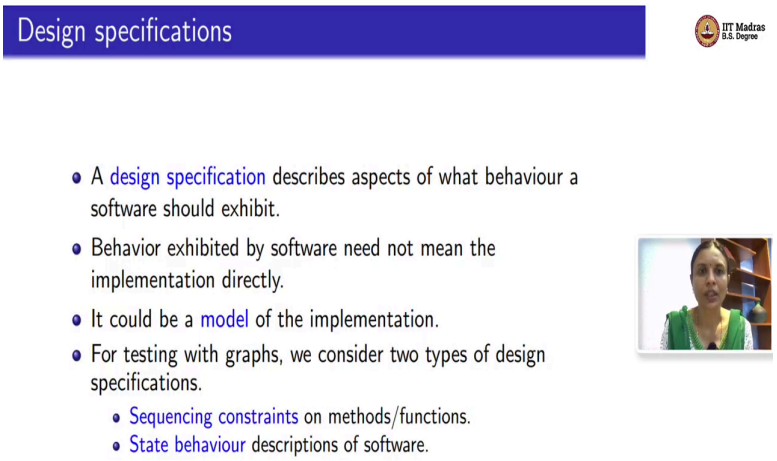
Software Testing
Professor Meenakshi D'Souza
Department of Computer Science and Engineering
Indian Institute of Technology, Bangalore
Specification Testing and Graph Coverage

Hello there. Welcome back. We will continue with graph based testing. We are done with using graphs for testing code. We saw at the unit level structure and coverage and then structural coverage criteria that a CFGs augmented with data definitions and uses. Then we also saw design integration, how to put together the modules, and call graphs were testing for call interfaces. What other graphs exists in software artefacts, assuming that we still want to do graph based testing, how can I continue to use graphs to test other kinds of artefacts?

We will see two more artefacts this week. One is what we generate from software specifications. I hope you remember by now what a specification is. So, we will see what kind of graphs could be generated from certain kinds of specifications and how can we use what we learned till now to be able to test those specification using graphs.

(Refer Slide Time: 01:18)

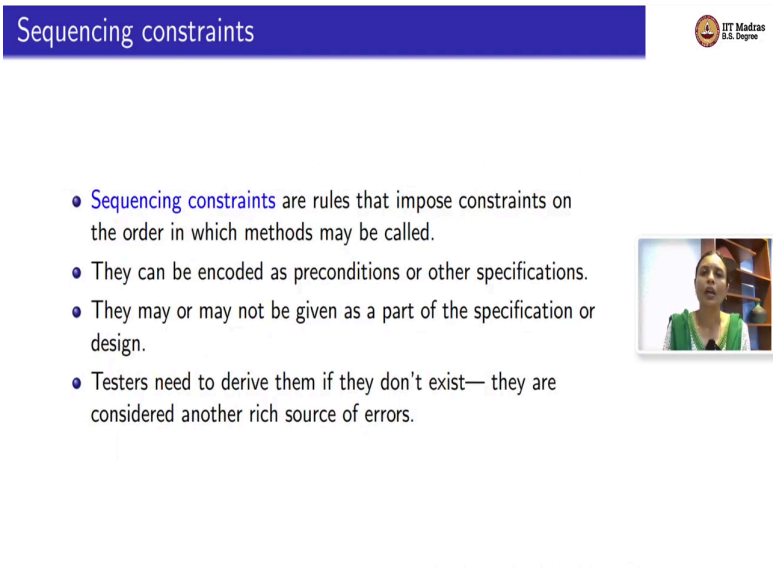


A design specification is what we are going to concentrate today in this lecture. It basically specifies what aspect or behaviour of a software should be subject to testing. Like for example, suppose you use date as one of the inputs to your software. It is taken for granted that you accept only valid dates. So, as a part of your software, you would have developed a written code which involves routines for validating a date.

Similarly, if you are using let us say, a data structure like a Q 2 as a part of your software development activity, then you would have methods like enQueue, deQueue that act as a part of your data structure. So, these constitute design entities and design specification tells that we have to test to check whether these things also exhibit correctly or not. So, behaviour can be exhibited by software. It may not mean the implementation. It could be parts or features of the implementation that we take for granted, which need not be a part of the main specification.

So, what kind of graphs are we going to generate? We are going to consider two types of design specifications. One is what we call sequencing constraints, on methods or functions or procedure calls. That is what we are going to see today. The next is certain sequencing constraints need information about what we call as a state. So, that is the next module where we talk about state behaviour based sequencing constraints of software.

(Refer Slide Time: 02:58)

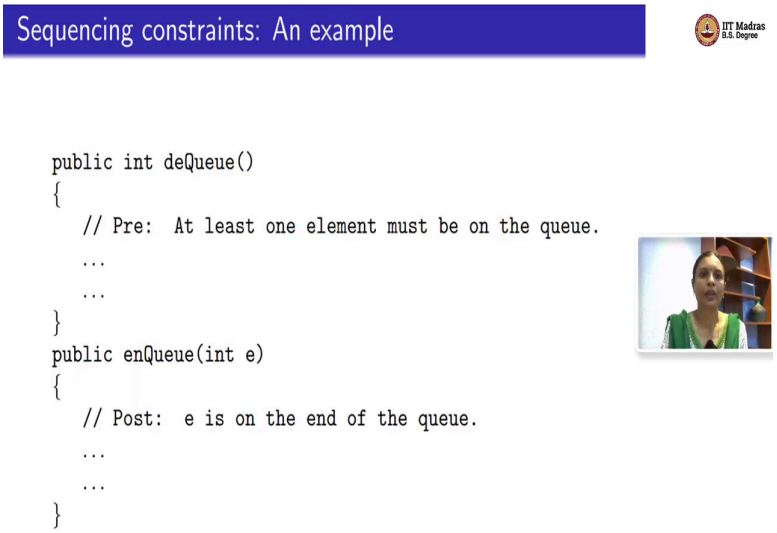


First part about sequencing constraints, what are they? They are basically a set of rules or conditions that impose the order in which some conditions on the order in which methods of a particular class may be called to implement a certain functionality. Typically, they are encoded as preconditions or as post conditions or could even be invariant conditions as a part of the software artefact. Sometimes they are given. Sometimes they may not be given. As I told you suppose we have a Q data structure as a part of our design, then certain methods like enQueue, deQueue come with that data type.

But for me to be able to deQueue the queue must be non-empty. This is a constraint but it may not be explicitly given. If it is not given then it is my duty as a tester to be able to infer or derive these constraints and test for them because they can be a rich source of errors. Like I told you earlier, suppose your application uses a routine date validation routine. Then, if you correct date should include dates and leap years and should not have 31 days in a month like November, things like that. And the year should be valid. Month cannot come with a number 13 or day of the month cannot be a number like 52. These are basically the checks that you do for a correct date.

Date validation routine should be a sequencing constraint or an implicit requirement that a tester should derive to be able to test the software artefact. Specifically, for the date validation routine, we may not be needing graphs, so I would not consider that example now. We will look at other examples where we can model sequencing constraints which are basically requirements on the sequence of method calls. How are we going to model them as graphs and use graph based testing to validate them?

(Refer Slide Time: 05:01)



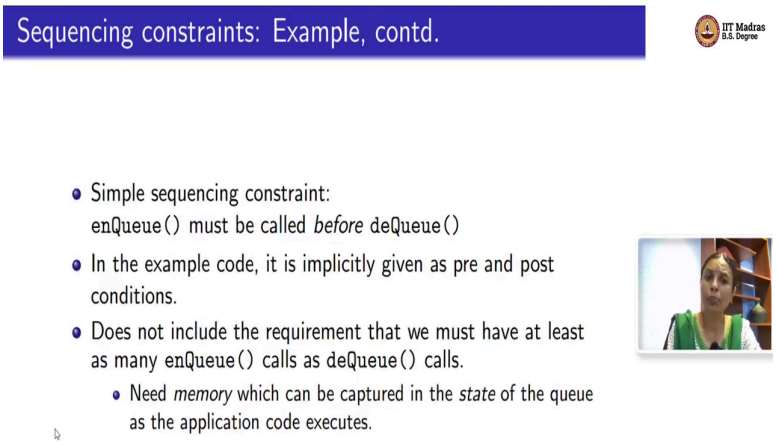
So, this is the queue example that I had told you just a few seconds ago. Let us say you had a method called deQueue, which basically removes the element at the top of the queue. At the end of the queue. There is one method call enQueue, which basically inserts this element e into the queue at the end of the queue. So, these two are two methods where any software that uses a queue are going to come with these two methods.

So, a precondition could be for de-queuing, that is for removing an element from the queue, a precondition could be at least one element must be present in the queue. Otherwise, what do I remove from the queue, there is nothing to remove. Date is dot dot dot that some method of the code that implements deQueue. That is not the focus for now. The focus is on deriving graphs for these kinds of preconditions post conditions.

Similarly, for enQueue, a post condition must be first to remove the element e from the end of the queue, then e better be at the correct end of the queue, V is somewhere in the middle of the queue or the second or third element in the queue, then by the way, Q works which is fee for order, I may not be able to dequeuer e or I may not be able to add e at the correct. Suppose condition for this would be after I have added it to the queue, it must be at the end of the queue.

So, I would like to keep in mind these preconditions and post conditions which are properties that enQueue and deQueue method should, if an application uses them, these are properties that they must be insured. And I would like to derive graphs to ensure these kinds of properties.

(Refer Slide Time: 06:40)



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.
   ...
   ...
}
```

A Simple Sequencing constraint is this is what I told you. Before the first deQueue is called there must be at least one enQueue operation. Something must be put inside the queue before I can remove things from the queue. So, enQueue must be called before deQueue. This is implicitly given as preconditions and post conditions in a code snippet like this, but we have to test for it.

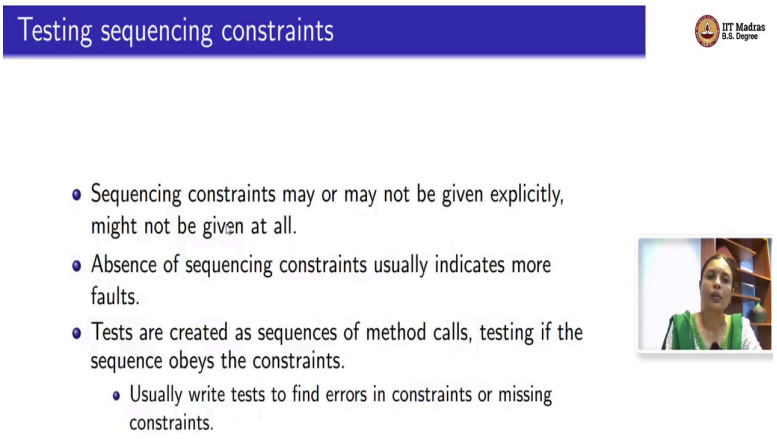
In fact, what is not given here something that we will have to test for, which we will do in the next lecture would be this also. If you see any application that uses these methods, some instances of deQueue some instances of enqueuer. deQueue removes from the end of the queue enQueue adds to the other end.

At any point in time, I should have added enough elements for me to remove that many elements. So, this kind of requirement that we must have at least as many enQueue calls as deQueue calls is an important part of sequencing constraint. Any application that uses enQueue and deQueue, there must be enough enQueues done for me to be able to do the deQueues. Clearly, if I have deQueue is number of deQueue is more than the number of enQueues, then the queue would, at some point leave an error message because I would be deQueueing from an empty queue.

So, for this, you see across all possible behaviours, the number of enQueue calls must be at least as many as the number of deQueue calls involves counting and remembering these numbers. So, this involves a notion of memory, whereas this is not the first one here. enQueue must be called before deQueue does not involve too much memory. So, this can be captured as a part of a state of a system. And as long as the queue has bounded size, the state

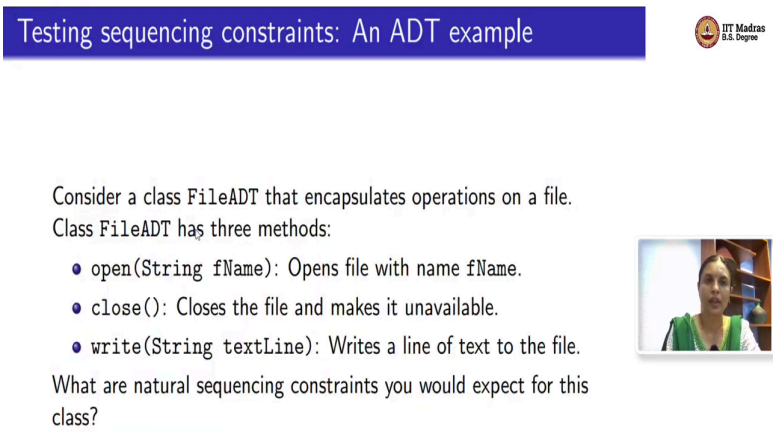
can be finite and such testing for such a sequencing constraint we will do in the next module. For now, we will check for sequencing constraints like these.

(Refer Slide Time: 08:40)



The first one. So, as I told you sequencing, constraints may not be given. If they are not given, then as testers, we have to derive them because they can be a rich source of faults.

(Refer Slide Time: 08:50)



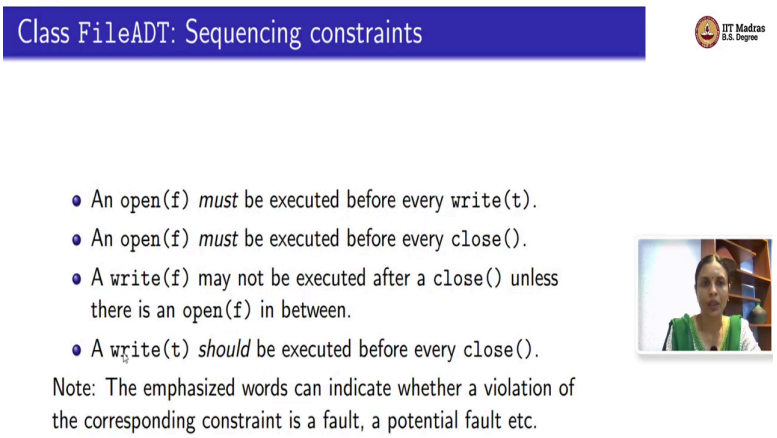
So, before we move on, here is another example. Let us say you have a class called file ADT, file abstract data type, which is a class of methods that manipulates files does operations on a file. So, let us say for now, as a simplistic case, this class has three methods. So, you could open a file with the name f Name, you could close a file that is already open, which makes it

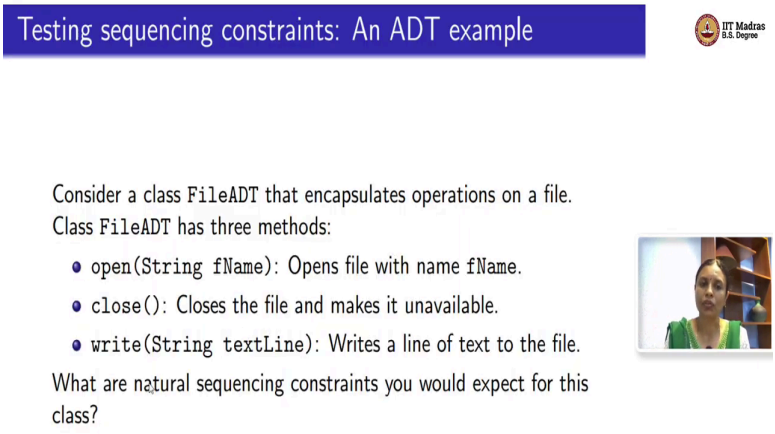
unavailable for use by the application anymore. Or you could add a line of text to the file that is already present through the right method.

So, there is a class called file ADT that has several methods that let you do operations on a file. Here are three sample methods. Open a file with a particular name. Here, in this case, f Name. Close an already open file. There is no exists argument here because it closed the file that is currently open. You could also write to the file, write a line to the end of the file. So, before we move on, maybe you should spend a few seconds understanding what would be natural sequencing constraints that you will expect for a class like this.

Suppose an application uses these three methods. What would be some natural sequencing constraints that will come?

(Refer Slide Time: 10:10)





So, here are some examples. First one is before I can write a line of text to the file, I must at least open the file. So, that is the first sequencing constraint. An open f must be executed before every write. The next one, an open f must be executed before every close. If a file is not even open, what can I close? I cannot close anything. So, if I call the close method, it will pop an error saying no file is open. So, some file must have been open for me to call the close method.

Similarly, this is another useful sequencing constraint. The third one, what is it say? It says a write may not be executed after a close unless there is an open in between. So, I have opened a file for some reason I did some operations, then we closed it. And then I want to execute a write. But I cannot do that because the file is already closed. It is not open for me to write. So, that is what this constraints is that arise when cannot be executed after a close unless there is an open in between.

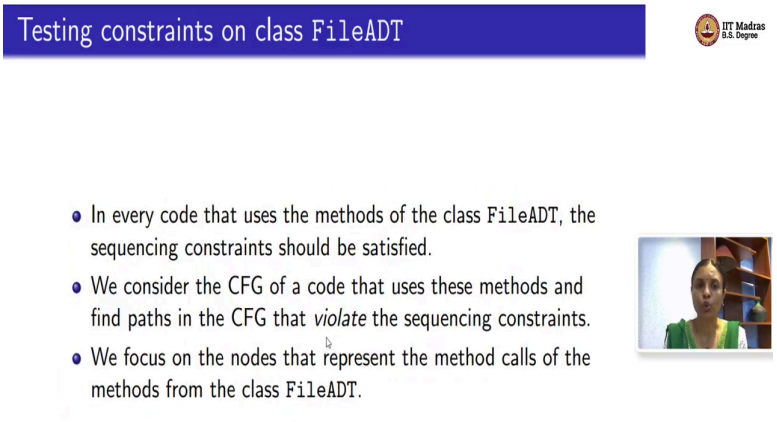
Similarly, the last one says, you cannot do dummy open close open close open close kind of operations. If I open a file, then I must do something like a write maybe or a read, I must do something with it before I close it. So, here it says write should be executed before every close. This is a sample. One can think of a whole set of other similar sequencing constraints. Now if you see in the sequencing constraints, I have written these kinds of words must, should, and all emphasized in italic text emphasized.

So, basically, what I wanted to say is that when you look at professional requirements written as a part of your let us say, job, when you look at a requirements document, you will have lots of these kinds of words used. And each organization will have their own interpretation of

what such a word means. For example, it could mean that if there is a presence of a word like must, it could mean that if it is violated, then there is definitely a fault. If there is a presence of a word like should and if it is violated, then it could mean that it is a potential fault. These interpretations can differ based on the software development organization.

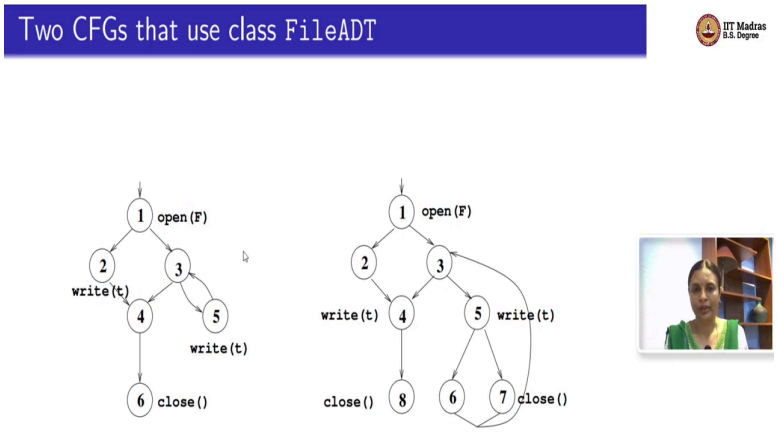
But as testers, when we read sequencing constraints like this, we should be careful and be aware of what do the meaning of these kinds of words mean? That is all I am saying. So, now, just going back to this example, there is a class file abstract data type. It has a whole set of methods. And let us say an application uses this methods to get some work done. And we have some sequencing constraints on the methods.

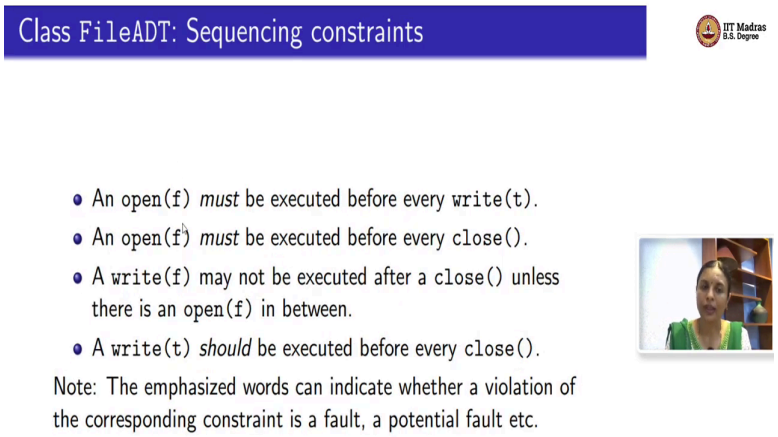
(Refer Slide Time: 13:04)



So, what I am going to show you now is, let us take the application that uses this method. Let us look at the control flow graph of that application. Along the way, in the control flow graph, you will have method calls nodes representing method calls that could call the methods file open or close or write. On those nodes in the underlying control flow graph, we want to enforce these kinds of sequencing constraints.

(Refer Slide Time: 13:32)





So, here are two such examples. So, these think of these as two control flow graphs representing two different applications. That happened to use methods from the class file ADT. In the first control flow graph given on the left, there are some nodes that are marked with this. So, at node 1, the application uses this call open file F. At node 2, the application writes something to the file that is F that is open. Node 3 is not labelled. It could be something but for now, it does not involve any method call from the class file ADT. And our goal is to check sequence in constraints on the file ADT.

So, I want the methods or the class file ADT. So, I am going to focus on the corresponding label nodes only. So, the rest of the nodes like nodes 3 and 4, which are unlabeled, execute

some piece of the application code which is not my current focus, so I am not considering those labels. Node 5 calls the method write, node 6 closes.

On the right, you have another control flow graph representing a second application. So, this control flow graph again in the beginning opens a file seems like a correct thing to do. And 2 and 3 are somethings not method calls of this class file ADT. So, we are not label them. 4 involves write, 8 involves close, 5 involves a call to the method write, 7 involves close. So, what I want you to do is to think for a couple of minutes, keep these two CFGs in mind one at a time, and look at these sequencing constraints. And see if any of these four sequencing constraints is violated by any of the two CFGs.

For example, let us look at the control flow graph on the left. At node one, the method open is called, let us say, the execution path that this application takes is 1, 3, does not enter the loop between 3 and 5. It skips the loop, goes to 4 and then goes to 6. So, there is a path from open to close, without a write in between. 1, 3, 4, 6 is apart from opening a file to closing the file without doing anything with the file. So, it is violating this last sequencing constraint that you see, which says the write should be executed before every close.

Similarly, let us look at this control flow graph on the right. So, here, what has happened is, suppose I traced this path 1 to 3 to 5 to 7, what is happening here at 1, the file is open. 3 maybe something else is happening. At line 5 or node 5, we write something to the file. And at 7, I close it. So, far, so good. Now from 7, there is a back edge to 3. So, I can from 3 come back to 5.

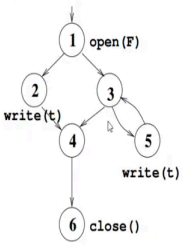
Now you see that there is a problem, because I had closed the file at 7. Now I go from 7 to 3 and then you take the edge from 3 to 5, I am attempting to write on a file that is already closed, that violates the sequencing constraint, third one. A write f may not be executed after a close unless there is an open in between. So, this explicitly violates the sequencing constraint. So, this is the goal of testing for sequencing constraints using graphs.

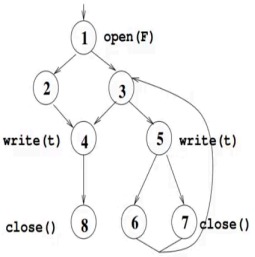
(Refer Slide Time: 17:23)

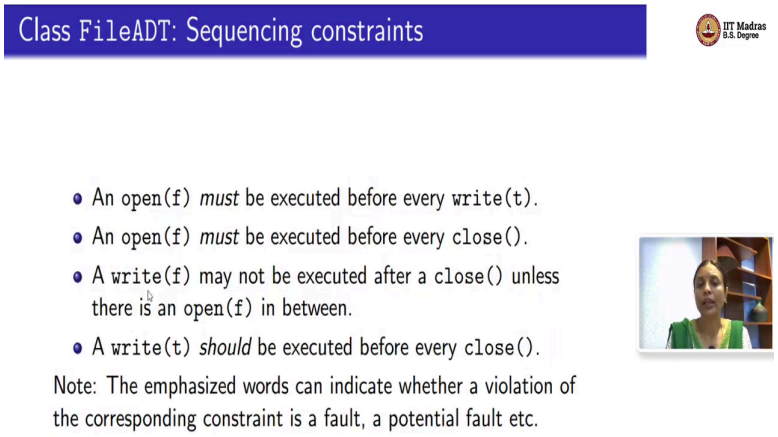
Static testing sequence constraints of CFGs

- 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.

Two CFGs that use class `FileADT`







So, we proceed by checking one constraint at a time, we first generate control flow graphs, focus or label the nodes that involve these method calls. And per sequencing constraint like this, we generate one test requirement. And our goal is to check whether that test requirement is satisfied or violated. It satisfied if all the control flow paths satisfy the test requirement or the constraint is violated if I can find one control flow path that violates this test requirement.

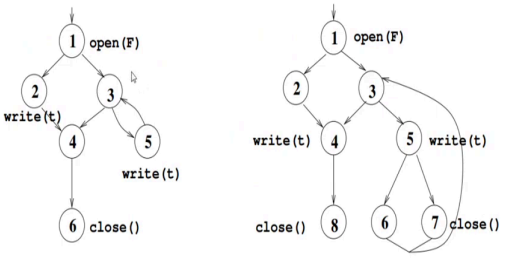
So, I proceed by checking each constraint. For constraint 1, we check whether there is a path from open F at node 1 to a write at nodes 2 and 5 that is on this control flow graph. Similarly, for constraint 2, I can check whether there is a path from open F at node 1 to a close at node 6. Similarly, for constraints 3 and 4, I can check and I realize that path 1, 3, 4, 6. This is what I told you earlier 1, 3, 4, 6 violates this constraint. This we can check statically. Once you have the control flow graph like this, you can check statically.

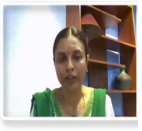
(Refer Slide Time: 18:41)

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.

Two CFGs that use class FileADT



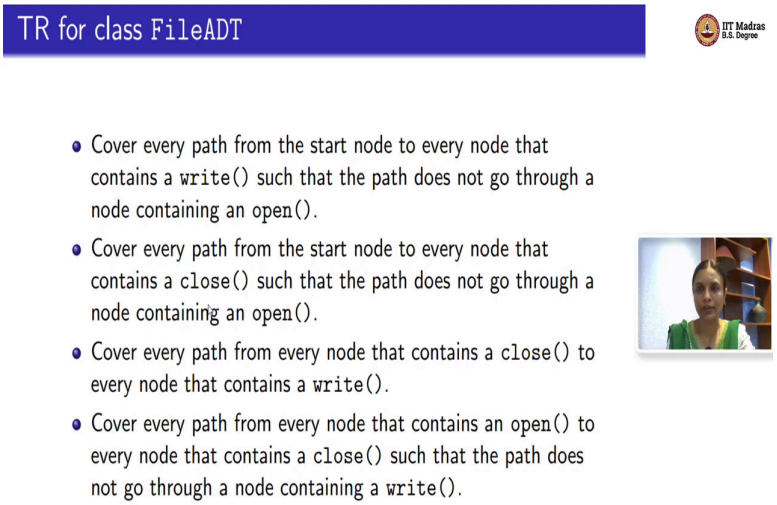


Some of them you have to check dynamically. And then the goal is again to find test parts that will violate sequencing constraints. For path 1, 3, 4, 6, which violated that sequencing constraint. It involves skipping this loop, if you remember skipping the loop 3, 5. Maybe the application is such that it Is impossible to skip the loop. So, you will end up doing a write if you go through the loop, only dynamic testing will find out that. For the path 1, 3, 4, 6 as I told you, it could be the case that the edge 3, 4 cannot be taken without going through the loop 3, 5, 3. This cannot be statically checked. You need to execute the application to ascertain something like this.

So, we basically write one or more test requirements per sequencing constraint and either statically or dynamically try to find if there is one execution of the application where the sequencing constraint gets violated. And then we apply it to all the applications or

programmes that use the methods from this class. This is what we do in sequencing constrained testing.

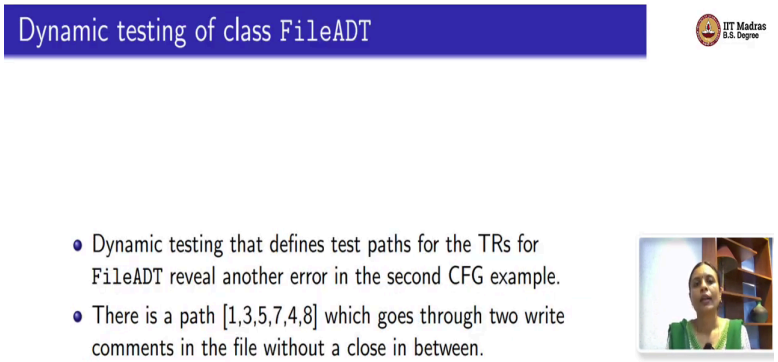
(Refer Slide Time: 19:52)



So, for the class example that we saw class file abstract data type, these would be the test set total set of tests requirements. We cover every path from start node to every node that contains a write such that the path does not go through an node containing an open. Similarly, we have to cover every path from start node to a node that contains a close, such that the path does not go through a node containing open.

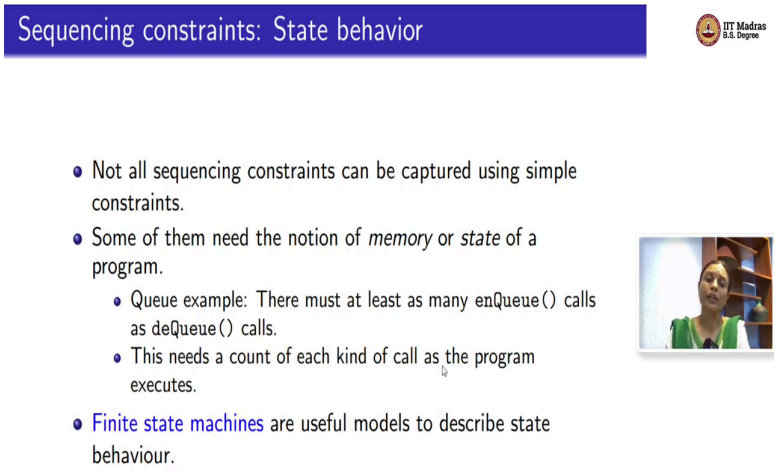
Third, cover every path from every node that contains a close call to a close sorry, to every node that contains a write. Last cover every path from every node that contains an open to every node that contains a close such that along the way, you go through a node that has a call to the write. So, I cannot do dummy open close open close, I should do something useful. And the useful thing that I do along the way could be write.

(Refer Slide Time: 20:53)



Dynamic testing defines these test paths for the test requirements and it can reveal errors. This is one thing that we traced if you remember path, there is a path 1, 3, 5, 7. Go back, which goes through two write comments in the file without a close and visually. So, that was violation of a test requirement.

(Refer Slide Time: 21:15)



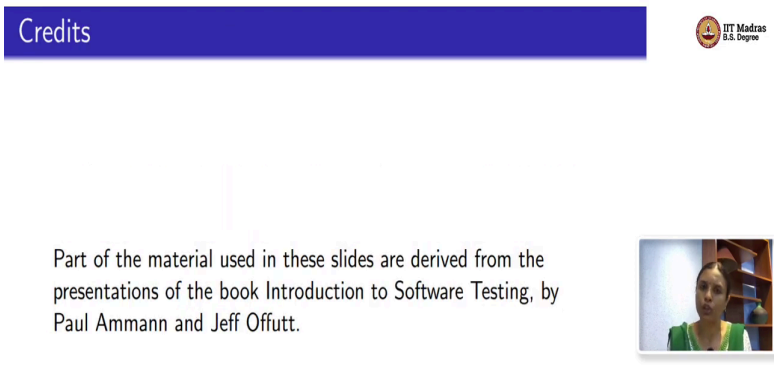
So, the next comes like for example, in the queue this thing, it cannot be a purely static structural path oriented behaviour. You might need information about data. In the queue example it could be information about contents of the queue. And the number of times enQueue operations have happened versus the number of times deQueue operations have

happened for you to say that one number is greater than or equal to the other number. So, not all sequencing constraints can be checked or captured using simple conditions and paths.

Some of them need for you to keep data need to store memory information. And you need to check for that. Like in the queue example, suppose you have to stack check for the sequencing constraint, which says that there must be at least as many enQueue calls as deQueue calls. You need a count of each kind of call as the application executes. For this, I cannot purely work with control flow graphs. I need slightly different graph models, which we call finite state machines.

So, in the next lecture, we will see what a finite state machine is and how I can use it to check for this kind of sequencing constraint.

(Refer Slide Time: 22:31)



So, I will stop here for now. I will see you in the next class. Thank you.