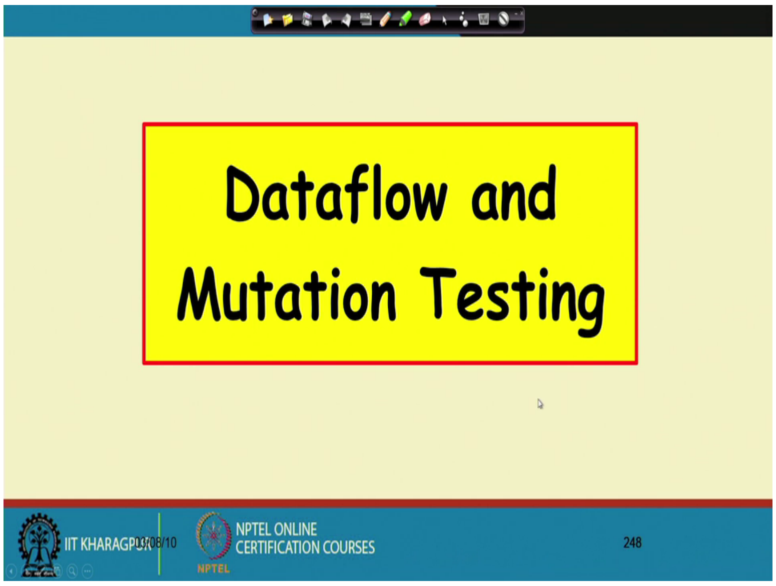
Software Engineering
Prof. Rajib Mall
Department of Computer and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 60
Dataflow and Mutation Testing

Welcome to this lecture. Over the last few lectures, we have looked at various white box testing techniques; specifically, we have looked at the coverage based testing, various types of coverage, various elements. Today, we will look at one more coverage based testing which is Dataflow testing and then, we will look at fault based testing which is Mutation Testing.

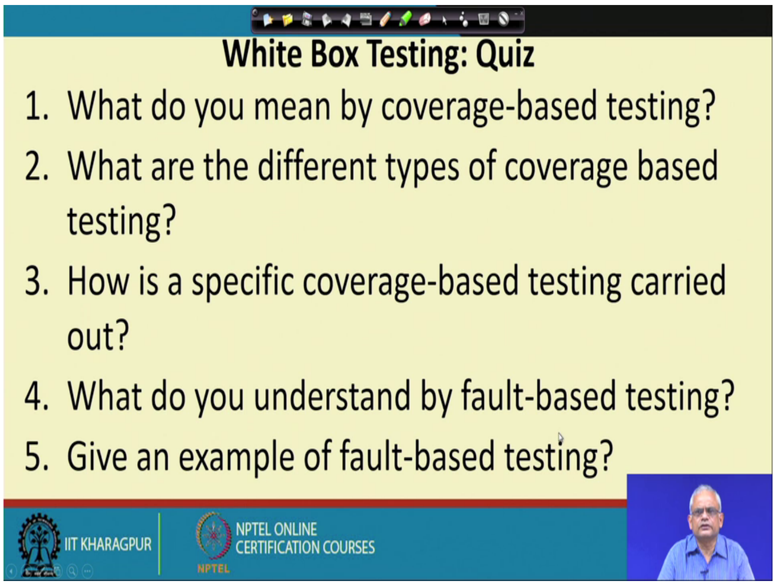
(Refer Slide Time: 00:49)



Dataflow and Mutation Testing

So, first we will look at Dataflow testing and then, will look at Mutation Testing.

(Refer Slide Time: 00:53)



White Box Testing: Quiz

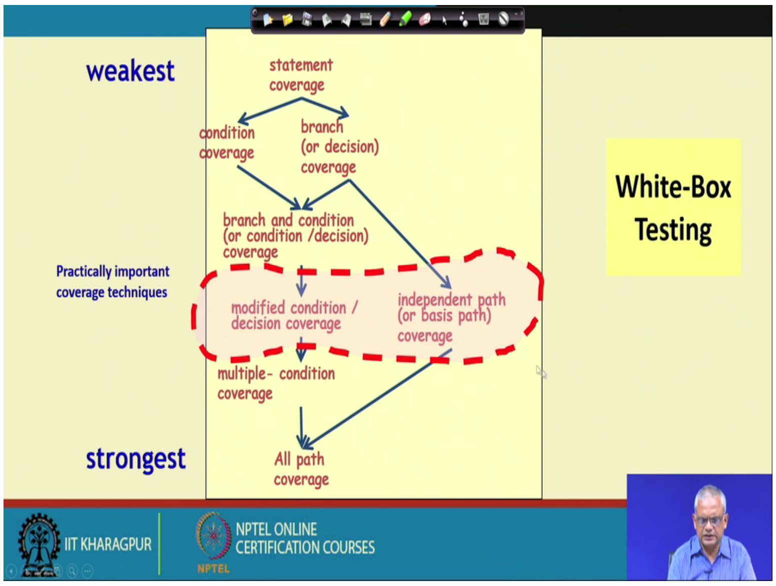
1. What do you mean by coverage-based testing?
2. What are the different types of coverage based testing?
3. How is a specific coverage-based testing carried out?
4. What do you understand by fault-based testing?
5. Give an example of fault-based testing?

But before we get started just a small quiz on White Box Testing. What do you mean by coverage-based testing? As you know by coverage-based testing, we mean that certain elements of a program need to be covered or executed by test case. The elements can be statements, control flows, decisions, conditions and so on. What are the different types of condition based coverage based testing? We had discussed many coverage based testing; starting with the statement coverage, then we had looked at the basic decision coverage, then multiple condition coverage, multiple condition decision coverage, MC-DC coverage and so on.

How is testing carried out in using a specific coverage-based testing? This we had said that you do not design test cases for a coverage based testing. But, you test the program using randomly generated test cases and then, measure the coverage using a dynamic analysis tool and keep on giving test inputs, until you reach the desired coverage value. What do you understand by fault based testing?

Here, we had said that unlike the coverage based testing, here we check whether the test cases are able to detect certain types of faults and for this, we inject fault into the program which we call as a mutated program and then, test using all the test cases and see, if the fault is getting exposed; if the fault is getting exposed by the test cases, then we know that the test cases are ok. But if the test cases are not able to detect the fault, then we know that testing is inadequate and we need to design additional test cases to detect the fault. Give one example of fault-based testing? And we had said that Mutation Testing is a prominent example of fault-based testing and today, we are going to discuss about Mutation Testing.

(Refer Slide Time: 03:50)

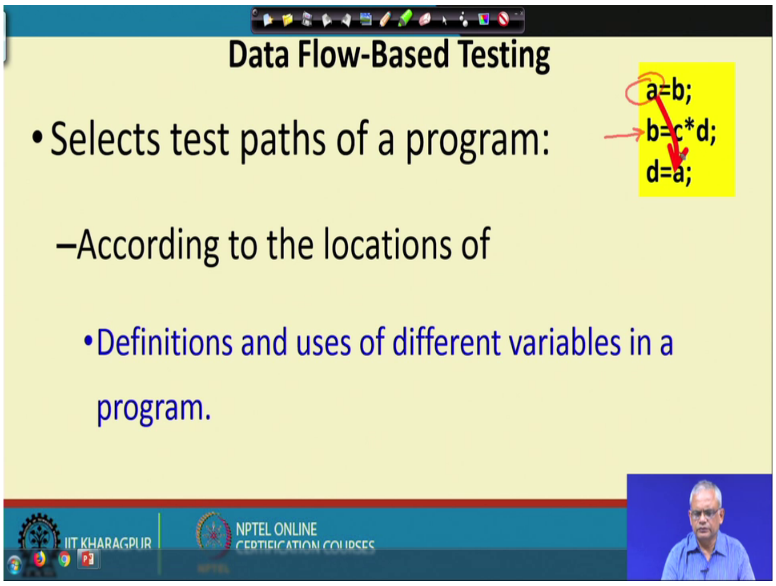

weakest
statement coverage
condition coverage
branch (or decision) coverage
branch and condition (or condition /decision) coverage
Practically important coverage techniques
modified condition / decision coverage
independent path (or basis path) coverage
multiple- condition coverage
strongest
All path coverage
White-Box Testing
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

We had discussed various types of testing. Now, let us see what is the inclusion relation between various types of coverage? The strongest coverage is known by all path coverage and the weakest coverage is known as the statement coverage and then, intermediately, we have condition coverage and then, the branch or decision coverage which are stronger than the statement coverage.

We have condition decision coverage which is stronger than both branch coverage and condition coverage and then, we have MC-DC coverage which is stronger than the branch that is condition decision coverage. And then, we have path coverage, the basis path coverage or independent path coverage and that is stronger than the branch coverage and then we have the multiple condition coverage or the MCC which is stronger than the MC-DC coverage.

But then, what about how do we compare MC-DC coverage with the path coverage? They are not strictly comparable; they are complimentary tests which means that for testing a program, we need to do both MC-DC test and the path test and this is popularly done for any non trivial software, we need to do both path testing and the MC-DC testing. Now, let us first look at the dataflow testing and then, we will look at the mutation testing.

(Refer Slide Time: 05:43)



Data Flow-Based Testing

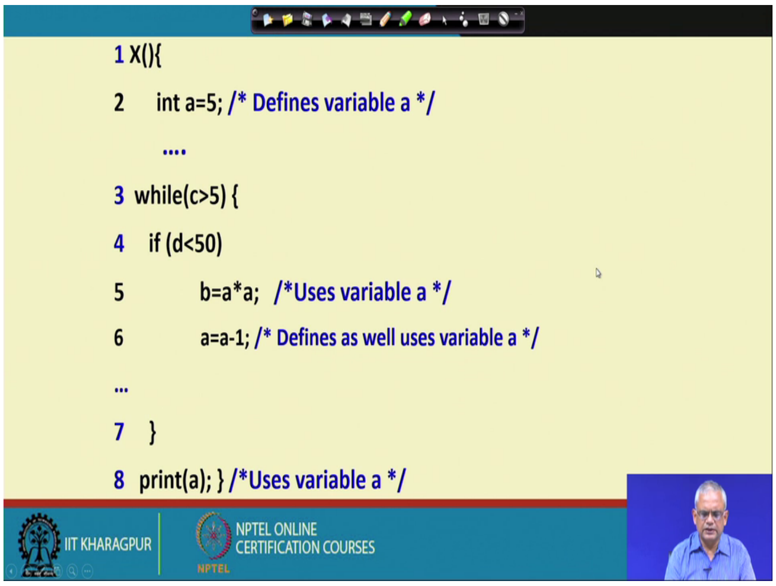
- Selects test paths of a program:
  - According to the locations of
    - Definitions and uses of different variables in a program.

```
a=b;
b=c*d;
d=a;
```

In data flow testing, we see that whether in the program certain definitions and uses of variables exercised. So, here given a small program a equal to b, b equal to c into d, d equal to a; we see here that a is assigned the value here. We say that a is defined, a is defined and the next statement b is defined, the third statement d is defined; but it uses a. So, a is defined in the first statement and is used in the last statement and this we call as a DU path.

So, the definition and use for various variables, we see that whether those are covered and that is the main idea between the dataflow testing. We can think of that the data is defined in the first statement and is used in the last statement here or in other words, the data flows from the first statement to last statement and here, we check whether all types of data flows are getting tested or not.

(Refer Slide Time: 07:35)

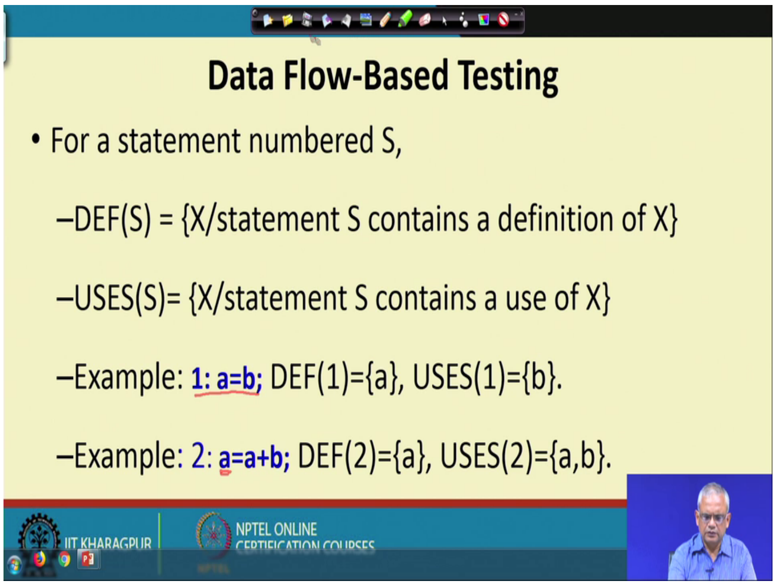


```
1 X(){
2    int a=5; /* Defines variable a */
       ....
3 while(c>5) {
4   if (d<50)
5          b=a*a;   /*Uses variable a */
6          a=a-1; /* Defines as well uses variable a */
...
7   }
8 print(a); } /*Uses variable a */
```

IIT KHARAGPUR   NPTEL ONLINE CERTIFICATION COURSES

Let us look at this program. This is the function and the first one, first statement assigns a value to a we say that it defines the variable a. Now the fifth statement it uses the variable a and defines variable b; the sixth statement, it defines a and also uses a. Similarly, the third statement uses c; fourth statement uses d; the eighth statement uses a. So, given any statement in a program, we can find out what are the variables that the statement defines or uses.

(Refer Slide Time: 08:34)



Data Flow-Based Testing

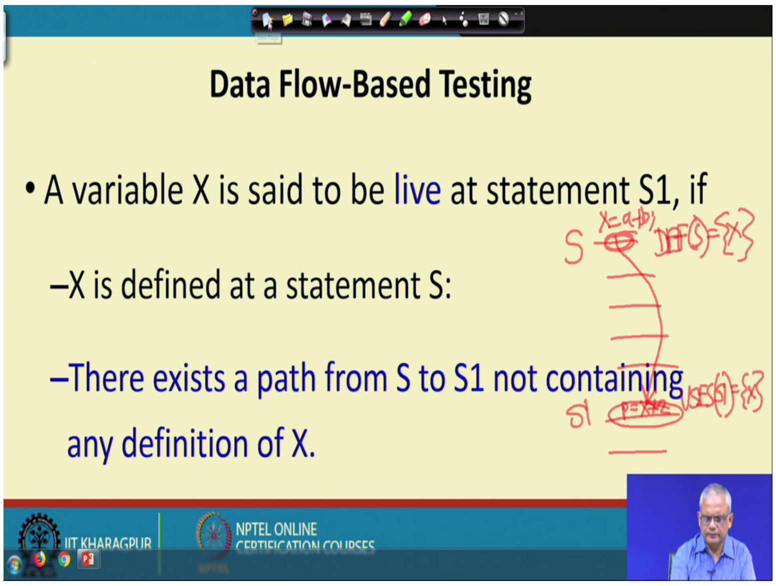
- For a statement numbered S,
  - –DEF(S) = {X/statement S contains a definition of X}
  - –USES(S)= {X/statement S contains a use of X}
  - –Example: 1: a=b; DEF(1)={a}, USES(1)={b}.
  - –Example: 2: a=a+b; DEF(2)={a}, USES(2)={a,b}.

IIT KHARAGPUR   NPTEL ONLINE CERTIFICATION COURSES

And here in the dataflow based testing, first we compute all the DEF and USES set in the, if S is a statement, a program statement, then the set DEF S is the set of all variable X such that the statement defines the variable X. So, if b is equal to a is a statement, then the DEF set for the statement will be b; b is defined b equal to a and USES will be the all the variables that are used in the statement.

Let us take some examples if we look at a statement like a equal to b; then, DEF 1 because the statement number is 1; DEF 1 is a and USES 1 is b. Similarly, if the statement number is 2 and it is a is equal to a plus b; then, DEF 2 is a because a is defined and also if we look at the right hand side of this equality a and b are also used here and therefore, we write USES 2 is a comma b.

(Refer Slide Time: 10:27)



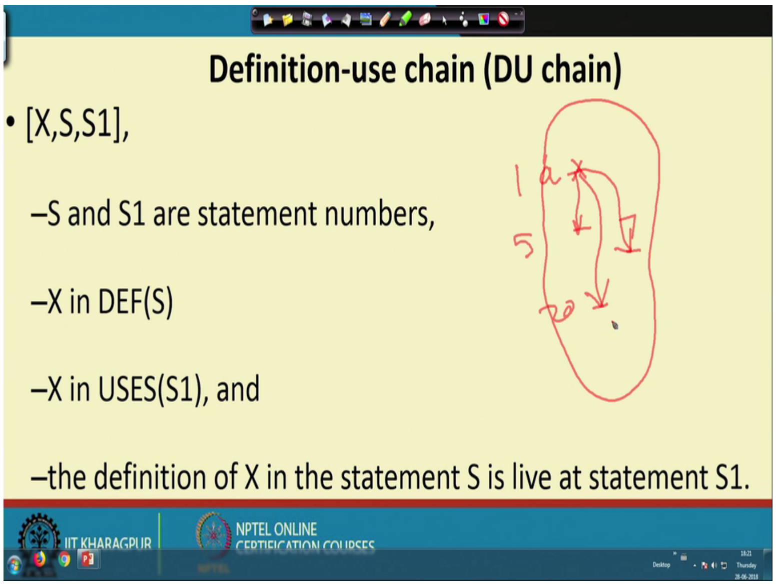
Data Flow-Based Testing

- A variable X is said to be live at statement S1, if
  - X is defined at a statement S:
  - There exists a path from S to S1 not containing any definition of X.

Now, we define the term live at a statement S 1. We say a variable to be live at a statement S 1, if the variable is defined at some statement S and there is a path from S to S 1 not containing any definition of X. So, that means if there are statements in the program and we have 1 statement here which is S and DEF S is X; that means, it must be having of the form X is equal to a plus b or something.

And there is another statement S 1 which has a use for X. So, USE of S 1 is the set X. It must be something like P equal to X into 2 or some such thing. So, that the value X, the value of the variable X is used here and then the statements in between they have no definition for X. So, if we define X to be 5, then the 5 value is available in S 1 and this is called as a the variable X is called as live at S 1. It is defined at S and use at S 1 and we say that X which is defined by at S is live at S 1.

(Refer Slide Time: 12:56)



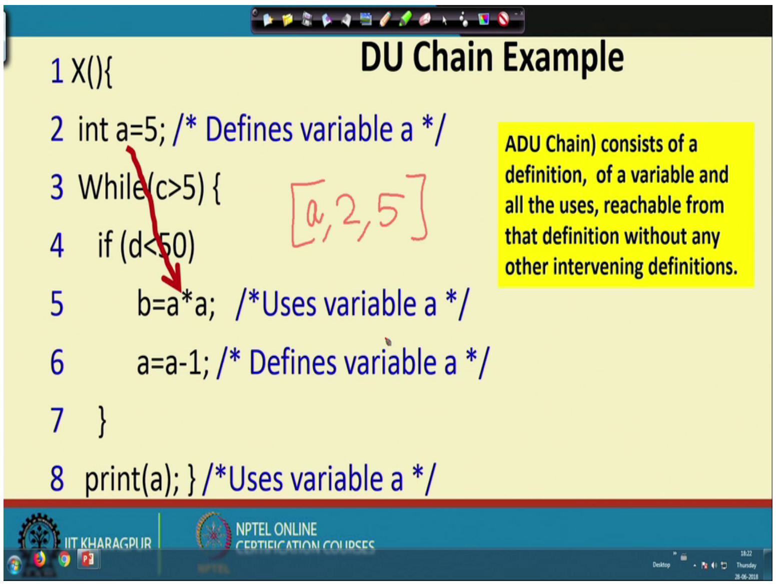
Definition-use chain (DU chain)

- [X,S,S1],
  - S and S1 are statement numbers,
  - X in DEF(S)
  - X in USES(S1), and
  - the definition of X in the statement S is live at statement S1.

Now, we can define a DU chain. It is the set of all variable which are live which is defined at S and used at S 1. It is the set of all variables X which is defined at S and used at S 1. So, in a program we might have different variables defined at different statements and which are live at other statements.

So, a variable a might be defined at some statement and it is live at some other statements; the statement number maybe 1 and this maybe 5, 7, 20 etcetera. So, 1 to 5 is a DU chain; 1 to 20 is a DU chain; 1 to 7 is a DU chain. So, a program can contain large number of DU chains because there are many variables and each variable which is defined at a place maybe live at several statements.

(Refer Slide Time: 14:31)



DU Chain Example

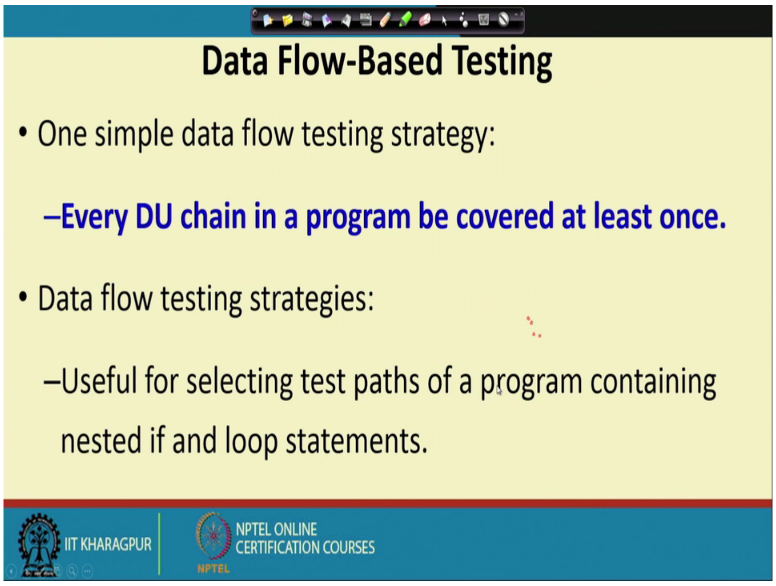
```
1 X(){
2 int a=5; /* Defines variable a */
3 While(c>5) {
4    if (d<50)
5        b=a*a;  /*Uses variable a */
6        a=a-1; /* Defines variable a */
7    }
8 print(a); } /*Uses variable a */
```

$[a, 2, 5]$

ADU Chain) consists of a definition, of a variable and all the uses, reachable from that definition without any other intervening definitions.

A DU chain consists of a definition of a variable and all uses the reachable from the definition without any intervening definition. Just to give an example here, if this is a function; a is defined here in the second statement and a is used in the fifth statement. So, a, 2 and 5 is a DU chain. We can find other DU chains here; maybe b is defined here and used at some other place and so on. A program typically has a large number of DU chains.

(Refer Slide Time: 15:40)

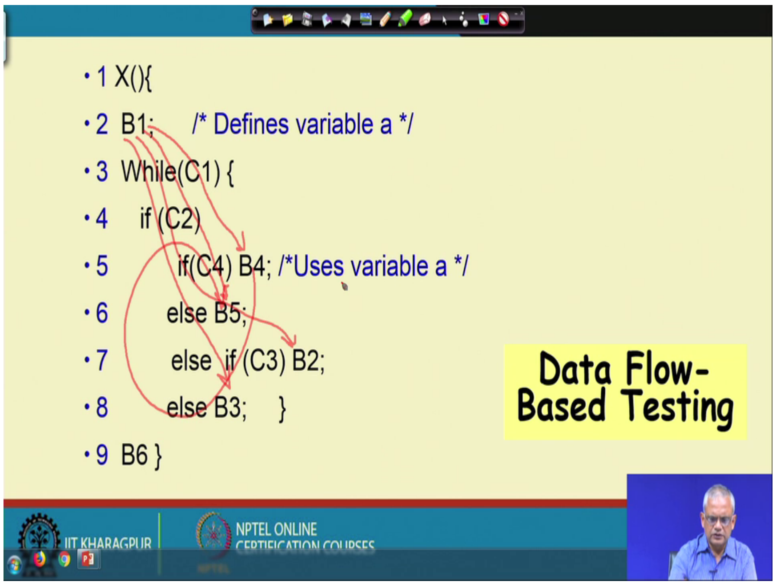


Data Flow-Based Testing

- One simple data flow testing strategy:
  - –**Every DU chain in a program be covered at least once.**
- Data flow testing strategies:
  - –Useful for selecting test paths of a program containing nested if and loop statements.

The simplest dataflow testing strategy is to require that every DU chain in the program is executed at least once; every DU chain in the program is executed at least once. But what about the path testing; does it ensure that all DU chains are covered? No. But what about all DU chains covered, does it imply all paths covered? No.

(Refer Slide Time: 16:30)



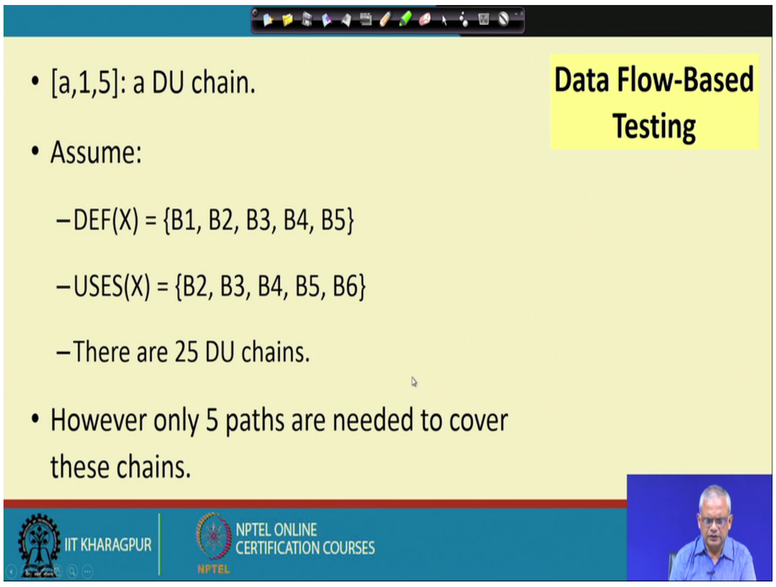
```
• 1 X(){
• 2 B1;      /* Defines variable a */
• 3 While(C1) {
• 4    if (C2)
• 5       if(C4) B4; /*Uses variable a */
• 6       else B5;
• 7       else if (C3) B2;
• 8       else B3;   }
• 9 B6 }
```

Data Flow-Based Testing

Let us look at 1 example. So, this is the program, where we have several statements that we have written as a block. So, there are 5 blocks here B1, B2, B3, B4, B5 and see here that once it enters a while loop, the different blocks can be reached based on the outcome of the conditions C2, C4, C3 and C1.

Now, if you want to compute the DU chains here, if a certain variable X is let us say used in defined in all the blocks and used in all the blocks. Then, how many DU chains are there ok? So, we have B1 to if the variable is defined in B1 and used in all these B4, B5, B2, B3. So, it can reach B2; it can reach B5; it can reach B2; it can be live at B2 and it can be live at B5. Now, what about if it is defined that B4 and used in B5 ok? Yes, in the loop it can be defined here and used here. Similarly, it can be defined in B4 and used in B2 and so on.

(Refer Slide Time: 18:08)



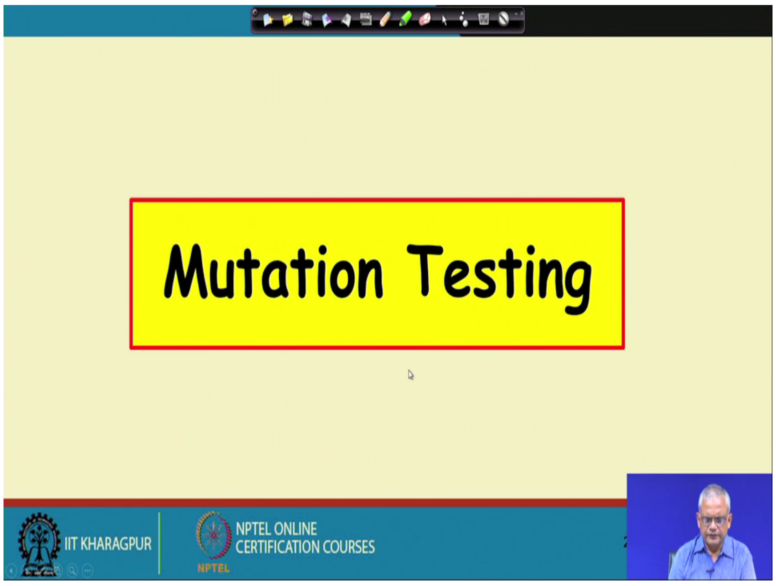
Data Flow-Based Testing

- [a,1,5]: a DU chain.
- Assume:
  - DEF(X) = {B1, B2, B3, B4, B5}
  - USES(X) = {B2, B3, B4, B5, B6}
  - There are 25 DU chains.
- However only 5 paths are needed to cover these chains.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

So, based on that computation, if B1, B2, B3, B4, B5 define the variable X and B2, B3, B4, B5, B6 use the variable X; then, we can see that from B1 it can come to B2, B3, B4, B5 and similarly, if it is defined in B2, it can also be used any of these. So, the total number of DU chains can become 25. But then, for the given program there are only 5 paths that are possible.

(Refer Slide Time: 18:43)

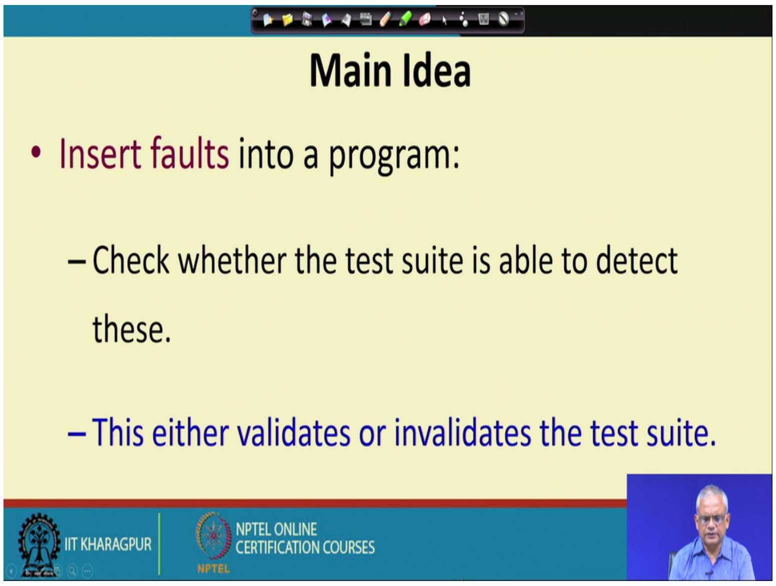


Mutation Testing

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

Because there are 4 conditionals 1, 2, 3, 4; 4 conditionals and that means, the cyclomatic complexity of this is 5 and 5 independent paths are possible and therefore, only 5 test cases can cover all the 5 independent paths; but then, the DU chains can be many. Now, let us look at Mutation Testing.

(Refer Slide Time: 19:31)

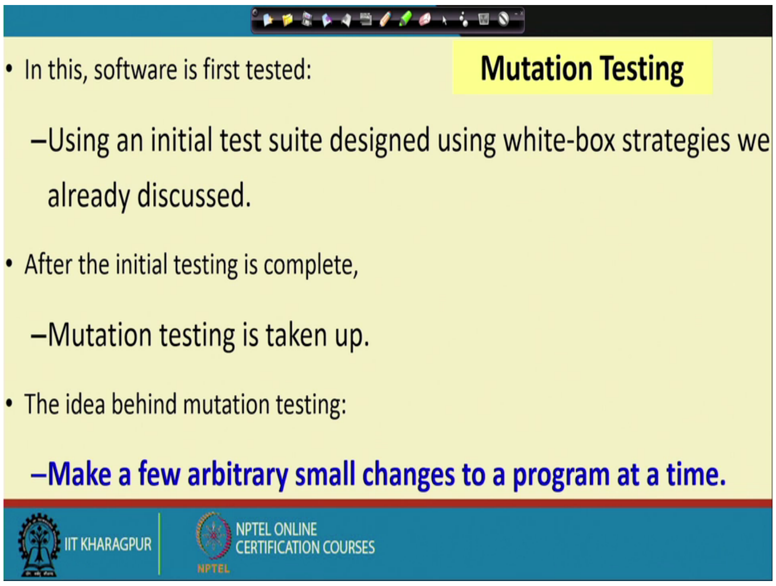


Main Idea

- Insert faults into a program:
  - Check whether the test suite is able to detect these.
  - This either validates or invalidates the test suite.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

(Refer Slide Time: 19:35)



Mutation Testing

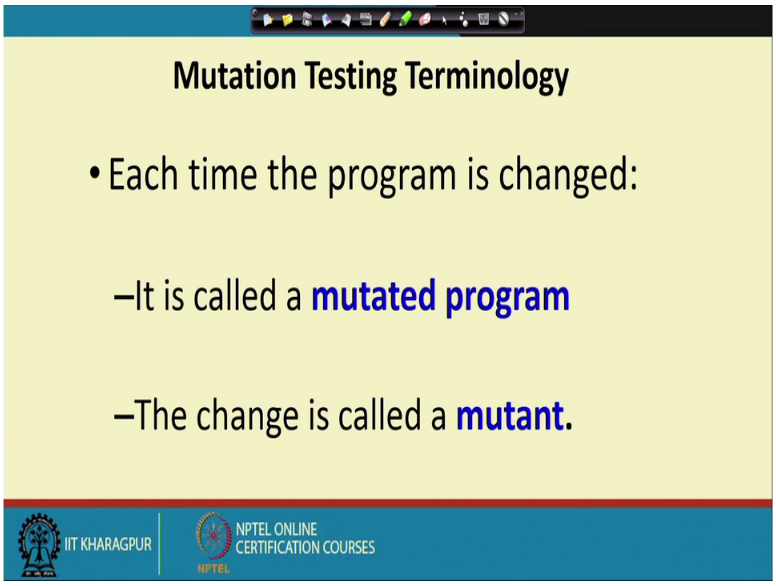
- In this, software is first tested:
  - Using an initial test suite designed using white-box strategies we already discussed.
- After the initial testing is complete,
  - Mutation testing is taken up.
- The idea behind mutation testing:
  - Make a few arbitrary small changes to a program at a time.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

In mutation testing, an initial test suite is designed by using some coverage based testing and then, once coverage based testing is over; then mutation testing is taken up. The main idea is to inject faults small changes into the program at a time and run the set up test cases. See if the change that we made is being detected, if it is not being detected by the test cases; then, we know that the test cases are inadequate. We need to introduce additional test cases until the change that was made is getting detected.

So, that is the main idea here. Given a program each time we inject faults by making small change in the program and then, check whether the test suite is able to detect the change and then based on that we either say that the test suite is and we inject one more fault or we just say that the test suite is and if it is not detected, we say that we need to add more test cases and we invalidate the test suite that is we say the test suite is not sufficient, we add more test cases to that.

(Refer Slide Time: 21:08)



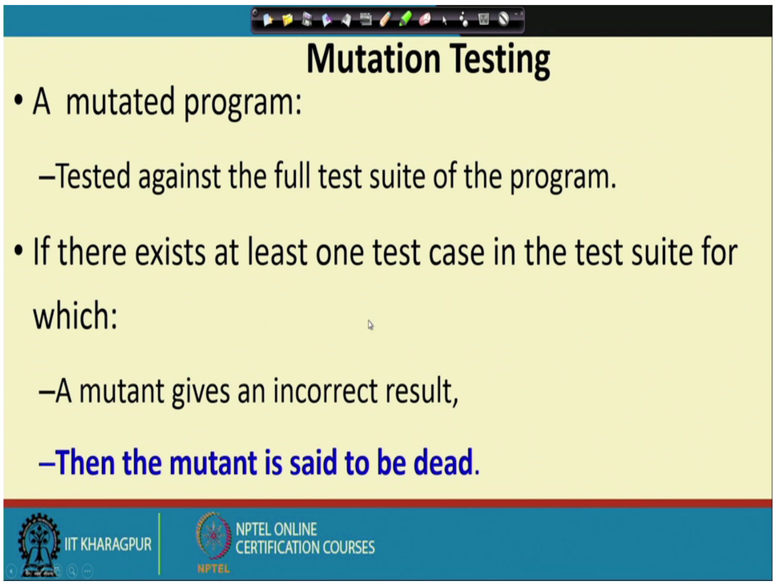
Mutation Testing Terminology

- Each time the program is changed:
  - –It is called a **mutated program**
  - –The change is called a **mutant.**

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

So, here each time we make one small change and the change is called as a mutant and once we have the mutant, the program is called as a mutated program.

(Refer Slide Time: 21:26)



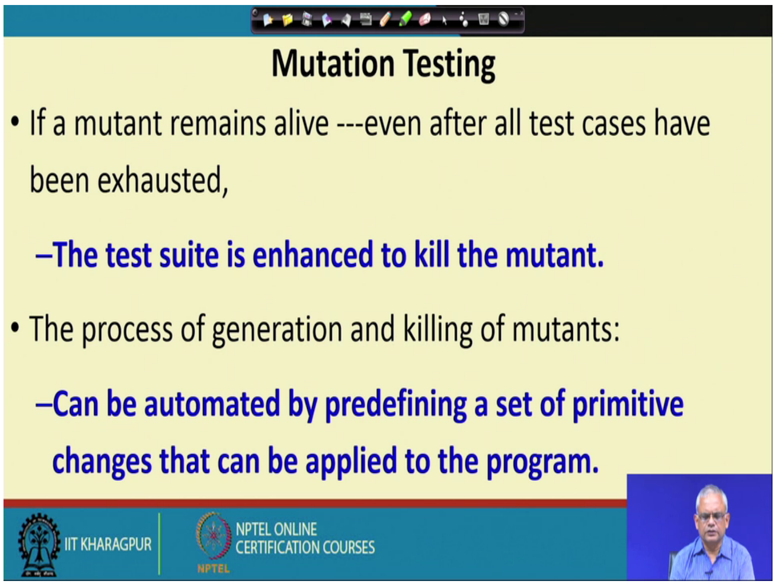
Mutation Testing

- A mutated program:
  - Tested against the full test suite of the program.
- If there exists at least one test case in the test suite for which:
  - A mutant gives an incorrect result,
  - **Then the mutant is said to be dead.**

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

And the mutated program is tested using the full test suite and then, if there is at least one test case which fails, then we say that the mutant is dead; but if the mutant is not dead, that is all test cases are exhausted and none of the test cases has failed. Then, we say that the mutant survives and we need to design additional test cases so that the mutant can get killed.

(Refer Slide Time: 22:05)



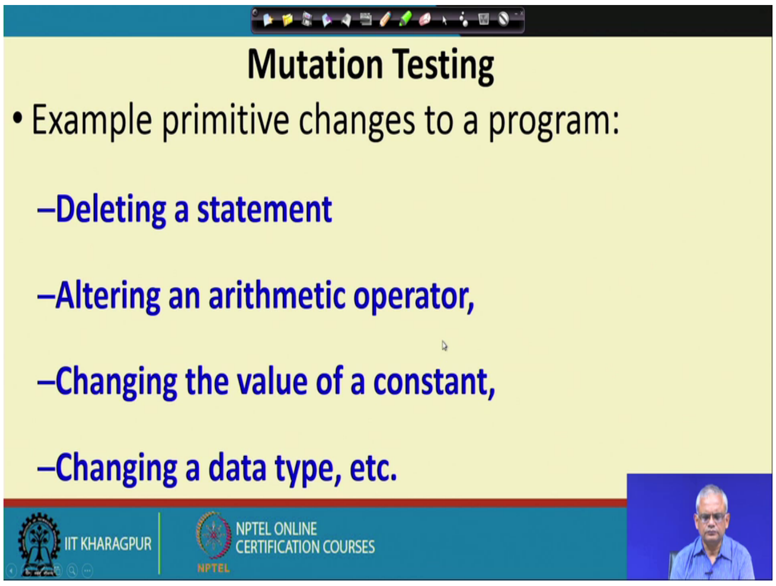
Mutation Testing

- If a mutant remains alive ---even after all test cases have been exhausted,
  - **The test suite is enhanced to kill the mutant.**
- The process of generation and killing of mutants:
  - **Can be automated by predefining a set of primitive changes that can be applied to the program.**

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

But here, for given program large number of mutants are possible because we can make many small changes to a program and therefore, when this is typically done by automated tool. In the automated tool it makes small changes and it runs the test suite and then, if there is a test suite fails, then the mutant is discarded and the new mutant is created.

(Refer Slide Time: 22:39)



Mutation Testing

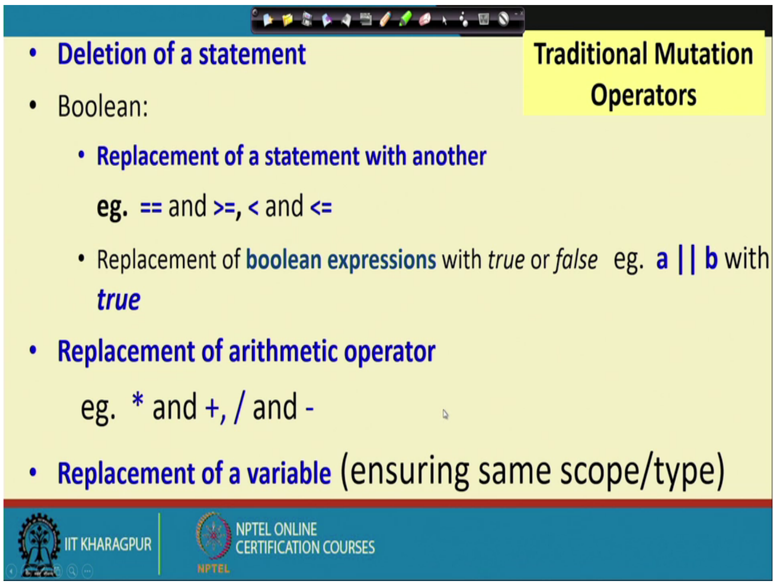
- Example primitive changes to a program:
  - –Deleting a statement
  - –Altering an arithmetic operator,
  - –Changing the value of a constant,
  - –Changing a data type, etc.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

But then, what are the types of mutants that can be possible; one mutant is delete test statement in a program; another mutant may be alter and arithmetic operator that is substitute a plus with minus or a multiplication with a division and so on. The third type of mutant maybe changes the value of a constant. A fourth type of mutant maybe change the data type that is an int into float. A fifth type of mutant maybe to change the logical operator equal to with not equal to or something.

So, given a program, we can have several primitives defined primitive types of changes and then, we carry out throughout the program and this may lead to millions or billions of mutants getting created.

(Refer Slide Time: 23:51)



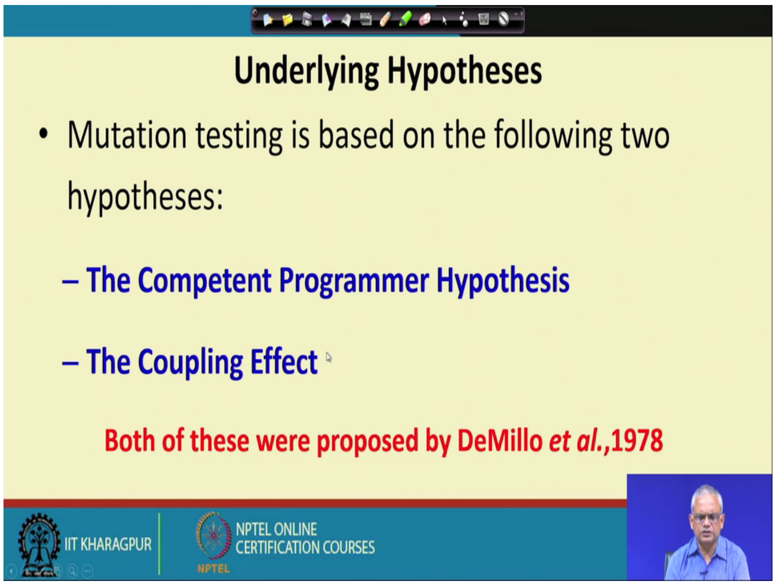
Traditional Mutation Operators

- Deletion of a statement
- Boolean:
  - Replacement of a statement with another
    eg. == and >=, < and <=
  - Replacement of boolean expressions with *true* or *false* eg. a || b with *true*
- Replacement of arithmetic operator
  eg. * and +, / and -
- Replacement of a variable (ensuring same scope/type)

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

Some of the traditional mutation operations are Delete a statement; Replace 1 statement with another. We can change some operators less than with less than equal to. We can replace the Boolean expression entire Boolean expression with either true or false. We can replace arithmetic operator multiplication with addition and so on. We can replace a variable a with b and ensuring that the scope and type of the variable are the same.

(Refer Slide Time: 24:42)



Underlying Hypotheses

- Mutation testing is based on the following two hypotheses:
  – The Competent Programmer Hypothesis
  – The Coupling Effect

Both of these were proposed by DeMillo *et al.*,1978

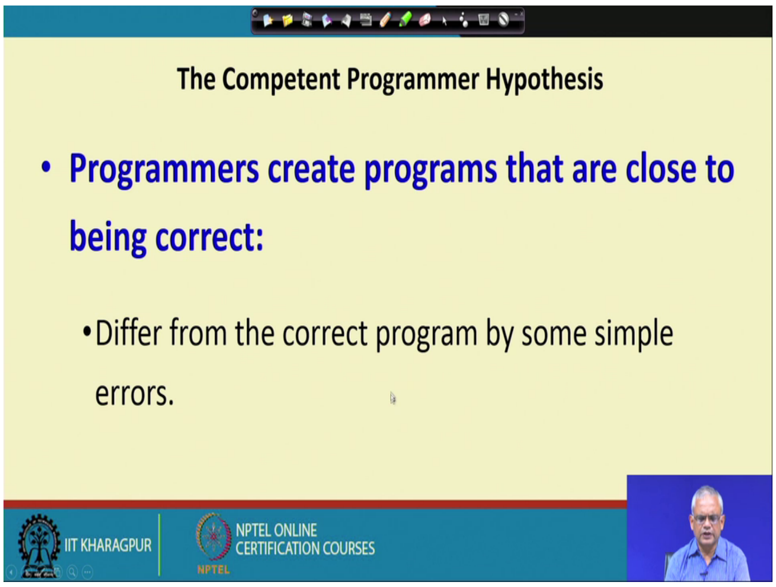
IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

Now, let us see the Underlying Hypotheses behind the mutation testing; there are 2 main hypotheses. The first one is called as Competent Programmer Hypothesis. So, this hypothesis says that programmers write programs; well, they are a good programmers. But occasionally, they makes small programming errors maybe while writing, they may exchange a plus with a minus or they may exchange Boolean operator, forget to write a statement etcetera.

And the other, hypothesis is called as coupling effect which says that the more complex bugs in a program are caused by a combination of several simple bugs. The simple bugs are called as the programming bugs just a small programming error and a complex bug maybe algorithmic bug; the programmer did not understand the algorithm well and so on.

So, the complex bugs are hard to introduce, but the coupling effect says that if we introduce several small bugs, they may act like a complex bug and both these hypothesis were proposed by DeMillo in 1978. And this form the corner store corner cornerstone of the mutation testing and this ensures that mutation testing is effective under these two assumptions the competent programmer hypothesis and the coupling effect.

(Refer Slide Time: 26:34)



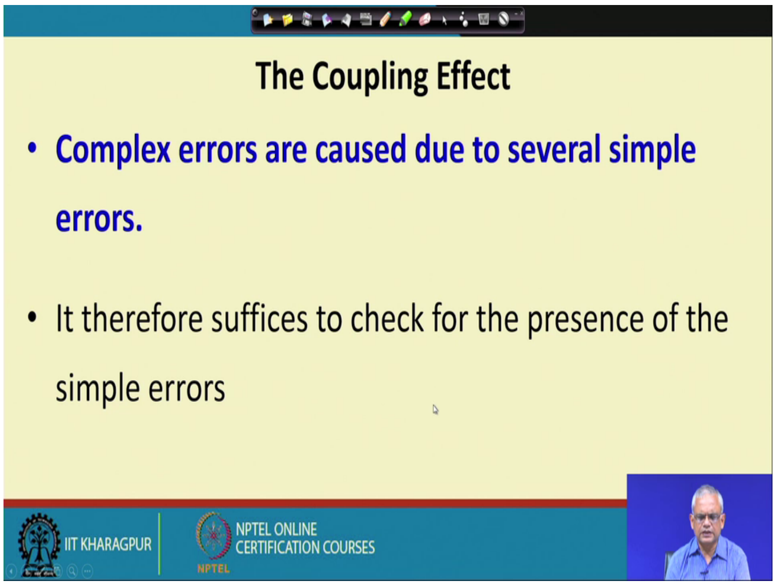
The Competent Programmer Hypothesis

- Programmers create programs that are close to being correct:
  - Differ from the correct program by some simple errors.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

The competent programmer hypothesis says that at anytime, the programmer may call, may introduce only very simple bugs.

(Refer Slide Time: 26:46)



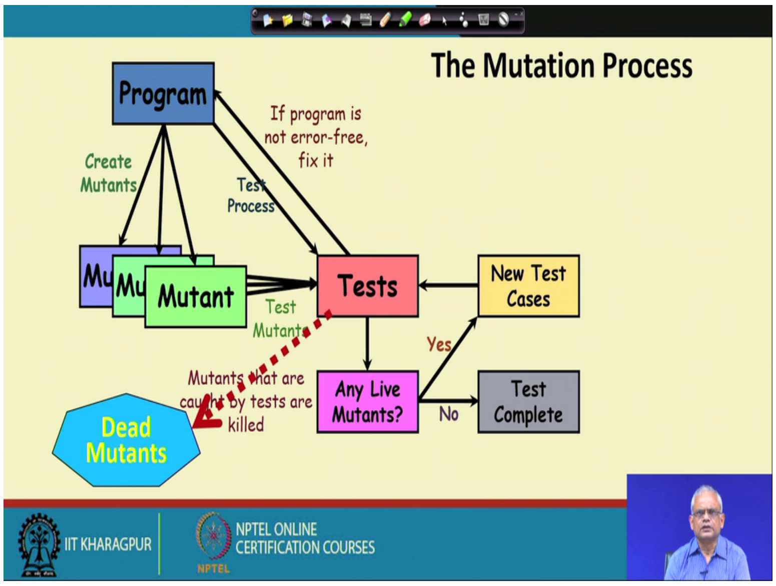
The Coupling Effect

- **Complex errors are caused due to several simple errors.**
- It therefore suffices to check for the presence of the simple errors

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

The Coupling effect complex errors are caused due to several simple bugs and due to the coupling effect, it is enough if we check with simple mutants. We do not have to really simulator create very complex mutants; just simple mutants several of them will suffice.

(Refer Slide Time: 27:16)



The Mutation Process

And now, we can pictorially depict the mutation testing process. We first use some initial test cases based on coverage based testing and so on, we test it and if there are problems we fix it and then, the mutation testing process starts. We create a large number of mutants and for each mutant, we test using the test cases and if the test case fails; then, the mutant is dead.

But, if there are any mutant is live, then we need to design further test cases. We augment the test cases and then we again carryout testing with other mutants and finally, as all mutants are being killed, the test is complete. So, we discussed the dataflow testing which is a coverage based testing and we looked at the mutation testing process and with this, we will complete this lecture already at the end of the time.

Thank you very much.