Lecture - 14
Dataflow and Mutation Testing

Welcome to this session. So far we had seen some basic concepts on testing and then, we had looked at black box testing, various black box testing techniques and then, we had looked at some white box testing techniques and we said that there are mainly two types of white box testing techniques. One is coverage based and the other is fault based and we started looking at the coverage based testing techniques. In the coverage based testing techniques, we looked at statement coverage branch or decision coverage, various conditions coverage techniques and then, we had also looked at path testing.

Today we will look at few more white box testing techniques. Let us get started, but before that we will just like to see how much you have understood and recollect about the previous discussions we had will just pose you few questions.

(Refer Slide Time: 01:33)

> **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?

The first question is what do you understand by a coverage based testing? What is coverage here? What is covered? So, to answer this question, you can check your own understanding. To answer this question, a coverage based testing essentially is test execution covers certain program elements

or executes certain program elements and the program elements that we consider can be statements, can be conditions, can be component conditions, can be paths and so on.
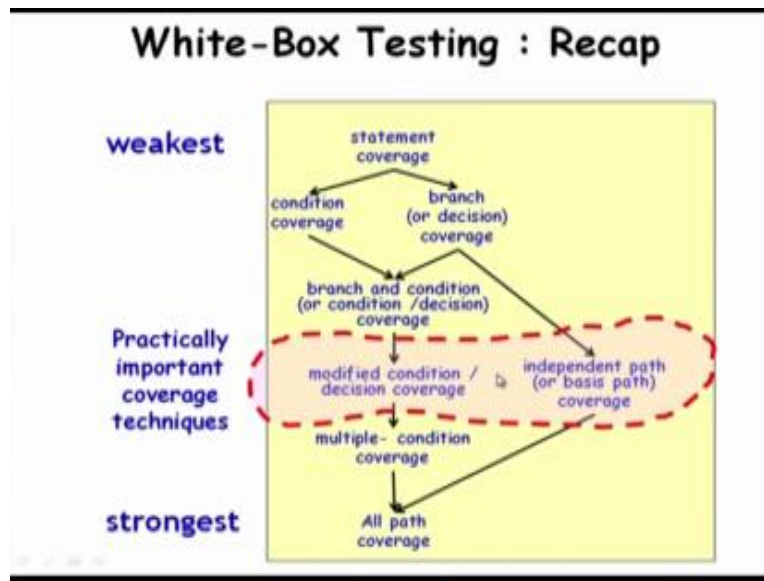
Now, let us look at one more question. What are the different types of condition coverage? What are the different types of coverage testing that we have seen so far and here I hope you remember the various types of coverage techniques. We have discussed statement coverage that was the weakest. Then, we looked at branch and decision coverage, we looked at basic condition coverage, we looked at basic condition with decision coverage, we looked at multiple condition coverage, the MCDC coverage and the path coverage.

Now, the next question is how is exactly the coverage based testing carried out? What does the tester have to do? Does he have to develop a graph for the program control flow graph and then, design the test cases. This, if you remember we had said that only for very trivial program, we can develop the test inputs that will cause the coverage of the program elements. For a practical program, we do not really design coverage based test suites. We give test inputs, random test inputs and then we have a tool which measures the coverage and we keep on giving inputs until a desired coverage metric is achieved. May be our metric is 100 percent statement coverage, 90 percent path coverage and so on.

Now, the next question is what do you understand by fault based testing? If you remember we had said that there are ( ) basically two categories of white box testing techniques. One is coverage based technique where we try to cover or execute certain program elements, where as in the other type of white box testing we introduced specific types of faults into the program and then, check whether the test cases are able to detect them. If the test cases are not able to detect them, we strengthen the test cases by adding more test cases. So, we have not really looked at any fault based testing techniques till now excepting for the basic concepts.

Today we will look at the mutation testing which is a fault based testing. Now, the next question is give an example of a fault based testing technique. This I think all will be able to answer. We just said mutation testing in an example of fault based testing.

Now, let us recollect some important aspects of the white box coverage based testing technique that we had discussed. We had said that the strongest white box testing possible is all path coverage and all path coverage as you might remember is covering all possible paths in a program, but then we had remarked that all path coverage is actually a very impractical criterion because in presence of loops, there can be very large number of paths, millions or billions of paths may be there and it is practically impossible to achieve all path coverage in the presence of when the program as loops and then, we had looked at the weakest white box testing technique which is the statement coverage and then, we had looked at the branch or decision coverage which is a stronger technique than statement coverage.

We had looked at the basic condition coverage, where every component condition or atomic condition is made to assume true and false values. We had looked at the branch and condition coverage or which is also called as condition decision coverage which is stronger than both condition coverage and the branch decision coverage and we had looked at the MCDC coverage which is stronger than the branch and condition coverage. We had looked and discussed the basis path coverage which we simply called as path coverage and sometimes, it is also called independent path coverage in the literature and then, we had looked at multiple condition coverage which is stronger than MCDC coverage, but then we said that multiple condition coverage can result in large number of test cases.
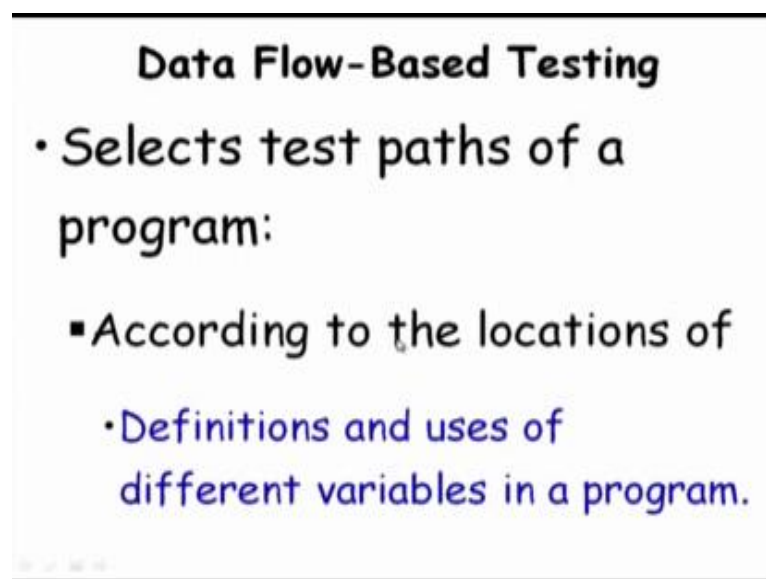
When the number of atomic conditions or component condition in a composite condition or a

decision statement is large, specifically we said that if there are ten component conditions, we will have 2^{10} test cases required for multiple condition coverage and if there are 20 or 30 atomic conditions in a conditional expressions, then we will have a huge number of test cases, millions and even billions of test cases required to achieve multiple condition coverage and therefore, the multiple condition coverage is not really considered a practical testing techniques and normally, no one insists that a program should be tested to achieve multiple condition coverage, but what normally is practically important, coverage techniques are the MCDC coverage and the path coverage.

So, normally when a program for commercially uses or uses by a large number of users is returned, the tester tries to test with the objective of meeting both MCDC coverage and path coverage and edge can be seen in this figure that both these are complementary test cases, sorry complementary testing. What we had made, what do you mean by complimentary testing? We had defined it earlier that achieving path coverage does not ensure that we have achieved MCDC coverage and similarly, if our test suite achieves MCDC coverage does not mean that we have achieved path coverage.

So, it is a good idea if you are testing a program which is going to be used by large number of users as most of the software industry, they test their program using two important techniques, the MCDC testing and the path testing. Now, let us look at one more important white box testing technique which is called as data flow testing.

(Refer Slide Time: 11:06)

Here in data flow testing, the main idea is that we have test cases such that the definition and uses of variables are covered. So, the test cases are defined or we define paths through the program based on what are the locations of definition and user specific variables. Let me just repeat that the location of definition and use of specific variables define paths through the program.

This is unlike the path coverage which you had discussed where the paths were defined on the control flow graph here. The paths are not defined on the flow graph control flow graph, but here we looked at what how does the data flow or where are the data definitions and uses occur and based on that we defined the path.

(Refer Slide Time: 12:20)


```
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 */
```

Just to give an example, let us look at this c program and here as we can see in statement two a is assigned 5 or we say that statement 2 defines variable a and then we have statement 3 which a uses of variable c, statement 4 as uses of d, but if you look at statement 5, we have use of variable a. So, definition of variable a, occur in statement 2 and the variable a is used in statement 5.

Now, let us look at statement 6. In statement 6, we have both uses of variable a and also definition of variable a because a appears on the LHS in addition to appearing on the RHS. Similarly on statement 8, we have again uses of variable a.

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(1)={a}, USES(1)={a,b}.

So, here for every statement we define two sets. One set is called DEF S. If the statement number is S, we define DEF S which is the set of all variables X, such that X contains a definition of X and typically a statement at most defines one variable. So, the definition set of a statement will typically be one variable whereas the uses set of a statement S can be many variables. The statement may use many variables and it is possible that it can assign value or defines value of one variable. Now, let us look at this example, a equal to b and here the statement number is 1. So, we write the DEF set of statement 1 is a, and the USES set of statement 1 is the set b.

Now, let us look at second example. So, maybe I could have written DEF 2 because I am number it as 2. So, DEF set of statement 2 is a. So, it defines the variable a, and the USES set of statement 2 is both a and b because a and b are both used.
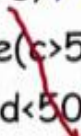
Data Flow-Based Testing

- 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 say that a variable X is live at a statement S1, the variable is live if x is defined at a statement S and from between S and S1, there is no further definition of X. So, a statement S, a variable x defined at a statement S is live at another statement S1 if there is no intermediate definition of that variable.

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 */
```

Now, let us see with respect to an example if a variable is live or not. So, this is the definition of a and this is the USES of a and there is no intervening definition of a and therefore, the definition of

the variable a is live at statement 5 and is also live at statement 6, but then it is not live. The definition at 2 is not live at statement 8 because it has been redefined in statement 6, but the definition of a at 6 is live at statement 8. So, from 2, the definition is live at statement 5.

(Refer Slide Time: 17:27)

Now, based on what we discussed, we can define a DU chain. A DU chain is a triplet consisting of the name of a variable the statement at which it is defined and the statement at which it is live. So, X is in the DEF set of S, X is in the USES set of S1 and there is no intervening definition of S, sorry of X between S and S1. This we call it as a DU chain.

One of the simplest data flow testing is that all DU chains in the program must be covered. That means that wherever there is a definition of a variable, the uses of that variable must be a test case, must cover those two statements. So, this we said that there is a, this is the simplest data flow testing. There are other data flow testing criteria, more advanced criteria, but we are not going to discuss those, but just looking at the basic concepts of data flow testing and the simplest data flow testing technique which is the DU chain coverage.

(Refer Slide Time: 19:10)

So, let us look at what will be the test cases required to achieve DU chain coverage. Now, let us look at these statement where we have a block of statements B1 which define a variable a, and then there are some conditions which use other variables. do not use a and the block B4. It uses variable a. Now, there are other variables as well.

(Refer Slide Time: 19:54)

Based on that we can compute the DU chains for the example that we discussed, a is a variable defined in block 1 and used in block 5 and this forms a DU chain. Now, assuming that X, the statement is used in different blocks, we will have a large number of chains, but then we need only a few paths to cover these chains because multiple chains can occur on different control flow paths.

Now, let us look at another white box testing technique which is the mutation testing and we had said that mutation testing is not a coverage testing, but it is a fault based testing technique where we introduce a specific type of fault into the program and then, check whether the test cases are effective against that type of fault. If not, the test case will be augmented with additional test cases to strengthen the test suite, so that the specific type of fault will be detected. Now, let us look at the basic idea here.
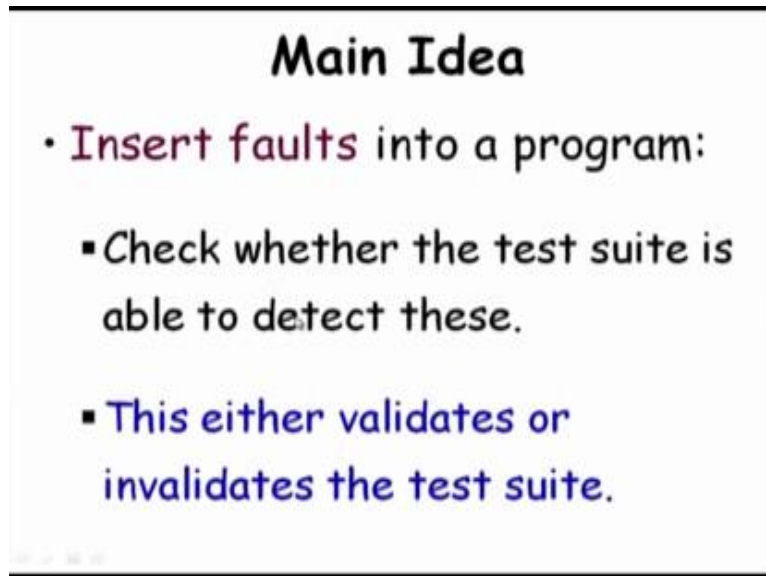
In mutation testing, first we use certain coverage testing techniques and we have some test cases to achieve coverage of some coverage technique. So, in initial test suite is designed using some white box coverage technique we discussed. Now, once the initial testing is complete, the initial testing may be mc dc testing or may be path testing or may be both and once we have tested using this, we want to check if the test is really effective against certain type of bugs and we carry out mutation testing.

The main idea behind mutation testing is we make small changes to the program and when we changed a program that essentially means a bug. So, the original program written by the programmer we make small changes to that. May be we changed an arithmetic operator from plus to minus or maybe we changed the type of a variable. These are some examples of small changes to a program and each small change to think of it is actually a bug, a type of bug.

So, now we run the test cases knowing that we have introduced a bug. We run the test cases again and check whether this bug is caught. That means, some test cases fail signaling that a bug has been introduced in the program.

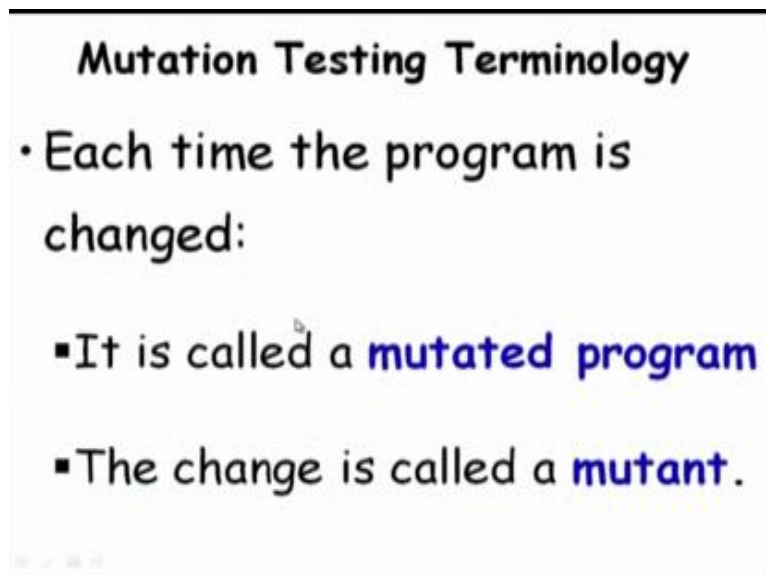## Main Idea

· Insert faults into a program:

- Check whether the test suite is able to detect these.

- This either validates or invalidates the test suite.

So, that is the main idea here. We insert fault into a program by constructing a mutated program. A mutated program has a simple bug and now, the test case is run and it may be quite straight forward to run the test cases. If we have a record and play tool, we just effortlessly run all test cases and check whether all test cases pass or some test cases have failed.

Mutation Testing Terminology

· Each time the program is changed:

- It is called a **mutated program**

- The change is called a **mutant.**

So, the terminology that we discussed here one is a mutated program. Mutated program is one where we deliberately introduced some minor fault and the type of change we apply is we call it as

a mutant.

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.

Now, if we have this mutated program and we run our test cases and then, some test cases fail, then we know that our test cases are actually testing well, they have caught the bug we introduced and we say that the mutant is dead, that means these type of bugs our test cases is able to detect and we might try new bugs to check whether the test cases are detecting those, but then if all test cases pass and we know that we introduced a bug, then our test cases are not good enough. We need to augment our test cases, add additional test cases until the bug that we introduced gets caught.

So, if the test cases pass for a mutated program, we say that the mutant is alive and that is the terminology used in mutation testing and then, we designed additional test cases, run further test cases until a test case fails flagging the bug that was introduced. So, one advantage of this mutation testing is that generation of mutated program making small changes and also, running all the test cases, both can be very easily automated and therefore, even though it is possible to generate a large number of mutants or in other words, even though we can have tens or hundreds of thousands of mutated programs, it is not a problem. We do not have to do it manually. We can generate all these mutated programs through a tool and also, we can run all the test cases and check whether these are passing or failing and therefore, the mutation testing technique is amenable for automation.

We will stop this session at this point and we will continue in the next session.

Thank you.