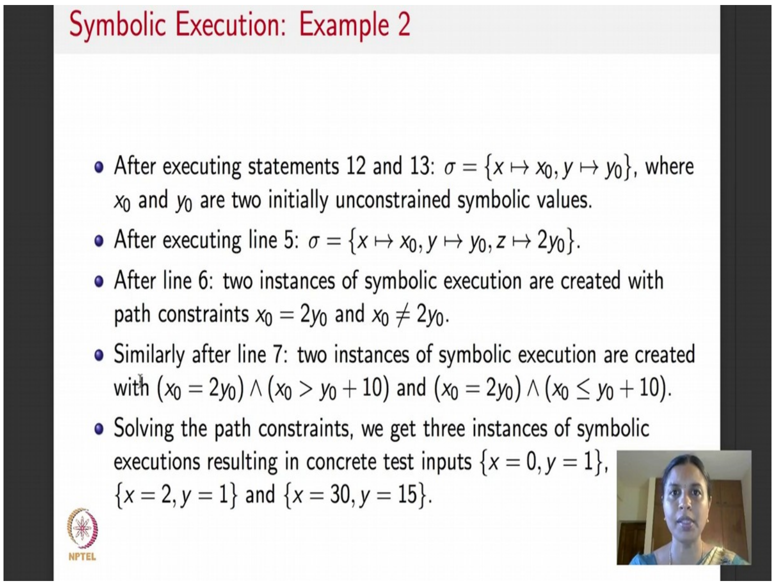
Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 51
Symbolic Testing

Hello again, I will continue with the symbolic execution. If you remember in the last class, I had introduced symbolic execution to you shown you a couple of examples and given you an overview of the technique. What I am going to do now is to continue with it, we look at the third example.

(Refer Slide Time: 00:20)

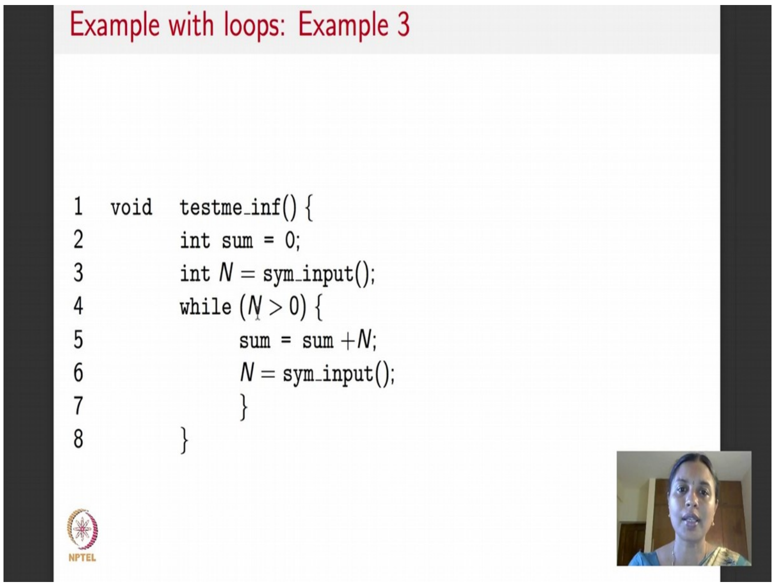


Symbolic Execution: Example 2

- 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 $\{x = 30, y = 15\}$.

So, here is another program; this time this program has a loop and the idea is to understand how symbolic execution works on programs that have loops.

(Refer Slide Time: 00:28)



Example with loops: Example 3

```
void   testme_inf() {
       int sum = 0;
       int N = sym_input();
       while (N > 0) {
             sum = sum +N;
             N = sym_input();
             }
       }
```

So, here is a piece of code is named as test me infinite, inf short for infinite, do not worry too much about the name; let us understand what the program does. It has an internal variable called sum which is an integer type initially assigned to 0, then it takes as another variable which is of type integer called N. How does it take it as? It takes it as input, it takes it as symbolic input which means what you take it as a symbolic input eventually substitute it with concrete value and then the program has piece of code.

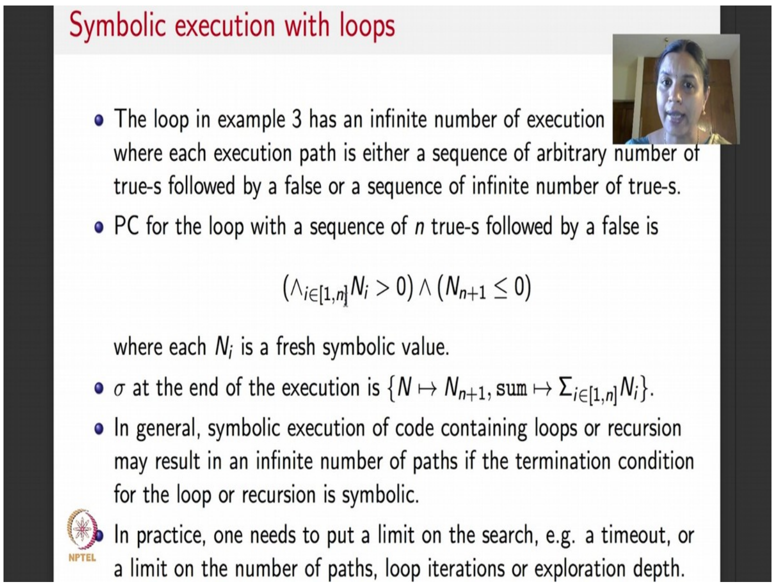
What does that code do? It has a while loop which says as long as the input that you have read, of integer type, is an input greater than 0, you keep adding it to the sum that you have collected till now and take the next input. If you happened to get an input that is less than or equal to 0, then this while loop gets aborted. Otherwise, this program, this while loop will run for ever as long as somebody gives it positive inputs. So, that is why this is called infinite. So, this program has a problem. The problem is, this program, the loop in the program, could be non-terminating and the whether the loop terminates or not is fully based on the next input that the user gives. I want to show or flag this as a potential error in the program because it has an input statement inside the while loop, it can go on running as long as the condition of the while loop passes.

What is the condition of the while loop? Condition of the while loop just says that the input that you have read just now is it greater than 0 or not. So, this program will go on running as long as somebody gives it positive inputs. I want to use symbolic testing to identify this as a problem with the program and tell the user that there is a problem. How will I do that? If you remember the steps of symbolic testing, what are the steps? Instead of giving concrete vales to inputs give symbolic values for every internal variable, collect a symbolic expression which is basically substitute the value of the input that comes in the expression with its symbolic value, collect path constraints that will tell you about the decision statements in the program. If the decision statements were simple if statements like the ones we saw earlier there would have been no problems. Here the decision statement, in this program is a while statement and it is a while statement which says N > 0.

Now, you could go by the previous times example and say that the path constraint will be symbolic value of N > 0, but then just that as the path constraint does not tell you; when this while loop is going to stop. You also remember, you also have to give a negation of the path constraint. For every path constraint that you encounter is the decision statement, an if or a while, you have to be able to give its negation also. What is negation of N > 0? That is N <= 0 and how will I give it as a path constraint for this, because it is a while loop it has to run for one or more iterations. In fact, it runs as long as N > 0 and when N is less than or equal to 0 it will stop running. Is that clear, that is what I specify in the path constraint.

This is what I was trying to explain it to you, I will go through this slide.

(Refer Slide Time: 04:05)



Symbolic execution with loops

- The loop in example 3 has an infinite number of execution where each execution path is either a sequence of arbitrary number of true-s followed by a false or a sequence of infinite number of true-s.
- PC for the loop with a sequence of n true-s followed by a false is

$$(\wedge_{i\in[1,n]} N_i > 0) \wedge (N_{n+1} \leq 0)$$

where each N_i is a fresh symbolic value.
- σ at the end of the execution is $\{N \mapsto N_{n+1}, \mathtt{sum} \mapsto \Sigma_{i\in[1,n]} N_i\}$.
- In general, symbolic execution of code containing loops or recursion may result in an infinite number of paths if the termination condition for the loop or recursion is symbolic.
- In practice, one needs to put a limit on the search, e.g. a timeout, or a limit on the number of paths, loop iterations or exploration depth.

Now, the loop in the example that we are seeing in this slide, example 3 has infinite number of execution paths as I told you where each execution path is a sequence of arbitrary number of true-s followed by a false, when it stops, or it can go on as a sequence of infinite number of true-s. So, the path constraint for a loop with the sequence of N true-s where N is unknown, it completely dependent, n is completely dependent on the user. This while loop let us say the user tries to give positive values of N first 3 times fourth input that he gives is negative, the while loop will stop.

Let us say the user tries to give positive values of N 100 times, 100 and first input this gives is negative then, 100 and after 100 iterations the while loop will stop. User could decide to give first 10,000 values as positive and then 10,001 th value as negative, the while loop will run till then. User continuously gives positive values, while loop will go on running that what is captured here. It says that there is some N whose value, I do not know and the path constraint says till that value of N that is for I is equal to one to small n, all the numbers that I get capital N i's are greater than 0 and after this N, there is a n+1 th value which is less than or equal to 0.

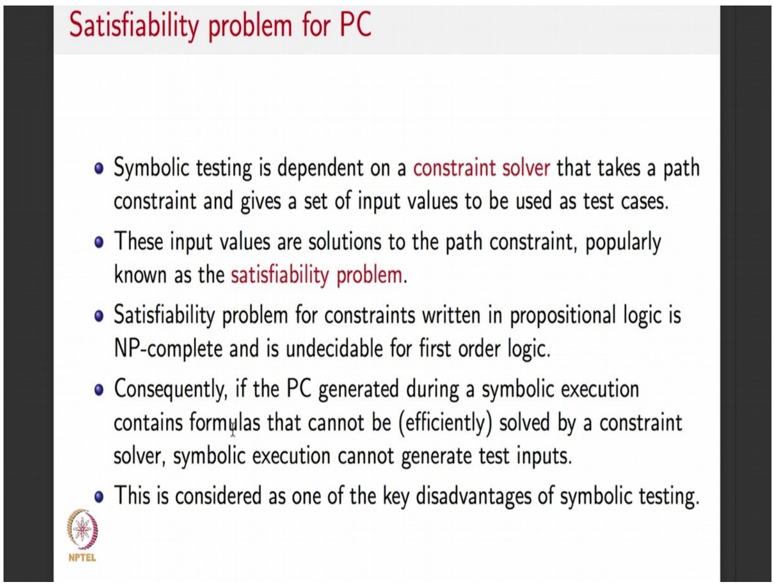
This N may or may not exist. If this then does not exists means what for no N, N + 1 <= 0 means, for every N and I will be greater than 0 which means the while loop can go on executing. That it what captures that succinctly and what will happen after at each step of the execution? Sum gets assigned to sum plus that value of N i and the new input of N i is read that is what is represented here sigma which represented the symbolic state which I had explained in the previous lecture looks like this.

There are 2 variables in the program N and sum. N will stop when the program go at the end of execution when the program gets N + 1 and what will sum be? Sum will be the summation of all the inputs that you have read till now. So, in general by using this example, what have we learnt? If I am doing a symbolic execution of a code that contains loops or recursion; recursion I have not shown you, but that holds for recursion also, it might result in an infinite number of the paths, if the termination condition for the loop is such that, it is also symbolic. For these kind of things, usually what people do is that they put a time out, they wait and then they say, no, there is a problem with the program.

I am doing this for too long, then they go and look at it and abort or they limit the number of paths, loop iterations or the exploration depth. So, this is how you collect symbolic constraints for loops. So, symbolic constraints for loop will have a particular number of iterations where the path constraint for the loop turns out to be true and then after the specified number of iterations the path constraint will be false.

In this case why I showed you this example directly is that in this case, the next iteration of the loop is dependent on the structure of the loop. This is particularly problematic case. So, I do not know whether I can stop it after a finite number of iterations or not. Or I do not know whether that n will exists or not. That is the case, then symbolic execution may or may not terminate and that truly reflects the problem in this program also, because concrete execution of this program also may or may not terminate. It will not terminate if a user continuously gives positive values. It will terminate, if the user eventually gives a 0 or a negative value.

(Refer Slide Time: 08:14)



Satisfiability problem for PC

- Symbolic testing is dependent on a constraint solver that takes a path constraint and gives a set of input values to be used as test cases.
- These input values are solutions to the path constraint, popularly known as the satisfiability problem.
- Satisfiability problem for constraints written in propositional logic is NP-complete and is undecidable for first order logic.
- Consequently, if the PC generated during a symbolic execution contains formulas that cannot be (efficiently) solved by a constraint solver, symbolic execution cannot generate test inputs.
- This is considered as one of the key disadvantages of symbolic testing.

So, now what is the biggest problem with the symbolic execution? Biggest problem with symbolic, when the symbolic execution end? When we are able to solve the path constraint and collect a set of test cases. In the case of this example, this was the path constraint to be solved. What is this path constraint solving means, find some N such that you give the first N values as numbers that are positive and N + 1th value as numbers that is <= 0. We can always find like this. As I told you the small n could be 100, whatever it will be satisfied. What was the path constraint for this one? For this piece of program that was given here, right, 3 path constraints was there one first if statement negating to be false and the second one was first and second; if statement both were true first if statement was true, second if statement was false, all the cases these path constraints have to be automatically solved. These were small programs that I told you for illustrative purposes, but for large program, imagine a large program with 10000 lines of code several different if statements and you symbolically execute, you collect a large constraint.

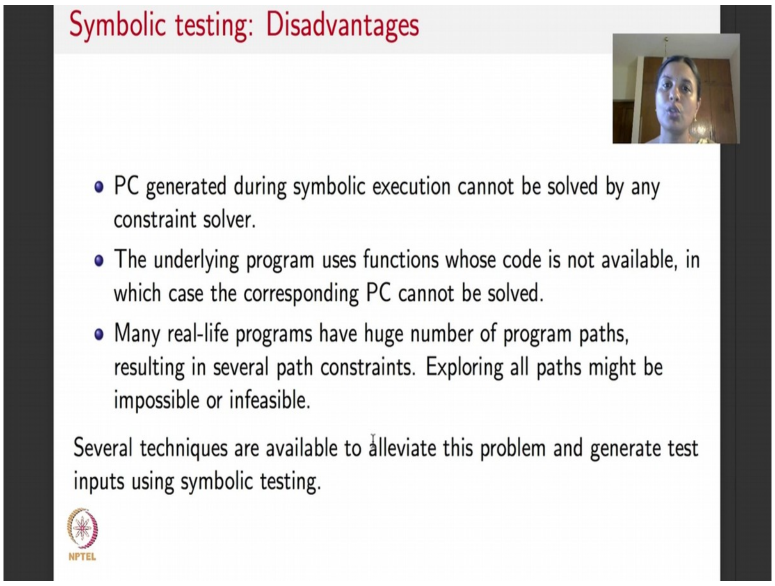
So, you have to give it to a constraint solver expect the constraint solver to be able to solve the path constraint for you. As I told you solving path constraints means looking at satisfiability problem in logic. If you remember in the lecture on logic that I did towards the first half of the course where before we began logic based testing, I had told you that satisfiability problem is a hard problem. Even for elementary logic like propositional logic, there no known polynomial time algorithm for solving satisfiability. Or in other words satisfiability problem for this is NP-Complete. We are not even talking about propositional logic. We are talking about predicate logic which means if there are variables that are of type integers that are not always Boolean. Satisfiability problem for predicate logic or first order logic is un-decidable which means there is no algorithm to do satisfiability.

But there are lots of so called constraint solvers which use exclusively designed heuristic based techniques to do satisfiability I will point you to references a couple of them towards the end of this course. So, suppose you get a path constraint and you are able to solve it using a constraint solver then well and good, you can do symbolic execution and drive the program towards the desired paths that you need or cover all the paths in the program. But let us say you are able to not solve the path constraint and get a collection of inputs then there is a problem symbolic execution may not be useful.

Why will you not be able to solve a path constraint, because there could be functions like $f(x) <= 0.5$ and you may not have the code corresponding to $f(x)$. Or let us say in a programming language like C, if you include the math.h library, you can write things like if $\log(x) < 0.005$ then you do something. Things like log may not be efficiently computable by consultant constraint solvers.

So, there could be several different reasons why arbitrary functions and predicates cannot be computed. So, if one such problem exists and constraints solver says I am not able to solve, then symbolic execution is not particularly feasible for that. In fact, this is considered to be one of the key disadvantages of using symbolic testing.

(Refer Slide Time: 11:47)



Symbolic testing: Disadvantages

- PC generated during symbolic execution cannot be solved by any constraint solver.
- The underlying program uses functions whose code is not available, in which case the corresponding PC cannot be solved.
- Many real-life programs have huge number of program paths, resulting in several path constraints. Exploring all paths might be impossible or infeasible.

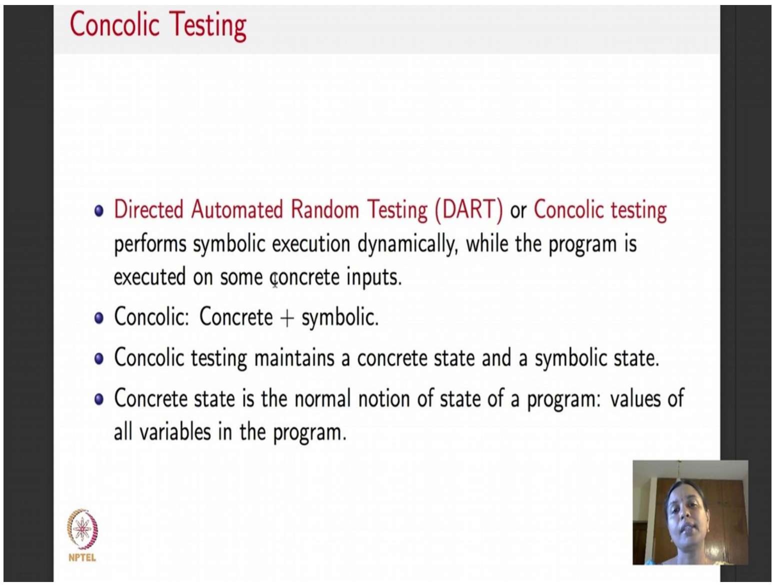
Several techniques are available to alleviate this problem and generate test inputs using symbolic testing.

So, to summarize all the disadvantages of symbolic testing, the PC or the path constraint as I told you just now generated during symbolic execution may not be solvable by any constraint solver then you are stuck, you can never get test cases. Why will it not be solvable, because the underlying program could use a function whose code is not available. Then how will you know what f(x) computes. This is very common when you are using web applications and all you might use functions whose code is not available. And as I told you many real life programs which have a few million lines of code have several different program paths.

So, as you go on symbolically executing a program there could be several decision statements and you might end up collecting a really long path constraint which is very difficult to solve. It is so long, it has so many variables that several constraint solvers might not find it feasible to be able to solve. In which case exploring all paths may not be feasible. In fact, symbolic execution was first introduced in the year 1976, quite old, almost 40 years from now. But then it is come into use only in the past 10 years it was dormant mainly because of these disadvantages.

Recently if people have devised a lot of techniques from 2005 onwards where some of the disadvantages that I have listed here can be overcome.

(Refer Slide Time: 13:12)



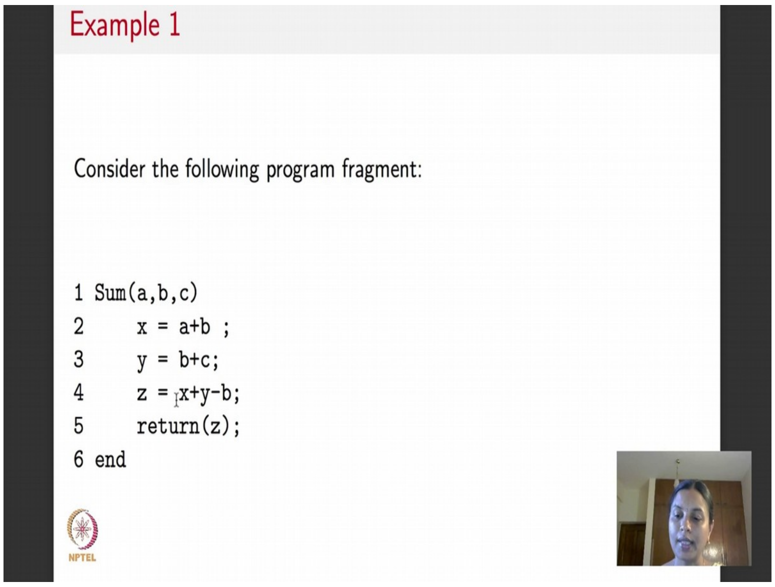
Concolic Testing

- Directed Automated Random Testing (DART) or Concolic testing performs symbolic execution dynamically, while the program is executed on some concrete inputs.
- Concolic: Concrete + symbolic.
- Concolic testing maintains a concrete state and a symbolic state.
- Concrete state is the normal notion of state of a program: values of all variables in the program.

So, one of the techniques that people have introduced that helps us to overcome the disadvantages of symbolic testing is what is called concolic testing. Concolic testing is short form for concrete plus symbolic: con colic. That is how they have coined the term, it means that you instead of keeping a symbolic value alone which we do in symbolic testing you keep the concrete value along with the symbolic value. We will learn one particular concolic testing technique called DART directed automated random testing. I will teach that to you in the next few lectures. We will see how it overcomes several of the disadvantages that we have listed of symbolic testing in this slide.

Concolic testing as I told you, keeps the concrete state along with the symbolic state. Concrete state is the normal notion of a state of a program, if you remember in the last lecture.

(Refer Slide Time: 14:20)



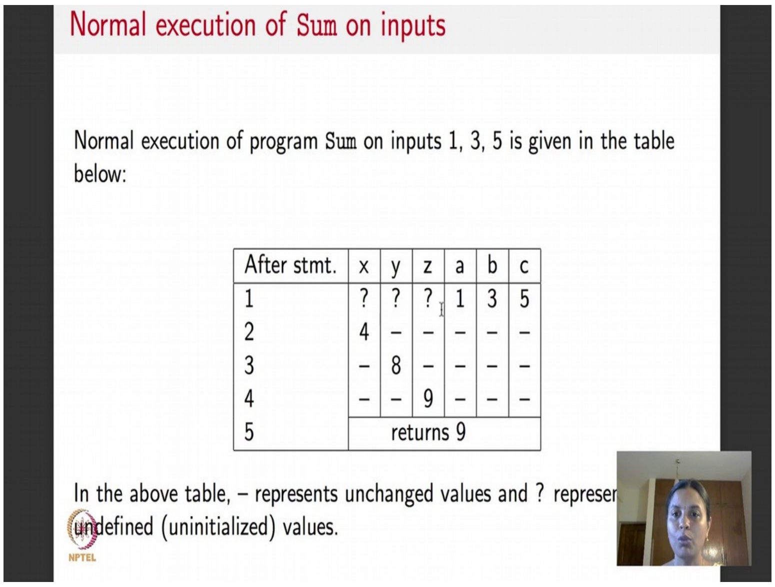
Example 1

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
```

If you would let me go back in my slides for a minute, we had this sum program if you remember, this program that calculated the sum.

(Refer Slide Time: 14:22)



Normal execution of Sum on inputs

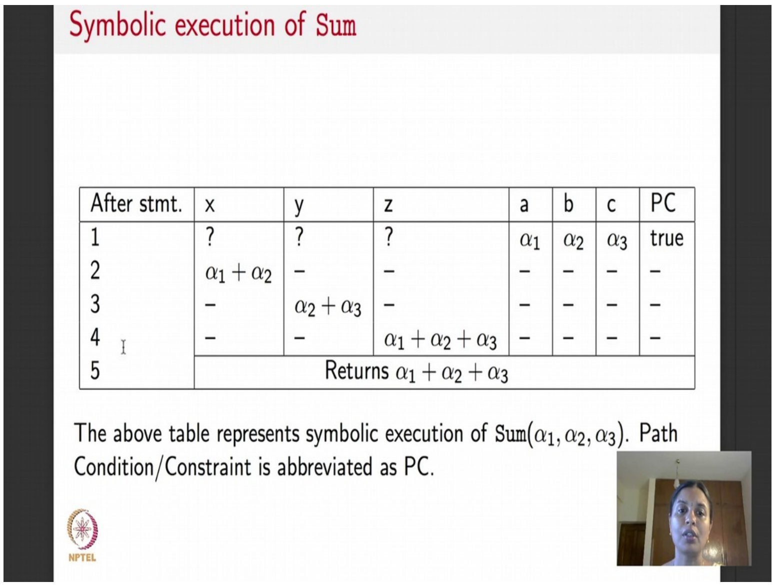
Normal execution of program Sum on inputs 1, 3, 5 is given in the table below:

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 ? represen undefined (uninitialized) values.

A concrete state would be value of a is 1 b is 3, c is 5, x is 4, y is 8, z is 9. After statement number 4 concrete state would be x 4, y 8, z 9, a 1, b 3, c 5.

(Refer Slide Time: 14:40)



Symbolic execution of Sum

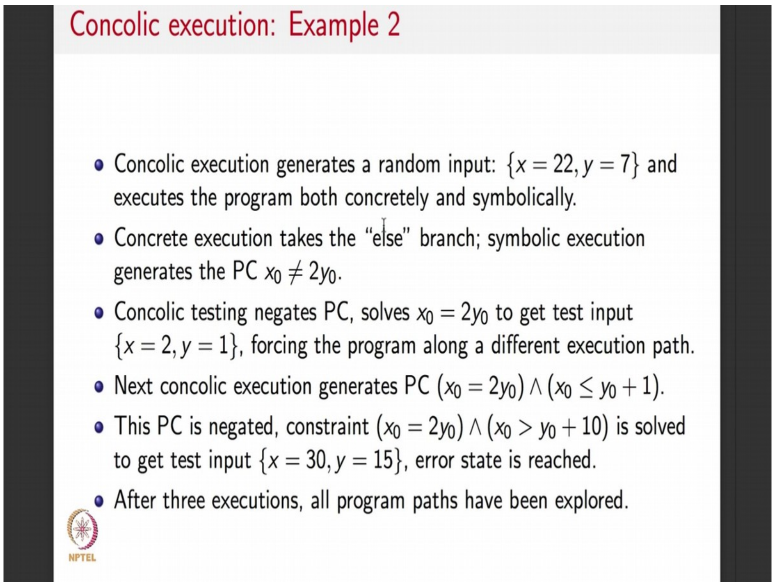
After stmt.	x	y	z	a	b	c	PC
1	?	?	?	α_1	α_2	α_3	true
2	$\alpha_1 + \alpha_2$	–	–	–	–	–	–
3	–	$\alpha_2 + \alpha_3$	–	–	–	–	–
4	–	–	$\alpha_1 + \alpha_2 + \alpha_3$	–	–	–	–
5	Returns $\alpha_1 + \alpha_2 + \alpha_3$						

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

After symbolic execution in line number 4, x would be α1 + α 2 symbolic expression, y would be α2 + α3 another symbolic expression, there would be another symbolic expression for z, the inputs a, b and c would have gotten symbolic values. So, now, I will move forward to concrete testing.

So, concrete testing or concolic testing keeps concrete state as I just told you which means it keeps concrete values; actual values and it will keep a symbolic state, it will keep the symbolic values and the expressions. Why does it keep both? The difference will be very obvious very soon. I will tell you and it what it tries to do is that by keeping track of both it has the convenience of moving back and forth. So, that is what it does.

(Refer Slide Time: 15:31)

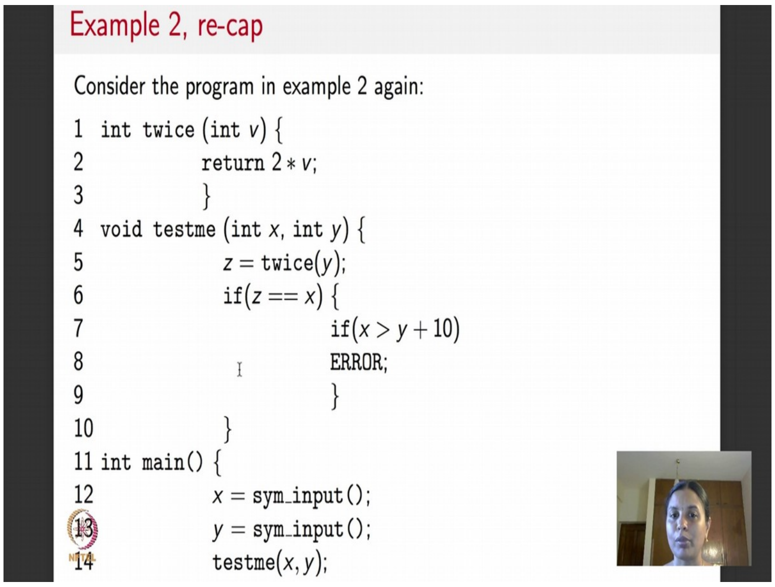


Concolic execution: Example 2

- Concolic execution generates a random input: $\{x = 22, y = 7\}$ and executes the program both concretely and symbolically.
- Concrete execution takes the "else" branch; symbolic execution generates the PC $x_0 \neq 2y_0$.
- Concolic testing negates PC, solves $x_0 = 2y_0$ to get test input $\{x = 2, y = 1\}$, forcing the program along a different execution path.
- Next concolic execution generates PC $(x_0 = 2y_0) \wedge (x_0 \leq y_0 + 1)$.
- This PC is negated, constraint $(x_0 = 2y_0) \wedge (x_0 > y_0 + 10)$ is solved to get test input $\{x = 30, y = 15\}$, error state is reached.
- After three executions, all program paths have been explored.

So, if I have to take the other example that I told you in the last class, again let me go back to show you that example.

(Refer Slide Time: 15:40)



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();
14             testme(x, y);
```

This was that example, if you remember, it did took 2 inputs x and y, did z as twice(y) and then it said z = x, if it is, then x + y > 10 then there is an error, this was the program. Here if you remember the symbolic expression was all these, right.

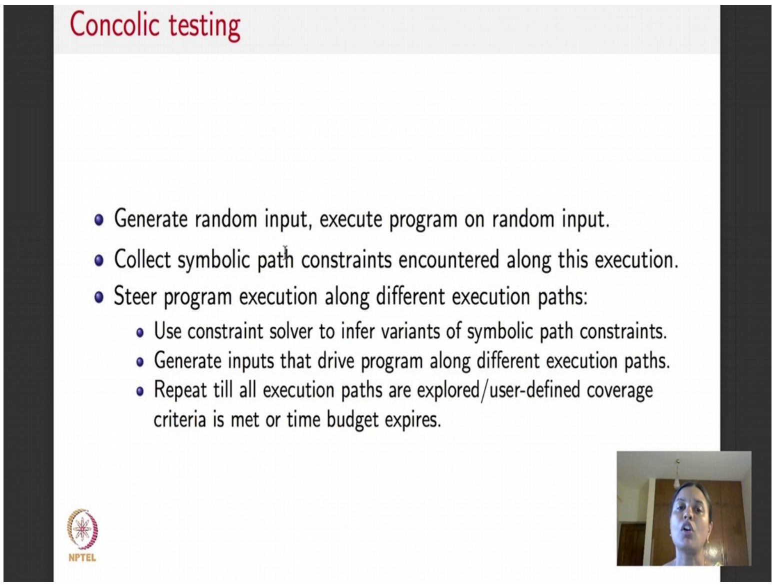
I will show you that slide also, symbolic state was x is x0, y is y0, z is 2 y0. These where the path constraints x0 = y0, x0 ≠ 2 y0 and this was after the second if statement was executed.

What we will now do is look at concolic statement. So, concolic execution, what will it do? It will keep a concrete state. How will it keep a concrete state, here is one particular way of keeping a concrete state. It will give a random input to the program. Let us say x = 22, y = 7 or any other choice. What it will do is that it will start executing the program. Once it starts executing the program, x = 22, y = 7. Go back to the code of the program. For this concrete input, this condition will it pass? It will not pass. So, this condition will go as fail which means the second if statement will not be evaluated.

So, that is what is written here. The concrete execution on this input will make the program take the else branch. Symbolic execution will generate saying the by the way, the program took an else branch by because x0 ≠ y0. What will concrete, concolic testing now do is that it says the program took the negation of this, right, I will negate this condition to see what the program will do if it takes the other one. So, it will negate this which means it will say x0, give inputs x and y such that x0 = 2 y0. Here is such input x is 2, y is 1. Now what will it do? This it will go to the next statement in the program.

The next statement in the program concolic testing will say I did this which was already my path condition. By the way I encountered one more if statement which was this and this is the part of the if statement that is satisfied by this test case. Now it will say it took one particular branch. I will take this condition, negate it to make it, there take it the other branch. So, when it negates it, it will get this condition x0 = 2 y0; that is not the predicate under focus, it will keep it as it is. This was the predicate under focus, here it was x0 <= 2 y0. I will say x0 > y0 + 10. Sorry, there is a typographical error here it should be 10 please read it as 10. So, now, it will correctly get the test input x = 13, y = 5 and it will hit the error statement in the program. So, this is how concolic execution works? This is just to meant to be an introductory example, I will explain concolic execution. In fact, this particular technique of concolic execution called DART, in detail to you.

(Refer Slide Time: 18:46)



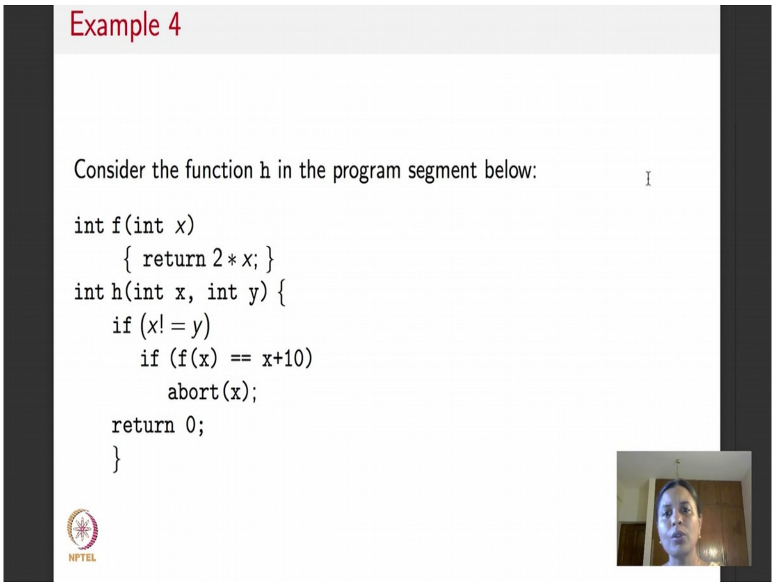
Concolic testing

- Generate random input, execute program on random input.
- Collect symbolic path constraints encountered along this execution.
- Steer program execution along different execution paths:
  - Use constraint solver to infer variants of symbolic path constraints.
  - Generate inputs that drive program along different execution paths.
  - Repeat till all execution paths are explored/user-defined coverage criteria is met or time budget expires.

So, what is concolic testing do? This is you can think of this as a concise summary of concolic testing. It has a program that it wants to test. What it will first do it will generate a random test input to the program. It will execute the program on the random test input. The program will take certain program paths. As the program takes certain program paths it will parallely symbolically execute the program. When it symbolically executes the program, it will collect constraints which said the program encountered the first if statement. The first if statement encountered in the program for this random input turned out to be true.

So, it will keep that constraint over symbolic variables and then it will move on. Let us say program encountered a second if statement second if statement on this run of the program turned out to be false. So, it will take the constraint corresponding to the first if statement negate the constraint corresponding to the second if statement, AND these two. It will keep collecting path constraints like this. Now this represents one concrete execution of a program on that random input and the whole set of path constraints. Now what concolic testing will do is that this path constraint is an AND of several different path constraints, right. So, it will systematically negate each clause in it and then try to see what the new execution of the program does, and for that it will use a constraint solver to get concrete values of inputs and it will repeat this till all the execution paths are done.

(Refer Slide Time: 20:19)



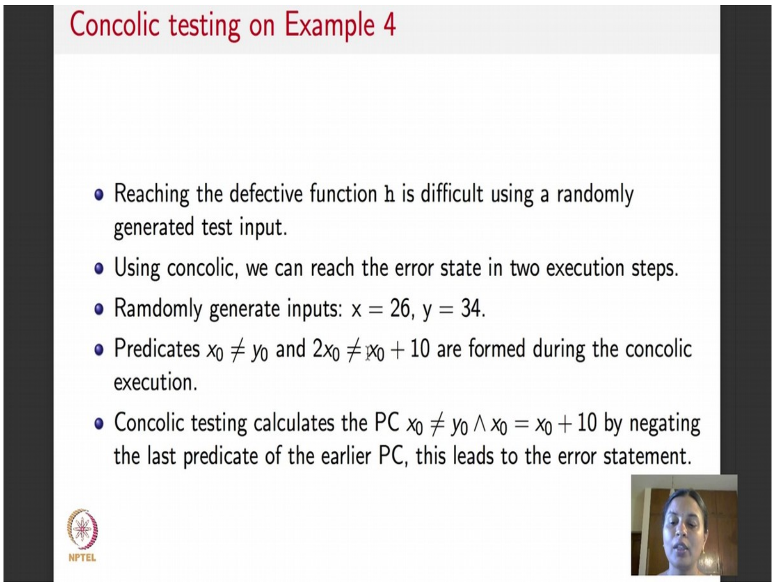
Example 4

Consider the function h in the program segment below:

```
int f(int x)
    { return 2 * x; }
int h(int x, int y) {
   if (x! = y)
      if (f(x) == x+10)
         abort(x);
   return 0;
   }
```

So, here is another example of concolic testing. So, this is a different function. So, it says there is a function f which says int x is an integer variable. Like the earlier thing it just did 2 * x, there is a function h which says it takes 2 arguments x and y and returns an integer. It has 2 if statements first one if x ≠ y, then it goes to the second statement second says if f(x) which is this f up here equal to x + 10, then you abort. Abort means you stop, there is an error in the program you return 0. If you notice that it is very similar to the earlier one; just a small piece of difference from the earlier example that we had.

(Refer Slide Time: 21:08)

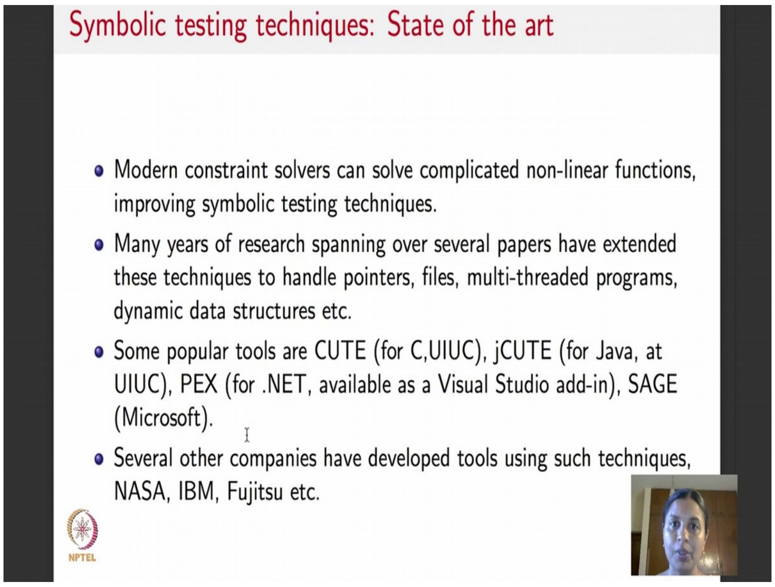


Concolic testing on Example 4

- Reaching the defective function h is difficult using a randomly generated test input.
- Using concolic, we can reach the error state in two execution steps.
- Ramdomly generate inputs: x = 26, y = 34.
- Predicates $x_0 \neq y_0$ and $2x_0 \neq x_0 + 10$ are formed during the concolic execution.
- Concolic testing calculates the PC $x_0 \neq y_0 \wedge x_0 = x_0 + 10$ by negating the last predicate of the earlier PC, this leads to the error statement.

Now, the problem is this is the place that I want to reach. Using concolic testing, how will I do? I will first randomly generate 2 inputs, let us say x is 30, 26, y is 34 any randomly generated inputs. Then 2 predicates are there, x0 ≠ y0 which I know because in this random input that is true and then 2 x0 ≠ x0 + 10. That is what this means right, because f(x) does 2 x and x + 10 is captured here. So, both these are taken. Path constraint looks like this, I take the second one, negate it then I will get something that will reach the statement. So, this will be true and this will be true, I would have reached the error. That is how concrete testing works, is that clear?

(Refer Slide Time: 21:54)



Symbolic testing techniques: State of the art

- Modern constraint solvers can solve complicated non-linear functions, improving symbolic testing techniques.
- Many years of research spanning over several papers have extended these techniques to handle pointers, files, multi-threaded programs, dynamic data structures etc.
- Some popular tools are CUTE (for C,UIUC), jCUTE (for Java, at UIUC), PEX (for .NET, available as a Visual Studio add-in), SAGE (Microsoft).
- Several other companies have developed tools using such techniques, NASA, IBM, Fujitsu etc.

I will explain it using an example and look at the general techniques of dart and so on in the next lecture. Before I stop, I would like to tell you that there are modern constraint solvers. I will give you some references when I begin the next lectures. Now there are several popular symbolic testing techniques tools that exist, there is a tool called CUTE symbolic testing you in a concolic unit testing engine for C developed by University of Illinois. This CUTE for Java which is again developed by University of Illinois at Urbana Champaign; the PEX which is available for dotnet code as a part of Microsoft visual studio the sage and so on and several other companies have developed symbolic testing techniques tools I will give you the URLs of all these and some constraints all words when I begin the next lecture. In the next lecture, what we are going to see is this dart technique in detail directed automated random testing, I will tell you what that is and how concolic testing works on that.

Thank you.