



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

**Lecture – 37**  
**Mutation Testing for programs**

Hello there, welcome to the third lecture of week 8. So, we are going to this lecture see how mutation testing applies through programs. What I am going to tell you today is explain in detail the terms that we saw in the last class. Ground string, mutation operator mutant, killing a mutant, will re understand all these terms, but by applying it to a particular program and then we will do a couple of examples explaining how to write test cases that will kill the mutants for this programs.

(Refer Slide Time: 00:46)

| Mutation testing for software artifacts: An overview |                       |                   |                          |                          |
|--|-----------------------|-------------------|--------------------------|--------------------------|
|  | For programs          | Integration       | Specifications           | Input space              |
| <b>BNF grammar</b>                                   | Programming languages | –                 | Algebraic specifications | Input languages like XML |
| <b>Summary</b>                                       | Compilers             |                   |                          | Input space testing      |
| <b>Mutation</b>                                      | Programs              | Programs          | FSMs                     | Input languages like XML |
| <b>Summary</b>                                       | Mutates programs      | Tests integration | Model checking           | Error checking           |



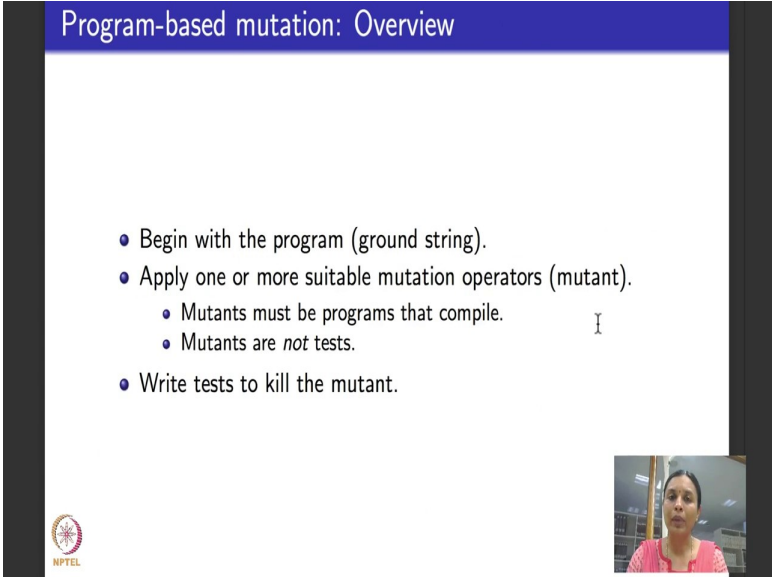
So, here is an overview of the rest of how the courses is going to go for mutation testing. Till now we saw grammars, I did not introduce you to BNF, but I introduced you to a context free grammar as it occurs in the syntaxes several programming languages and when I apply grammar based testing through programming languages, I usually test compilers. Compiler testing is a very involved area we want deal with it in the course.

No known application of direct grammar based testing is available for design integration. Grammar based testing is available for algebraic specifications. Again because this we will involve deep algebraic specification knowledge I am going to skip this part of the

course, but we will apply grammar based testing to do input formats like XML towards the end of this module of mutation testing, we will see this part. We also saw mutation testing in the previous lecture. Mutation testing again applies to all the software artifacts, it applies to program source code of programs where it mutates programs we will see that today and in the next lecture. Mutation testing can be applied for design integration it mutates operators that led to the test basically how one module calls another module and all the integration operators that a particular programming language has to offer. Mutation testing can be applied to specifications especially to finite state machines. There are mutation operators is available for model checkers like SMV, NuSMV but because this course is not on modeled checking I will skip this part, but we will see mutation has its apply to input languages like XML.

So, what are we going to do today's lecture we will begin with program based mutation continue into the next lecture after that I will tell you program based mutation, but for integration testing and then we will see how mutation testing compares for producing invalid inputs by applying it to languages like XML.

(Refer Slide Time: 02:49)



Program-based mutation: Overview

- Begin with the program (ground string).
- Apply one or more suitable mutation operators (mutant).
  - Mutants must be programs that compile.
  - Mutants are *not* tests.
- Write tests to kill the mutant.

The slide includes a small video inset in the bottom right corner showing a woman in a pink shirt speaking. In the bottom left corner, there is a logo for NPTEL (National Programme on Technology Enhanced Learning).

So, that is the plan for the rest of the modules on mutation testing. So, of program based mutation which we are going to begin in this lecture what do we do? We basically begin with the given program. Please remember that in mutation testing the given program that you begin with is called the ground string. Then I apply one or more suitable mutation

operators to get a mutant or a mutated operator, mutated program. Typically its exactly one mutant that I apply. Remember that mutants must be valid programs that must compile for programs for sure and mutants are not test cases. We have to write test cases to kill a mutant. What was the definition of killing a mutant that we saw? Killing a mutant means that the behavior of the test case of the original ground string program is different from the behavior for the same test case in the mutated program.


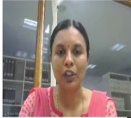
(Refer Slide Time: 03:39)

A simple example of mutation, contd.

One mutation of the Min method:

```
int Min(int A, int B)
{
    int minVal;
    minVal = A;
    if (B<A)
    {
        minVal = B;
    }
    return(minVal);
}
```

```
int Min(int A, int B)
{
    int minVal;
    minVal = A;
    Δ1 minVal = B;
    if (B<A)
    {
        minVal = B;
    }
    return(minVal);
}
```



Here is a simple example. What is this Java, what is this program to, it could be Java it could be C whatever it is the if I have just given a tiny little code segment does not matter which programming language it is. So, here is his method called min it takes 2 arguments A and B which are both integers and then it computes the minimum of A and B. That is what the job is supposed to do, which is the less amongst A and B. So, it initializes a local variable called min value where it stores the value of the minimum it says let min value be A to start with. If B happens to be less than a then you change the min value to be otherwise let min value continue to be whatever it is A or B it returns the min value. Simple enough right? Suppose we want to test this program using mutation test what do we do. So, here is the original method given on the left hand side. What I have done it is I want to check whether this program correctly computes the minimum value.

So, I will say is this code correct I assigning minimum value to be A and then say if B is less than A then you reassigned minimum value to be B. Is that correct? What will happen if I use the same code, but instead of assigning minimum value to A, I assign minimum value as B. So, that is called one mutation. So, I take the same program which is given on the left hand side and then I consider the same program and I make one mutation which is given here. How will you read this as? Delta one this triangle or delta one read it has one mutation applied to this program on the left which is the ground string. What is the mutation? Instead of this statement, min val A you remove that statement and replace it with min val B. In other words, assuming that this program came from a grammar at some point in the grammar the string the statement has a string would have got generated min val A.

At the same time instead of generating a substitute A with B. So, on the right hand side how do I read this? This is the same copy of the program which is the ground string on the left hand. So, the ground string is on the left hand side, the mutant or the mutated program is on the right hand side. What is the difference between the ground string and the mutant? That is this one statement. The statement min val equal to A is not there. Even though I have returned it here for the sake of completeness in the muted program it is not there instead of that this statement is there min val equal to B. So, what is the mutation or the change that I have done? I have taken this statement in the ground string changed this A with B. That is how you read the program on the left hand side. Is that clear? Instead, in fact, to this is just one change you could make several other changes.


(Refer Slide Time: 06:21)

A simple example of mutation, contd.

Six different mutations of the Min method. Results in six different programs, each with one mutation.

```
int Min(int A, int B)
{
    int minVal;
    minVal = A;
    if (B<A)
    {
        minVal = B;
    }
    return(minVal);
}
```

```
int Min(int A, int B)
{
    int minVal;
    minVal = A;
    Δ1 minVal = B;
    if (B<A)
    Δ2 if (B>A)
    Δ3 if (B<minVal)
    {
        minVal = B;
    }
    Δ4 Bomb();
    Δ5 minVal = A;
    Δ6 minVal = failOnZero(B);
    return(minVal);
}
```



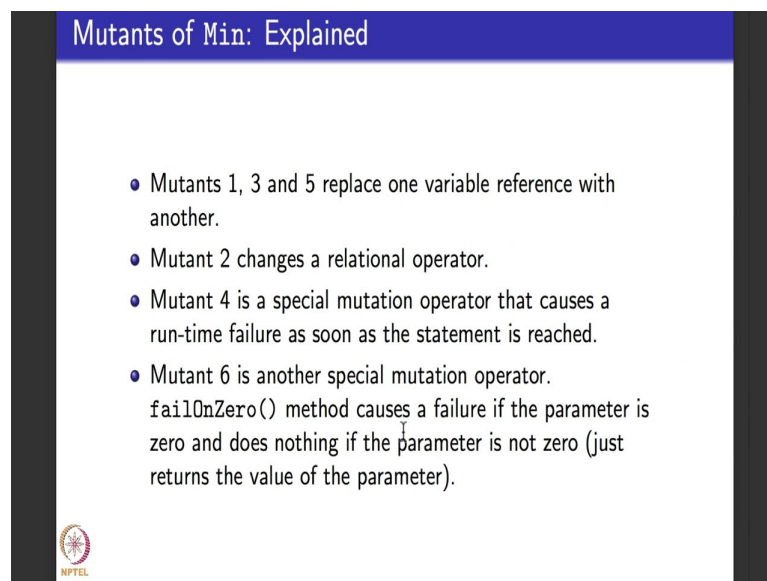
So, what I have done here is a given you examples of 6 different mutants for the same min method. How do I read this? 6 different mutants the one on the left hand side is the ground string or the program that is being tested for mutation testing. I can create 6 different mutants how do I create 6 different mutants, they are labeled here: delta 1, delta 2, delta 3, delta 4, delta 5 and delta 6. How do these come into picture? delta 1 means take out this statement min val equal to A replace it with min val equal to B. All other parts the program remain the same. What is delta 2 mean ? Take out the statement if B less if B less than A, instead of that, keep this statement if B greater than A. So, that gives me a second mutated program from the ground string on the left.

Similarly, the third mutated program says take out this if statement B less than A, instead of that keep this B less than min val. And what do these mean? These mean that you insert a new statement called bomb instead of min val equal to B and I tell you what the statement bomb is. It reads the little funny, but it make sense. The fifth mutation says instead of min val equal to B. What will happen if you do min val equal to A, that is the fifth mutation. Sixth mutation says instead of doing min val equal to B you do min val is equal to fail on 0 of B. Fail on 0 is again another special mutation operator for Java, I will tell you in a minute what it is. The rest of the program is same. So, how do I understand this? The program on the left hand side is called the ground string. From the ground string I pick one statement at any point and time and I make a change to that

statement. On the right hand side I have shown you put together in one consolidated program how 6 different changed mutated programs look like.


The first mutated program takes this statement min val equal to A, replaces A with B the rest of the program is the same. The second change is you check the statement B less than a instead change it to B greater than A, that is the second mutation. Third mutation says instead of the statement B less than A change to B less than min val. Forth mutation says instead of min val equal to B call this method called bomb. Fifth mutation says instead of min val equal to B call min val equal to A. Sixth mutation says instead of min val equal to B call min val equal to fail on 0 of B. Please remember even though I have put it inside one program, what I actually I am depicting in the single program are 6 different mutants that you can apply. At any point in time one at a time to the given ground string program on the left to get 6 different mutated programs.

(Refer Slide Time: 09:18)



**Mutants of Min: Explained**

- Mutants 1, 3 and 5 replace one variable reference with another.
- Mutant 2 changes a relational operator.
- Mutant 4 is a special mutation operator that causes a run-time failure as soon as the statement is reached.
- Mutant 6 is another special mutation operator. `failOnZero()` method causes a failure if the parameter is zero and does nothing if the parameter is not zero (just returns the value of the parameter).

 NPTEL

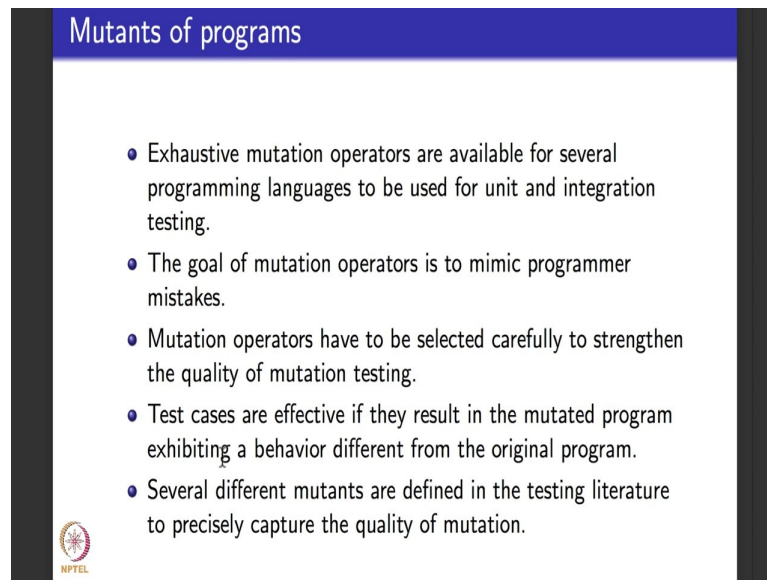
So, this one consolidated thing corresponds to 6 different mutations, please remember that. So, this is what I explained to you. Mutants 1, 3 and 5 are basically called variable reference mutants because they replace one variable reference for the other. Let us go back and see, mutant one replaces A with B, mutant 3 replaces A with min val, mutant 5 replaces B with A. So, it just changes the variable reference. Mutant 2 changes a relational operator. Let us see that. If you go back mutant 2 changes this less than 2 greater than that is what is this mutant 2. Mutation 4, I told you right, that is the bomb

statement which seem related to funny that is a special mutation operator. It is called like that because it causes a runtime failure as soon as the statement is reached. You might wonder what is the point in putting the bomb statement instead of doing this min val equal to B.

Typically there is something called random testing or crash testing a program bomb statement is very useful to see how the program behaves when it randomly crashes. So, by apply this bomb mutation by replacing min val equal to B with that bomb statement I am checking for random crashes in the program. The 6<sup>th</sup> mutant is another very special mutant operator this mutant fail on 0 what it does is it is a method that causes of failure if it is parameter is 0. If the parameters non 0 it does nothing. Does nothing meaning what? It just returns the value of the original parameter as it was. So, if you go back this was the sixth mutation I took this statement min val is equal to B in the ground string and instead of B I put fail on 0 of B. So, it is like saying what would happen to this program if the value of B was 0. In testing usually during unit testing and debugging, it is recommended the programmers give every variable's value as 0 and test.


So, this fail on 0 mutant is very useful for that. It basically take this ground string program. By changing B to fail on 0 B it checks how the program behaves if the parameter B is 0. If the parameter B is non 0 it will just replace it with the parameter and behave as of the original program would behave, but it will tell you what happens if the parameter is 0.

(Refer Slide Time: 11:38)



### Mutants of programs

- Exhaustive mutation operators are available for several programming languages to be used for unit and integration testing.
- The goal of mutation operators is to mimic programmer mistakes.
- Mutation operators have to be selected carefully to strengthen the quality of mutation testing.
- Test cases are effective if they result in the mutated program exhibiting a behavior different from the original program.
- Several different mutants are defined in the testing literature to precisely capture the quality of mutation.

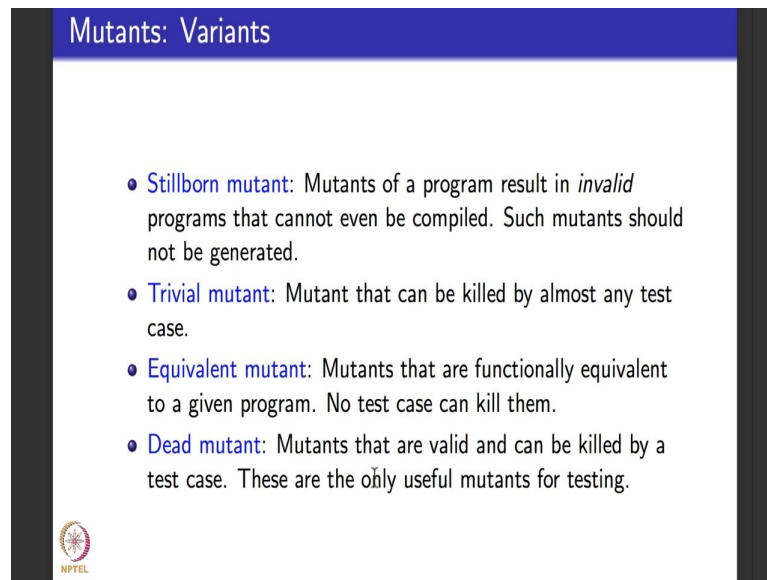


So, it is useful for making a variable 0 which is recommended practice in testing and to test a program for that. So, how do I create mutants of programs? As of always told you from the time we began mutation testing, exhaustive list of mutation operators are available for several different programming languages. They can be used for unit testing like we saw in this example of program. They can also be used for integration testing. In the next lecture I will show you exhaustive list of simple mutation operators that can be applied to Java programs and to programs in C.

The goal of mutation operator is to mimic simple programmer mistakes. If you see this example you will understand the statement right. Maybe the programmer made a mistake. This is a correct program, the ground string, may be instead of writing A if we written B then how would the program behave erroneously. Similarly instead of less than suppose the programmer I had written greater than then how would the program behave erroneously? That is what this mutation operators do they make small changes that will reflect typical mistakes that a programmer we will make and then see how the change makes the program behave differently. Mutation operators, which of these operators to apply? We saw 6 for that example the choice is up to us. Sometimes we might want to test for some parameters of the program then you choose mutation operators accordingly. Sometimes you might have audit requirements with say that you test for such requirements then you might have to apply all the recommended mutations, it completely depends on what is the current status of testing.



(Refer Slide Time: 13:25)



The slide has a blue header with the text "Mutants: Variants". Below the header, there is a bulleted list of four types of mutants. At the bottom left of the slide, there is a small circular logo with the text "NPTEL" below it.

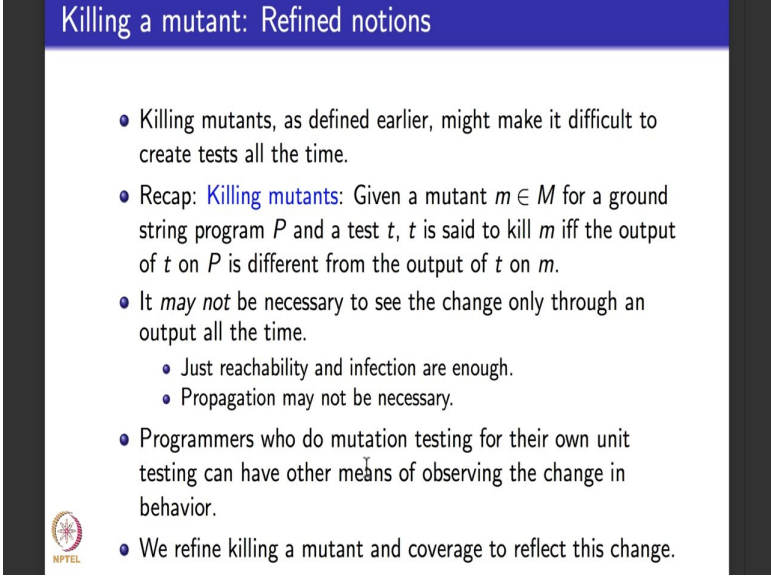
- **Stillborn mutant:** Mutants of a program result in *invalid* programs that cannot even be compiled. Such mutants should not be generated.
- **Trivial mutant:** Mutant that can be killed by almost any test case.
- **Equivalent mutant:** Mutants that are functionally equivalent to a given program. No test case can kill them.
- **Dead mutant:** Mutants that are valid and can be killed by a test case. These are the only useful mutants for testing.

Test cases have to be effective in the sense that they have to make the mutated program come up with the different output than the original ground string programs. And as I told you testing literature is studded with different mutation operators and testing for mutation operators. There are several variants of mutants that one can define. In fact, these terms might be hilarious for that how its lingered on in the mutant mutation testing community. So, you say mutant is still born if the mutants of a program result in invalid programs that cannot even be compiled. As I told you right, when I mutate a program I need another program that is a muted program to be valid, syntactically valid because I need to be able to execute my test case on that. But suppose I mutate in such a way that I get a program that is in tactically invalid and cannot even be compiled then that mutation mutant is what is called stillborn mutant. We do not really need such mutants, they are not useful for testing.

A trivial mutant is a mutant that can be killed by almost any test case. What do we mean by killed by almost any test case? That it is so easy that you do not have to put in much effort to make it different. It is almost easy to apply and it is not useful for testing how the program behavior changes. An equivalent mutant is mutant that is functionally equivalent. In the sense that if you taken the ground string program you made a change the ground string program resulted in a mutant but you no matter what you do you cannot find test case to kill it. In the sense that the ground string program and the mutated program always produce the same output on every possible test case, in which


case the mutant is called an equivalent mutant. There is one more kind of mutant which is called a dead mutant, these are mutants that are valid but they can be killed by almost any test case right. That can be not by almost any test case sorry, but they can be killed by some test case. These are the real mutants that are needed for our testing.

(Refer Slide Time: 15:12)



**Killing a mutant: Refined notions**

- Killing mutants, as defined earlier, might make it difficult to create tests all the time.
- Recap: **Killing mutants**: Given a mutant  $m \in M$  for a ground string program  $P$  and a test  $t$ ,  $t$  is said to kill  $m$  iff the output of  $t$  on  $P$  is different from the output of  $t$  on  $m$ .
- It *may not* be necessary to see the change only through an output all the time.
  - Just reachability and infection are enough.
  - Propagation may not be necessary.
- Programmers who do mutation testing for their own unit testing can have other means of observing the change in behavior.
- We refine killing a mutant and coverage to reflect this change.



So, we saw the notion of killing a mutant what it means? It means that given a program and a test case, I apply mutation to a program get a mutated program. If the test cases such that the behavior of the original program on the test case is different from the behavior of the mutated program on the test case, then you say that the mutant is killed. It may not be necessary to see the change only by observing the output all the time. If you remember we saw these reachability infection propagation and reveal model, RIPR model in the first week of this course. All that I want to know is that the program behavior of the ground string is different from the program behavior of the mutated program. I need not wait for the program to make, reflect the change all the way down the output. Maybe somewhere in between a program I can do a print test to see if the behavior at that point in this statement is actually different, that is all I am interested in.


It might be too much to expect that the if the behavior that is changed at that statement that was mutated in the program is propagates all the way to the output. It might be difficult to do that also. So, sometimes I require that it propagates all the way to the output, sometimes I am happy that the behavior at that mutated statement is different

from the ground string. We have to refine the notion of killing a mutant to understand this.

(Refer Slide Time: 16:36)

Killing a mutant and coverage: Refined notions

- **Strongly killing mutants:** Given a mutant  $m \in M$  for a ground string program  $P$  and a test  $t$ ,  $t$  is said to **strongly kill**  $m$  iff the output of  $t$  on  $P$  is different from the output of  $t$  on  $m$ .
- **Strong Mutation Coverage (SMC):** For each  $m \in M$ , TR contains exactly one requirement to strongly kill  $m$ .
- **Weakly killing mutants:** Given a mutant  $m \in M$  that modifies a location  $l$  in a program  $P$ , and a test  $t$ ,  $t$  is said to **weakly kill**  $m$  iff the state of the execution of  $P$  on  $t$  is different from the state of execution of  $m$  immediately after  $l$ .
- **Weak Mutation Coverage (SMC):** For each  $m \in M$ , TR contains exactly one requirement to weakly kill  $m$ .



So, we make killing the mutant strongly killing mutants and weakly killing mutants. Strongly killing is what we saw as killing a mutant earlier. Strongly killing mutant means given a mutant, for a ground string program  $B$  and the test case  $t$ ,  $t$  said to strongly kill the mutant if the output of the program is different from, the output of the mutated program is different from the output of the ground string program, which means propagation has happened all the way and revealing is also happened, the test case is managed to strongly kill it.

So, I change mutation coverage to strong mutation coverage which says for each mutant test requirement contains exactly one requirement to strongly kill that mutant. A weaker notion of killing a mutant which is what I explained to you know is that I might do a mutation by making one mutation operator to a program, but all that I want to know is that at that statement is there a change in the behavior of the program. I may not expect the change to propagate all the way to the output. That is what weakly killing a mutant captures. So, what is weakly mutant? Given a mutant that modifies a particular location  $l$  or a statement in the program and the test case  $t$ ,  $t$  as said to weakly kill them mutant if the state of the program  $p$  at, on applying  $t$  is different from the state of the mutated program immediately after that statement  $l$  at which mutation was applied. These are


clear? So, just as for strong mutation coverage we take mutation coverage and refine it to weak mutation coverage. What is weak mutation coverage say? For each weak mutant  $M$ ; TR contains exactly one requirement to weakly kill  $m$ .

(Refer Slide Time: 18:23)

Min method: Mutant 1

Consider the first mutant of Min method:

- Reachability: Always satisfied (True) as it is on the first statement.
- Infection: Value of A and B must be different;  $A \neq B$ .
- Propagation: Mutation version of Min must return an incorrect value, i.e., statement in the if block must not be executed;  $(B < A) = \text{false}$ .
- Full test specification:  $\text{True} \wedge (A \neq B) \wedge ((B < A) = \text{false}) \equiv (A \neq B) \wedge (B \geq A) \equiv (B > A)$ .
- Test case  $A=5, B=7$  will kill mutant 1. Original program will return 5, mutated version will return 7.



So, we will go back to the min method. So, I will show you the program and the first mutant. So, this was the min method. This is the first mutant that I applied basically take the statement min val equal to A; change it to B. Now I want to understand is this mutant strong can be strongly killed, can be weakly killed, is it a dead mutant, I want to understand what it is. So, what do I do I go ahead and do my reachability infection and propagation condition. Reachability means what I need to be able to reach that statement. If you go back for a minute and see I can always reach the statement because it is write the second statement, so, reachability is true. Propagation means what, I mean infection means what, that this test should be different because this is the only a test based on whether it passes or fails this statement will be executed right. If this passes the result is going to be different if this condition fails the result is going to be different.

So, that is what infection deals with reachability is always satisfied as it is in the first statement. Infection means the value of A must be different from B that is when the if condition will be tested. Propagation means mutated version of minimum must return an incorrect value that is the statement in the if block must not be executed. If it must not be executed means what the condition of the if statement B less than a must be false. So, the

full test specification for reachability infection and propagation says you just AND them, reachability is true, infection is  $A \neq B$ , propagation is  $B \leq \text{false}$ . It is simplified this logical predicate true and  $A \neq B$  gives me  $A \neq B$ ;  $B \leq \text{false}$  is the same as  $B \geq A$ . So, if and these 2 condition it means  $B \geq A$ ;  $B$  is not equal to  $A$  and  $B$  is greater or equal to  $A$ . So, the equal to  $A \neq B$  cancel out I just get  $B \geq A$ .


So, what it means it means that give 2 values which are inputs to the program  $A$  and  $B$  such that  $B$  is greater than  $A$ . Then you will be able to kill that mutant. So, some test case like this like for example,  $A$  is equal to 5,  $B$  is equal to 7 which satisfies this condition  $B$  greater than  $A$  will kill the first mutant right. Why, because the original program we will correctly return 5 which is the minimum value. Mutated program will return seven because I have change the assignment from  $B$  to  $A$  and the if condition then didn't pass. Is that clear please.

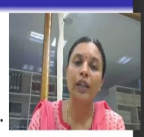
(Refer Slide Time: 20:57)


Min method: Mutant 3

The third mutant of Min method is an equivalent mutant.

- Intuitively, `minVal` and `A` have the same value at that statement in the program, so replacing one with the other has no effect.
- Reachability is true; infection condition is  $(B < A) \neq (B < \text{minVal})$ .
- It is also true that  $(\text{minVal} = A)$  (assertion).
- Simplifying, we get  $(A \neq \text{minVal}) \wedge (\text{minVal} = A)$ , a contradiction. This means that no value satisfying the conditions exist.







So, now let us look at third mutant we will go back and see what the third mutant is. So, here is the program and here is the third mutant. What is the third mutant do? It changes this if statement in particular it changes the predicate in the if statement if replaces if  $B$  less than  $A$  it replaces it with if  $B$  is less than `min val`. That is instead of doing a directly it replaces it with  $B$  is less than `min val`.

So, this is the original program or the ground string. In this program this statement replace like is the mutated program. Lets understand what are reachability infection and propagation conditions for that mutant I believe that it will not have that big a change because if you see just before that in the program the value min val was assigned to A right. So, min val and the variable A have the same value at that statement in the program, when that if statement is executed in the program. So, even normally you should understand that replacing one with the other in the if statement should not have any effect, which means what. Let us look at what how does it reflect on the reachability infection and propagation conditions. So, reachability as always is true because I can always reach that statement. What does infection condition mean? Infection condition is that the if should be test which means B should be less than A and the fact the B is less than A should not B the same as B been less than min val because the mutant replaces A with min val right.


Independently just from the statement before this we know that min val is equal to A, we know that min val is equal to A. So, what we mean by that? We know at because the statement holds if I simplify it what do I get? I will get a is not equal to min val from this part because B is less than A and it is not the same as B is less than being min val. And I also know from this assertion that min val equal to A. What do, if read this independently you see that A is not equal to min val and A is equal to min val. This is the contradiction right. This the contradiction logic means what it means that I cannot find any test case value that will do infection and propagation for this condition, which means this is an example of an equivalent mutant.

(Refer Slide Time: 23:25)

Another example

```
1  boolean isEven(int X)
2  {
3      if(X < 0)
4          X = 0-X;
5      if(float)(X/2) == ((float)X)/2.0
6          return(true);
7      else
8          return(false);
9  }
```

NPTEL





So, to understand reachability infection and propagation well, we will go through another small example. So, here is an example that tests if a number, given number, is Boolean or not. If it is Boolean it returns true otherwise it returns false. So, Boolean is a method I mean, is even is a method that returns a Boolean value, takes an integer as its argument. It first checks if X is less than 0, if X is less than 0 it makes it a positive number then it assigns X to 0 and then divides it by 2. If it divides it by 2 and every number should be divisible by 2, that is what this check condition, if it is divisible by 2 then it returns true.

Otherwise it returns falls this program is simple enough to be clear. Let us look at the reachability infection and propagation for one mutant in the program. What is this mutant? This mutant instead of keeping this original statement as X is equal to 0 minus X replace is with X is equal to 0.

(Refer Slide Time: 24:25)

Another example, contd.

- The mutant in line 4 is an example of weak killing.
  - Reachability:  $(X < 0)$ .
  - Infection:  $(X_T \neq 0)$ .
- Consider the test case  $X = -6$ . Value of  $X$  after line 4:
  - In the original program: 6
  - In the mutated program: 0.
- Since 6 and 0 are both even, the decision at line 5 will return true for both the versions, so propagation is not satisfied.
- For strong killing, we need the test case to be an odd, negative integer.

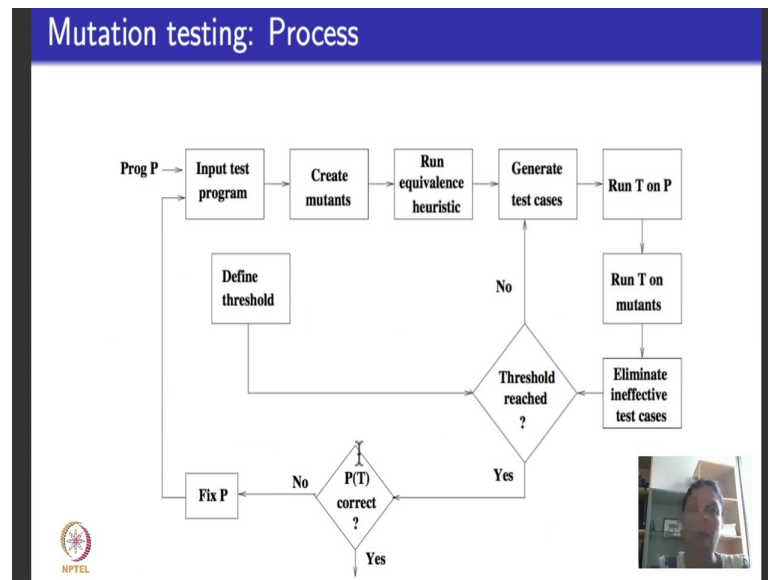


So, again what is reachability for this mutant. Reachability means this if condition should be passed. That is how I get to statement number 4. For this if condition to be pass value of  $X$  must be less than 0. So, that is what is returned here  $X$  is less than 0 for it to be infected  $X$  must not be equal to 0 why if  $X$  is equal to 0 then both 4 and the mutated version of 4 we will return the same value, but if  $X$  is a number that is not equal to 0 and in fact, less than 0 then mutated version of 4 we will make it 0, but the non mutated the original 4 we will not make it 0. So, infection is that.

So, let us consider a test case of the form  $X$  is equal to minus 6 right. So, in the original program the value will be 6. In the muted program the value will be 0. But then if you see both 6 and 0 are even numbers which means both of them will become divisible by 2 at line number 5 in which case the program will return true for both versions, I want the program to return false value for one version. How do I do that ? I have to give odd negative integer. If I give an odd negative integer then here there will be infection and propagation will also happen because the condition in line number 5, will return 0 as even number because of the mutated version of the program whereas, in the original version that it will be still be an odd positive number which is not divisible by 2. So, it will return it as false. So, far reachability infection and propagation which results in strong killing of mutants, I want the test case to be odd negative integer.



(Refer Slide Time: 26:06)




So, here is the summary slide that explains the process of mutation testing. I begin with the program P, my ground string which is my input program to be tested by using mutation. I create one or more mutants for the program. How do I create mutants? I have a standard list of mutation operators which I will be telling you in the next module. Using one or more of those mutation operators, I create different mutants. Please remember that per one mutant I have to use only one mutation operator, it is not recommended that you use more than one mutation operator. Now if the mutant turns out to be equivalent to the original program then there is no point in trying to test it. So, usually there are these things called equivalence checkers which return yes or no for a large number of cases which basically tell you if the mutated program is the same as the original program.


If it is then there is no point. If it is not then I generate test cases and I run the test cases on the program first, then I run the test cases on the mutated program. Because they are not equivalent the output should be different. If the outputs are not different because propagation is not achieved then the test case is called ineffective, I eliminate it. And I will usually have a threshold that is defined independently by let us say a quality auditor or somebody which tells you how many mutants to test the program for. If I have reached my threshold then I finish, exit. If there are any errors that I found then I fix the program and I go back. If I have not reached my threshold then I generate more test cases and repeat this process. So, this is how broad level mutations testing works step by step.

(Refer Slide Time: 27:48)

### Mutation testing for source code: Summary




- Mutation testing for source code is one of the most powerful forms of testing source code.
- We need to generate an *effective* set of test cases.
- Next lecture: Mutation operators for typical source code.




So, here is the summary of mutation testing for source code. Its one of the most powerful forms of testing source code. In later this week I will tell you how mutation operators and mutation testing subsume several other coverage criteria that we have seen. It is very powerful for both unit testing and for integration testing. And then the biggest problem is we need to pick up an effective set of mutants that will give us an effective test setup test cases such that if the original program  $P$  has an error using mutants we will be able to identify the error.

(Refer Slide Time: 28:21)

### Mutation testing for source code: Summary



- Mutation testing for source code is one of the most powerful forms of testing source code.
- We need to generate an *effective* set of test cases.
- Next lecture: Mutation operators for typical source code.



What I will do in the next lecture is I will tell you about typical mutation operators that you can use for programming languages like C or Java and that will help you to define mutated programs from ground strings.

Thank you.