



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

Lecture - 43
Mutation Testing: Grammars and Inputs

Hello everyone, welcome to the next lecture of week 9. So, with this lecture I will finish mutation testing or syntax based testing.

(Refer Slide Time: 00:20)

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





So, this is, this was what we had as an overview picture of mutation testing for various software artifacts. So I said through the span of my lectures, we would look at mutation testing along these lines.

(Refer Slide Time: 00:34)

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

Focus of this lecture: Mutation for input space grammars and XML.



Here is an update of what we have done and where we are. So, what did we discuss? Mutation testing or syntax based testing is of two kinds: one directly based on grammar the other based on mutating programs. Grammar based testing can be applied for programs and mutation can be applied for programs. Grammar based testing cannot be applied for integration but mutation based testing can be applied for integration. Both grammar based and mutation based can be applied for specifications and again both grammar based and mutation based can be applied for directly testing input spaces.



What have we done? We completed all the things that are marked in green. We did grammar based testing for programming languages at an abstract level, I just told you what the coverage criteria on grammars are. We also did mutation testing for programs at the unit testing level and later on we specifically looked at mutation testing for design integration of generic kind and we went on and looked at mutation operators for specifically testing object oriented integration like, for a language like Java.

I told you that next we will do what is called input space partitioning based mutation with XML as one case study where I generate mutation based on grammars of input spaces. Specifically for grammars of XML formats for inputs as many languages. So, that will be the focus of this lecture. And as we continue to say the ones that I strike out we will not do because it is slightly outside the focus of this course.

(Refer Slide Time: 02:05)

BNF grammars for inputs

- BNF grammars are also used to define precise syntax for input of a program or method.
- E.g., consider a program that processes a sequence of deposits and debits.
 - Deposit format: *deposit account amount.*
 - Debit format: *debit account amount.*
 - Regular expression for input format:
 $(\text{deposit account amount} \mid \text{debit account amount})^*$
 - Describes a sequence of deposits and debits (in a abstract way).
 - E.g. inputs: deposit 739 \$ 12.35, debit 544 \$ 20.



So, this lecture we are going to look at grammars for inputs. Grammars for inputs are again given in backus naur form as a context free grammar specifically or as a regular expression and when are they used, what are they used for, they used for defining precise syntax of how the inputs look like. Most of the times inputs may not be just a simple integer or a Boolean value or a floating point number. Inputs will have a lot of structure, input could be a entire webpage, the form that you filled into a web page the data that you read from a database, it could have a lot of structure. Inputs with a lot of structure are described using formal languages specifically regular expressions or grammars. For example, consider a program that operates as a part of a banking system and let us say that program processes the sequences of deposits and debits.

So, the format of deposit will look like this, it will say deposit which is like a keyword which tells you the action to do this amount into the account. Similarly the format for debit would be debit which is like a keyword, this amount into this account, is that clear. Now what are the transactions that you can do as a part of a bank account? You can either deposits and amount into an account or you can debit an amount into an account and you can repeat this transaction any number of times in any order. So, this is the regular expression that describes that input transaction sequence. So, it says the input transaction is given as a regular expression which is the star of two entities.

The first expression says deposit account amount which says deposit this amount into the account, the second expression says debit account amount which means debit this amount, the given amount, into the account. There is an 'or' connecting them which means you can do one of these actions and after that there is a star. Star in regular expressions as we saw indicates that you can do zero or more occurrences in any order of either this or this, which means you can either deposit a given amount into an account or you can debit a given amount from an account and any bank transaction is a sequence of inputs of arbitrary deposits and debits.

Now, this could be an input. How do you read this? You say deposit into the account number 739, this amount, 12.35 dollars. Similarly you could say debit into the account number 544, the amount 20 dollars.



(Refer Slide Time: 04:36)

BNF grammars for inputs, contd.

- Regular expressions might not suffice for some programs' inputs.
- We can use BNF grammars in that case to express a larger set of inputs.
- Grammar notation for the above example:

```

bank ::= action*
action ::= dep | deb
dep ::= 'deposit' account amount
deb ::= 'debit' account amount
account = digit3
amount ::= '$' digit+ '.' digit2
digit ::= 0|1|2|3|4|5|6|7|8|9
  
```

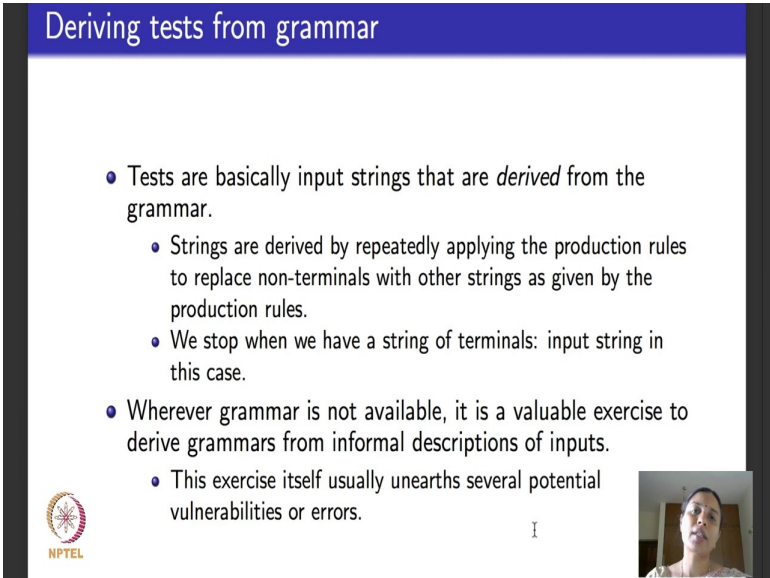



So, regular expressions, this is simply given as regular expressions, but sometimes as we saw regular expressions may not be enough and you might need slightly more expressive structures like context free grammars or arbitrary grammars which can be expressed in a normal form like Backus-Naur form. So, here is a same sequence of input transactions. In this slide I had given it to you as a regular expression. Here I am giving it to you as a grammar. How do I read it, I read it as follows.

So, what does a bank transaction? Bank transaction is a sequence of zero or more actions denoted by action star, then the next rules is action could be a deposit or a debit, dep for short form for deposit, deb short form for debit. A deposit is basically deposit the key word deposit some amount into an account number, debit is similarly the key word debit which depicts a fixed action of an amount into the account. Account is typically a 3 digit number, amount is given in dollars or any currency and its one or more digits followed by a decimal point followed by at least two digits, digits could be anything from 0 to 9 is that clear. So, I using this grammar I could derive an expression which looks like this.

So, I could say either deposit a particular amount which is this into this account what is it say, it says account is always a three digit number and amount is always in dollars. In this case, it has 0 it has one or more digits which is denoted by digit to the power of plus means 1 or more digits it cannot be 0 digits because you have to deposit or debit a nonzero number and it could also have decimal digits up to 2, decimal point two, is that clear.

(Refer Slide Time: 06:25)



The slide is titled "Deriving tests from grammar" in a blue header. It contains a list of four bullet points. The first bullet point states that tests are input strings derived from the grammar. The second bullet point explains that strings are derived by repeatedly applying production rules to replace non-terminals. The third bullet point states that we stop when we have a string of terminals. The fourth bullet point states that where grammar is not available, it is a valuable exercise to derive grammars from informal descriptions of inputs, and that this exercise usually uncovers potential vulnerabilities or errors. In the bottom left corner is the NPTEL logo, and in the bottom right corner is a small video inset showing a person speaking.

- Tests are basically input strings that are *derived* from the grammar.
 - Strings are derived by repeatedly applying the production rules to replace non-terminals with other strings as given by the production rules.
 - We stop when we have a string of terminals: input string in this case.
- Wherever grammar is not available, it is a valuable exercise to derive grammars from informal descriptions of inputs.
 - This exercise itself usually unearths several potential vulnerabilities or errors.

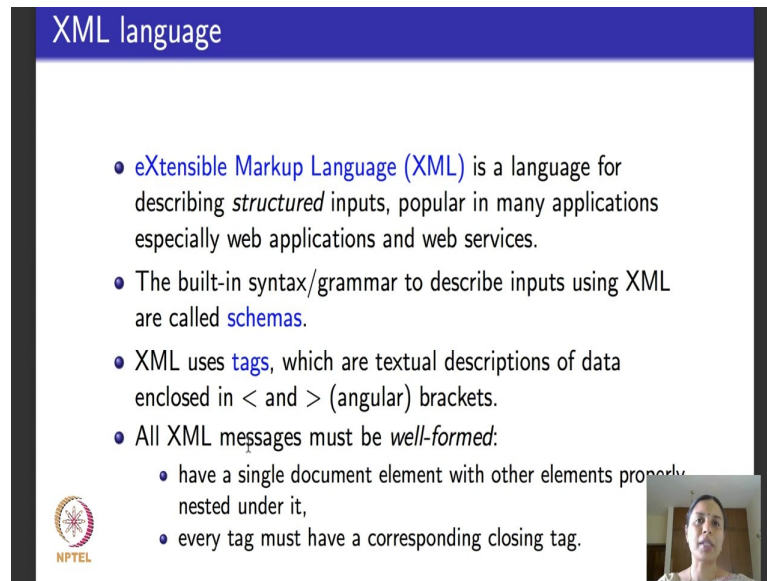
So, how do I derive tests from these grammars? Tests are basically input strings that are derived from the grammar. How do I derive? You follow the normal notion derivation that we saw for grammars when we began mutation testing last week.

So, what is the normal notion of derivation go like? You begin with the start symbol, designated start symbol and you use the production rules. Keep applying production rules by picking up one non terminal that is available at any given point in this string derived so far. Replace that non terminal by picking an appropriate rule that replaces that non terminal with another string. Keep doing this still you are left only with a string of terminals. In other words no more non terminals to replace. When you stop there that is the string that you derive and that belongs to the input space corresponding to the grammar that is given for that particular software artifact.

Whenever grammar is not available, you might wonder how are we going to use it? Usually it is very wise to be able to as a tester derive the grammar yourself from informal description of the input given. Some description or the other of the input will always be given because the software artifact and its input have to be described. Otherwise software development does not progress, but to be able to automatically derive tests you need grammar notations which are syntactically formal notations for describing inputs.

If it is not there then, as a tester, it is completely worth spending some time deriving the grammar from the informal description given. In the past empirical studies I have shown that just this exercise of deriving grammar from an informal description of the input sometimes unearths nontrivial bugs or vulnerabilities in the inputs to the software. So, it is a valuable exercise in its independent right and is also useful for mutating and producing inputs to a grammar.

(Refer Slide Time: 08:15)



The slide is titled "XML language" in a blue header. It contains a bulleted list of points about XML. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a person's face.

- eXtensible Markup Language (XML) is a language for describing *structured* inputs, popular in many applications especially web applications and web services.
- The built-in syntax/grammar to describe inputs using XML are called *schemas*.
- XML uses *tags*, which are textual descriptions of data enclosed in < and > (angular) brackets.
- All XML messages must be *well-formed*:
 - have a single document element with other elements properly nested under it,
 - every tag must have a corresponding closing tag.

We will see XML as a specific example because it is a very popular input format that is used across several different applications. So, what is XML? This module is not intended as an exhaustive introduction to XML please feel free to read up details about XML from other resources. I assume that you know some basics of it and I will use it mainly as a language to describe inputs and as a case study to show you how test cases are generated from the input.

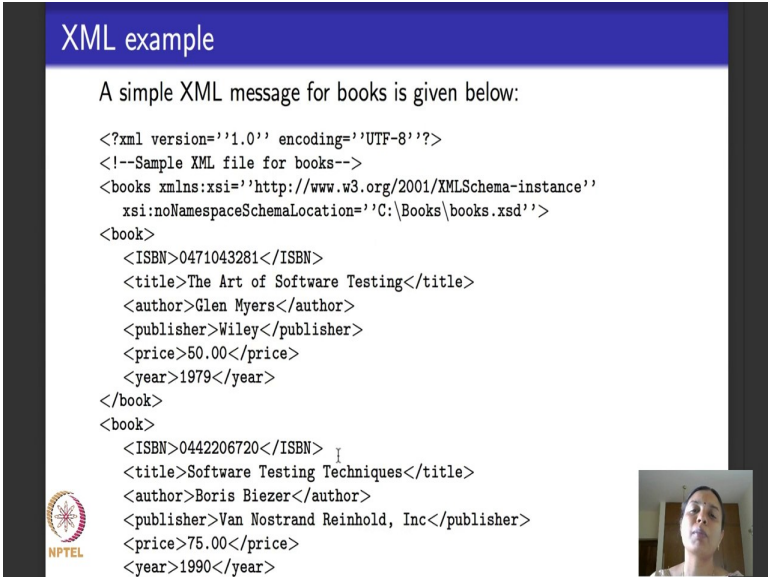
So, XML expands out to eXtensible Markup Language. It is a language for describing inputs that have a lot of structure in them, its popular across many applications typically its popular across web applications and web services. Of course, I basically work with a lot of embedded software design and development and if you see almost all the languages that I work with which is Simulink, AADL which are appropriately embedded design and architecture languages, they all have XML to describe their input language format. So, XML like all other structured formats, has a built-in syntax or a grammar to describe.

The built-in syntax or grammar that is available in XML it is what is called a collection of schemas. We will not introduce schemas thoroughly, but you can look up schemas from any standard book or reference on XML. In addition XML inputs use a notation called tags. What are tags? You can think of tags as some kind of textual description of data and the textual description of data is enclosed within angular brackets. Angular

brackets means this is the opening of the tag this is the closing of a tag. Each tag comes with this angular bracket and then for each open tag there is a corresponding closed tag. Inside each tag the some kind of a structured text that is described.

Typically we note that all XML messages must be well formed. What do we mean by that? They typically have a single document element with that other elements properly nested within it. There is a structure that is described in a nested fashion and every tag as I told you that has an opening tag, must also have a corresponding closing tag. So, in this sense if you know HTML which is an old language, it is a lot like HTML. It is another markup language, but of course, it has a more, a lot more sophisticated richer language to describe structured data formats.

(Refer Slide Time: 10:33)



XML example

A simple XML message for books is given below:

```
<?xml version='1.0' encoding='UTF-8'?>
<!--Sample XML file for books-->
<books xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:noNamespaceSchemaLocation='C:\Books\books.xsd'>
  <book>
    <ISBN>0471043281</ISBN>
    <title>The Art of Software Testing</title>
    <author>Glen Myers</author>
    <publisher>Wiley</publisher>
    <price>50.00</price>
    <year>1979</year>
  </book>
  <book>
    <ISBN>0442206720</ISBN>
    <title>Software Testing Techniques</title>
    <author>Boris Biezer</author>
    <publisher>Van Nostrand Reinhold, Inc</publisher>
    <price>75.00</price>
    <year>1990</year>
  </book>
</books>
```

The slide features the NPTEL logo in the bottom left corner and a small video inset of a person in the bottom right corner.

So, here is a simple example of how an XML message for books will look like. So, what does it say? If you ignore the first four lines, we will come back to that it says you concentrate on this. So, it says book opening tag is here the corresponding closing tag is here. So, the opening tag is given by this angular brackets book, the closing tag is given by the angular brackets book with an additional forward slash here.

Now what are the a further structures that are available inside the tag book? Inside the tag book there could be several other tags ISBN number is one tag, title of the book is another tag, author is another tag, publisher is another tag, the price of the book is

another tag and the year in which the book was published is another tag. Of course, this is not exhaustive, you could have additional details describing the book like for example, you could say whether it is which edition, first edition, second edition, whether it is a paperback and so on.

And if you notice each of these sub tags have an opening tag and a closing tag and the syntax is fairly clear right. So, opening tag looks like this within angular brackets. After the text that comes inside the opening tag, there is a corresponding closing tag which is the same as opening tags, tag, has angular brackets in addition has a forward slash. So, it says each book tag has ISBN number, title, author, publisher, price and year. So, this describes one book, the book with this number, title art of software testing, author Glen Myers, publishers Wiley, price is 50 dollars, let us say year of publication is 1979.



Now there could be one more book tag which again gives the same details for another book. That book's ISBN number is this. the title of the book is called software testing techniques, author is Boris Biezer, publisher is Van Nostrand Reinhold, price is so much, year is 1990 and so on. So, and what is the top part, top part is some kind of a header information it tells you which is the version and the encoding of XML that you are using and this is a comment which tells you the this is a sample XML file meant to describe books and this is the place where I kept the schema from. In this schema books dot x s d, which I have kept here I can describe the following, which I have not given you the schema.

(Refer Slide Time: 12:47)

XML example

The following are described in the schema for the XML for books (actual schema not given):

- A books XML message can contain any number of book tags.
- Each book tag contains six pieces of information:
 - title, author and are strings.
 - price is of type decimal numeric, has two digits after decimal point, lowest value is 0.
 - ISBN has up to 10 numeric characteris.
 - year is an integer with four digits.
- The criteria used for grammars can be used to derive XML messages for test inputs.



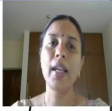

Please note that we have not described the schema, but I will tell you in English what the schema will describe in structured language of XML.

So, it says the schema will say the following it says this which begins here XML message can contain any number of book tags which are these. There is one book tag here, two book tags that I have shown. Each book tag contains 6 pieces of information title author and publisher, there are strings, price is a decimal numeric has two digits after the decimal point lowest value is 0. There will be a schema to describe this within XML I have not given you the schema.

Please remember that I have just described it in English ISBN is a number that has up to ten numeric characters year is an year with 4 digit and once I have the schema, the schema looks exactly like my grammar - it has terminals it has non terminals, it has rules that tell you how strings of non terminals and terminals can be derived. And so I can use the same criteria that I told you for grammars when we did that module to be able to derive XML messages for test inputs. In this case the test input will be a specific book or a specific set of books and I use the grammar to be able to derive those books. The grammar is obtained directly from the schema description of the XML message.

(Refer Slide Time: 14:10)

Mutation for input grammars

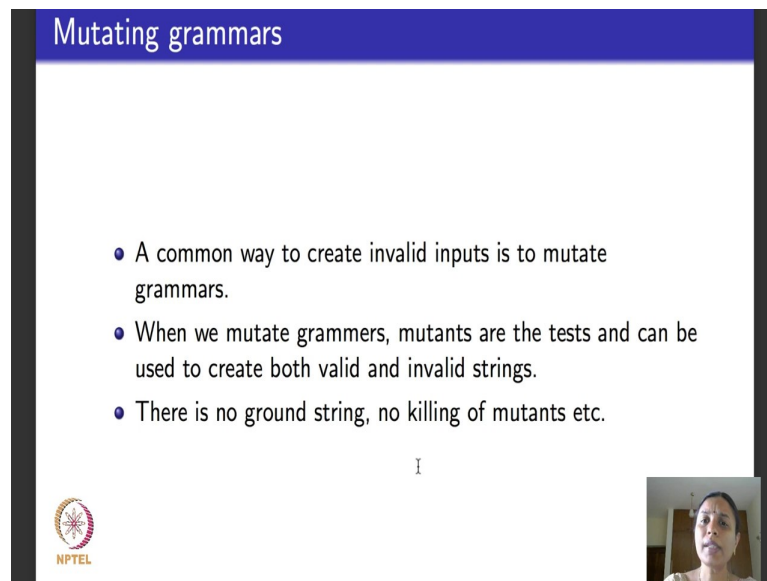


- While testing a program, tester needs to provide *malformed* inputs and see if the program *rejects* or *handles* them in a proper fashion.
- Such *invalid inputs* imply a lot about the functionality of the program and also check if the program/system handles faults properly.
- Security testing, safety critical system testing etc. all recommend or routinely test a system against invalid inputs.
- Input grammars can be mutated to generate invalid inputs as well.

Now, moving on when I consider mutation for input space grammars like XML or any other grammars while testing a program what is the tester, in addition to normal inputs the tester is this also useful if the tester provides malformed or invalid inputs. Why are invalid inputs necessary? Invalid inputs are necessary because it helps to check if the program does fault handling correctly. Invalid inputs also imply a lot about the core functionality of the program on correct inputs.

If the program works correctly on invalid inputs also then there is definitely wrong, something wrong with the functionality of a program. Typically in areas like security testing, testing for embedded systems, they insist that you do tests with invalid inputs at every level, specifically at system level and they have known to reveal a lot of vulnerabilities in the system. Just as we saw that standard mutation operators for input grammars can be used to generate valid inputs, mutation for input grammars can also be used to generate invalid inputs. Once you generate an invalid input you give it to a software artifact like a program and see how the program behaves on that invalid input.

(Refer Slide Time: 15:28)



The slide has a blue header with the title "Mutating grammars". Below the header, there are three bullet points:

- A common way to create invalid inputs is to mutate grammars.
- When we mutate grammars, mutants are the tests and can be used to create both valid and invalid strings.
- There is no ground string, no killing of mutants etc.

At the bottom left of the slide is the NPTEL logo. At the bottom right is a small video inset showing a person speaking.



Now we will see what are the common ways to create invalid inputs by mutating grammars. Common ways to create valid inputs by mutating grammars, we already saw, when we introduced grammars. Now I will tell you how to create invalid inputs by mutating grammars. Please note that when we mutate grammars the mutants that we create are the test cases themselves. This was not the case when we mutated programs. When we mutated programs, we started with the program if you remember we called that program a ground string, we mutated that program, the resulting program that we got was another program that had to be tested.

We have to separately write test cases to kill the resulting mutant program. When we apply mutation for input space grammars the mutated string is directly the test case right. So, there is no notion of killing ground string, those things do not exist and the test case could be valid input, the test case could be an invalid input and that as is, acts like a test case.

(Refer Slide Time: 16:33)

Mutating grammars

- **Non-terminal replacement:** Every non-terminal symbol in a production is replaced by other non-terminal symbols.
- This is a generic mutation operator, several different invalid strings can be produced.
- In the example for deposit and debit, the rule `dep::='deposit' account amount` can be mutated to create `dep::='deposit' amount amount and` and `dep::='deposit' account digit`.
- This gives the following tests: deposit 739 \$12.35 mutated to obtain deposit \$19.22 \$12.35 and deposit 739 1.



So, here are some standard operations to mutate grammar, what does grammar have? If you remember, grammar has non terminals, terminals, production rules and a designated non terminal called the start symbol. So, the mutation operators that we will see will basically work around these entities of a grammar.

So, the first mutation operator that we will see is what is called non terminal replacement. What does it do? Every non terminal symbol in a production rule in a grammar is replaced by other non terminal symbols. This is a very generic mutation operator. Remember typically large grammars for inputs typically XML grammars will have several different non terminals. So this says you can pick up any non terminal replace it with any arbitrary non terminal. So, it is like having probably dozens or even hundreds of rules just of this kind. As always while doing mutation you have to pick; which are the non terminals that I want to replace to create the invalid inputs that I want to test the program for.

And pick only those. Do not randomly apply non terminal replacement for replacing every non terminal, it does not make sense. For example, if you remember this grammar I will go back to that slide and show you the grammar for a minute. This was the grammar that we had which basically gave a sequence of input strings that either could debit an amount or deposit an amount from a particular account. So, for that grammar if

we had one of these rules which says a deposit is a string called deposit with the key word and it deposits an amount into the account, this, which are the non terminals here.



The non terminals could be amount or account. Deposit within quotes is like a terminal string, it is meant to be fixed its meant to indicate a keyword called deposit. So, the non terminals are amount an account. So, if I consider this mutation operator I can replace amount with account, amount with something else and so on that is what the first mutation I have done. The first mutation I have taken this production rule replaced this non terminal account with another occurrence of non terminal amount. So, instead of giving depositing an amount to an account you just give this invalid input which deposits an amount into an amount. So, we will see how the program that handles these input transactions will manage this input. Ideally it should have some exception handling to manage this another mutation.

That I could create would be to replace an amount with a single digit number, this would be a valid mutation, it will not produce an invalid string. Maybe it will be useful for testing what the program does on this particular valid input. So, these are just examples. Again non terminal replacement says that when you have a production rule in a grammar you could pick any non terminal and replace it with any non other non terminal both to produce valid inputs and invalid inputs. In this example we took this, it had two non terminals account and amount. First mutation that I created I replaced the non terminal account with amount, second mutation that I created I replaced this non terminal amount with the other non terminal that I had in the grammar which was digit. This resulted in an invalid mutant, this resulted in a valid mutant.

(Refer Slide Time: 19:44)

Mutating grammars

- **Terminal replacement:** Every terminal symbol in a production is replaced by other terminal symbols.
- This can again create several different mutants, some of them not appropriate.
- For the same e.g., the rule
`amount::='$$' digit+ '.' digit2`
can be mutated to create
`amount::='.' digit+ '.' digit2`, `amount::='$$' digit+ '$' digit2` and `amount::='$$' digit+ '1' digit2`.
- This gives the following tests: deposit 739.12.35, deposit 739 \$12\$35 and deposit 739 \$12135.

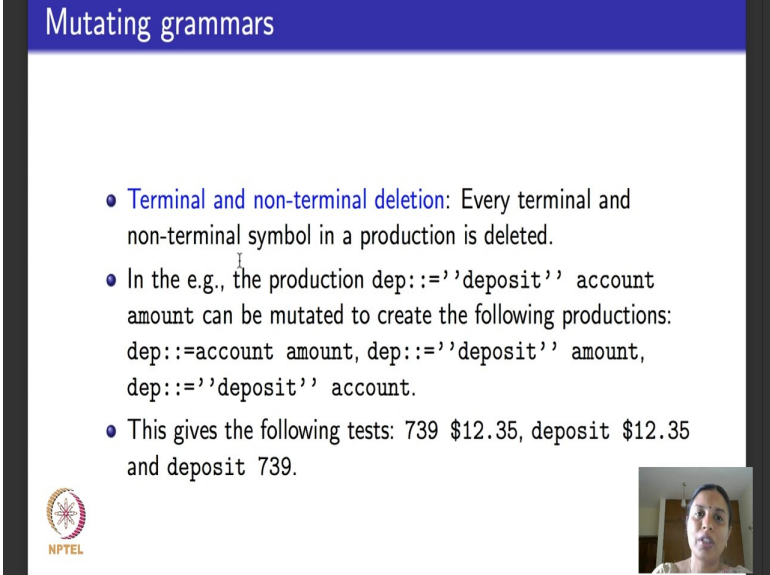


The next rule for mutating grammars would be instead of replacing non terminals you do terminal replacement, which means every terminal symbol in a production is replaced by other terminal symbols. Again this is a very generic rule, can create several different mutants, some of them completely junk, inappropriate, it is up to us as testers to pick and choose the one that we want wisely. For example, suppose I had this rule which was also a part of the grammar, amount equal to dollar and zero, one or more digit followed by a decimal point followed by exactly two digits, which basically gave the amount that could be deposited or debited with reference to an account. That can be mutated by replacing this dollar with a dot. That will create an invalid mutant, we will see what the program does.

Second kind of mutation replaces this dot with a dollar. We will see what happens because the ones within codes that terminals and I am considering terminal symbol replacement. So, I cannot replace digit, so I replace only the terminals. The third mutation, what is it do? It replaces the dot with a one. So, all these three mutations which replace one terminal symbol with some other terminal symbol create what are called invalid mutants, which could be used to test how the program that does, deals with these bank transaction handles these invalid mutants. So, if I apply the first mutant I will get a test case that looks like this. The program should figure out that this is junk input to handle it. Similarly if I apply the second mutant I get a test case which looks like this which is also junk input program should handle it. Third one is not a junk input it says



you deposit a fairly large amount into an account, if you have it why not the person is lucky.

(Refer Slide Time: 21:31)



Mutating grammars

- **Terminal and non-terminal deletion:** Every terminal and non-terminal symbol in a production is deleted.
- In the e.g., the production `dep ::= 'deposit' account amount` can be mutated to create the following productions:
`dep ::= account amount`, `dep ::= 'deposit' amount`,
`dep ::= 'deposit' account`.
- This gives the following tests: 739 \$12.35, deposit \$12.35 and deposit 739.



So, the third kind of production rules says you delete the terminals and the non terminals. Every terminal and every non terminal in a production is deleted. Of course if you delete the whole thing you left no grammar left. So, you pick and choose a subset that you would want to delete and see when the restricted grammar is applied it will generate obviously, lesser input strings, what happens to that. For example, there was this production rule the that we had in the grammar for bank transactions. This can be mutated to create the following: I just delete the terminal deposit from that rule, that is the first mutant that I get. Second mutant, I delete the account from the rule, so I just get deposit amount. Third mutant, I delete the amount from the rule, so I get deposit account.

All three of them will produce junk or invalid or malformed test cases. We have to figure out what the program does. If the program does not handle exceptions on these test cases we have to build in those exception handlers.

(Refer Slide Time: 22:27)

Mutating grammars

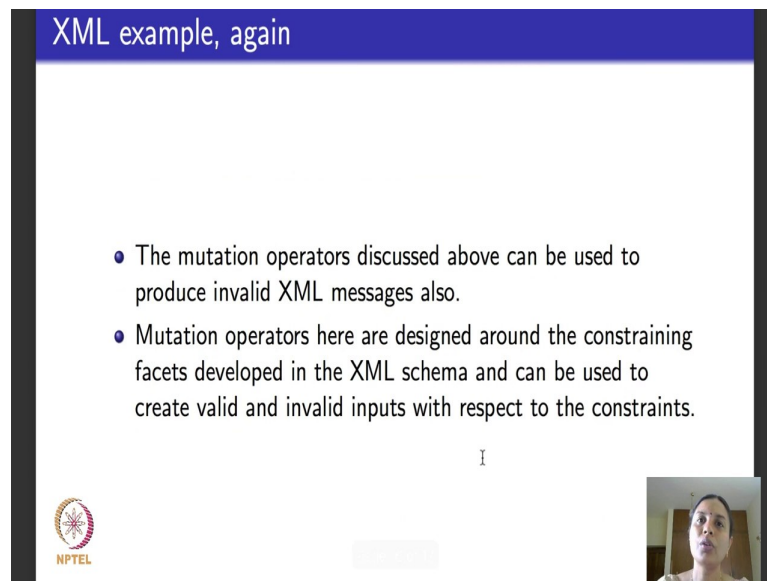
- **Terminal and non-terminal duplication:** Every terminal and non-terminal symbol in a production is duplicated.
- This mutation is sometimes called *stutter* operator.
- In the e.g., the production $\text{dep} ::= \text{'deposit' account amount}$ can be mutated to create the following productions:
 $\text{dep} ::= \text{'deposit' 'deposit' account amount},$
 $\text{dep} ::= \text{'deposit' account account amount}$ and
 $\text{dep} ::= \text{'deposit' account amount amount}.$
- This gives the following tests: deposit deposit 739 \$12.35, deposit 739 739 \$12.35 and deposit 739 \$12.35 \$12.35.



The next kind of production rule is, instead of deleting them you duplicate terminals and non terminals. Duplicate means what, you create another copy of one terminal or one non terminal. Again please remember always in mutation testing only one mutation at a time. So, do not duplicate two terminal symbols or in this in this case, do not delete two non terminal symbols or terminal symbols, only one operator at a time. So, because I duplicate, the term for duplication in computer science theories called stuttering.

So, the duplication operator is also called a stutter operator. For example, if I consider the same thing deposit which is deposit and into an account and amount, it can be mutated by replicating the word deposit twice, by replicating the non terminal account twice, by replicating the non terminal amount twice. All the three cases, except for maybe the, no sorry not the last one, the first two cases it will create invalid test cases because if the program cannot handle two word deposits, two keyword deposits. Second case the program cannot handle two account numbers. In the third case the program may or may not handle the amount based on whether there is a space or not. If you include this space, the program will consider it as an invalid input. All the three cases you all get invalid input except maybe for the third case where sometimes you all get valid input and this will again test if the program is handling invalid inputs correctly. It should not do anything, it should raise an exception and stop.

(Refer Slide Time: 23:56)



The slide has a blue header with the text "XML example, again". Below the header, there are two bullet points:

- The mutation operators discussed above can be used to produce invalid XML messages also.
- Mutation operators here are designed around the constraining facets developed in the XML schema and can be used to create valid and invalid inputs with respect to the constraints.

At the bottom left of the slide is the NPTEL logo. At the bottom right is a small video inset showing a person speaking. There is also a small "I" character and a white rectangular box near the bottom center of the slide.



So, now all these rules that we saw, four rules which is terminal replacement, non terminal replacement, terminal and non terminal duplication, terminal and non terminal deletion, they can be applied for XML example that we saw again. For the sake of succinctness, I have not really worked out an XML example. Feel free to ping me in the forum if you have any doubt and I will be able to explain it to you.

When specifically for XML people usually design mutation operators that check for constraints, like in the earlier thing we saw right that ISBN must be a 10 digits numeric string. So, they will make ISBN 0 empty string and see how the program handles it and so on. So, they are designed to work around the constraining artifacts to create invalid inputs with reference to the constraints.

(Refer Slide Time: 24:44)

Mutation testing: Summary

- Mutation or syntax-based testing is a powerful testing technique.
- Mutation operators are based on the underlying grammar of the software artifact.
 - Grammar always available for programming languages.
 - Might have to be derived for inputs.
- Mutation applied for programs need to be killed by test inputs.
- Mutation applied for inputs are tests themselves.
- Check the tool muJava available at: ^I
<http://cs.gmu.edu/~offutt/mujava/> to do mutation for Java programs.



So, here is a summary of mutation testing. This will be the end of mutation testing that we will see. Mutation or syntax based testing is a very powerful testing technique as I told you, it subsumes several other coverage criteria as we saw in one lecture module. They are based on the underlying grammar of the software artifact. For every programming language, grammar is available. For inputs whenever grammar is not available, it is a worth exercise to derive the grammars for these inputs.

Mutation when applied to programs result in mutated programs, they have to be killed by designing test cases. We saw two notions of killing: strong and weak killing whereas, mutation when applied to inputs directly result in test cases. So there is no notion of ground strings or killing mutated programs. And specifically if you are interested in doing mutation for Java, I recommend that you check out this tool that is available as a part of a project from George Mason and a University in China, it is a pretty good tool to do mutation for Java programs.

I also know of a couple of other tools for mutation for XML and for old Fortran programs which is a very good tool called Mothra, but feel free to Google for mutation tools for other kinds of languages. So, in the next lecture I will give you a summary of what we have done till now.

Thank you.