.
- Testers need to derive them if they don't exist— they are considered another rich source of errors.

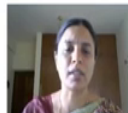




So, what is the sequencing constraint? Before I introduce you to what is a sequencing constraint, I would like to begin with a small example.

(Refer Slide Time: 02:54)

Sequencing constraints: An example

```
public int deQueue()  
{  
    // Pre: At least one element must be on the queue.  
    ...  
    ...  
    public enqueue(int e)  
    {  
        // Post: e is on the end of the queue.  
        ...  
        ...  
    }  
}
```



Page 5 of 17

Let us say this is just another abstract snippet of some design it says that there are 2 methods here. Both the methods deal with the queue data type, you know what a queue data type is right? A queue is a first in first out list or an array of elements. So, what are the typical operations that come with a queue? The typical operations that come with a queue are enqueue and dequeue - enqueue inserts an element into the queue dequeue

removes an element that is already present at the other end of the queue. So, there are these 2 methods - one is `deQueue` which is removed the element from the queue and assign it to an integer variable the other is insert an integer into a queue right. And I have not really given the code for these methods, but if you see there are 2 comments that are written here, for `deQueue` there is a comment which is named `Pre` which says at least one element must be on the queue.

So, you can read this as, it is a precondition which says that there must be at least one element in the queue. Why is this thought of as a precondition? Let us say the queue was empty. Then how do I `deQueue` anything. I will not be able to `deQueue`, `deQueue` will give me an error. So, to be able to `deQueue` I need at least one element in the queue. So, I state it like a precondition and if you read what is written in `enQueue`, I have written again as a comment the first condition which says that `e` is at the end of the queue. So, this is which basically says that the queue is like a FIFO data structure, first in first out data structure. So, when I insert an element in the queue it goes to the end of the queue and for me to remove an element from the queue the precondition is that there must be at least one element in the queue.

So, these can be thought of as basic constraints that tell you what methods `enQueue` and `deQueue` should satisfy. The methods `enQueue` and `deQueue` could be used as a part of a mainstream functionality that is meant to cater to a particular requirement of a software. But wherever they are used these preconditions and post conditions basically say what will happen or what should happen when they are used. Precondition for `deQueue` says that the queue should not be empty. So, there should have been at least one `enQueue` operation without another `deQueue` operation before the current `deQueue` operation.

And the post condition for `enQueue` says that after an element is inserted into the queue it goes to the end of the queue. So, these are what are called sequencing constraints. So, what is a sequencing constraint? These can be thought of as the rules that impose some constraints in the order in which the methods in a particular piece of code can be called. Like for example, for me to be able to do `deQueue` I should have done at least one `enQueue` in the past. So, these can be written as preconditions, post conditions, asserts or any other way of writing it and they are requirements that we impose on the design of the software.

On other thing that I told you that has to be noted is that these are typically not given as a part of the specification document or as a part of the requirement document. These are written along with design typically by designers to ensure that their design works correctly in the sense of it is necessary that these constraints are satisfied for the design to be able to meet its expected functionality. Suppose they are typically not given then the onus is on the tester to be able to derive them and test them.

(Refer Slide Time: 06:46)

Sequencing constraints: Example, contd.

- Simple sequencing constraint:
enQueue() must be called *before* deQueue()
- In the example code, it is implicitly given as pre and post conditions.
- Does not include the requirement that we must have at least as many enQueue() calls as deQueue() calls.
 - Can be handled by state behavior techniques.

NPTEL

Page 6 of 17

So, this is the queue example that we saw of a simple sequencing constraint on a queue which basically says that enQueue must be called before a deQueue. If that is not the case then the queue could be empty because it begins with an empty queue and there is nothing to deQueue from right. So, I do not want to keep trying to deQueue from an empty queue. So, insist that there must be at least one enQueue operation before a deQueue operation.

Here in this example, such a sequencing constraint is given implicitly as pre and post conditions. In fact, you can write several such other sequencing constraints for the queue example itself. Another simple sequencing constraint is something like this--- it says that there must be at least as many enQueue calls as there are deQueue calls. Why do we need this? Let us say I have a queue the queue contains at any point in time at some point in time let us say it contains 3 elements. So, what can I do? I can do continuously 3 deQueues. So, how did the three elements come into the queue? There must have been 3

enQueue operations in the beginning before the 3 deQueue operations. It can never be the case that there were only 2 enQueue operations, if there were only 2 enQueue operations in this queue then the queue cannot have three elements right assuming that the queue is empty to start with.



So, what it says is that to be able to deQueue I should have enQueued in element in the past. So, at any point in time the following hold as an invariant: the number of enQueue operations should be greater than or equal to the number of deQueue operations. So, if you see the word number matters here; we are talking about the number of enQueue operations and the number of deQueue operations. Whenever I am counting the number of operations related to a particular piece of software execution I typically you cannot write it as a simple assertion or a sequencing constraint. I need what are called state based models for that. We will not look at state based models in this lecture that will be the focus of next lecture. But what I wanted to tell you as a part of this lecture is that when I take a queue example or any other kind of example which is meant to be like a data structures serving as a part of a method, I could write constraints which talk about how the data structure should work correctly such constraints are called sequencing constraints.

Some sequencing constraints can be written as simple invariants like we saw here enQueue must be called before a deQueue. Some sequencing constraints need to deal with history or memory or number and they need state machines to be able to write them which we will see in the next lecture.

(Refer Slide Time: 09:20)

Testing sequencing constraints

- Sequencing constraints may or may not be given explicitly, might not be given at all.
- Absence of sequencing constraints usually indicates more faults.
- Tests are created as sequences of method calls, testing if the sequence obeys the constraints.
 - Usually write tests to find errors in constraints or missing constraints.



So, as I told you sequencing constraints typically are not explicitly given say they always have to be written by the designer and it could very well be the case to the designers mis-writing them if the designers miss writing them then the onus is on the tester to be able to identify, write them and test for all these sequencing constraints to be met. Like interfaces, when you go into integration testing absence of correct sequencing constraints or a software not adequately tested for correct sequencing constraints is meant to lead to a rich source of errors and onus is on testers to be able to identify and eliminate these errors during integration testing.



(Refer Slide Time: 10:01)

Testing sequencing constraints: An ADT example

Consider a class FileADT that encapsulates operations on a file.
Class FileADT has three methods:

- open(String fName): Opens file with name fName.
- close(): Closes the file and makes it unavailable.
- write(String textLine): Writes a line of text to the file.

What are natural sequencing constraints you would expect for this class?

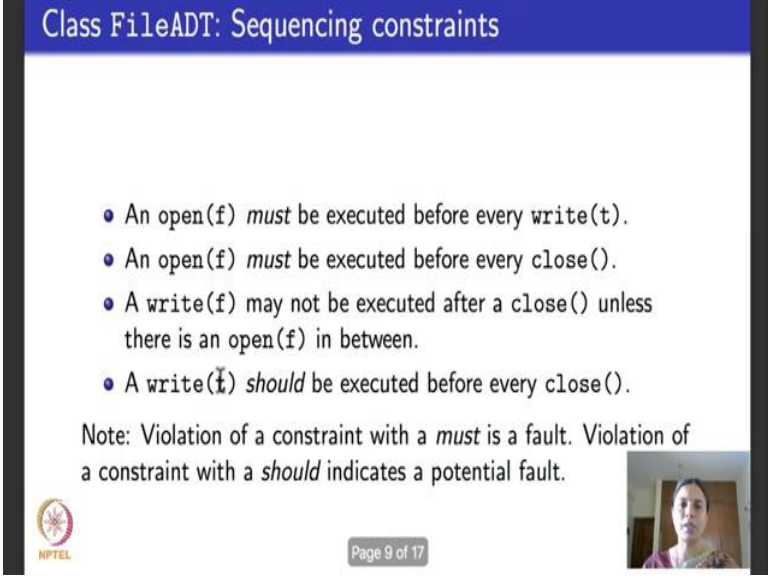


So, we will see how to test for sequencing constraint through an example. Consider a class called file ADT, ADT stands for abstract data type. This is the class that encapsulate operations on a file, that file has some kind of data. What is the typical bare minimum set of operations that you would do on a file? You could open a file which is done through this command: open file name with the name fName. So, this is a method open which let us you to open a file name with the name fName and once a file is opened you could close it. I have not given an argument as the name of the file here just for convenience sake, but you could as well write close a string followed by file name. So, that you know which file to close because I have not written an argument for close here I assume through the examples that we look at that I am working with only one file at a time. Suppose it happens to be the case that you are working with more than one file you need to give close operations for that file name also.

And once the file is open before being closed you would want to work with a file you could read data from the file or you could write on to a file. So, here is a method that writes on to a file. So, this method writes a line of text called text line which is a string to the file.

So, there is a class file abstract data type in short, file ADT which encapsulates three methods open file, close file and write data to a file. So, if you think about sequencing constraints for any code that uses these methods. So, there is a code that deals with some file operations and in the process of dealing with those file operations it uses methods that are available as a part of this class. What would be sequencing constraints that you will write for such kind of code? The obvious kinds of sequencing constraints are as follows.

(Refer Slide Time: 11:55)



Class FileADT: Sequencing constraints

- An `open(f)` *must* be executed before every `write(t)`.
- An `open(f)` *must* be executed before every `close()`.
- A `write(f)` may not be executed after a `close()` unless there is an `open(f)` in between.
- A `write(t)` *should* be executed before every `close()`.

Note: Violation of a constraint with a *must* is a fault. Violation of a constraint with a *should* indicates a potential fault.

Page 9 of 17

A file must be opened before you can write on to it, an open file must be closed before you finish operations, better not leave it open. A write may not be executed after a close unless there is an open in between. So, what it says is suppose you open a file and then you write and then you close the file and then you have to open it once again to be able to write. So, once you close the file you cannot write on to it without opening it again and then the fourth sequencing constraint says that a write should be executed before every close.

So, what is the fourth one say? Fourth one says write should be executed before every close. So suppose this is violated then what will happen then it means that a piece of software is trying to do some dummy operations on a file, it could just open a file close a file, open a file, close a file without doing anything. Even it does not read from a file, it does not write on to a file. So, it is just keeping itself busy, but doing nothing. So, we would want to prevent such things. So, we say that once a file is open do something with it, do something like a write with it before you close it.

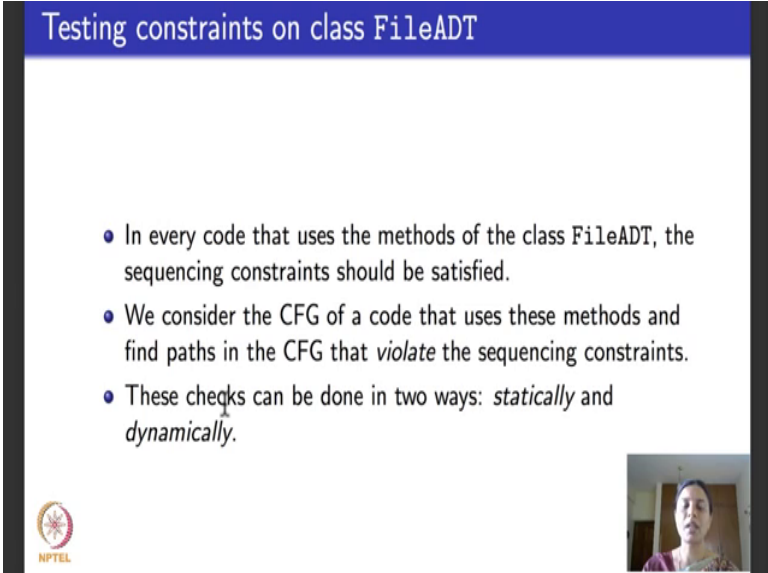
The other thing that you must notice in this slide is that I have put words like *must* in the first 2 constraints and the word *should* in the fourth constraint in italic font, what do we mean by that? Typically when we write requirements for specifications or constraints you will encounter these words a lot when you write requirements in English: 'may', 'must', 'shall', 'should' and what do they usually mean? Usually the presence of a word

must means that if that requirement or constraint is violated then there is a fault for sure and the word like 'should' should be interpreted as if that requirement or constraint is violated then there might be a fault. So, it is clear even for this example that something like that holds.

Suppose, let us say the second requirement consider second constraint it says open f must be executed before every close. So, suppose this is violated which means what somebody has tried to close a file that was not even open, that is not possible so it will definitely result in a fault and it is a must sequencing constraint. Look at the fourth sequencing constraint, it has the word 'should', should means that somebody is trying to write on to a file I mean somebody is opened the file and closed the file without writing on to it. It is harmless in the sense that it does not really cause the serious fault, but it is useless also. So, a requirement with a 'should' may or may not result in a fault. So, we have to be careful about when we use must and should kind of requirements, these are typically done by requirement engineers who write software requirement or by architectures specify the design.

Now, I have class that contains the three methods.

(Refer Slide Time: 15:09)



The slide has a blue header with the text "Testing constraints on class FileADT". Below the header, there are three bullet points:

- In every code that uses the methods of the class FileADT, the sequencing constraints should be satisfied.
- We consider the CFG of a code that uses these methods and find paths in the CFG that *violate* the sequencing constraints.
- These checks can be done in two ways: *statically* and *dynamically*.

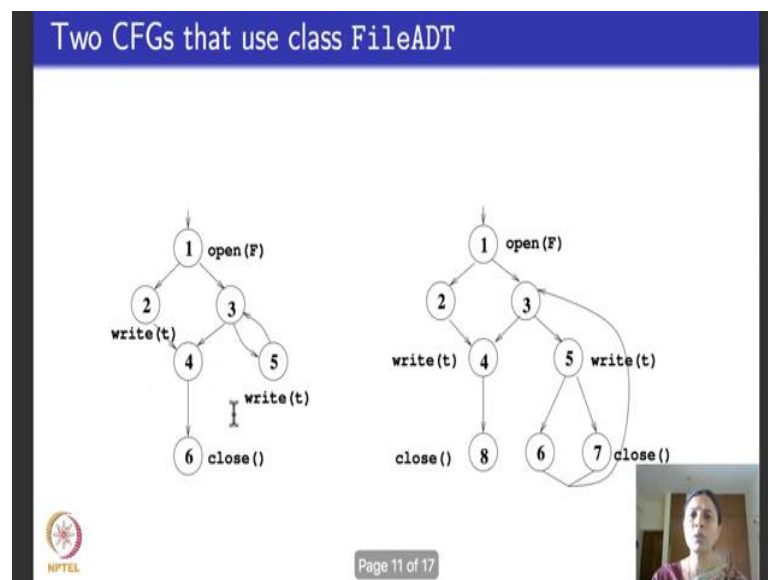
In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a person speaking.

These are the sequencing constraints on the method. Now what I want to do is the following. Consider a piece of code that uses methods from this class, then that piece of code should satisfy all the sequencing constraints that we have written right, which

means it must open a file before it writes on to it and promptly close the file and it must not ideally open and close a file without doing a write operation in between. Those are the kind of things that we do. So, what we do is that we consider the code that uses the methods from this class you consider the control flow graph of that code, then what will be your test requirement and test paths? They would be test paths in the control flow graph that check if any of these constraints are violated.

Typically if sequencing constraints there are two ways of doing it, you can try to violate the sequencing constraints in a static way or you could try to violate the sequencing constraints in a dynamic way.

(Refer Slide Time: 16:08)



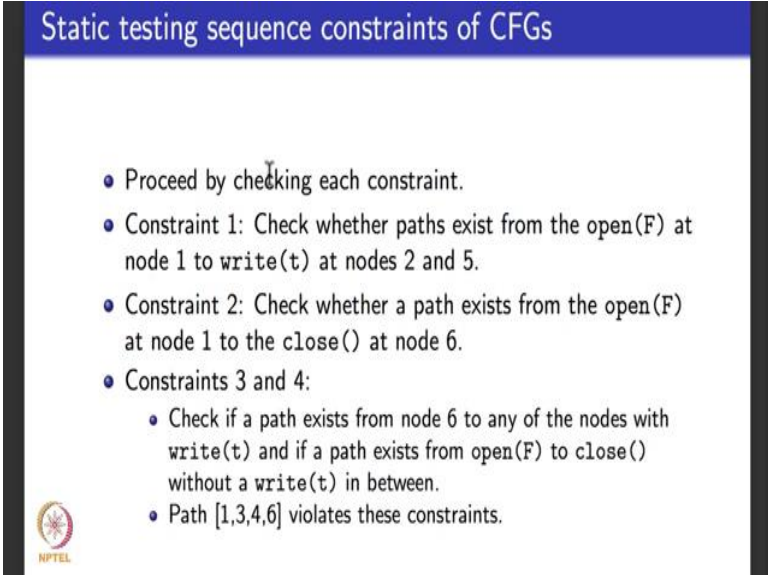
So, we first look at static way. So, before we move on to that here are 2 examples of 2 different pieces of code that use the methods of the class file ADT. On the left hand side it is the CFG corresponding to one piece of code that uses methods in a particular way. On the right hand side this CFG corresponding to another piece of code that uses a method in another particular way. So, let us focus on the CFG in the left hand side at node 1 it calls the method open F which opens the file F and node 2 it writes data on to that file, at node 3 it goes into a loop where it repeatedly writes data to the file several times and then when it comes out it closes the file. This is the control flow graph corresponding to some code that uses these methods. I have again abstracted out and

given you only the details of the CFG as relevant to this lecture, in between it could do several other operations we have not depicted that in the CFG.

So, here is a CFG of another graph that uses the same methods from the class file ADT. Always it begins with open; please remember that unless it begins with open if it directly does a write or a close that will violate the sequencing, one of sequencing constraints. So, it always begins with the open and then like the earlier code, it can do one write and close the file or it can write and close several times in a loop which is this right hand side 3, 5, 7 and get back.

So, now what we will do is that if this is the first CFG of some code, this is a second CFG of some of the code, we will take these sequencing constraints and see whether these two pieces of code or CFGs that uses these methods do they satisfy the sequencing constraints or do they violate the sequencing constraints? As I told you this can be done in two ways: statically or dynamically. What I mean by statically? Statically means I do not execute the code. I consider the CFG as a graph and then try to see if sequencing constraints are violated. Typically static analysis is not considered a path of testing, but I will just tell you for the sake of completeness here.

(Refer Slide Time: 18:34)



Static testing sequence constraints of CFGs

- Proceed by checking each constraint.
- Constraint 1: Check whether paths exist from the open(F) at node 1 to write(t) at nodes 2 and 5.
- Constraint 2: Check whether a path exists from the open(F) at node 1 to the close() at node 6.
- Constraints 3 and 4:
 - Check if a path exists from node 6 to any of the nodes with write(t) and if a path exists from open(F) to close() without a write(t) in between.
 - Path [1,3,4,6] violates these constraints.

NPTEL

So, how the static testing of sequence constraints a CFG work? It proceeds by checking one constraint after the other. Go back to the constraint slide, there are four constraints - first says open must be executed before write, second says open must be executed before

close, third says write cannot be executed after the close unless there is an open in between, fourth says that write should be executed before every close.

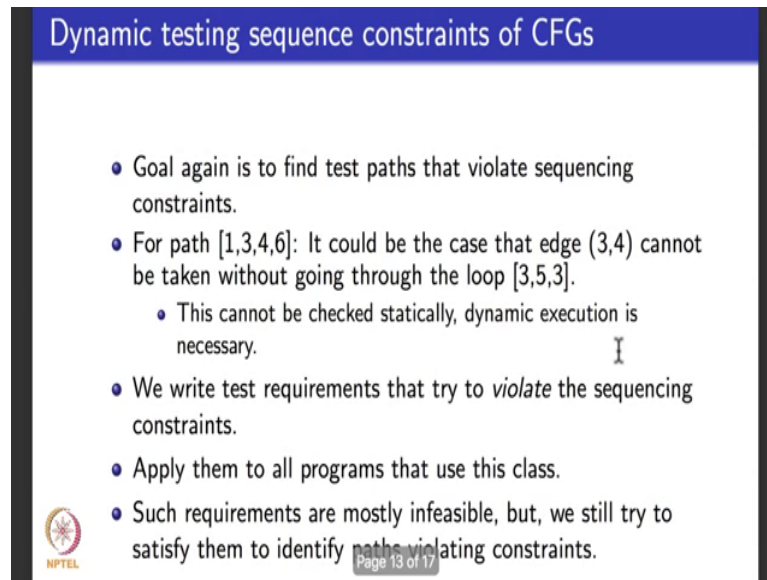
So for constraint one, we check if there is a path from an open to a write, where is open here I am focusing only on the CFG on the left hand side for the purposes of this, open is at node 1, there are writes at node 2 and node 5. So, for the first constraint I check whether there is a path from the open at node 1 to writes at node 2 and node 5. It so happens then there are, so we doing fine as far as the first sequencing constraint is concerned.

For the second one similarly you check whether there is a path from an open at node 1 to the close and node 6. It so happens that there are, if you see 1, 2, 4, 6 is one path, 1 2 3 4 is one path, 1, 2, 3, 5, 3, 4, 6, this one path. So, there are paths, so as far as the second requirement open must be executed before every close I am still doing fine.

For constraints 3 and 4, I have to check if there is a path from node 6 to any of the nodes with a write and if there is a path from open to close without write in between. So, there is a path from node 6 which is a close to any of the writes. So, if you see there is a write that happens at node 5 before close, there is a write that happens at node 2 before the close. But if you see there is a path from open at node 1 to close at node 6 without a write. Which is that path? That path goes through node 3, I open the file I go to node 3, I do not do anything specifically I do not do a write then I go to four there I skipped the loop that is and then I close. So, I have open the file and close the file without doing a write in between. So, which is the constraint that it violates? It violates the fourth constraint, it violates the constraint that the write should have been executed before a close, that is what is given here. Path 1, 3, 4, 6 violates the constraint.

Please note that in this slide all the constraints we are reasoning about with reference to this CFG, the CFG on the left hand side. The CFG example of the right hand side I will use it when I do dynamic testing.

(Refer Slide Time: 21:10)



Dynamic testing sequence constraints of CFGs

- Goal again is to find test paths that violate sequencing constraints.
- For path [1,3,4,6]: It could be the case that edge (3,4) cannot be taken without going through the loop [3,5,3].
 - This cannot be checked statically, dynamic execution is necessary.
- We write test requirements that try to *violate* the sequencing constraints.
- Apply them to all programs that use this class.
- Such requirements are mostly infeasible, but, we still try to satisfy them to identify ~~paths violating~~ constraints.

HPTEL

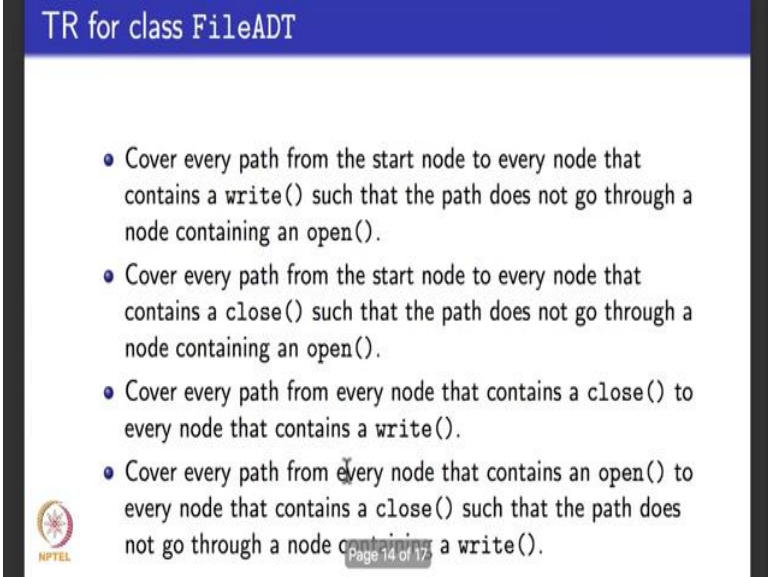
Page 13 of 17

Now when it comes to dynamic testing my goal is somehow to be able to find paths that violate sequencing constraints. So, I write test requirements or TRs that tell you what to cover. So, I will say cover for sequencing constraint, second sequencing constraint and so on for each of the sequencing constraint and my goal is to find test paths that violate the satisfaction of the sequencing constraint, right. So, if I consider dynamic testing and I go back to the first CFG the path 1, 3, 4, 6 which potentially violated constraint 4 could not be realistic.

If you remember I told you whenever we look at control flow graphs, it many times may not be feasible to write a path that all together skips a loop. 1, 3, 4, 6 in this CFG is such a path. It could be the case the loop was a do while loop and you are forced to execute one iteration of the loop at least to be able to do. So, suppose you have that then when you do dynamic testing you will know about it because you actually execute the software and then path 1, 3, 4, 6, might not be an actual violation of a sequencing constraint, but in static testing you will flag it as a potential violation of the fourth sequencing constraint, but when you do dynamic testing you will know whether it was actually a false positive or it is a genuine violation of a sequencing constraint. It will be a false positive if the code for the CFG forces you to enter the loop it will not be a genuine violation it would be a false positive, but if the code allows you to skip the loop then it will be a genuine violation of the fourth constraint. You can find out this through dynamic testing.

So, when the other thing that we do in dynamic testing as I told you is to write test requirements that violate the sequencing constraint and apply these test requirements to all the programs that use the methods from this class right. Typically for large programs is actually quite difficult to write all the sequencing constraints, to write the TRs for each of the sequencing constraints and to be able to get tests pass and its usually undecidable, but we will not really worry about proving which are the programs for which it is decidable or undecidable, the idea is to be able to understand how it is done.

(Refer Slide Time: 23:41)



TR for class FileADT

- Cover every path from the start node to every node that contains a write() such that the path does not go through a node containing an open().
- Cover every path from the start node to every node that contains a close() such that the path does not go through a node containing an open().
- Cover every path from every node that contains a close() to every node that contains a write().
- Cover every path from every node that contains an open() to every node that contains a close() such that the path does not go through a node containing a write().

Page 14 of 17

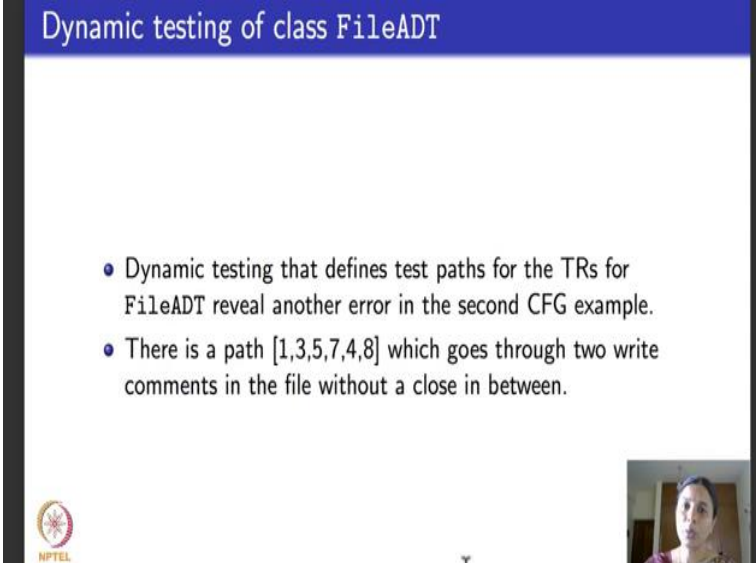
So, what I do now is I write test requirements for all the constraints this one TR for every constraint, there were four constraints so there are four TRs. First one says that you cover every path from the initial node to every node that contains the write such that the path does not go through an open, this will violate the first constraint right. So, it says you cover every path that goes through a write, but without going through an open. Let us go back and see what constraint one said open must be executed before every write. Suppose you are able to find a path that will meet this TR then you have violated the first constraint.

Similarly, for second constraint, second constraint says that before a close a file must be open. So, the test requirement to violate the second constraint will say you cover every path from the start node to every node that contains the close command call to the close method such that the path does not go through an open method. Suppose you find the test

path you have successfully found a test path that violates the second constraint. Similarly for the third sequencing constraint your TR says you cover every path from every node that contains a close to the path the node that contains a write which means the write is happened after a close. Fourth TR says cover every path from the node that contains an open to a close without going through a write.

I have basically written the same sequencing constraints, but in terms of coverage criteria that violate the sequencing constraints, I hope this is clear.

(Refer Slide Time: 24:25)



The slide has a blue header with the text "Dynamic testing of class FileADT". Below the header, there are two bullet points:

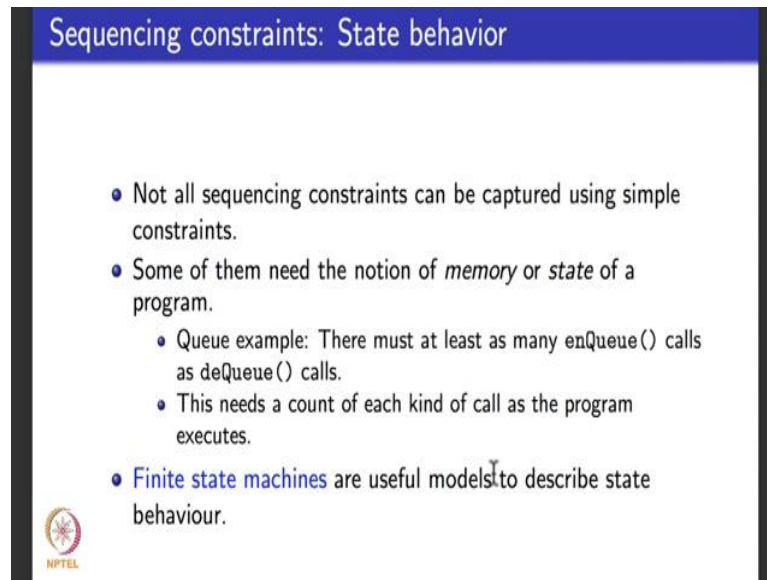
- Dynamic testing that defines test paths for the TRs for FileADT reveal another error in the second CFG example.
- There is a path [1,3,5,7,4,8] which goes through two write comments in the file without a close in between.

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

So, if I do dynamic testing I will find that again the fourth constraint is violated. So, in this case we will look at the second CFG this one. How is the fourth constraint violated? There is this path, 1, 3, 5, 6, 3 which tries to write and not close a file. It tries to write once, goes back to 3 tries to write once more and does not close the file and after that only closes a file.


So, through dynamic testing I will be able to find out whether there is a violation of the sequencing constraint or not.

(Refer Slide Time: 26:08)



Sequencing constraints: State behavior

- Not all sequencing constraints can be captured using simple constraints.
- Some of them need the notion of *memory* or *state* of a program.
 - Queue example: There must at least as many enqueue() calls as dequeue() calls.
 - This needs a count of each kind of call as the program executes.
- Finite state machines are useful models to describe state behaviour.



So, in summary, what do I do? I identify sequencing constraints if they have not been documented by designers. What is my test requirement? I write test paths that try to cover in the test cases in such a way that if my test path is found then one of the sequencing constraints is violated. So, I write one test coverage criteria for violating each of the sequencing constraints and suppose I identify in a test path that achieves this coverage criteria then I have successfully found a path in the graph that violates the sequencing constraint which leads to an error.

Sequencing constraints can be of two types: the first type this plain example like we saw for the queue example or the file open, file close example. Sometimes you can have sequencing constraints that need to count the numbers that happened in the past, the number of certain kinds of operations that happened in the past. Those are called state full sequencing constraints. And in the next module will introduce finite state machines or finite state automata with which we can work to model state behavior related sequencing constraints.

Thank you.