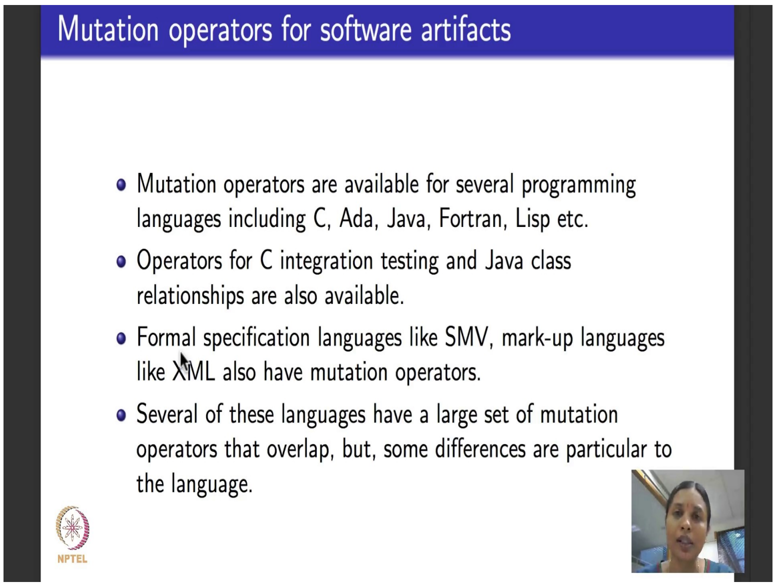
Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 38
Mutation testing: Mutation operators for source code

Hello again welcome to the fourth lecture of week 8. If you remember we were been doing mutation testing throughout this week. Last lecture I took an example of a small program, we did two methods and I told you how to apply mutation to particular statements in the program, how to write test cases that would kill those mutants. We distinguish between two notions of killing strong feeling and weak killing. We saw examples of both kind, we also saw examples of an equivalent mutant for the first method that we saw. How are these mutation operators obtained, how these statements are mutated? These statements are mutated by using an underlying set of mutation operators that are available for several different programming languages.

(Refer Slide Time: 01:02)



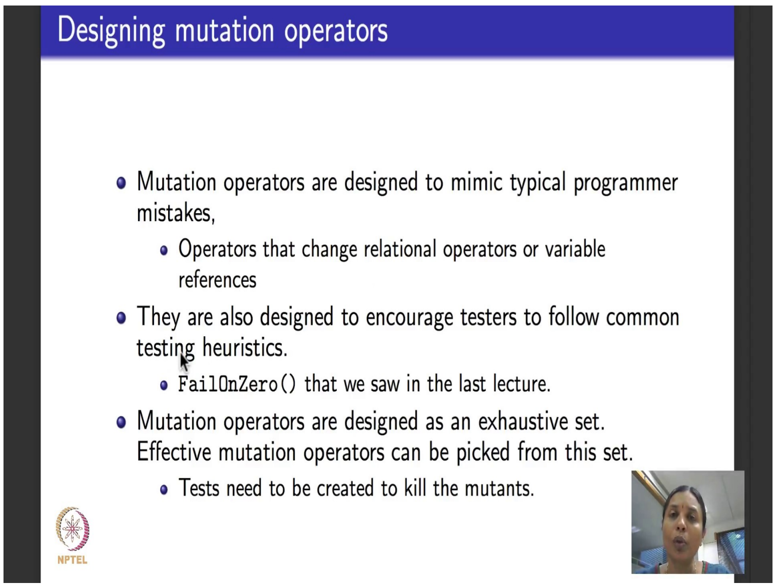
Mutation operators for software artifacts

- Mutation operators are available for several programming languages including C, Ada, Java, Fortran, Lisp etc.
- Operators for C integration testing and Java class relationships are also available.
- Formal specification languages like SMV, mark-up languages like XML also have mutation operators.
- Several of these languages have a large set of mutation operators that overlap, but, some differences are particular to the language.

So, the focus of this course is to understand mutation operators. As I have been telling you mutation operators are available for many programming languages like C, Java, Ada, Fortran, Lisp and so on. They are available for integration testing of C which focuses purely on procedures calling each other. They are available for integration testing of Java which focuses on class integration testing, methods calling each other and so on which we will also see.

Then we will come to specification, mutation operators are available for formal specification and modeling languages like SMV and NuSMV. They are available for markup languages like XML which we will see as a part of this course. Some, each of these languages we know is different, it forms slightly different subset, has different syntax and more or less the same set of programs can be written with any programming language. So, what we will see today is like a common set or a generic set of mutation operator that cuts across all the different programming languages and we will also see mutation operators that apply at the individual statement level within a program.

(Refer Slide Time: 02:06)



Designing mutation operators

- Mutation operators are designed to mimic typical programmer mistakes,
  - Operators that change relational operators or variable references
- They are also designed to encourage testers to follow common testing heuristics.
  - `FailOnZero()` that we saw in the last lecture.
- Mutation operators are designed as an exhaustive set. Effective mutation operators can be picked from this set.
  - Tests need to be created to kill the mutants.

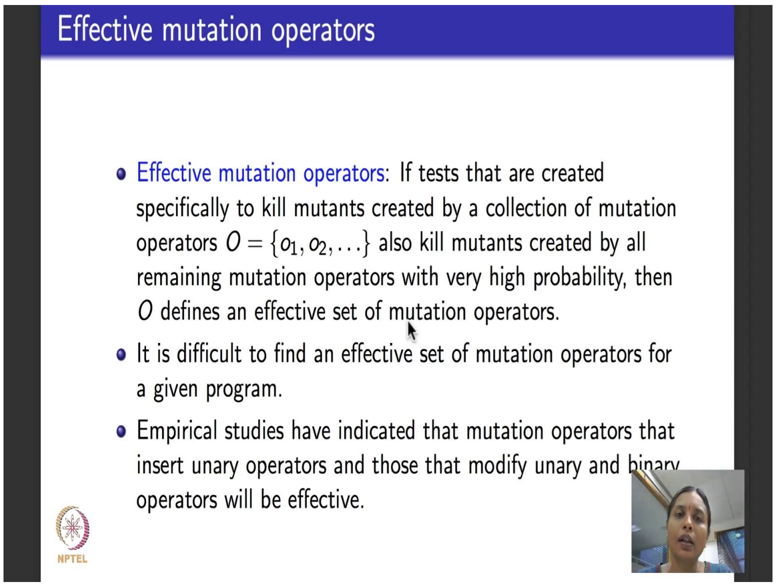
So, these are not the operators that are going to focus on integration testing. As I told you here they are going to focus on unit testing.W we are going to mutate individual statements in the program by making small changes to each statement in a program as and when it is necessary to test the execution of a program. What are these mutation operators designed for? Typically programmers when they unit test their code, why do they do unit test? It unit testing helps them to detect simple mistakes that these programmers make themselves. For example, instead of writing less than or equal to they could have written less than, instead of writing array index as beginning from 0 they could have written as array index as beginning from one and so on. So, these are simple programmer mistakes.

And what we are going to see is a reasonably exhaustive list of mutation operators that try to mimic or mutate to mimic these simple programmer mistakes. These could be operators that change relational operators as I told you ,maybe he suppose to do less than or equal to, but instead the programmer did less than. Or maybe greater than or equal to instead he did greater than, or maybe he had to do i plus plus instead of that he did j plus plus so that he changed variable name or a variable reference.

So, we are going to see mutation operators that mimic these kind of mistakes they make small changes that mimic these mistakes and see how the program behaves if the mutant can be killed or not. They are also a design to encourage testers to follow common testing heuristics. What do we mean by that? One common testing heuristic which every individual tester is suppose to do see is to see how the individual method that he or she has written, where in the variables in that method become 0, is it fine. Because maybe somewhere in the code there is a division by 0 or there is division by some number that could potentially become 0. So, every programmer the ones on each programmer to test their code to see if all exceptions including variable names being 0 are being handled correctly.

So, for that we will see a specific mutation operator which we already used in the last lecture called fail on 0. Mutation operators that we will see will be fairly exhaustive in the sense that it will be like a laundry list of all possible different mutations that you can do to a particular statement. Typically most of the times it is not the case that a programmer has to use each of the kind of mutation operators that we will be presenting today to be able to test his or her code. But as a list the onus on these tools and on us to be able to provide an exhaustive list. We never know which of these operators from the list is going to be useful to a programmer for his or her code. But this list is exhaustive, but the use is not meant to be exhaustive. Use is meant to be selective as and when needed for specific pieces of code for unit testing.

(Refer Slide Time: 05:26)



Effective mutation operators

- Effective mutation operators: If tests that are created specifically to kill mutants created by a collection of mutation operators $O = \{o_1, o_2, \ldots\}$ also kill mutants created by all remaining mutation operators with very high probability, then O defines an effective set of mutation operators.
- It is difficult to find an effective set of mutation operators for a given program.
- Empirical studies have indicated that mutation operators that insert unary operators and those that modify unary and binary operators will be effective.

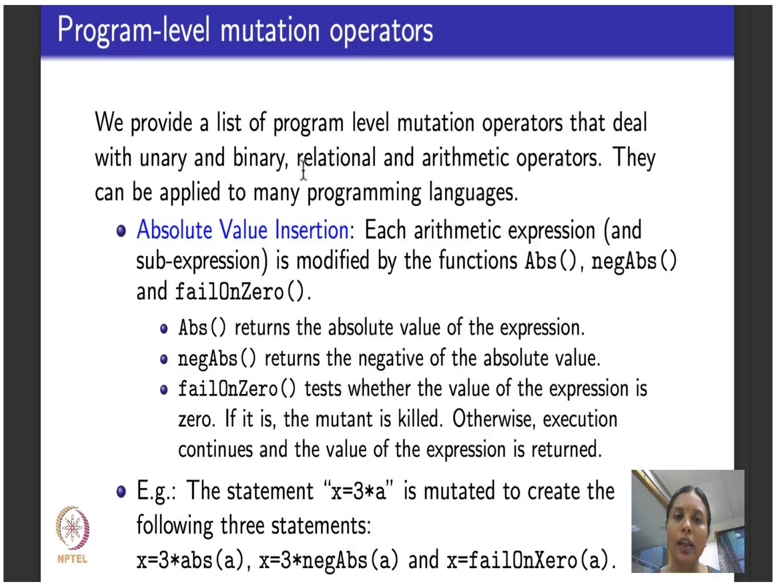
And while mutating a program and testing it the programmer has to create tests to be able to kill the mutant, either strongly kill it or weakly kill it. So, when I said in this exhaustive list of mutation operators, how will a test of pick which are effective mutation operators? How do I know which set of operators are effective. So, here is a definition of effective mutation operators. It says if tests are created specifically, let us say you start with a collection of mutation operator, call that set as the set O, given as O 1, O 2, O3 and so on, some finite set of mutation operators. You have made these mutations one at a time to the given piece of program and you are writing test cases to kill these mutation operators.

If the test cases that you have killed, written to kill, these mutation operators also happened to kill other unused mutation, the remaining mutation operators with very high probability. You may or may not know whether they were actually killed, but let say they happen to kill the other remaining operators with very high probability, then you say that the underlying set of mutation operators that a programmer has to picked is turning out to be effective. So, just to repeat it, a programmer has a method with him and now the idea is to apply some set of mutation operators to test that individual method written by the program. Let say the programmer picks let say three different set of mutation operators to apply from the exhaustive listing.

So, when do we call these 3 mutation operators is being effective? Let us say the programmer after testing with a reference to these 3 mutation operators also happened to realize that these 3 mutation operators also happen to kill other mutation operators that he did not apply, then you say that the 3 mutation operators chosen by the programmer is an effective set of mutation operators. So, it will be a great thing if per se given a method or a piece of language, if you could have algorithms that determine right up front which are the effective set of mutation operators. Unfortunately it is a difficult problem , an NP complete or sometimes an undecidable problem to be able to determine an effective setup mutation operators, even at the level of an individual method.

But typically various empirical studies in software engineering have indicated that mutation operators of this kind, those that insert unary operators, and those that modify unary and binary operators, have proven to be very effective from the point of view of this definition. So, that is the list that we are going to see in this lecture today.

(Refer Slide Time: 07:59)



Program-level mutation operators

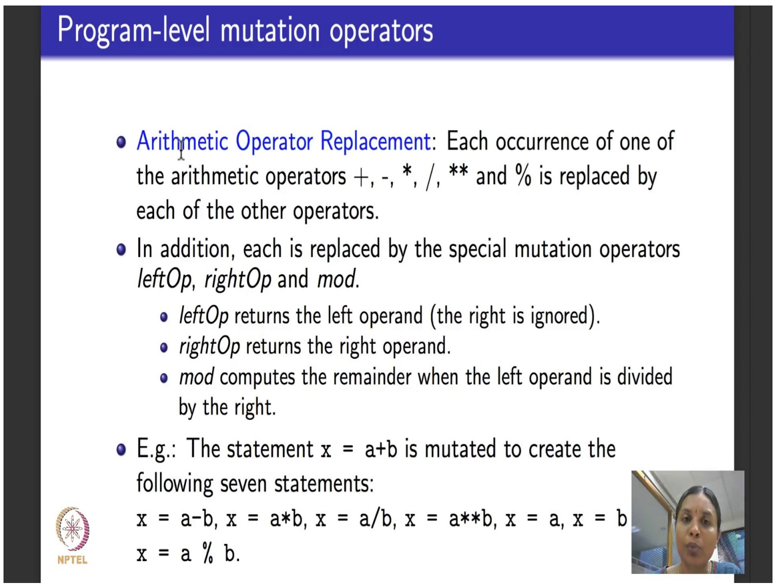
We provide a list of program level mutation operators that deal with unary and binary, relational and arithmetic operators. They can be applied to many programming languages.

- Absolute Value Insertion: Each arithmetic expression (and sub-expression) is modified by the functions `Abs()`, `negAbs()` and `failOnZero()`.
  - `Abs()` returns the absolute value of the expression.
  - `negAbs()` returns the negative of the absolute value.
  - `failOnZero()` tests whether the value of the expression is zero. If it is, the mutant is killed. Otherwise, execution continues and the value of the expression is returned.
- E.g.: The statement "`x=3*a`" is mutated to create the following three statements: `x=3*abs(a)`, `x=3*negAbs(a)` and `x=failOnXero(a)`.

So, what we do now is that I will provide a list of program level mutation operators. What do they deal with? They deal with unary, binary, both relational and arithmetic and logical operations. They cut across several different programming languages. They can be applied for C, for Java or for any other programming language that has a similar syntax. And I will be giving you an exhaustive list of mutation operators, there will be so many different ones. Please remember that you need not apply, given a piece of program or a method you need not apply every kind of mutation operators. Choose a small subset from this such that you can make use of this list of mutation operators to be able to effectively mutate your program in test.

It is useless to pick everything from this list because it is a fairly large list. So, what are we what are the exhaustive large list of mutation operators that at the program level that we are going to see. So that is what will the rest of the lecture be. So, I will give each of these collection of mutation operators that we are going to see a name like this. That name will be written in blue in every slide. So, it just tells you what is the kind of mutation that we are going to apply. The first collection of mutation operator that we are going to see is what is called absolute value insertion.

(Refer Slide Time: 09:19)

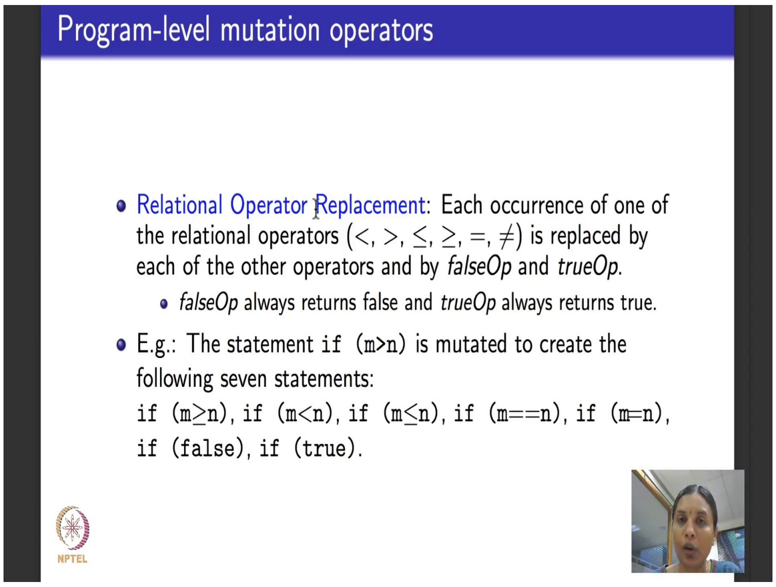


Program-level mutation operators

- Arithmetic Operator Replacement: Each occurrence of one of the arithmetic operators +, -, *, /, ** and % is replaced by each of the other operators.
- In addition, each is replaced by the special mutation operators *leftOp*, *rightOp* and *mod*.
  - *leftOp* returns the left operand (the right is ignored).
  - *rightOp* returns the right operand.
  - *mod* computes the remainder when the left operand is divided by the right.
- E.g.: The statement `x = a+b` is mutated to create the following seven statements:
  `x = a-b, x = a*b, x = a/b, x = a**b, x = a, x = b`
  `x = a % b.`

We will see what it is. Later I will tell you set of mutation operators called arithmetic operator replacement, we will after that do relational operator replacement, after that do conditional operator replacement, after that do shift operator replacement and so on, we will see a fairly large list.

(Refer Slide Time: 09:25)

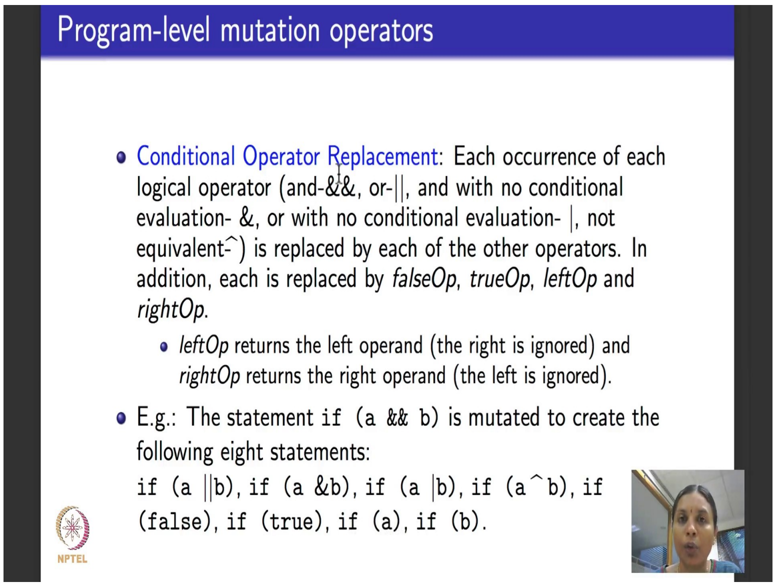


Program-level mutation operators

- Relational Operator Replacement: Each occurrence of one of the relational operators ($<$, $>$, $\leq$, $\geq$, $=$, $\neq$) is replaced by each of the other operators and by *falseOp* and *trueOp*.
    - *falseOp* always returns false and *trueOp* always returns true.
- E.g.: The statement `if (m>n)` is mutated to create the following seven statements:
  `if (m≥n)`, `if (m<n)`, `if (m≤n)`, `if (m==n)`, `if (m=n)`, `if (false)`, `if (true)`.

So, each slide will have one collection of mutation operators that belong to one particular category, and remember typically when you test a program only a small number from this exhaustive set is enough to test a program.

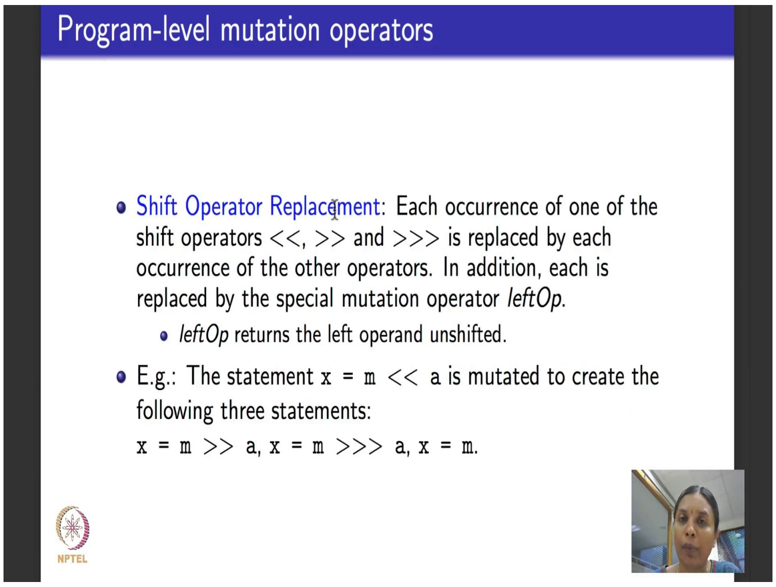
(Refer Slide Time: 09:29)



Program-level mutation operators

- Conditional Operator Replacement: Each occurrence of each logical operator (and-&&, or-||, and with no conditional evaluation- &, or with no conditional evaluation- |, not equivalent-^) is replaced by each of the other operators. In addition, each is replaced by *falseOp*, *trueOp*, *leftOp* and *rightOp*.
    - *leftOp* returns the left operand (the right is ignored) and *rightOp* returns the right operand (the left is ignored).
- E.g.: The statement `if (a && b)` is mutated to create the following eight statements:
  `if (a ||b)`, `if (a &b)`, `if (a |b)`, `if (a ^b)`, `if (false)`, `if (true)`, `if (a)`, `if (b)`.

(Refer Slide Time: 09:32)



Program-level mutation operators

- Shift Operator Replacement: Each occurrence of one of the shift operators <<, >> and >>> is replaced by each occurrence of the other operators. In addition, each is replaced by the special mutation operator *leftOp*.
  - *leftOp* returns the left operand unshifted.
- E.g.: The statement `x = m << a` is mutated to create the following three statements: `x = m >> a`, `x = m >>> a`, `x = m`.

So, we begin with our first set which we call absolute value insertion. So, what we do here? Let us take a statement, if you look at this example here down the slide, let us take a statement like x is equal to 3 star a. It is a normal assignment statement. What I am going to do is that I am going to change this to replace the variable a, with it is absolute value call it as abs of a, I am going to replace the variable a with a negation of its absolute value this is called neg abs of a, and I am going to replace a with 0 through the mutation operator fail on 0. So, for this statement which looks like this a is an absolute value that a is a variable, whose which, for which I am mutating by considering 3 possible different mutation operators. Instead of a I take the absolute value of a that is this first mutated statement that I get. The second one instead of a, I take the negation of the absolute value of a that is the second mutated statement. The third one instead of a, I do fail on 0 of a which makes a as 0. So, those are the 3 mutation operators.

So, this I can do for any statement and given any program. So, I say whenever I find an arithmetic expression or a sub expression, and I find a variable in that arithmetic sub expression then I can apply 3 different possible mutations to that variable. One possible mutation returns the absolute value of that expression or the variable, one possible mutation returns the negation of the absolute value of the expression of variable, the third possible mutation tests whether the value of the expression is 0 or not. If it is happen to be 0 then we say that the mutant fail on 0 is killed, otherwise the mutant is not killed, the execution continues and the value of the expression as it was originally in the program is returned.

So, the next set of mutation operators that we are going to consider belong to this category of operators called arithmetic operator replacement. If you see in a typical programming language, what are the arithmetic operators that are there? There is addition, plus, subtraction, minus, multiplication, star, division, exponent, mod. So this is an reasonably exhaustive list of possible arithmetic operators. Now if you go down and look at the example, let us say there is a statement that look like this: x is equal to a plus b. It has one arithmetic operator in it, this is the plus. So, I could create 7 different mutations of the statement based on my need, where I replace this arithmetic operator plus with either minus or star or slash, which is division or exponent. Or there are two other special things where I knock off one of the operands, I retain only the left operand, here I retain only the right operand or I could replace this arithmetic operator with a mod. So that is what it says. It says take any statement which has an arithmetic operator in that each occurrence. So, this example the arithmetic operator that we begin with this plus, but it could be the case that the statement original statement was x is equal to a minus b, in which case you choose something else in this list of arithmetic operators to mutate it with, I hope that is clear.

So, what it says is that the statement or the expression could have any of these arithmetic operators. So take that arithmetic operator out replace it with some other arithmetic operator from this list based on you need. Two special kind of other mutations can also be done. One mutation is called leftOp, what is it do? It returns only the left operand, the right operand is ignored that is what we did here for this example. If I had a statement x is equal to a plus b , if I apply the mutation leftOp it results in this, x is equal to a. That is the change that I make the statement to see what would the program do. Similarly rightOp returns just the right operand. So, if I had the left one is ignored. So, if I had x equal to a plus b and I apply rightOp, then I get x is equal to just b, a is ignored and what is mod do mod computes the reminder when the leftOp is divided by the rightOp, the standard thing. So, is that clear.

So, in summary what we saying there could be statements in the program, which have arithmetic operations in them. One possible way to mutate such statements would be replace that arithmetic operator that you find in that expression in that statement. If it is a plus you could replace it with minus, star, slash, mod and so on. If it is a minus you could replace it with plus, star, slash, mod and so on. So, this collection of mutation operators put together is called arithmetic operator replacement. Moving on, just as arithmetic operators, you could do relational operator replacement. Again what are the exhaustive list of relational operators that are available in programming languages? There is less than, there is greater than, less than or equal to, greater than or equal to, equal to and not equal to.

Let us say there was an expression that look like this. I had an expression which says m is greater than n, I had a predicate sorry which says m is greater than n. My goal, may be I do not know may be the programmer should have written m greater than or equal to n. So, I want to mimic and see is this a mistake paid by the programmer. So, I mutate the predicate m greater than n by replacing the relational operator greater than with greater than or equal to. So, or maybe sometimes the programmer intended to do less than. So, I could mutate the predicate if m greater than n by replacing this relational operator greater than with the relational operator less than. So, what it says is that any occurrence, each possible occurrence of one of the relational operators, which is this list, 6 of them could be replaced by each of the other relational operator. In addition you could do two special mutation operations: one is called falseOp, one is called trueOp. What is falseOp do? FalseOp will replace this whole predicate with false, trueOp will replace this whole predicate with true.

For example if I had an if statement that look like this: if m is greater than n, then I could create seven different mutations from that. I could replace this greater than relational operator with great than or equal to create this statement if m greater than or equal to n, or I could replace greater than with less than to create the statement, greater than with less than or equal to, greater than with equal to, greater than with not equal to, I am sorry this should be not equal to, then greater than be just false, maybe it is a contradiction, it useful to test it for this, and greater than, m greater than n be just true to test for being a tautology. So, is that clear please. So, what are we saying we say that if I have a predicate, a logical continuing a relational operator, I could create a mutant of that predicate or a clause by replacing the actual relation in the relational operator by any of the other ones that are available. Two special ones, I could replace the whole clause by true, I could replace the whole clause by false.

Moving on, similarly I can do conditional or logical operator replacement. So, what is conditional or logical operator replacement? So, what are the various logical operators? They are and, they are or, and with no conditional evaluation, denoted by a single ampersand, an or with no conditional evaluation which is denoted by a single vertical bar, and not equal to which I denote by this caret symbol. So, I could replace each of these with the other to create so many different mutants. In addition, I also have 4 special mutation operators. What are the special mutation operators? True and false, which are very similar to this one that we saw here, false returns false, true returns true and I have leftOp and rightOp which were similar to the one that we saw in the arithmetic expression rightop and leftOp. So, true returns trueOp returns true, falseOp returns false, leftOp returns the left operand, the right operand is ignored, righOp returns the right operand and the left operand is ignored.

So, for example, if I had a predicate that looked like this and an if statement with that predicate. Let say if a and b then I could mutate the predicate by replacing this relational operator and with any of the other ones. So, I could replace and with an or to get this, I could replace and with, an and with no conditional evaluation to get the second expression here, I could replace this and with an or with no with conditional evaluation to get the third one. This is replacing and with not equivalent, this is the fourth one, this is replacing and with false, this replacing a and b with true, this is knocking of the b from a this is knocking of the a from a and b. I hope this is clearly right. It is a fairly routine thing, I take one operator of one kind I can substitute it with any other operator of any other kind. Now you might ask why cannot I replace this and with e with let say plus? That would not make sense right. Why, because the type of a and b is meant to be Boolean and I should replace one operator with an another operator of the same category. Please remember that the mutants that I create from programs still have to be valid in the sense that the resulting program has to compile.

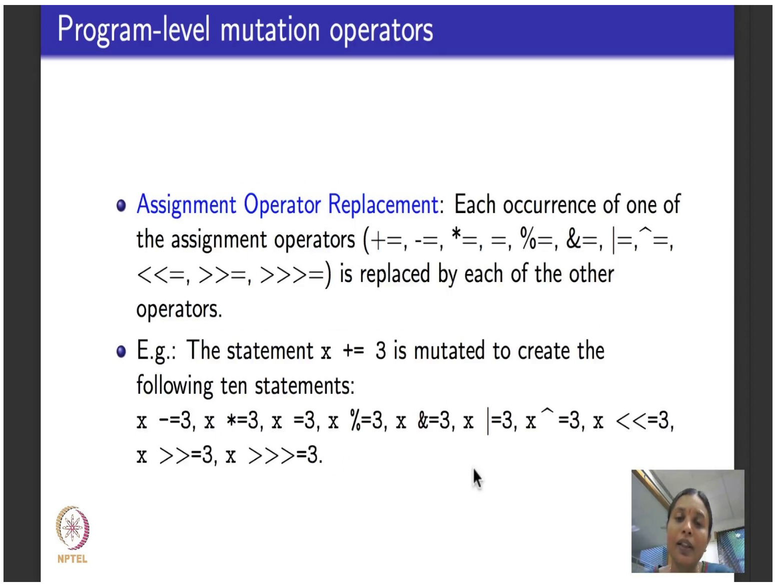
So, arithmetic operators that we saw here can be replaced with other arithmetic operators only. Conditional operators, relational operators can be replace to the other relational operators only. Conditional operators can be replace with conditional operators only and so on for all the other groups that we are going to see for the remaining part of this lecture also. So, the next set of mutation that you could have do is regarding the shift operators. You might have seen this left shift, right shift and right shift has two categories signed and unsigned. So, this is the left shift operator, this is the signed right shift operator, this is the unsigned right shift operator. Unsigned right shift is the counterpart of left shift, signed right shift is meant to take care of negative numbers also.

So, each occurrence of these shift operators can be replace by the other occurrence. In addition I have this special mutation operator which just returns the left operand which is unshift. So, if I had a statement like this which says x is equal to m left shift a, I can mutate this statement to create 3 different mutations. I can replace this left shift with unsigned, signed right shift, I can replace with this left shift with unsigned right shift or I can replace this left shift by knocking of the left shift and the a, do not shift and return only m. So, these are 3 mutations that are possible when you have statements in your program that deal with shift operations.

Moving on, suppose you had statements in your program that deal with bitwise logical operators. That is, they work a lot like and an or, but when given a sequence of bits or binary numbers, they and, and or, or exclusive or, bit by bit by applying the same truths table for and or. So, each occurrence of each of the bitwise operators the bitwise and the bitwise or the bitwise exclusive or, can be replace by each of the other operators when we do mutation operators here. In addition I can also do leftOp and rightop like I did for arithmetic and other categories. LeftOp always returns the left operand right one is ignored, rightop returns the right operand left one is ignored.

So, suppose I had a statement like this: x is equal to m bit wise and n, then I can create 3 possible mutants of the statements. I could replace this bitwise and by bitwise or, this is a first mutant, I could replace this bitwise and bitwise exclusive or, the second mutation, I could knock off the second left, the right operand and consider only x is equal to m, I could knock off the left operand and consider only x is equal to n.

(Refer Slide Time: 22:16)



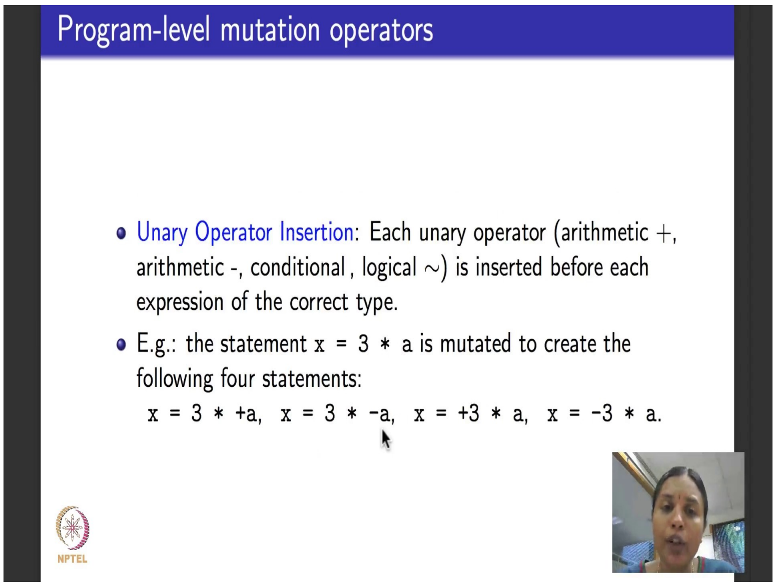
Program-level mutation operators

- Assignment Operator Replacement: Each occurrence of one of the assignment operators (+=, -=, *=, =, %=, &=, |=,^=, <<=, >>=, >>>=) is replaced by each of the other operators.
- E.g.: The statement x += 3 is mutated to create the following ten statements:
  x -=3, x *=3, x =3, x %=3, x &=3, x |=3, x^=3, x <<=3, x >>=3, x >>>=3.

So, these kind of mutations are possible dealing with bitwise operations. Similarly I could also consider these assignment operators plus equal to, minus equal to, star equal to, slash equal to, percentile equal to and so on and I could replace each occurrence of each kind of assignment operator with the other occurrence.

For example if I had a statement which said x plus is equal to 3, I can mutate it to create ten different statements. I could say x minus equal to 3, x star equal to 3, x slash equal to 3, x slash equal to 3, x a mod equal to 3 and so on and so forth.

(Refer Slide Time: 22:49)



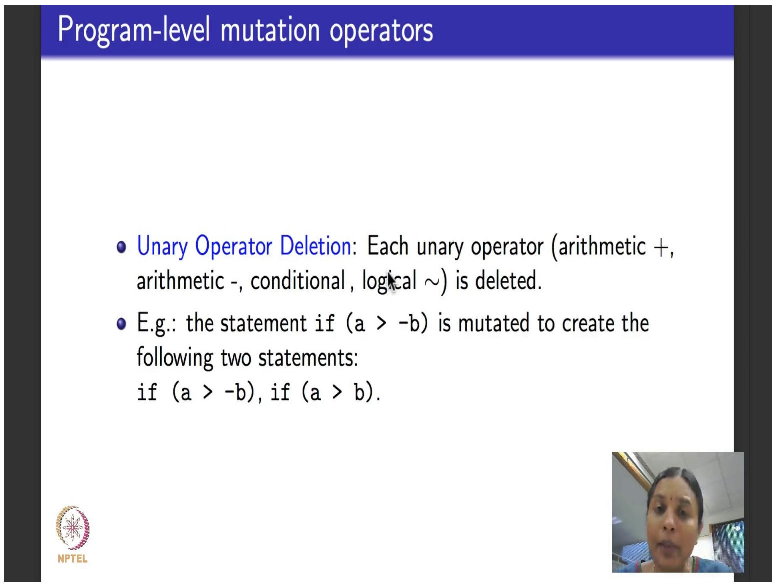
Program-level mutation operators

- Unary Operator Insertion: Each unary operator (arithmetic +, arithmetic -, conditional , logical ~) is inserted before each expression of the correct type.
- E.g.: the statement x = 3 * a is mutated to create the following four statements:
  x = 3 * +a, x = 3 * -a, x = +3 * a, x = -3 * a.

Now the other kind of mutation that I can do is unary operator insertion. What do we mean by that? Let us begin by understanding the example. Suppose there is a statement which says x is equal to 3 into a. Now I can replace this a with minus with plus a, create one mutation, this mutation turns out to be equivalent to this because if a is positive then plus a has the same sign as a, if a is negative also then the same thing. So, I list this for the sake of completeness, but these two are equivalent. Then I could replace this a with minus a create another mutation.

Similarly, I could replace 3 with plus 3. Here again it is just an equivalent mutant might as well not created because it gives the same expression, but it listed here for be the for the sake of being exhaustive. I could replace this 3 with a minus 3 to create such an expression. So, that is class of mutation operators is what is called unary operator insertion. What do I do here? Each unary operator which is the arithmetic plus, the arithmetic minus, the conditional and the logical is inserted before each expression of the correct type. Here I had arithmetic expression. So, I am considering only the arithmetic unary operators. Later I could consider the logical or the conditional unary operators.

(Refer Slide Time: 24:07)



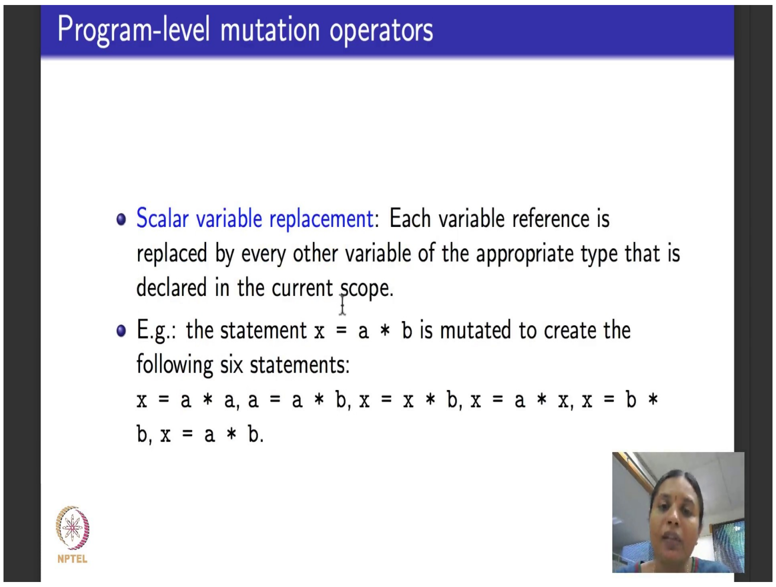
Program-level mutation operators

- Unary Operator Deletion: Each unary operator (arithmetic +, arithmetic -, conditional , logical ~) is deleted.
- E.g.: the statement `if (a > -b)` is mutated to create the following two statements:
  `if (a > -b)`, `if (a > b)`.

Just as replacing, I could delete a unary operator all together. Each unary operator can be deleted, it could be the plus, it could be the minus, it could be a conditional operator, it could be a logical negation I can delete it.

For example, here I say if a is greater than minus b, I could say a is greater than minus b which is the same as this or I could remove the minus b and say a is greater than b. So, I could remove that. Now this is a very useful thing which we also used in the example that we saw in the last section. There are variables where I find an expressions throughout my piece of program.

(Refer Slide Time: 24:33)



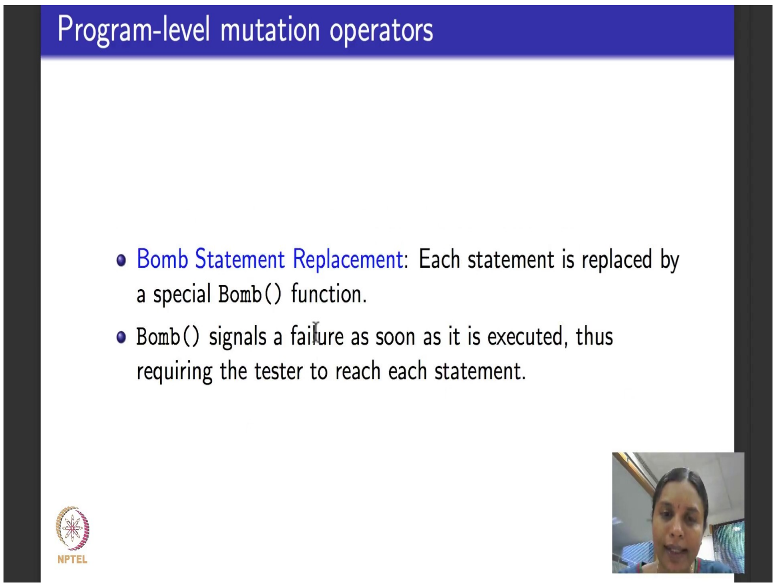
Program-level mutation operators

- Scalar variable replacement: Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.
- E.g.: the statement `x = a * b` is mutated to create the following six statements:
  `x = a * a`, `a = a * b`, `x = x * b`, `x = a * x`, `x = b * b`, `x = a * b`.

And those variables are called scalar variables because they are not as vectors, there is one individual variable. What I can do is I can mutate the program by considering each variable reference and replace it with some other variable reference such that the type matches. So, the expression continues to get evaluated and not only the type should match, the other variable reference that I replace this with should be within the scope, that is what is true this thing. I cannot pick up a another variable far away somewhere else in the program and suddenly substitute it here. It should be within the scope because if it is within this scope then things like declaring the variable and all other things will be taken care of, and mutating this way is effective than replacing with a random undeclared variable.

So for example if I had a statement x is equal to a star b, then what can I do? I can take this and replace this b with a to create an expression like this, x is equal to a star a. I could replace the x with a to create another expression like this, I could replace the a with x to create this third expression. I could replace b with x to create the forth expression, I could replace a with b to create this expression and finally, I retain the same thing, is that clear. There is no point now in picking up a completely new variable let say c that is out of the scope of this statement and replace x is equal to c in to b maybe the program will complain saying c is not been declared and so on. So, you do not want to risk. Remember programs have to compile, they have to be valid. So, when I do always replacement I am always replace variables such the type of the variables match at sorry and the variable that is declared is within the current scope.

(Refer Slide Time: 26:29)

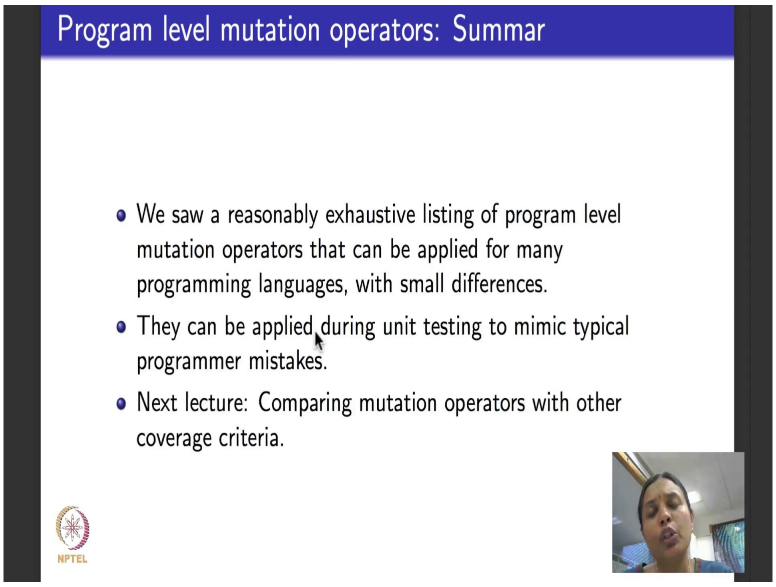


Program-level mutation operators

- Bomb Statement Replacement: Each statement is replaced by a special Bomb() function.
- Bomb() signals a failure as soon as it is executed, thus requiring the tester to reach each statement.

Finally we saw this last time. There is a special mutation available called bomb statement what is this bomb statement do? Bomb signals a failure as soon as it is reached. It is like making the program fail at that statement and see what happens. This is another common testing heuristics which many companies that recommend unit tester has to do. So, they say reach each statement. So, one way to make sure that the program execution has reached a particular statement is to replace that statement with a bomb. When the program reaches that statement, it will terminate. So, you know that the program is reached that statement.

(Refer Slide Time: 27:07)



Program level mutation operators: Summar

- We saw a reasonably exhaustive listing of program level mutation operators that can be applied for many programming languages, with small differences.
- They can be applied during unit testing to mimic typical programmer mistakes.
- Next lecture: Comparing mutation operators with other coverage criteria.

So, bomb is a very useful mutation operator. So, what did we do in this lecture? We saw an exhaustive list. Please remember the word is exhaustive, there is no point in considering each and every mutation operator that we saw today for every program that you test for. You should carefully select from this list, which is the one that I want to test in my program, which is the property, which is the statement.

If I want to test this statement for one particular property which would be a good mutation operated to pick from the list that we learnt today, that is how you should use it. And typically it is applied during unit testing to mimic typical programmer mistake as I told you. In the next lecture what I will be telling you is we saw mutation testing and one question that might naturally arise in your mind is, how does mutation testing compare to graph testing or how does it compare to logical predicate testing because they were also white box testing techniques right. So, that is what we will see, we see that mutation testing in fact is very powerful, it subsumes a lot of criteria from graph base testing and from logical base testing.

Thank you.