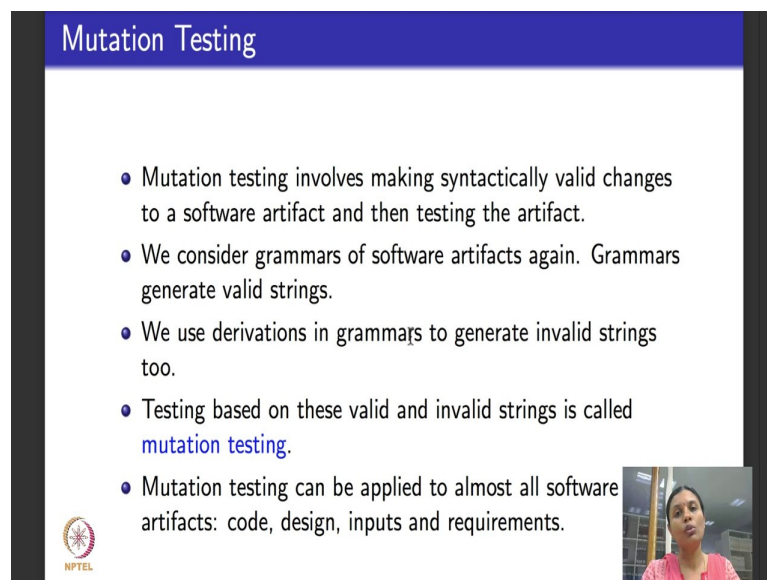


Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 36
Mutation Testing

Hello, welcome to the second lecture of week 8. So, what are we going to do today? Last time I told you that mutation testing means testing for the syntax of a program, syntax of a program deals with regular expressions and grammars? So, I gave you a very brief quick introduction to regular expressions and context free grammars and this time what I am going to do is I am going to introduce you to mutation testing.

(Refer Slide Time: 00:38)



The slide is titled "Mutation Testing" in a blue header. It contains a list of five bullet points explaining the concept of mutation testing. The NPTEL logo is visible in the bottom left corner of the slide content area. A small video inset of the professor is in the bottom right corner.

- Mutation testing involves making syntactically valid changes to a software artifact and then testing the artifact.
- We consider grammars of software artifacts again. Grammars generate valid strings.
- We use derivations in grammars to generate invalid strings too.
- Testing based on these valid and invalid strings is called **mutation testing**.
- Mutation testing can be applied to almost all software artifacts: code, design, inputs and requirements.

Next week we will see how to apply mutation testing to source code. So, what is mutation testing involve? The term mutation in the in biology or generically means you make a change. So, in biology you make changes cells undergo mutation they undergo changes. When we apply mutation testing in the context of software testing, we say a software artifact undergoes the change.

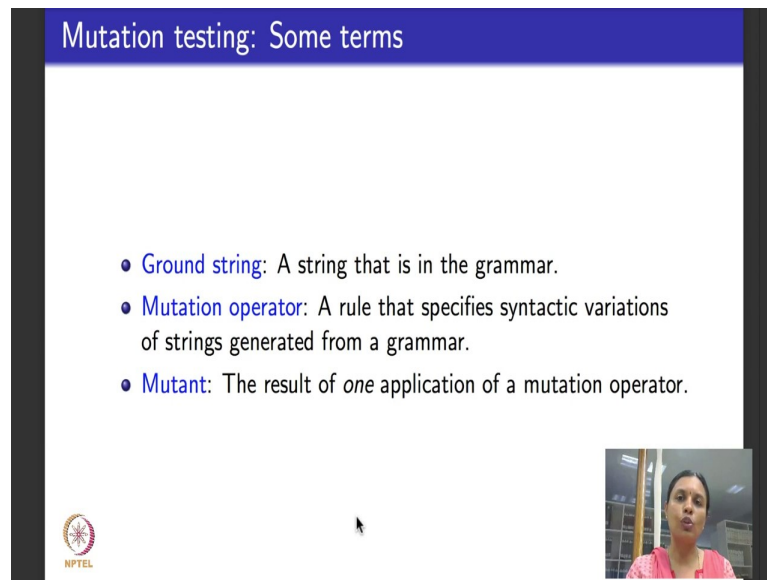
So, the software artifact that could be mutated or changed could be a program, could be an input, could be a design document, it could be one of the several different things. So, mutation testing means you make changes, most of the time the changes are syntactically valid, I will explain to you what that is to a software artifact and then test the artifact.

When I say syntactically valid change I mean the following. So, you consider the software artifact we have be piece of source code or a program now I make a change to see how the changed program behaves with reference to the original program. So, for me to be able to test the changed program the first thing that I must do is to be able to compile the changed program.

So, the changed program must be syntactically valid, it should be a compilable program. So, while doing mutation testing we mutate by making changes. How are the changes defined? The changes are defined by using mutation operators and where do the mutation operators come from they come from grammars of software artifacts. Grammars typically generate valid strings everything to the generate is syntactically valid as per the production rules of a grammar. We use derivations in grammars to generate valid strings. And sometimes we use derivations to generate invalid strings also. When do we generate invalid strings in the grammar? We generate invalid strings only when we consider mutation as being applied to the inputs of a program. When we mutate a program itself then we expect the program to be valid because it needs to be compilable.



But sometimes we want to know how a piece of program, given program, behaves on inputs that are invalid. How does it handle invalid input? So, when we do that we mutates to produce invalid string from the grammar corresponding to the input domain. So, testing based on generating these valid strings and invalid strings for different artifacts is what is called mutation testing mutation testing as I told you can be applied to program source code, it can be applied to design integration, it can be applied to input space and it can be applied to change the requirements themselves.

(Refer Slide Time: 03:08)



Mutation testing: Some terms

- **Ground string:** A string that is in the grammar.
- **Mutation operator:** A rule that specifies syntactic variations of strings generated from a grammar.
- **Mutant:** The result of *one* application of a mutation operator.

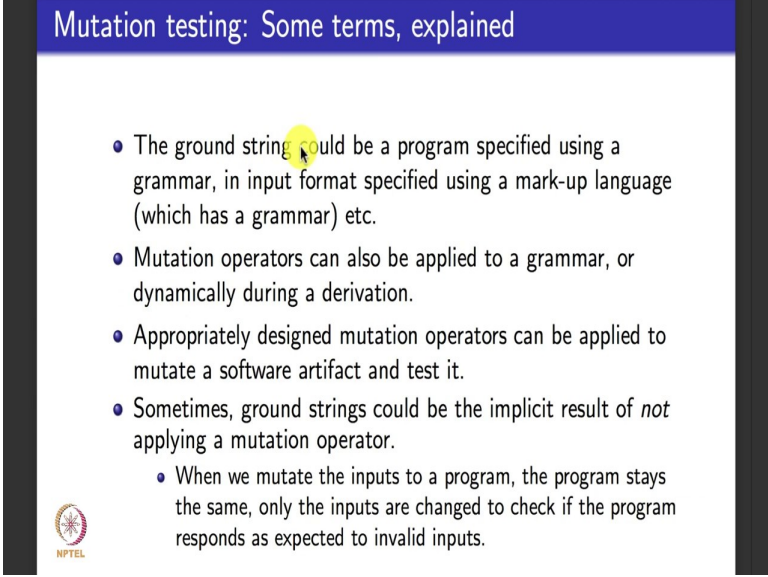
So, here are some terms that we will use throughout our lectures in mutation testing. So, I have a particular software artifact that I am trying to mutate or make a change in, that software artifact is what is called a ground string. So, ground string is a software artifact that I want to mutate. In other words it is the string in the grammar of the corresponding software artifact, it is a string in the underlying grammar, and how do I mutate? I mutate by making what is called a mutation operator. What is a mutation operator? A mutation operator is a production rule or a rule that specifies syntactic variants of strings that can be generated from the grammar. Every string generated by grammar as we saw in the last lecture comes by applying a sequence of rules one after the other till we get a string only of terminals.

At some point during the sequence of derivations that I apply, I decide to apply another rule instead of the original rule that was used in the string and such a rule that I apply instead of the original rule is what is called a mutation operator. So, I begin with the ground string, I apply a mutation operator, after applying mutation operator I have changed a mutated the ground string. The changed or mutated ground string is what is called a mutant. So, a mutant is the result of applying exactly one application of a mutation operator. So, you might ask why exactly one why not more? You can apply more as long as the changes are syntactically valid and you are confident of how the mutated program will behave, but typically in mutation testing it is recommended that

you apply mutation operators one at a time. So, we will discuss this, we will revisit this question once again very soon in this lecture.


But just to move on, before we move on I would like to summarize, let us say you testing a java program using mutation testing, the original Java program that you begin with is called a ground string. You take the Java program apply one mutation operator, that is one change to the underlying grammar from which the Java program comes by applying some rule instead of the other rule the change that you make is called mutation operator or applying a mutation operator. The resulting string is another syntactically valid Java program whose behavior you want to test against the original Java program.

(Refer Slide Time: 05:36)



Mutation testing: Some terms, explained

- The ground string could be a program specified using a grammar, in input format specified using a mark-up language (which has a grammar) etc.
- Mutation operators can also be applied to a grammar, or dynamically during a derivation.
- Appropriately designed mutation operators can be applied to mutate a software artifact and test it.
- Sometimes, ground strings could be the implicit result of *not* applying a mutation operator.
 - When we mutate the inputs to a program, the program stays the same, only the inputs are changed to check if the program responds as expected to invalid inputs.

 NPTEL

So, the changed the Java program is what is called a mutant. So, the ground string could be a program as I told you could be a Java program, it could be specified using the grammar of Java, it could also be an input format specified using a markup language.

Several different things take XML format, several different software programs take XML format corresponding to an entity as their input format. Sometimes I want to be able to test the software artifact by changing the input to a syntactically invalid input or another valid input and see how the program behaves under these invalid inputs. When I do that I used= the grammar to generate implicit invalid inputs also. So, when I do that I get invalid inputs and in that case ground strings result as those that arise by not applying a mutation operators. For example, as I told you when we mutate the inputs to a program;

program remains the same it is still the ground string, but the mutation operator is applied on the inputs to a program the program and the resulting mutant are the same, but the mutated input is what is changed and in thus this cases it can be invalid also.

(Refer Slide Time: 06:48)


Example of mutant

Consider the grammar that we saw in the last lecture:

```

stream := action*
action := actG | actB
actG   := "G" s n
actB   := "B" t n
s      := digit1-3
t      := digit1-3
n      := digit2 · digit2 · digit2
digit  := 0|1|2|3|5|6|7|8|9

```



- Strings generated by the grammar: G 17 08.01.90, B 13 06.27.94
- Two valid mutants: B 17 08.01.90, G 43 08.01.90
- Two invalid mutants: 12 17 08.01.90, G 23 08.01

So, here is a simple example of how to apply mutation at a basic grammar level. Next lecture I will show you how to apply mutation to source code by taking concrete examples of programs. If you remember we saw this grammar in the last lecture and these was some valid strings that we were generated by grammar. This grammar used notations that were a combination of regular expression notations and grammar notations and this vertical bar is to be read as 2 rules: this rule and then this rule. So, here are some examples of strings generated by grammar. This was generated by this grammar G 1708.01.90 and B 13 06 27 94. Here are 2 examples of valid mutants.

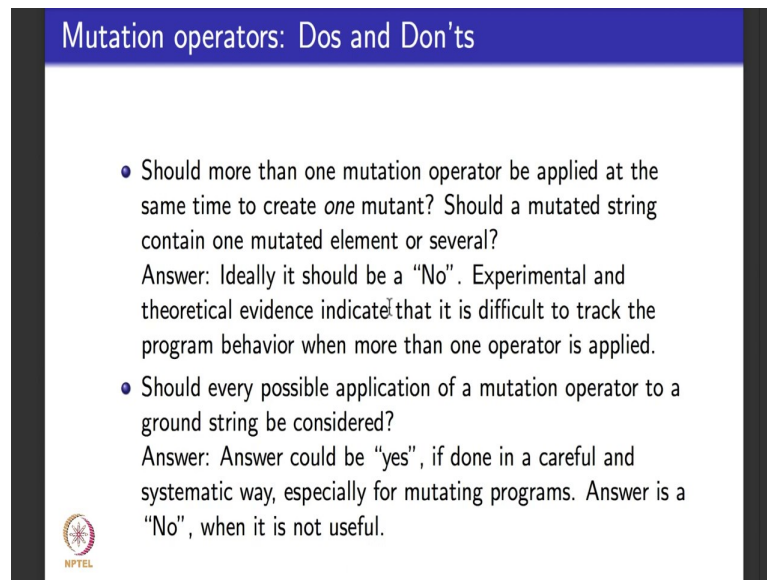
How did I get this? How did I get the first mutant? I took this string this string had a derivation in the grammar at some point this G was derived. In fact, right in the beginning G will be the first terminal string to drive after the third or fourth step in the derivation when this production rule is applied. Instead of now this G whenever this production rule was applied, I picked up the same derivation which gave this at that point instead of deriving G, I applied the 4 production rule and got B instead. So that is the only change. So, at some point in the derivation of this string I decided to take out one

step in the derivation and replace it with a different production rule. In particular whatever was giving me G I replaced it with B. So, this is a valid mutant.

So, here is another example. I take the same string G 17 08 01 90. At some point this number 17 was generated because this rule was applied S goes to digit 1 2 3. It picks up randomly a digit that is between 1 and 3 length long and applies it and here the digits that was picked up was 1 and 7 and it was of 2 digits long. Instead of picking up 1 and 7 at pick up 4 and 3. So, from this I get this valid mutant. Here are examples of 2 invalid mutants from the string, the first invalid mutant says you take this string G 17 08 01 90; instead of G you get 12. Why is this an invalid mutant? If you see this grammar you will realize that no there is no way that this particular string can be generated you can never general 12 17 08 01 090. The way this grammar structured every word of this form we will begin with the G or a B because it begins with the 12 and it cannot be generated by the rules or the grammar we call it an invalid mutant. Maybe these are inputs to a program and you want to check if the program throws in an error saying this input format is invalid.


Here is another example of an invalid mutant. This is the same thing G 23 08 01 and the last 2 digits of this date like entity is missing. Here is another, this is an example of invalid mutant because there is no way I can generate this string from this grammar again because when I reach this stage where I apply this GSN or BTN and I finished applying for S, now I have to apply for n at 1. In 1 shot I generate this whole thing I generate digit 2 dot digit 2 dot digit 2. So, this one if you see does not have the second dot and the third 2 digits. So, it can never be generated in this grammar.

(Refer Slide Time: 10:12)



Mutation operators: Dos and Don'ts

- Should more than one mutation operator be applied at the same time to create *one* mutant? Should a mutated string contain one mutated element or several?
Answer: Ideally it should be a "No". Experimental and theoretical evidence indicate that it is difficult to track the program behavior when more than one operator is applied.
- Should every possible application of a mutation operator to a ground string be considered?
Answer: Answer could be "yes", if done in a careful and systematic way, especially for mutating programs. Answer is a "No", when it is not useful.

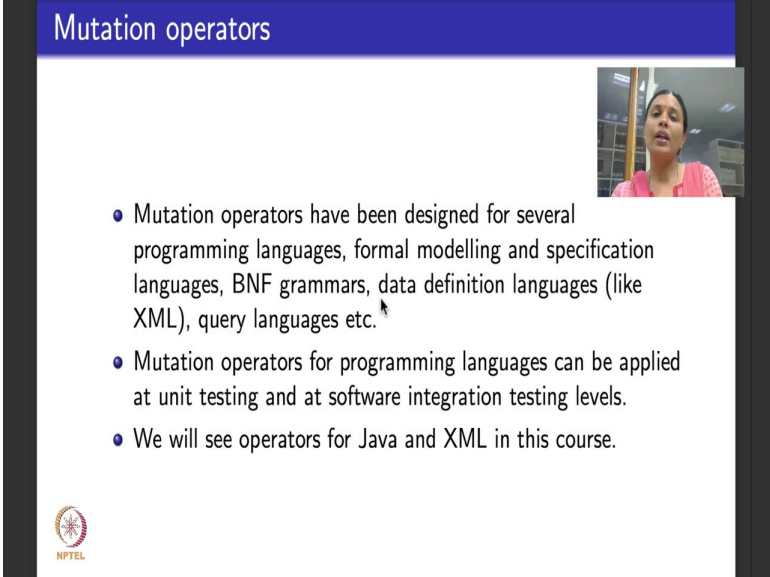


So, that is why it is an invalid mutation. So, here are some do's and do not's as far as mutation operators are concerned. The first question that we discussed a little while ago in this lecture was, should I apply exactly one mutation operator to get a muted mutant or should I apply more than one mutation operator to get a mutant?

So, that is the first question. So, should it contain just one mutation operator or it can contain several mutation operator. The ideal recommendation that people use a mutation testing is do not do more than one. Work with exactly one mutant, it is good enough and after analyzing that you work with the next mutant. Experimental and theoretical evidence indicate that it is usually very difficult to track the program behavior when more than one operator is applied. Because you do know whether the change in the mutated program is because of which operator that I apply. Even if you apply to you do not know with the change might be difficult for large programs to isolate and say that this changes because of the first operator, this behavior changes because of the second operator it is very difficult to do that. So, always the voice thing to do is to apply one mutation operator at a time. The second question that you might want to ask you should every possible application of a mutation operator to a ground string be considered? In the sense that I have a grammar of a programming let us say language, a grammar usually fairly exhaustive and if you try to generate mutation operators based on the grammar you will get lots and lots of operators, hundreds, sometimes even thousands of operators.

So, now for a given program they could be several different ways of mutating it. Should I consider every possible mutation and test that is like exhaustive testing with reference to mutation. The obvious answer is no, but sometimes for small programs people say yes also. So, yes if you can do it carefully and in a systematic way know if you do not know that it is not useful. Typically it is not useful to do exhaustive mutation testing.

(Refer Slide Time: 12:14)



The slide is titled "Mutation operators" in a blue header. It contains a list of three bullet points. In the top right corner, there is a small video inset showing a woman with dark hair wearing a pink top, speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

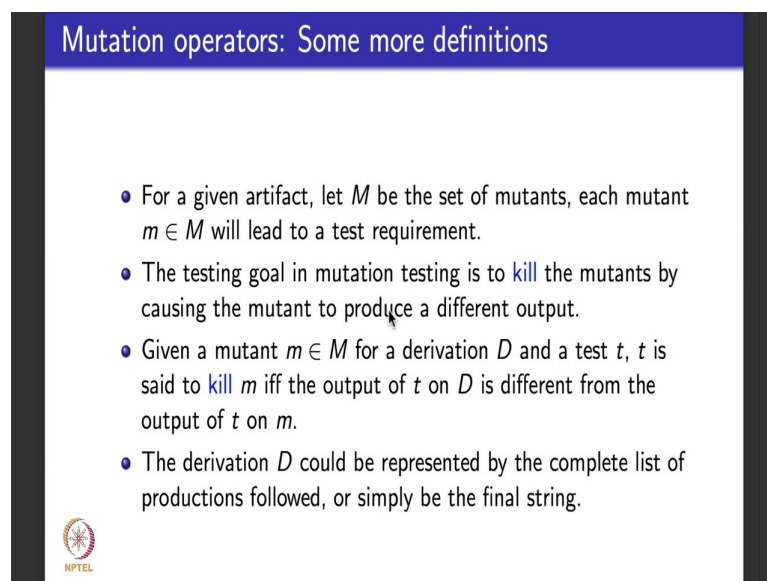
- Mutation operators have been designed for several programming languages, formal modelling and specification languages, BNF grammars, data definition languages (like XML), query languages etc.
- Mutation operators for programming languages can be applied at unit testing and at software integration testing levels.
- We will see operators for Java and XML in this course.

So, the answer is a no. Now, what are the mutation operators? Mutation operators take the grammar and make one change with reference to the grammar that is what I told you. Mutation operators have been designed for several different programming languages. We will see one for java in the next couple of lectures they have been designed for formal modelling and specification languages especially for model checkers like new SMV and SMV, they have been defined for several different grammars and Backus Naur form. I keep using this term be enough I did not really introduce you to you the term be enough we just saw context free grammars.

But there almost always given in Backus Naur form the fact that it comes in this normal form may not be very important for our lectures. So, I skipped introducing the BNF part of the context free grammars when we did context free grammars. We do not really need, but it is important to know that they do not come in any format they always come in BNF format. Mutation operators are also available for data definition languages like XML, they are available for several different query languages SQL and so on. Mutation

operators for programming languages, which are the phases they can be applied in ? They can be applied while doing unit testing and valuing integration testing there are integration level mutation operators that are also available. So, what we will see from the next lecture on words is we will see some mutation operators for java for good number of lectures and when I end mutation testing towards the end we will see mutation operators for XML also.

(Refer Slide Time: 13:44)



The slide has a blue header with the text "Mutation operators: Some more definitions". Below the header, there are four bullet points. In the bottom left corner, there is a circular logo with a star and the text "NPTEL" below it.

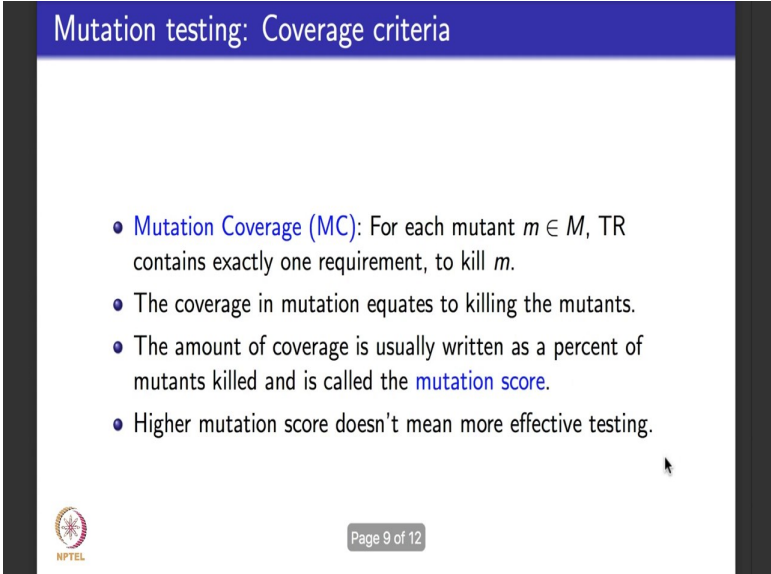
- For a given artifact, let M be the set of mutants, each mutant $m \in M$ will lead to a test requirement.
- The testing goal in mutation testing is to **kill** the mutants by causing the mutant to produce a different output.
- Given a mutant $m \in M$ for a derivation D and a test t , t is said to **kill** m iff the output of t on D is different from the output of t on m .
- The derivation D could be represented by the complete list of productions followed, or simply be the final string.

Some more definitions related to mutation testing before we move on. For a given artifact like source code or requirements or design let m be the set of total number of mutants. Each mutant m in m we will lead to a different test requirement. Testing goal in mutation testing is to be able to kill the mutants. What does it mean to kill the mutant? It means the following you take the original program called ground string apply one mutation operator get a mutant or a mutated program. Now take a test case, run the original program on that test case, take the mutated program, run the same test case on the mutated problem. If these 2 programs, the original program and mutated program, produce outputs that are different for the same test case then you say the test case has killed the mutant right. So, given a mutant m for a derivation d and test t , the test t is set to kill the mutant m if and only if the output of t on d is different from the output of t on m . I hope that is clear right. Take a program, apply one mutation operator, get a mutated program. Pick a test case, run it on the original program, run it on the mutated program.

If the 2 programs produce different outputs then you say the test case is killed my mutant.


Is it definition is very important because the core of mutation testing and this lies and understanding or designing test cases for killing mutants. So, the derivation d could be represented by the complete list of productions or simply by the direct result and final string which could even be the program.

(Refer Slide Time: 15:24).



Mutation testing: Coverage criteria

- **Mutation Coverage (MC):** For each mutant $m \in M$, TR contains exactly one requirement, to kill m .
- The coverage in mutation equates to killing the mutants.
- The amount of coverage is usually written as a percent of mutants killed and is called the **mutation score**.
- Higher mutation score doesn't mean more effective testing.

 Page 9 of 12

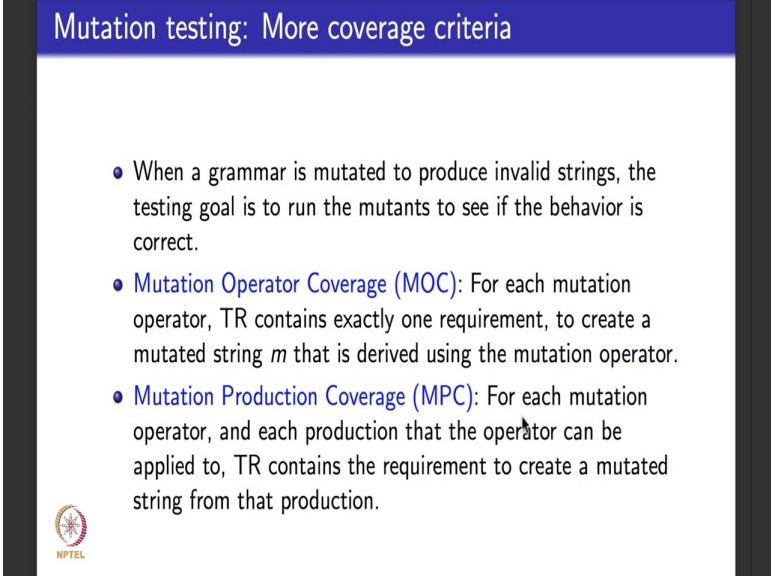
So, here are some coverage criteria for mutation testing. The first coverage criteria that we define is what is called mutation coverage. So, for each mutant that you apply to the original ground string, for each mutant m in M , test requirement or TR contains exactly one requirement, you kill that mutant. So, when I make a change to a ground string and get a mutant I should be able to write a test case to kill the mutant. So, if I am able to do this for every mutant then I have achieved what is called mutation coverage. The amount of coverage that you achieve in mutation testing is usually written as the percent of mutants killed. Like for example, suppose I have a small Java program and it happens to be the case that there are 15 different mutants that I can get from this Java program which means what I can make 15 different kind of changes one at a time to get 15 different other Java programs.

Let us say out of these I manage to write test cases that kill 15 of these mutations. Let us say the test cases that I right I am able to kill only let us say 10 of these 15 different

mutations that I have done. The remaining 5 mutants of this given Java program are such that no matter what kind of test case it is it can never kill in the sense that the behavior of the original program will be the same as the behavior of the mutated program. If that is the case if you manage to kill only 10 out of 15 mutants then the mutation score says that you have killed only 10 out of 15 mutants which means you have roughly killed about 75 percent of the mutants that you created. So, the amount of coverage is usually means the amount of killing of the mutants that you can do, it is usually written as a percent of mutants killed and is called mutation score.

Suppose for all the 15 different mutants that you wrote you managed to write test cases that killed all the 15 different mutants that does not mean that you have tested in a more effective way. It is in fact, a myth to belief that the more mutants that I write and kill the more effective that I have tested the program. These 2 do not really correlate to each other. Another kind of coverage criteria over mutation testing or mutation operator coverage and mutation production coverage.

(Refer Slide Time: 17:30)



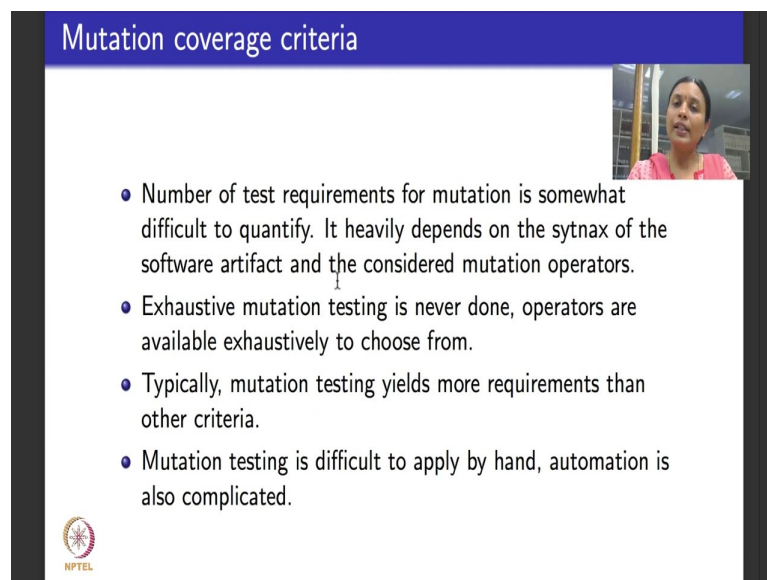
The slide has a blue header with the text "Mutation testing: More coverage criteria". Below the header, there are three bullet points. The first bullet point discusses testing mutants for correct behavior. The second bullet point defines Mutation Operator Coverage (MOC). The third bullet point defines Mutation Production Coverage (MPC). In the bottom left corner, there is a circular logo with a star and the text "NPTEL" below it.

- When a grammar is mutated to produce invalid strings, the testing goal is to run the mutants to see if the behavior is correct.
- **Mutation Operator Coverage (MOC)**: For each mutation operator, TR contains exactly one requirement, to create a mutated string m that is derived using the mutation operator.
- **Mutation Production Coverage (MPC)**: For each mutation operator, and each production that the operator can be applied to, TR contains the requirement to create a mutated string from that production.

So, when a grammar is mutated to produce invalid strings, like we do for mutating inputs, the testing goal is to run the mutants to see if the behavior is correct. So, in which case, mutation operate operator coverage abbreviated as MOC says for each mutation operator TR contains exactly one requirement, to create a mutated string m that is derived using that particular operator.


So, suppose at some point and time there was an operator that said you use less than or equal to and you are able to create a mutation where instead of less than or equal to you use greater than or equal to. So, this is called one mutation operator. So, I rate a test case to cover this mutation operator. Similarly I can define what is called mutation production coverage. What does that say? For each mutation operator and each production the operator is applied to, test requirement contains a requirement to create a mutated string by using that production. So, the production rule that changed the greater than or equal to 2 less than or equal to use the rule make the change and write a test case to kill that mutant, resulting mutant program.

(Refer Slide Time: 18:44)



Mutation coverage criteria

- Number of test requirements for mutation is somewhat difficult to quantify. It heavily depends on the syntax of the software artifact and the considered mutation operators.
- Exhaustive mutation testing is never done, operators are available exhaustively to choose from.
- Typically, mutation testing yields more requirements than other criteria.
- Mutation testing is difficult to apply by hand, automation is also complicated.



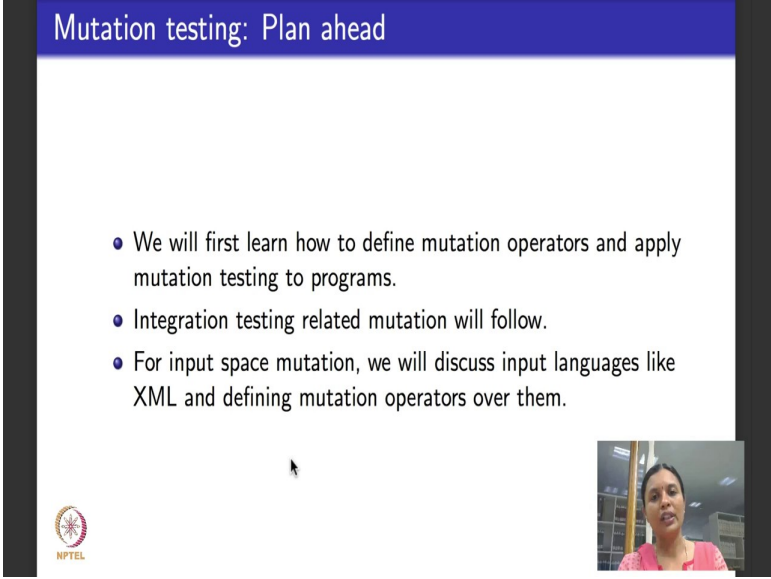
So, these are the 3 mutation coverage criteria. In fact, we will see them for programs and see how they apply. The number of test requirements that I need to achieve mutation coverage criteria is somewhat difficult to quantify.

It basically heavily depends on the syntax of the underlying software artifact or the program and what are the mutation operators that I have considered. Typically as I told you exhaustive mutation testing is never done, but operators are available as a fairly large exhaustive sets to choose from because you never know which operator we will come to use and it is a fairly routine mechanical programming task to generate mutation operators given the grammar of a software artifact. So, even though an exhaustive set of

mutation operators are available, mutation testing is never exhaustively done by applying every possible operator that is available, people usually do not do that.

Typically mutation testing is supposed to subsume a lot of other coverage criteria. We will see precisely a comparison to several other coverage criteria that we saw and how mutation testing subsumes which of those. Mutation testing is very difficult to apply by hand because it involves grammars and there are mutation testing tools available. Towards the end I will point you to links of these kind of tools. In fact, mutation testing is very difficult, it is difficult to apply by hand, it is difficult to automate also. Mutation operators are difficult to automate, but the per say testing process, after you have created the mutated program designing a test case to kill the mutant, there automation is difficult and you need human intervention and domain knowledge. In which operator to apply, they are also you need human intervention and domain knowledge.

(Refer Slide Time: 20:22)



The slide has a blue header with the text "Mutation testing: Plan ahead". Below the header, there is a white rectangular area containing a bulleted list. In the bottom right corner of the slide, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

- We will first learn how to define mutation operators and apply mutation testing to programs.
- Integration testing related mutation will follow.
- For input space mutation, we will discuss input languages like XML and defining mutation operators over them.

So, in the next lecture what are we going to see we learn how to define mutation operators how to apply mutation testing to programs or source code to begin with. Then we will apply mutation testing to do design integration. Finally, we will apply mutation testing to a markup language like XML and understand how mutation testing applies to input space to create invalid inputs and how the programs react to invalid inputs. So, that is the plan for mutation testing.

Thank you.