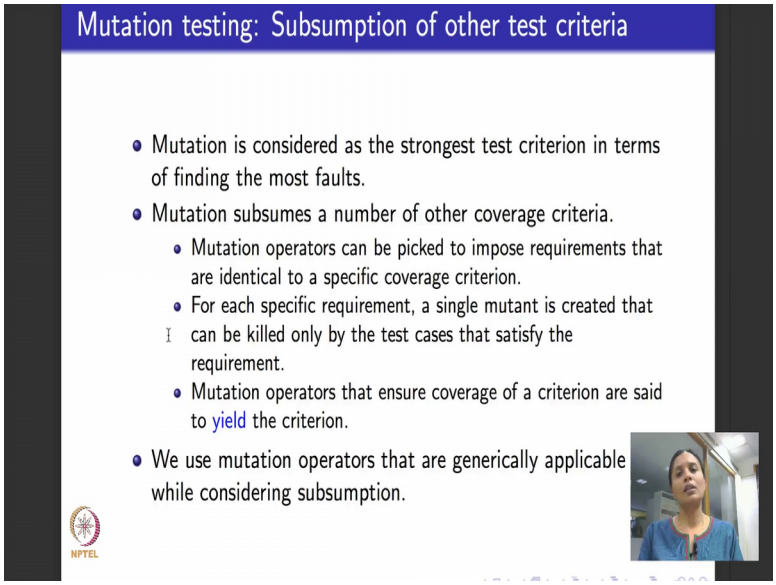
Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 39
Mutation testing vs. graphs and logic based testing

Hello there, we are in the last lecture of week 8. We will be doing mutation testing. I told you began this week by introducing you to regular expressions and grammars. Then we introduce generic terms and mutation testing. Then we applied mutation testing to source code we saw 2 examples of that, in the last lecture I gave you a reasonably exhaustive list of mutation operators, that you could use to mutate a program for doing unit testing within a method within a piece of code, that lecture make changes to the statements in a program.

Next week we will apply mutation testing for design integration. But before we do that as I ended my last lecture as I told you, we saw mutation testing before that we saw graph testing logical predicate testing. How does mutation testing compared to all the other testing criteria that we saw?

(Refer Slide Time: 01:08)

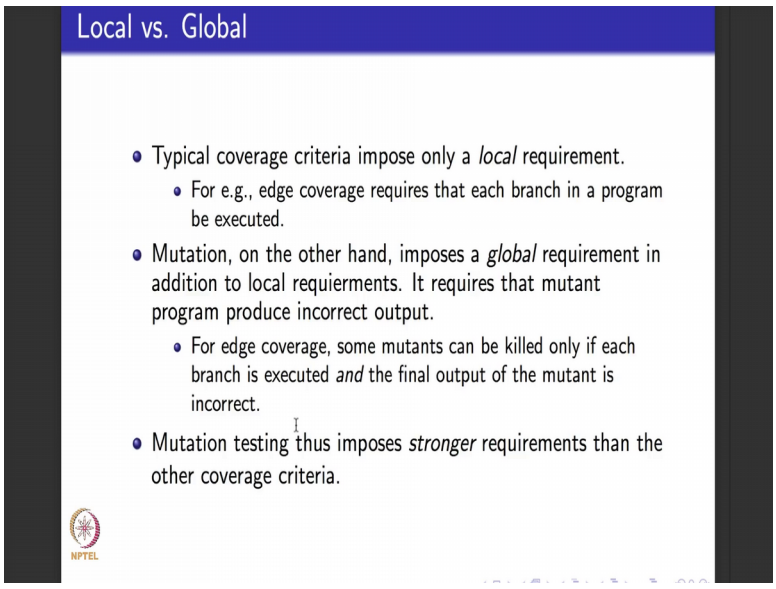


I told you that mutation testing is considered as one of the strongest test criteria in terms of finding capability of finding the most number of faults. So, if you go by that then it should subsume a good amount of graph be testing, and a good amount of logical

predicate based testing also. So, in this lecture we will see which are the graph criteria that mutation testing is going to subsumed, which are the logical predicate criteria that mutation testing is going to subsume.

So, mutation operators can be picked to be a subsumed the kind of criteria we want. Let us say suppose we want to make a claim that mutation is testing subsumes edge coverage. Then I say in the control flow graph to achieve edge coverage for the program, what are the mutation operators that I should pick in the program for this to happen? So, I pick my mutation operators to make mutation testing subsumed the kind of coverage criteria I want to do. For certain coverage criteria it is possible to pick mutation operators to do this, for certain criteria it is not possible to pick mutation operators that do this. So, for criteria which it is not possible to pick mutation operators to do this, mutation testing does not subsume those kind of criteria. Mutation operators that ensure that a particular criteria is covered put together are called what are called that make the mutation testing subsume the coverage criteria what are called yielding the criteria right. So, we use mutation operators that are generically applicable not specific to any programming language not specific to any entity that way when we consider subsumption.

(Refer Slide Time: 02:50)



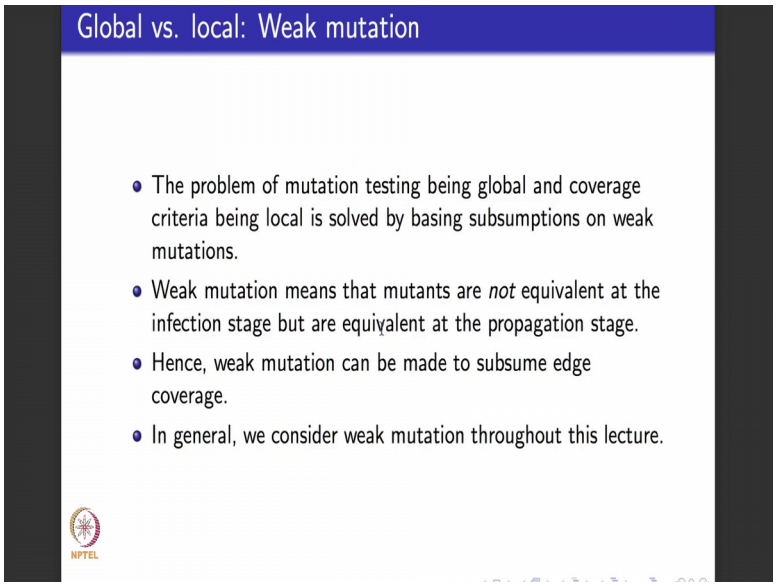
So, if you look at coverage criteria like graph coverage criteria or logical coverage criteria, you will realize that they are somewhat local coverage criteria. They impose only a local requirement, what do we mean by a local requirement? Let us take an

example. You remember we saw edge coverage when we did graph based coverage, and we later realized that predicate coverage and logic coverage is the same as edge based coverage. What is that require? What is the local requirement that it imposes? It says that every branch in a program be executed. It does not talk about anything related to inputs and outputs of a program that is what mutation testing talks about, but edge coverage says in the program in the control flow graph of the program execute every edge.

So, it talks about a requirement that is local to the edges of the control flow graph of a program, but mutation testing imposes a global requirement in addition to the local requirement. It requires that the mutated program produce an in correct output. We saw notions of weak mutation and strong mutation, if you remember. Weak mutation does not need a program to produce incorrect input, it requires that infection happens at that statement. But strong mutation needs a program I mean strong killing a mutation needs a program to produce a different output. So, that is what is called a global requirement.

So, in some sense because of this mutation testing imposes stronger requirements on the software coverage than other coverage criteria. The problem of mutation testing being global and coverage criteria being local how do you solve it? As I told you when I say mutation testing is global back in this slide, why do we I mean by global? We mean that it is global because we want the mutant to produce an incorrect output, which means in the reachability infection propagation model I want the mutant to ensure conditions of reachability of infection, and of propagating all the way to the outputs to produce a different output. We later refine this notion of killing to say that this is actually strongly killing the mutant. We could have the option of weekly killing mutant.

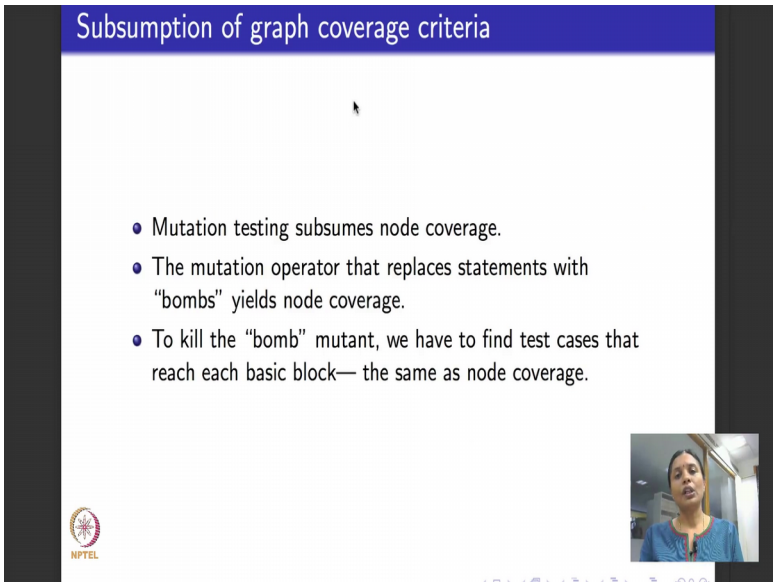
(Refer Slide Time: 05:00)



By which we satisfy only reachability an infection need not propagate. So, that is the notion of mutation that we will use through most of this lecture except towards the end when we look at data flow criteria.

So, we consider weak mutation means mutants are not equivalent at the infection stage, but can be equivalent at the propagation state. So, weak mutation can be thought of us making mutation testing local to the statement or the set of statements that we are considering in the graph. So, using weak mutants we can compare mutation testing to other coverage criteria.

(Refer Slide Time: 05:42)

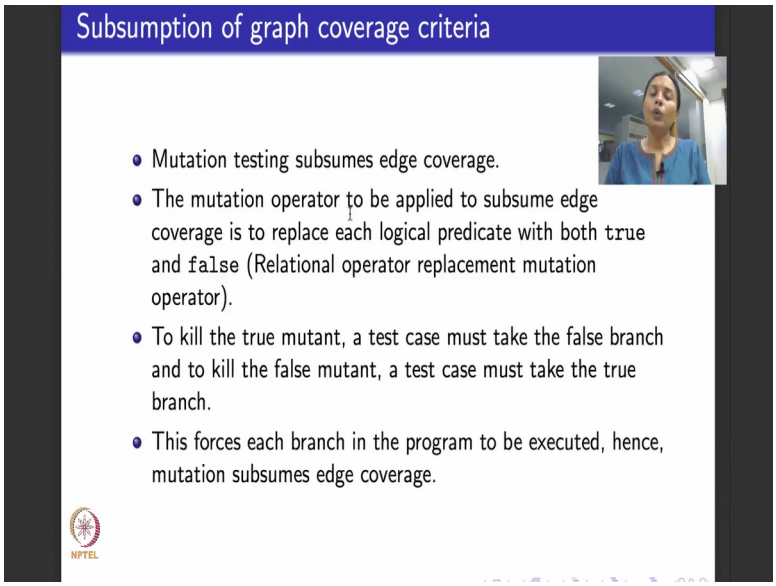


What are the Graph coverage criteria that mutation testing is going to subsume? Mutation testing subsumes node coverage, why does it subsume node coverage? So, node coverage for graphs when it is apply to programs mean in each or execute every block of statements and every statement in the program. If you remember last lecture I had told you about this bomb mutation operator. Bomb whenever it is placed at a particular statement as a particular statement in the program, causes the state program to fail at that point.

So, let say you want to achieve node coverage, what you do is take a program draw it is control flow graph of a program, you want a test case that will achieve node coverage for this program. How do you do it by using bomb statement? Let say you want to target a node 7 which corresponds to some particular statement in the program. You put a bomb statement in the mutated version at that place in the program, and write a test case to kill that mutant. When you write a test case to kill that mutant, the place of the statement of the node at which you put the bomb statement will be visited by the test case, and hence that will be covered by the test case. So, by using bomb statements at a appropriate places in the program we can ensure the node coverage is met.

So, mutation testing subsumes node coverage criteria.

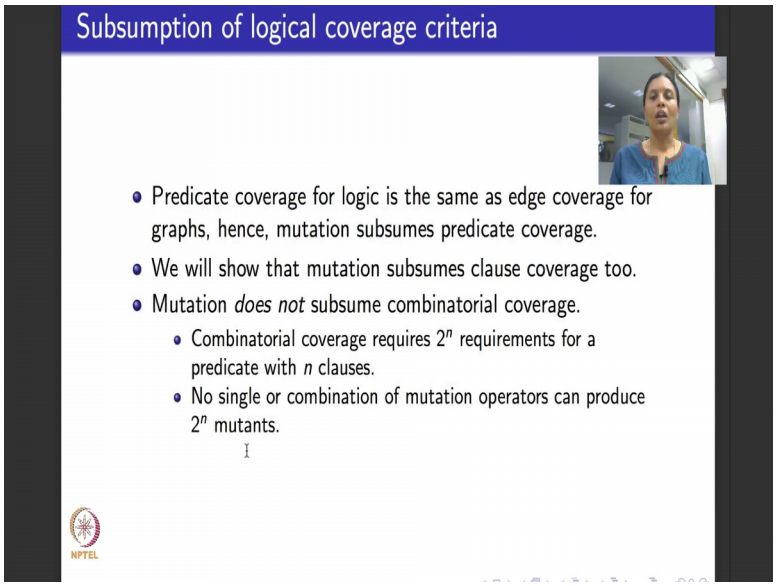
(Refer Slide Time: 07:02)



Mutation testing also happens to subsume edge coverage criteria, why is that? So, how does edge coverage happen, why do you get different edges in a node. We saw that edge coverage is the same as predicate coverage, from a particular node in the graph corresponding to a program. We take 2 different edges because that node had a decision statement in the program. One edge results because the decision statement evaluated to true, one edge results because the decision statement evaluated to false. So, I can it is natural that I use those mutation operators, right? So, I say I will do this is relational operator replacement mutation operator that we saw in the last lecture. I will replace each logical predicate that I encounter in the program which true once to create another mutant. Now if I write test cases to weakly kill these 2 mutants, one test case to take the true edge because I have replace the predicate with true one test case would take the false edge because I have replace the predicate with false. Between these 2 test cases I would have achieved edge coverage for this program.

So, mutation testing does subsume Edge coverage.

(Refer Slide Time: 08:15)



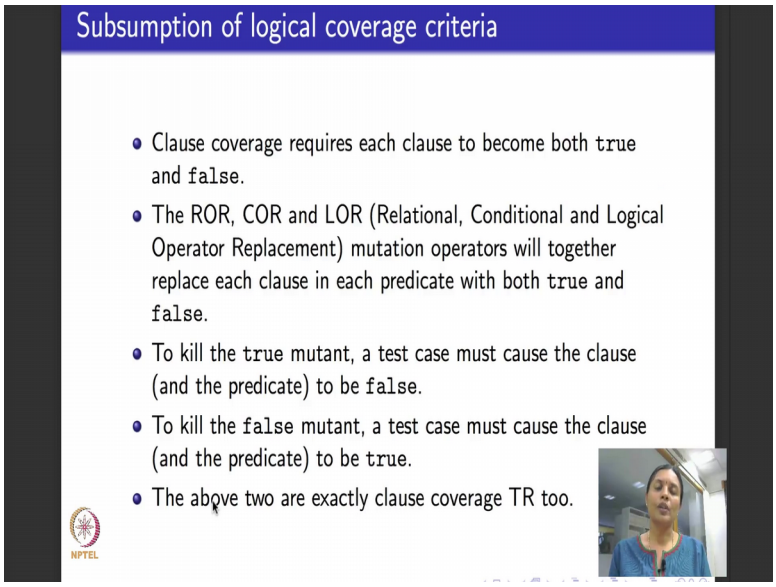
Now let us move on to logical coverage criteria, what are the various logical coverage criteria that mutation testing subsumes? By the way, before we move on for graph coverage criteria mutation testing just subsumes node and edge coverage. It does not subsume other coverage criteria, like prime path coverage, complete path coverage, and all because it does not deal with behavioral aspects of such behavioral aspects of programs. Now will logic coverage criteria it so happens that mutation testing subsumes clause coverage, predicate coverage, and some amount of active clause coverage also. It is does not subsume inactive clause coverage and combinatorial coverage. So we will see why.

Predicate coverage for logic as I told you is the same as edge coverage for graphs. And just in the previous slide I explained to you about how mutation testing subsumes edge coverage. Since the predicate coverage is the same as edge coverage, mutation testing subsuming edge coverage implies such mutation testing also subsumes predicate coverage. You just have to make the predicate true once, and make the predicate false once. We will show that mutation testing subsumes clause coverage. Before we move on to show that I would also like to tell you that mutation testing does not subsume combinatorial coverage. You remember what combinatorial coverage is? It says right every possible values of true false for each of the clauses in a predicate and write test cases that will test the predicate for each possible value of true false values for each of the clauses, so combinatorial coverage because of this each possible combination being

considered. We understand requires 2 power n requirements for a predicate with n clauses. Because each clause can take to 2 true or false totally n clauses for 2 power n possible values can be there. No single or combinational mutation operators can produce 2 power n different mutants it is very exhaustive and difficult, because it is very exhaustive and difficult we say that mutation testing does not subsume combinatorial coverage.

Now, what we will show is why does mutation testing subsume clause coverage.

(Refer Slide Time: 10:29)



What is clause coverage tell you? If you recall clause coverage says each clause has to be tested to be true once and tested to be false once. If you remember in the last lecture we saw these 3 categories of mutation testing operators. Relational operator replacement which replace the less than less than or equal to greater than or equal to and not equal to with each other. Conditional operator replacement which replace the AND or NOTs (Refer Time: 11:01) with each other. Logical operator replacement which replace the bit wise and bitwise or and bitwise XOR with each other.

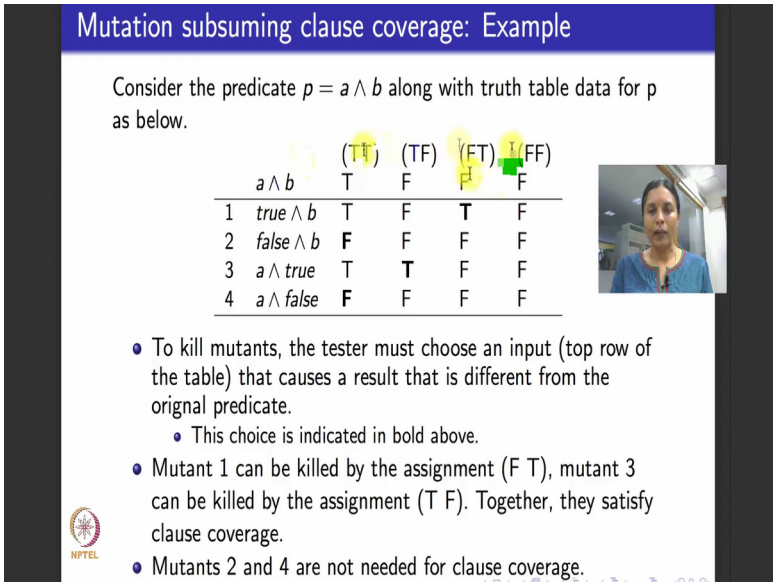
Between these 3 mutation operators you could alter each clause in the predicate to be come true once and become false once. Why because each clause in a predicate will be a combination of these operators, and the blind mutation operator that you could apply is to replace one with true and one with false wherever it is applicable. Wherever it is not then you reverse you negate the relational operator to make it false, and retain the

relational operator as it is to make it true. Suppose there was a clause with says x is less than or equal to y which was a part of a predicate, then x is less than or equal to y can be made true as it is, to make it false you replace the less than or equal to in the clause by using a relational operator replacement of greater than. So, instead of writing x less than or equal to y, you write x greater than y.

So, this is variant that will make the clause false. So, like this each clause in a predicate can be make true by retaining as it is if it is turns out to be true, and can be made false by flipping the relational arithmetic or conditional or logical operators that are involved in the clause. So, to kill the true mutant a test case must clause the clause and the predicates to be false, which is what it will do, to kill the false mutant the test case must clause the clause and the predicate to be true because it has to do different output then the original program. And this is exactly what clause coverage does right because each clause is may true once and false once in this process.

So, mutation testing does subsume clause coverage. I will explain this part the detailed example to you.

(Refer Slide Time: 12:49)



Let us consider this predicate p is equal to a and b it has 2 clauses a and b. So, I have depicted this table in a slightly different way. So, it let me explain this table to you. So, what it says is this is the predicate here a and b, and right on top here this TT TF FT FF within brackets tell you, that the a clauses a and b in the predicate both are true in the

case of TT, TF means a is true b is false FT means b a is false b is true. FF is both a and b are false. Now what did I tell you I told you here that each clause can be made true once and false once using any of these operators. So, there are 2 clauses here. So, this part the second column in this table, I had made a true once false once retain b as it is in this part rows 3 and 4 I have made b true ones false ones retain a as it is.

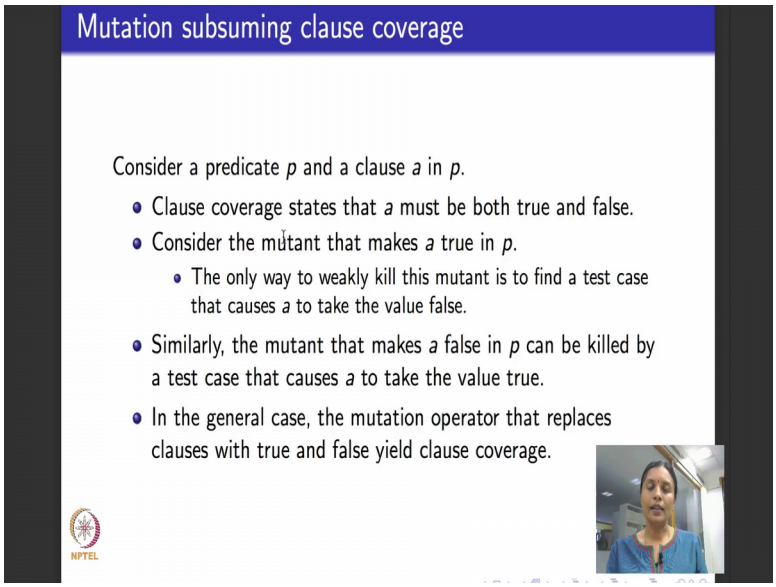
Now, let me see what happens to this. So, what do these mean these mean what is the value that the predicate takes right. So, if both a and b are true, what is true and b take it becomes true. If a is true and b is false then, what is true and b become it becomes false. If a is false and b is true there is no a here, but basically dependent on the value of b because b is true it is true, and now both are false in particular b is false. So, it becomes false to this part of the table, I have populated as the true false values that these predicate where one clause is replaced with true and false will take. The top part indicates truth false assignments for a and b, this row here indicates what happens to this predicate a and b. Remember to kill the mutants, tester must choose an input which is here right on top row of this table that causes the result to be different from the original predicate.

So, let us say this is the first mutant, what have I done in this mutant? I have taken a made it true. So, now, let us see which is the place where this resulting predicate mutated predicate is different from the original predicate a and b here both are true not different here both are false they are the same again not different. Here the original predicate is false, but the mutated predicate is true. So, the test case that will kill the first mutation makes a false and b true, is that clear? So, this that is why it is been colored in bold here, it is been indicated in bold similarly, if I take this as the original predicate and this is the mutated predicate, where I replace a with false to be able to kill this mutated predicate, the only test case that will kill will be this one a is true b is true because the value of the original predicate here is true whereas, the value of the mutated predicate is false.

Similarly, for the other rows. Now if you take mutant one mutant one can be killed by the assignment false and true this part indicated by bold as I told you. And mutant 3 can be killed by the predicate true by the assignment a is equal to true b is equal to false which is indicated as this bold true here because it differs from the value of the predicate. Between these 2 test cases if you notice a is made true once false once. And a is b is made false once true once. So, it does achieve clause coverage. In fact, we just mutants one and 3 we can achieve clause coverage for this predicate. So, mutation testing does

subsume clause coverage. In this example the mutants 2 and 4 even though I have given them in this table for completeness are not necessary we might as well not done them at all.

(Refer Slide Time: 16:39)

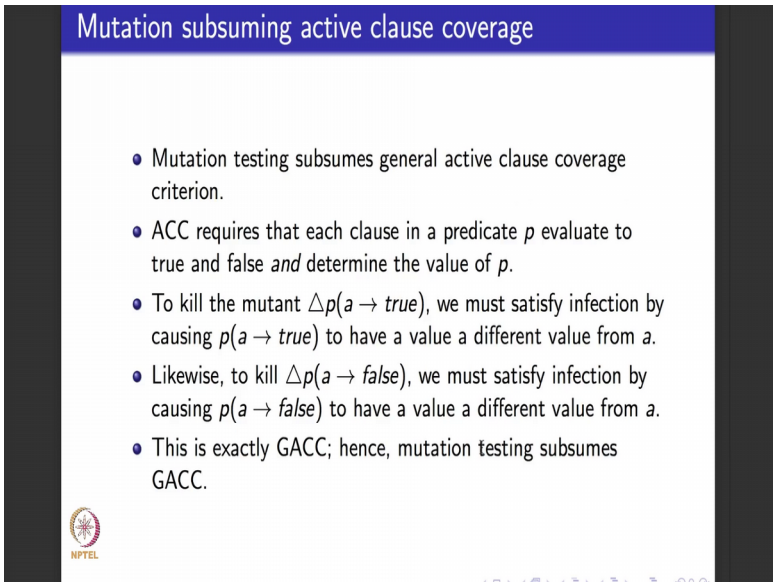


Now to generalize this consider a predicate and let a be a clause in the predicate. Clause coverage for a says a must be true and must be false. Now consider a mutant that makes a true in the predicate. The only way to weekly kill this mutant is to find a test case that causes a to take the value false, is that clear?

Similarly, the mutant that makes a false in the predicate can be killed by a test case that causes a to take the value true, that is what happened in this example right. Here was made true the only test case that killed this mutant of this predicate if you see a was made false. In the third mutant that we considered b was made true, the only test case that killed the third mutant b here which is this value is made false as what is returning. So, it has to differ in the opposite way by differing. The opposite way it make sure that each clause that I am considering currently in the predicate is made true once and false once. So, it does manage to achieve cross clause coverage.

So, how do I generalize this? Take any predicate that involves a few clauses. How do I make mutation coverage subsume clause coverage? Just choose a mutation operator that replaces each clause with true and false and then you will get clause coverage immediately.

(Refer Slide Time: 17:55)



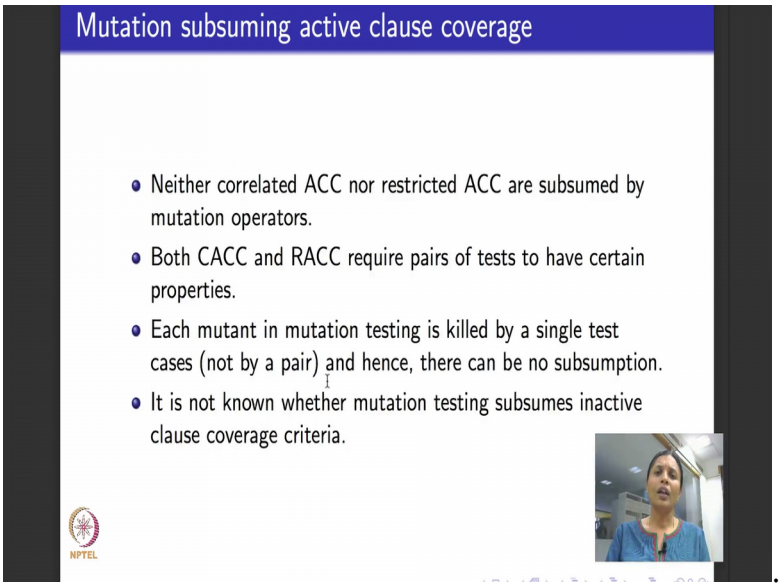
Mutation testing also happens to subsume active clause coverage. If you remember we had seen 3 different active clause coverage criteria when we did logical predicate base testing. The 3 criteria that we saw were generalized active clause coverage criteria called GACC. Correlated active clause coverage criteria called CACC. Restricted active clause coverage criteria called RACC. It so happens that mutation testing subsumes only GACC, cannot be made to subsume CACC and RACC. I will tell you why So. So ACC, what is ACC criteria require? Active clause coverage criteria requires that each clause in a predicate evaluate to true once and false once and while doing so, also determine the value of the predicate. So, which means what are the value that the predicate takes when the clause is true should be different from the value that the predator takes when the clause is false.

So, to suppose let us say I take the predicate p, please read this phrase that I have written here, as in the predicate p, I mutate or change p by replacing a with the value true, is that clear? So, in the predicate p I take a every occurrence of a and p and replace it with true. So, I said this is the mutated predicate with every occurrence of a replaced by true. How will we satisfied infection? We will satisfy infection by causing this was the original predicate this is the mutated predicate these 2 should have different values. Similarly when I take p and replace a with false, we can again satisfy infection by having the original predicate have a different value from the mutated predicate. This is exactly what is generalized active clause coverage criteria. Because it says that each clause becomes

true and false here for example, if a is an arbitrary clause that we have chosen a becomes true and false once and to kill it the predicate should have a different value in both the cases. And this is exactly the definition of generalized active clause coverage criteria.

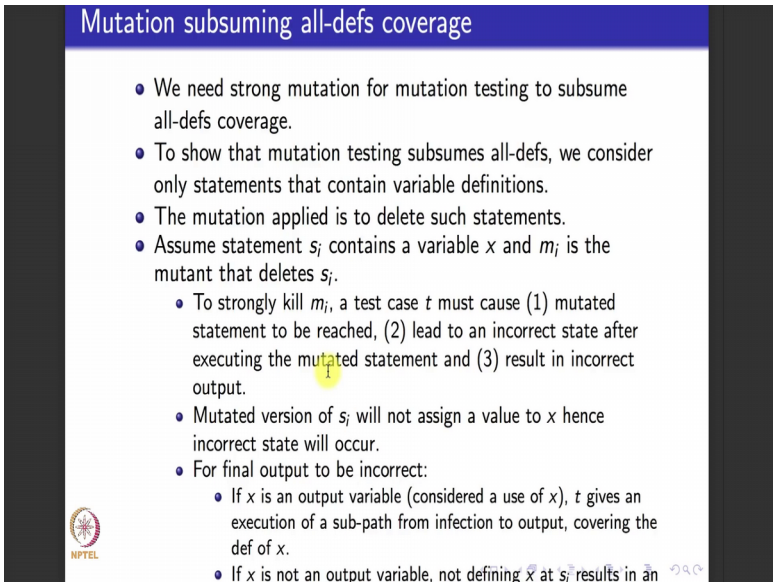
Hence mutations are testing subsumes GACC.

(Refer Slide Time: 20:08)



Now moving on, as I told you CACC and RACC are not subsumed by mutation testing or mutation operators cannot be designed to subsume CACC and RACC, why is that so? If you remember the definition of CACC and RACC, it says while the major clauses determining the truth value of the predicate the minor clauses in one case should all have different values. In the other case the minor clauses should have all the same values. So, it not only imposes a condition on one clause in the predicate it imposes conditions on all the other remaining clauses in the predicate. So, they come as pairs of conditions, each mutant in mutation testing is meant to kill only one condition that can be implemented at a test case not as a pair of condition, mutation testing is not meant for that, as I told you the wisest thing to do in mutation testing is to apply one mutation operator, at a time and because we apply one mutation operator at a time we really cannot (Refer Time: 21:07) to CACC or RACC. Hence we say mutation testing as we do it in this course will not subsume correlated active clause coverage criteria and restricted active clause coverage criteria.

(Refer Slide Time: 21:22)



Finally we go Back to graph coverage criteria. I would like to tell you that mutation testing subsumes all defs. If you remember what is all defs criteria. All defs is that there must be a path which is def clear from every definition to one criteria that which is there from every definitions to one possible use. In this case we cannot use weak mutation. So, is we switch back to strong mutation. So, strong mutation means that the output should be different not only reachability and infection, but propagation should also be met. So, what to do we do the do to all defs condition? We consider every statement in the program that contains definitions of variables.

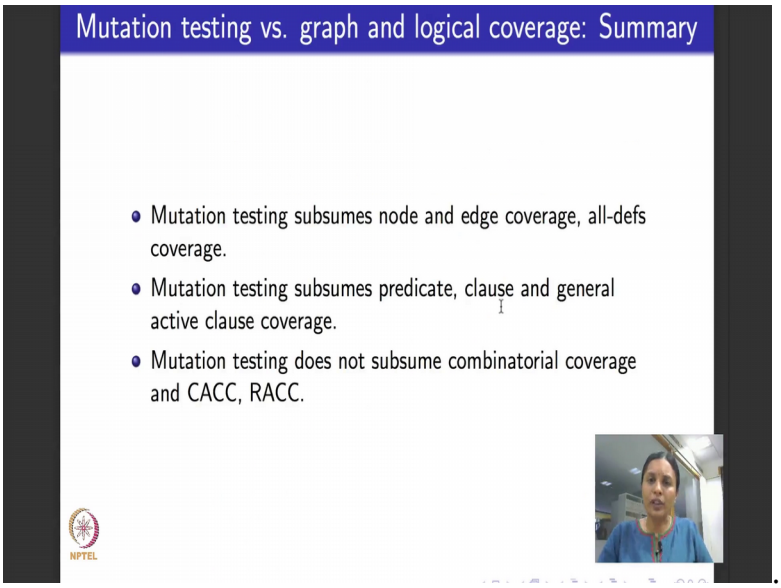
What we do to mutant is that we simply remove that statement, is that clear? We simply remove that statement. When you remove that statement what happens? Assume that there is a statement s i that contains the definition of a variable x, and m i is the mutant that removes or deletes the statement s i too strongly kill the mutant program m i, a test case must cause the following. It must cause the mutated statement to be reached. It must lead to infection, which is lead to an incorrect state after execution of the mutated statement. And it must lead to propagation, which is result in an incorrect output. Mutated version of s i will not assign the value to x because s i itself is not there.

So, the assignment itself is missing. So, an incorrect state will definitely occur propagation will definitely happen. So, the final output to be incorrect if this variable x that I am considering is itself an output variable then t gives execution of a sub path from

infection to the output variable. If x is not output variable then not defining x itself results in an error I am sorry, I just realize that the last line is missing here, please read it as if x is not an output variable then not defining x as s i results in an error on it is own right. Either way I thing we anyway have issues. So, what I am saying is to cover all definitions consider every statement in a program that contains the definition.

The mutation that I apply is remove that statement completely remove that statement from the program. And the claim is that if that statement is removed you will definitely meet all the 3 conditions of reachability infection and propagation. So, the output of the mutated program is guaranteed to be different from the output of the original program because one single statement is missing. Because the statement is missing, mutation testing subsumes all definitions coverage.

(Refer Slide Time: 24:00)



So, just to summarize what we learnt today, in the graph coverage criteria mutation testing subsumes node edge coverage amongst the structural criteria. And in the data flow criteria it subsumes all defs coverage. In logical coverage criteria mutation testing subsumes predicate coverage clause coverage and generalized inactive coverage. It does not subsume combinatorial coverage CACC and RACC. It is not known whether mutation testing subsumes inactive clause coverage criteria. It is also not known whether mutation testing subsumes all uses criteria. So, if you pursuing research here are 2 good problems for you to work on see and understand, and see if you can (Refer Time: 24:44)

mutation testing, how does mutation testing relate to inactive clause coverage criteria. And how does mutation testing relate to all uses or all du paths criteria. So, next week we will begin with applying mutation testing for design integration. That is it for this week.

Thank you.