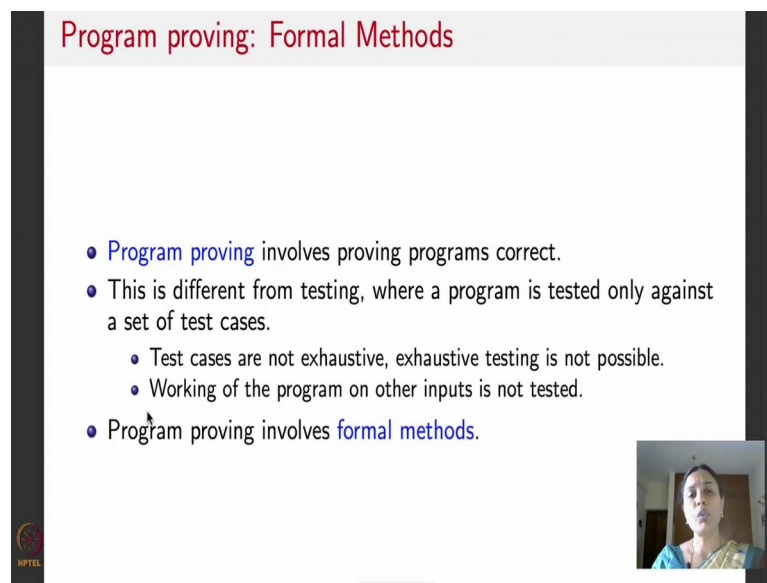


Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 50
Symbolic Testing

Hello, there I am going to begin a completely new module with this lecture. What we are going to do a bag of techniques called symbolic testing techniques which I said I would do at the end of this course. So, symbolic testing broadly includes a new set of testing techniques, which are actually speaking more than a decade, more than 3 decade old, but have become popular now. So, over the next 4, 5 lectures we will exhaustively understand symbolic testing.

(Refer Slide Time: 00:43)



Program proving: Formal Methods

- Program proving involves proving programs correct.
- This is different from testing, where a program is tested only against a set of test cases.
 - Test cases are not exhaustive, exhaustive testing is not possible.
 - Working of the program on other inputs is not tested.
- Program proving involves formal methods.

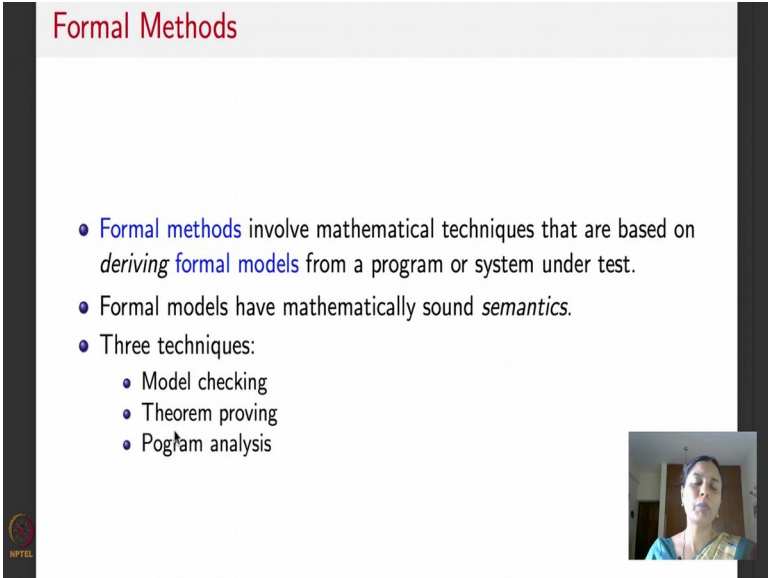
So, before I move on with symbolic testing if you remember right in the first week, I told you about the things that we will not do as the part of the testing course, and one of the things that I said we will not do is related to proving programs correct, which broadly use a bag of techniques collectively known as formal methods. Program proving means taking a program and showing that it works correctly no matter what. What do we mean by no matter what? That is if given any input that is legally acceptable by a program, it will always produce correct output. Now one thing to be noted before we move on is that program proving is different from program testing. When I test a program all that I do is

to give some inputs using carefully designed tests selection methods which we saw almost throughout the course, and check if the program works on these test inputs correctly.

Now, we all know exhaustively you cannot go on giving test inputs to a program, it is not feasible and it is not necessary also. That is not the goal for testing. It is to be able to cleverly design test case techniques to find the error. Given that we cannot exhaustively test a program what we can do using testing is to show that on the inputs that we have tested, the program does the way it is supposed to be doing, it either works correctly or it has an error. We cannot make any claim about the inputs that we have not tested the program on. Hence program can never be proved to be correct on the other inputs.



Program proving deals with proving a program to be correct and that involves a bag of techniques as I told you which are called formal methods. What are formal methods? I am not going to introduce you even to a cursory extent of what formal methods is, I am just listing here so that if you are interested you could look up for other courses on NPTEL or other material in this area.

(Refer Slide Time: 02:30)



Formal Methods


- **Formal methods** involve mathematical techniques that are based on *deriving formal models* from a program or system under test.
- Formal models have mathematically sound *semantics*.
- Three techniques:
 - Model checking
 - Theorem proving
 - Program analysis

Formal methods is the word formal says involves mathematical techniques that are based on taking a particular software artifact co design requirements, you derive formal models from that and then you verify or validate a program. Formal methods because their formal, have mathematically sounds semantics. There are 3 broad categories of formal

methods available; model checking, theorem proving and program analysis. There is a currently running NPTEL course on model checking.

(Refer Slide Time: 03:18)



Program testing vs. proving

- Program testing and program proving can be considered as extreme alternatives.
- In testing, one can be assured that sample test runs work correctly by carefully checking the results.
- Correct execution of program for inputs not in the sample is still in doubt.
- In program proving, one formally proves that program meets its specification (of correct behavior) without executing the program at all.
- Not all proofs can be automated and most proofs need to be done by hand.

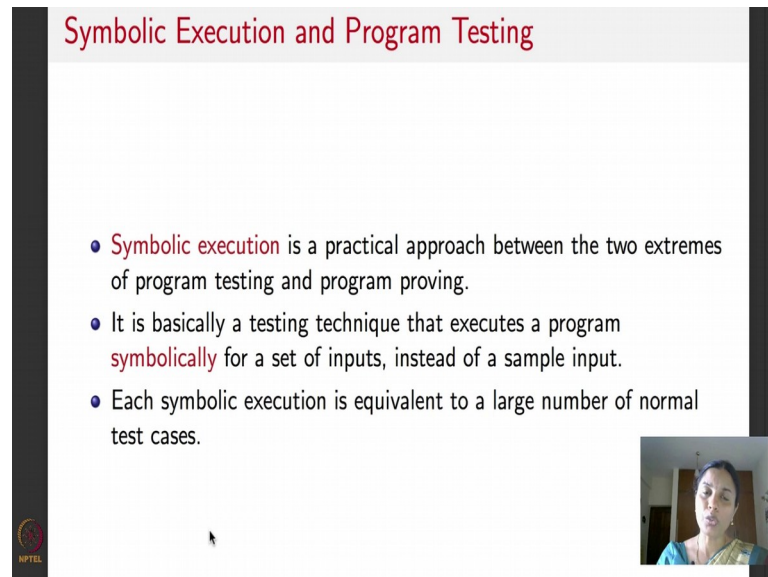
Now, program testing as I told you that is what we have dealt with in the course, testing a program. The other extreme is program proving which is proving a program correct. They can be thought of as extreme alternatives, one does not replace the other. They both can be used together and it does not if you have proved a program correct that does not mean that you need not test a program, if you have tested a program it does not mean that you need not prove a program to be corrected. They are complementary in fact, they consider extreme alternatives.

In testing as I told you can be assured that the sample test runs work correctly by checking the results, the actual output does not match the expected output or not. And correct execution of program on inputs that have not been tested is still in doubt. No sweeping claims about the program is correct, because it is working on all my test inputs that is wrong to state. But program proving you do not actually execute it one vector at a time one test vector at a time, you use a bag of techniques that are listed here to mathematically prove that a program meets its specification.

What is its specification? Is any requirement about a property of a program; could be a requirement that states that the program needs its functionality of a certain kind and so on. It may or may not involve executing the program and the thing to be noted is that

unlike testing, where there is heavy automation, program proving cannot be automated at all. Typically model checking is where there is automation possible, but it is not scalable for all kinds of software. In theorem proving and in program analysis full automation is never possible, proofs are done by hand.

(Refer Slide Time: 05:00)



The slide has a title bar at the top that reads "Symbolic Execution and Program Testing". Below the title, there is a list of three bullet points. In the bottom right corner of the slide, there is a small rectangular video inset showing a woman speaking. In the bottom left corner, there is a small logo for "NPTEL".

- Symbolic execution is a practical approach between the two extremes of program testing and program proving.
- It is basically a testing technique that executes a program symbolically for a set of inputs, instead of a sample input.
- Each symbolic execution is equivalent to a large number of normal test cases.


What we are going to do symbolic execution as I told you. Symbolic execution can be thought of, or has been introduced by several papers that deal with this, as a practical approach that or else stands in between program testing and program proving. Symbolic execution is basically a testing technique, that is why we are looking at it in this course. It again executes a program, but it does not execute the program the way we have seen it till now, it executes the program in a particular style called symbolic execution that is the term. So, we will understand what that term is today. Each symbolic execution can be thought of as executing a program on large set of actual inputs.

(Refer Slide Time: 05:40)

Example 1

Consider the following program fragment:

```
1 Sum(a,b,c)
2   x = a+b ;
3   y = b+c;
4   z = x+y-b;
5   return(z);
6 end
```



So, I will explain it through an example, here is a small piece of code very small piece of code what does it do? It takes 3 numbers a, b and c. Do not worry about what the types are. It is not given here, it is not important for us. Let us assume without loss of generality that they are integers, takes these 3 numbers a b and c and then what does it do it computes the sum of a b and c. But it does it in the roundabout way, first it adds a and b stores it in x, then it adds b and c stores it in y.

Please remember it is supposed to add a with b and then with c, but by then what it does is that because it is adding b twice between x and y when it finally, computes z by doing $x + y$ it subtracts one b. This is just some way of writing program may not be the most intuited way, but for some reason let us say it takes it like this, and then it returns z which is the sum of a b and c. Let us say I give you the task of testing this program, what will you do? You will give some values to a, b and c based on some criteria and then test the program. You will see what for these values of a, b and c that you have given, what is the expected output. Then you will run this piece of program and record the actual output and see if the expected output matches the actual output. If it does then you say that the test case is passed.

(Refer Slide Time: 07:02)

Normal execution of Sum on inputs						
Normal execution of program Sum on inputs 1, 3, 5 is given in the table below:						
After stmt.	x	y	z	a	b	c
1	?	?	?	1	3	5
2	4	-	-	-	-	-
3	-	8	-	-	-	-
4	-	-	9	-	-	-
5	returns 9					

In the above table, - represents unchanged values and ? represents undefined (uninitialized) values.

So, our actual test statement will look like this. So, let us say the inputs you are going to give to the program a, b and c are 1, 2 and 3. I have written it here: a is 1, b is 2, c is 3. I will come back and tell you what this question marks are. What is question marks mean is that as you give statement, as you give inputs a, b and c as 1, 2 3, let us say you maintaining this table which records what happens after these 5 statements in the program are executed. What are the 5 statements in the program? You go back to the program, these are the 5 statements. The first statement is just the function name second statement is this, third statement is this, fourth statement is this, fifth is the return statement, sixth is the end statement.

So, you keep this table, where you give inputs a, b and c as some concrete values, in this case let us say 1, 2 and 3, and then in the statement you record the values of the variables in the program as the program executes. There are 3 other variables in the program x, y and z. x and y can be thought of as internal variables, z is the variable that is returned. So, after statement one which is the first statement in the program, which let us you pass inputs as a, b and c, x, y and z are not assigned any values, that is why there are these 3 question marks. After statement number 2 executes x becomes a + b, what is a, 1. What is b? 3. So, a + b is 4. So, x becomes 4.


I have put these dashes they can be interpreted as values have not changed, in the sense that I still do not know what y is, I still do not know what z is, a remains 1, b remains 3, c

remains 5. If this is confusing you copy exactly what was there in the previous row here that is what we mean by this dash. Now after statement number 2 which computes y as $b + c$, y gets assigned value 8 all other values remain the same. Similarly after statement number 3, what does z do? z is $x + y - b$. That will be $4 + 8$ which is 12, b is -3, $12 - 3$ is 9. What is line number 5 do? Line number 5 returns 9. Using this statement you see actual output is the same as expected output, yes, then you say my test case is passed, otherwise you say my test case is failed.

Now, let us say you decide to give another set of inputs to the program Instead of 1 3 and 5, may be you will give 0, 0 and 0 to see what happens how the program behaves. For the same set of inputs, you will again do the same thing, you will give the inputs look at the expected output look at the actual output compare and see if the test case is passed or failed. You can repeat the process for another set of inputs, repeat the process for another set of inputs and do on carry on till test a criterion is met, or till you are satisfied as a tester that the program is working correctly or when you reach a state where you know that the program is erroneous. In this case the program segment that is given here is not erroneous. So, it does work correctly.


(Refer Slide Time: 10:23)

Symbolic execution of Sum



After stmt.	x	y	z	a	b	c	PC
1	?	?	?	α_1	α_2	α_3	true
2	$\alpha_1 + \alpha_2$	-	-	-	-	-	-
3	-	$\alpha_2 + \alpha_3$	-	-	-	-	-
4	-	-	$\alpha_1 + \alpha_2 + \alpha_3$	-	-	-	-
5	Returns $\alpha_1 + \alpha_2 + \alpha_3$						

The above table represents symbolic execution of $\text{Sum}(\alpha_1, \alpha_2, \alpha_3)$. Path Condition/Constraint is abbreviated as PC.



Now, let us assume instead of giving these concrete values as 1, 3 and 5, or any other concrete values let us say I want to execute it symbolically. What I will do in that case is that, I will say a is not 1 or any concrete value, a is some symbolic value, let us say α_1 . b

is another symbolic value let us say α_2 , c is another symbolic value let us say α_3 . I will come back and tell you what this last column is. The rest of the table exactly is the replica of this table, but instead of having concrete values here, it executes it on symbolic values. How do you interpret symbolic values at this stage?

Think of symbolic values as being place holders for concrete values. So, when I say a is α_1 , you can think of a as being taken the symbolic value α_1 , where α_1 will eventually be 1 concrete value. Similarly b is also symbolically α_2 , α_2 could represent any abstract value for b that matches the type of b, eventually be substituted by a concrete value for b. Similarly for c. The rest of the table is the same--- it says in statement 1, x, y and z are not yet assigned. After statement 2 x becomes $\alpha_1 + \alpha_2$ y and z remain the same, all these remain the same. x becomes, x remains the same here, after statement 3 y becomes $\alpha_2 + \alpha_3$ z is not yet assigned, α_1 α_2 α_3 are the same, after statement 4, x and y are the same as before, z becomes this and so on is this clear? What does the program return?

This table is the same as this table there are 2 differences; one is here a, b and c are given concrete values as input, a, b and c are given abstract or symbolic values as input everything else is the same. The last column represents what is called a path constraint or a path condition. PC is short form for path constraint. Path constraint here is marked as true. You can think of it as you remember our reachability, infection and propagation condition, you can think of PC as a set of conditions that collect those predicates that predicates, corresponding to reachability, infection and propagation.

If you see this program, it does not have any decision statements any branching statements. So, reachability is always true. Infection and propagation will also be eventually true. There is nothing like path constraint here. So, the path constraint is marked as true right. So, this is how a program is symbolically executed. So, just to repeat what I said, you want to symbolically test or execute a program, the program looks like any other normal program written let us say C or Java or whatever programming language. For normal testing you would give concrete values to inputs like in this case you would give a, 1, b, 3, c, 5. In symbolic testing you would give symbolic values to inputs.


And then what do you do? You collect expressions corresponding to internal variables because you have not given concrete values you cannot evaluate an expression $x = a + b$, you can only collect another expression in terms of the symbolic variables. One natural doubt that might come to your mind or that you might ask your self is, what am I doing differently than what is given in this program here? Instead of writing $a + b$, I am writing $\alpha_1 + \alpha_2$, instead of writing $b + c$ I am writing $\alpha_2 + \alpha_3$ and so on. That is natural, in some sense we are not doing anything. We just saying that α is an arbitrary place holder value for a β is another arbitrary place holder value for b, α , sorry α_2 is another arbitrary place holder value for b, and α_3 is an arbitrary place holder value for c, that is exactly what is symbolic execution.

(Refer Slide Time: 14:25)

Example 2

Consider the following program:

```
1 int twice (int v) {
2     return 2 * v;
3 }
4 void testme (int x, int y) {
5     z = twice(y);
6     if(z == x) {
7         if(x > y + 10)
8             ERROR;
9     }
10 }
11 int main() {
12     x = sym_input();
13     y = sym_input();
14     testme(x, y);
15     return(0);
16 }
```



I will illustrate it with another example, this time it is a slightly bigger code. So, let us go through the code first. So, here is a program. So, it has a function called twice which takes an argument of type integer given as variable v and it returns an integer type. It is just a one line function. What does it do? It multiplies v by 2 and returns it nothing else. Now there is another program which is called test me. These names are given just for improving readability there is nothing else to it. This program test me returns what? It takes 2 arguments x and y.

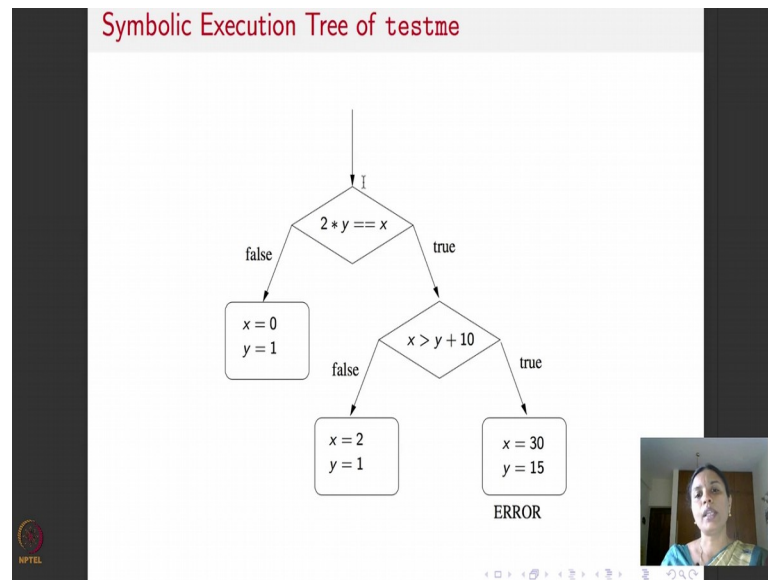
So, what does it do the first line it calls this function twice. To it might as well directly said $z = 2y$, but I have just said this to illustrate a particular point that I will come back

to. So, it calls the function twice which basically multiplies its argument by 2 assigns it to z and then what does it do. It says if z is equal to the other input x , then you go and check for one more condition. If $x > y + 10$, then you say the program reaches an error statement reaches an error state or error condition. I have not explicitly written what the error condition is, but indicated more like that there is an error just by writing it as error. It may not syntactically be legal in any programming language, but think of it as representing an abstract piece of code that indicates an error or you can think of it as printing an error state.

So, what does this program do? This program is very simple: takes 2 arguments x and y does z is equal to $2y$, then passes 2 if conditions, checks, first checks if $z = x$, then checks if $x > y + 10$. Both the if conditions have passed it says there is an error, otherwise it does not do anything, this is the program. Then this is being called by main calls tests me, and then it says return 0 and I have also said what are inputs x and y to test me? I say x is given as a symbolic input read this `sym_input` as x is being given as a symbolic input, similarly y is being given as a symbolic input.

Now, before we move on and understand how symbolic execution works for this program, what do I hope to achieve? I am telling you that there is an error in the program using symbolic testing my goal would be to find this error. When can I reach this error? I can reach this error if this if condition passes and if this if condition passes. You might want to spend a minute thinking about which will be the values of x and y for which both the if conditions will pass. By this is a small enough program if I look at this program for a couple of minutes and study its code, I will be able to come up with 2 values for x and y such that both the if conditions pass and I will reach the error statement for sure.

(Refer Slide Time: 17:35)



Let us say instead of staring at this program and thinking about it manually yourself, you want to be able to systematically do it. How will you do it? Let us say here is a way. So, I will draw what is called an execution tree of a program. What is this execution tree of a program tell you? This can again be related to our reachability, infection and propagation condition. Right in the very first statement after calling this function twice(y), I can always reach here. There are no if conditions, this is the first place where a decision takes place. If this if condition itself is false then this program exits.

Suppose this if condition is true then it goes into the second if condition, that is what this node captures here. So, it says, this is a first if statement that is encountered, it can be true or it can be false. If it is false then the program does something, if it is true then the second if statement is tested which in turn, can be true and false. Now what are these 3 leaf nodes in the tree that I have given you. I have given you an example test case that will make this if condition false. For example, if x is 0 and y is 1, then this if condition will be false why? Because it says 2 is equal to 0 which is false.


So, this can be thought of as a test case that will make this false. If this is true which in some test case it could be either this or this, I will go ahead and test the next statement. So, a test case for example, x = 2, y = 1 will make this one true and then this one false. Similarly a test case for example, x = 30 and y = 15, will make both the if statements true and this is the test case that corresponds to the error.

So, ideally to exhaustively test this program or to do edge coverage or branch coverage or predicate coverage for this program, you make this statement true once false once, you make this statement true once false once, that is what this does. This test case will make this statement false, the first if, then any of these test cases will make the if, this if statement true and this exclusively make this second if statement false, this will make the second if statement true which will hit the error. So, symbolic execution, what it does is, it will go back to this program, it try to not give concrete values for x and y, it will give symbolic values for x and y, and what it will do it collect constraints corresponding to these if conditions. Towards the end, it will try to solve the constraint and try to automatically obtain these value, that is what symbolic execution does.

(Refer Slide Time: 20:11)

Symbolic Execution

- Symbolic execution uses **symbolic values** instead of concrete data values as input.¹
- Program variables are represented as **symbolic expressions** over the symbolic input values.
- Hence, output values computed by a program are expressed as a function of symbolic input values.
- **Symbolic state** σ maps variables to symbolic expressions.
- **Symbolic path constraint**, PC, is a quantifier-free, first-order over symbolic expressions.



So, symbolic execution uses symbolic values instead of concrete values as inputs, inputs means test case inputs. Program variables are represented as symbolic expressions over the symbolic input values. Like for example, here going back to the previous sum code, there were 3 program variables x, y and z, when I symbolically executed them, I represented them using symbolic expressions I represented x as $\alpha_1 + \alpha_2$, y as $\alpha_2 + \alpha_3$ and so on right. So, that is what is written here. Program variables which are internal variables in a program that do not correspond to inputs now become symbolic expressions as the program runs.


The symbolic expression can be thought of first it is an arithmetic expression that is there already in the program, but has symbolic variables because it is not being evaluated. Output values are also a function of symbolic input variables. What is a symbolic state? Symbolic state at any point in a time gives me the symbolic values of all the variables. Like for example, here going back to the sum program, if I read this at any point in time I get the symbolic state. It says x is $\alpha_1 + \alpha_2$, y is $\alpha_2 + \alpha_3$, z is not yet assigned value, a is α_1 , b is α_2 , c is α_3 , the program is executing statement number 3. That is what is symbolic state will say.

Symbolic path constraint, it is what? We have introduced logic, if you remember as per the logic that we introduced, is a quantifier free predicate logic formula or it is a quantifier free first order logic formula. Predicate logic is the same as first order logic. Obviously, there are no quantifiers and the path constraint is a normal predicate logic formula. If you go back here, the path constraint to reach this statement will be this is true and this is true. The path constraint to reach this statement would be not of this because that is when this becomes false. For an example program like this, there are no path constraints or path constraint is just the Boolean constant true, because there are no decision statements in this program.


So, I will just quickly repeat what I said in this slide. Symbolic executions symbolically executes the program over inputs, all the internal variables of program, every internal variable will have some expression in the program that evaluates it. Instead of evaluating it you just substitute it with symbolic variables and keep it as symbolic expressions. A symbolic state will be a large to pull that gives symbolic values or symbolic expressions for the inputs and the variables in the program. Along with that, I collect path constraints which can be thought of as, all the decision statements that you encounter in the program take the statement as it is when it is true, takes the negation of the statement when it is false keep AND-ing them and since there are no quantifiers here, it is a quantifier free first order logic or a predicate logic formula.

(Refer Slide Time: 23:25)

Symbolic execution in software testing




- In software testing, symbolic execution is used to generate a test input for each execution path of a program.
- An execution path is a sequence of true and false, where true (respectively, false) at the i th position in the sequence denotes that the i th conditional statement encountered along the execution path took the "then" (respectively, "else") branch.
- All the execution paths of a program can be represented using a tree, called the **execution tree**.




What is symbolic execution do? In software testing symbolic execution is used to generate a test input for each execution path of a program. As I told you if you go back to this tree, an execution path is a sequence of true false values right? If I say this is a path of length one which makes first condition false, this is the path of length 2 which makes the first condition true, second condition false and so on. So, a path constraint or an execution path is a sequence of true and false values, where true at the i th position in the sequence denotes that the i th conditional statement is true or it took the then branch and a false in the i th position denotes that the i th conditional statement took the else branch. This should be obvious as we see it in that example and we represent the execution path using an execution tree, this is an execution tree.

(Refer Slide Time: 24:24)

Symbolic Execution: Symbolic state σ



- At the beginning of symbolic execution: σ is initialized to an empty map.
- At a read statement: symbolic execution adds the mapping assigning the variable being read to its new symbolic variable, to σ .
- At every assignment $v = e$: symbolic execution updates σ by mapping v to $\sigma(e)$, the symbolic expression obtained by evaluating e in the current symbolic state.




So, at the beginning of the symbolic execution the state, symbolic state sigma is assigned an empty map, which is dash dash dash, a question mark question mark question mark. At every read statement assigns a symbolic variable, at every assignment statement it assigns a symbolic expression, like here. For example, there was this statement which can be thought of as a read statement, after statement one symbolic values are assigned, after statement 2 symbolic expressions is assigned, that is what I mean. At every read statement, symbolic execution adds the mapping assigning variable being read to it is symbolic variable. At every assignment statement it adds symbolic expression.

(Refer Slide Time: 25:06)

Symbolic execution: Path constraint PC

- At the beginning of symbolic execution: PC is initialized to true.
- At a conditional statement $\text{if}(e) \text{ then } S_1 \text{ else } S_2$: PC is updated to $\text{PC} \wedge \sigma(e)$ ("then" branch) and a fresh path constraint PC' is created and initialized to $\text{PC} \wedge \neg \sigma(e)$ ("else" branch).
- If PC (PC') is satisfiable for some assignment of concrete to symbolic values, then symbolic execution continues along the "then" ("else") branch with σ and PC (PC').
- Note: Unlike concrete execution, both the branches can be taken resulting in two execution paths.
- If any of PC or PC' is not satisfiable, symbolic execution terminates along the corresponding path.



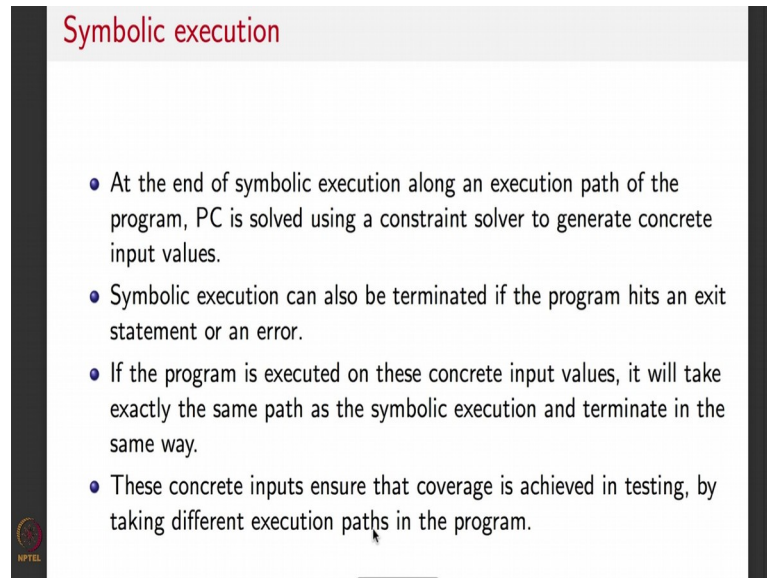
Now, path constraint, path constraint initially is true to start with. The moment you get a first if statement if e then S else 1, S1 else S 2, this is how it looks like. e is actually the condition whatever your path constraint was you and it with sigma of e which means take e substitute with symbolic expression AND it. This is for the then branch when that conditional value is to true and you have to and it with negation of sigma of e to represent the else branch for the condition to be false.

Like for example, if I go back here there were 2 path constraints, one for this if and one for this if. Till I come here path constraint is true and then I will do $\text{true} \wedge x \text{ for } 1$, and I will do $\text{true} \wedge \neg z \text{ is equal to } x$. So, $\text{true} \wedge z \text{ is equal to } x$ will make this takes to true branch true, and $\neg z \text{ is equal to } x$ will make this statement take the false branch. And when I come here I already had $\text{true} \wedge z \text{ is equal to } x$, if I negated with AND of if I and it with $x > y + 10$ then I get this if I and it with $\neg (x > y + 10)$ then I get this. So, that is what I said. As you are running a program executing a program you would have a path constraint, for every if statement that you encounter you AND it with e once, and you end it with negation of e once. AND-ing it with e intuitively represents the program taking the then branch and ending it with negation of e represents the program taking an else branch.

Finally what you do you would have collected a path constraint like here you would have collected this constraint and this constraint. You solve it, means you give it to a solver will return value say for this to be true and this to be true here is one value. If x is 30 y is 15 both are true. This becomes your test case that you execute the program on and similarly to how do you get this test value, you solve this and negation of this, then this will be the test case how do you get this test case you solve negation of this.

So, each path constraint is solved and if there is a assignment then the program will continue along the then or the else branch right. If a path particular path constraint is not satisfiable for some reason cannot be solved then you will say symbolic execution terminates then and there.

(Refer Slide Time: 27:39)



The slide is titled "Symbolic execution" in red text at the top. It contains a bulleted list of four points. The first point states that at the end of symbolic execution, a path constraint (PC) is solved using a constraint solver to generate concrete input values. The second point states that symbolic execution can be terminated if the program hits an exit statement or an error. The third point states that if the program is executed on these concrete input values, it will take exactly the same path as the symbolic execution and terminate in the same way. The fourth point states that these concrete inputs ensure that coverage is achieved in testing by taking different execution paths in the program. The NPTEL logo is visible in the bottom left corner of the slide.

- At the end of symbolic execution along an execution path of the program, PC is solved using a constraint solver to generate concrete input values.
- Symbolic execution can also be terminated if the program hits an exit statement or an error.
- If the program is executed on these concrete input values, it will take exactly the same path as the symbolic execution and terminate in the same way.
- These concrete inputs ensure that coverage is achieved in testing, by taking different execution paths in the program.

So, at the end of symbolic execution path constraint is solved using a constraint solver. There could be third party constraint solvers. If you remember when I introduced propositional logic to you I told you that the satisfiability problem for propositional logic is an NP-complete problem, it is a hard problem. And the satisfiability problem for predicate logic in general is undecidable, but there are lots of constraints solvers: dReal is one of them that I use, that you can use to solve these path constraints.

And if concrete test values come, then they become your concrete test inputs. What do these concrete test inputs achieve for you? Let us say the concrete test inputs will achieve, by how many path constraints you have here? 3 different path constraints. Two path constraints, but their combinations give you 3 conditions. By giving it in solving what you guaranteed is, you are guaranteed exhaustive coverage for program. With just 3 test cases, you can completely cover this program. Test this program for every behaviour that is possible in the program that is what is written here. So, it says that the program is executed on the concrete input values that the constraint solver gives you, it will take exactly the same path as the symbolic execution and terminate in the same way. These concrete inputs ensure that coverage is achieved in testing. You will be able to take different test paths in the program.

(Refer Slide Time: 29:09)

Example 2, re-cap

Consider the program in example 2 again:

```
1 int twice (int v) {  
2     return 2 * v;  
3 }  
4 void testme (int x, int y) {  
5     z = twice(y);  
6     if(z == x) {  
7         if(x > y + 10)  
8             ERROR;  
9     }  
10 }  
11 int main() {  
12     x = sym_input();  
13     y = sym_input();
```

We will see a few more examples where this is illustrated very clearly. Like for example, this is that we saw the same code I am showing you here again.

(Refer Slide Time: 29:17)

Symbolic Execution: Example 2

- After executing statements 12 and 13: $\sigma = \{x \mapsto x_0, y \mapsto y_0\}$, where x_0 and y_0 are two initially unconstrained symbolic values.
- After executing line 5: $\sigma = \{x \mapsto x_0, y \mapsto y_0, z \mapsto 2y_0\}$.
- After line 6: two instances of symbolic execution are created with path constraints $x_0 = 2y_0$ and $x_0 \neq 2y_0$.
- Similarly after line 7: two instances of symbolic execution are created with $(x_0 = 2y_0) \wedge (x_0 > y_0 + 10)$ and $(x_0 = 2y_0) \wedge (x_0 \leq y_0 + 10)$.
- Solving the path constraints, we get three instances of symbolic executions resulting in concrete test inputs $\{x = 0, y = 1\}$, $\{x = 2, y = 1\}$ and $\{x = 30, y = 15\}$.

I will just walk you through the steps of symbolic execution, after executing these statements 12 and 13, the symbolic inputs x_0 and y_0 are assigned to x and y and after executing line 5 which is this line, x_0 stays as x_0 , y_0 stays as y_0 , z becomes $2y_0$. This is a symbolic expression and after line 6 where you encounter first if statement, one which passes x as $2y_0$, the other one is failing x as equal to $2y_0$.

Now, you in an passes will go to the inner if statement where you take this if condition $x_0 = 2 y_0$, and AND it with the condition of the second if statement and you take the same condition $x_0 = 2 y_0$ and AND it with the negation of the if statement which amounts to $x_0 \leq y_0 + 10$. So, you have 3 constraints that you have collected, first one is $x_0 \neq 2 y_0$, second one is $x_0 = 2 y_0 \wedge x_0 > y_0 + 10$, and third one is this.

You let us say you give these 3 constraints to a constraint solver, then you will get some values like this. These are just one set of sample values, any values that solve these equations satisfy these equations are good enough, this is a small example. So, you could just look at it and do this yourself. So, the value of symbolic execution may not be seen, but let us say you are running it on a large enough example, then actually collecting these constraints and solving them will help you to drive program execution to a particular path.

So, symbolic execution is valuable from that point of view. So, I will stop here. In next lecture I will continue with symbolic execution. We will look at additional examples of code that involves loops and so on.

Thank you.