

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


Lecture – 53
DART: Directed Automated Random Testing

Hello again, continuing from where we left in the last lecture on DART, I had ended with this explanation where we talked about the execution model of DART. I will begin by briefly recapping what we ended with in the last lecture.

(Refer Slide Time: 00:27)

Execution model of DART: Program Execution


- Let **C** be the set of conditional statements and **A** be the set of assignment statements in P .
- A **program execution** w is a finite sequence in $\text{Execs} := (A \cup C)^*(\text{abort}|\text{halt})$.
- They further simplify program execution and view w as being of the form $\alpha_1 c_1 \alpha_2 c_2 \dots c_k \alpha_{k+1} s$ where $\alpha_i \in A^*$, $c_i \in C$, and $s \in \{\text{abort}, \text{halt}\}$.
- An alternate view is to consider $\text{Execs}(P)$ is a tree, with assignment nodes having one successor, condition nodes have one or two successors and leaves being labeled by **abort** or **halt**.
- Each input vector results in an execution sequence, as a path in this tree.




We assume for simplicity sake, that every program has only assignment statements or an if condition which is represented as a conditional statement. Assignment statement is represented in the set A . Conditional statements are represented on the set C . So, program execution is a series of assignment statements and condition statements and always ends with an abort or halt statement. $\text{Execs}(P)$ can be thought of as an execution tree that consists of all program executions which we saw an example of when we did symbolic execution. Assignment nodes will have only one successor because there is an immediate next statement and that is only one. Conditional nodes will have a branch out based on whether the conditional is true or false and leaves will all be labeled by abort or halt.

(Refer Slide Time: 01:14)

Symbolic evaluation of expressions



- DART maintains a **symbolic memory** S that maps memory addresses to expressions.
- While the main DART algorithm runs, it will evaluate the symbolic path constraints using this algorithm and solve the path constraints to generate directed test cases.
- To start with, S just maps each $m \in M_0$ to itself.
- The mapping S is completed by evaluating expressions symbolically: algorithm in the next slide.
- Only linear path constraints can be solved. Hence, whenever the path constraints become non-linear, the algorithm just uses concrete values instead of symbolic values



So, now we will see how DART symbolically evaluates the expression. So, to symbolically evaluate the expression, DART maintains what is called a symbolic memory. What is a symbolic memory? Symbolic memory basically tells you for every variable which is a memory address in which the variable is stored. So, it maps memory addresses to expressions. For inputs, it is the directly the name of the variable itself, but remember when we did that example for internal variables other variables, let us say if you had $z = x + y$, then you need to update the value of x by using an expression for x plus y as $x_0 + y_0$, where x_0 is the symbolic expression for x and y_0 is the symbolic expression for y . So, that is what we mean by a symbolic memory.

Symbolic memory denoted by S is a map from memory addresses to expressions. When the DART algorithm runs, it will evaluate symbolic path constraints using the algorithm when solved the path constraint to get directed generate test cases. Solving, it will give it to a third party constraint solver. To start with the initial the inputs will all be in S . The mapping of S is completed by evaluating expressions symbolically. So, for each kind of expression I will tell you how to symbolically evaluate the expression. Before we understand what to how to symbolically evaluate the expressions, fairly straightforward, whenever an actual variable comes in the expression you substitute the symbolic variable instead of the actual variable in the expression. But then ultimately we going to substitute, use these expressions in your path constraints to give it to a constraint solver, that is where the problem begins. As I told you constraint solvers cannot handle all kinds


of path constraints because the problem is un-decidable. In particular constraint solvers cannot handle path constraints that are not linear that is, path constraints of the form $5x^2 \neq 0, 3x^3 > 0$.

Any kind of non-linear expression; it is not good at handling. So, DART will remember this is one of the disadvantages of symbolic testing. So, whenever it encounters a non-linear path constraint, it drops it, in the sense that path constraints become non-linear algorithm, the DART algorithm; we will directly use the concrete values instead of symbolic values. That is how, it that is why DART keeps the concrete values set.

(Refer Slide Time: 03:50)

Symbolic evaluation of expressions

```
evaluate_symbolic(e, M, S) =  
match e :  
  case m :  
    if m ∈ domain S then return S(m)  
    else return M(m)  
  case * (e', e'') :  
    let f' = evaluate_symbolic(e', M, S);  
    let f'' = evaluate_symbolic(e'', M, S);  
    if not one of f' or f'' is a constant c then  
      all_linear = 0  
      return evaluate_concrete(e, M)  
    if both f' and f'' are constants then  
      return evaluate_concrete(e, M)  
    if f' is a constant c then  
      return * (f', c)  
    else return * (c, f'')
```





So, here is how symbolic evaluation of expressions happen. There is a function called evaluate symbolic which symbolically evaluates an expression e in the context of a memory M and already available symbolic expressions capital S. So, what could be various kinds of expressions.

(Refer Slide Time: 04:15)

Execution model of DART: Symbolic variables

- **Memory** \mathcal{M} : a mapping from memory addresses m to say 32-bit words.
 - $+$ denotes updating of memory. For e.g., $\mathcal{M}' := \mathcal{M} + [m \mapsto v]$ is the same as \mathcal{M} except that $\mathcal{M}' = v$.
- **Symbolic variables** are identified by their addresses.
- In an expression, m denotes either a memory address or the symbolic variable identified by address m , depending on the context.
- A **symbolic expression** or just an expression, e is given by
 - m , c (a constant), $*(e, e')$ (multiplication), $\leq (e, e')$ (comparison), $\neg e'$ (negation), $*e'$ (pointer dereference) etc.
- Symbolic variables of an expression e are the set of addresses m that occur in it.
- Expressions have no side effects.



If you go back and see the various kinds of expressions could be just a variable a constant arithmetic expression a relational expression a logical expression or a pointer expression.

So, I have written only multiplication, but as I have told you in the last class, it represents all arithmetic operators this represent all relational operators, this represents all logical operators. So, we will do a case by case analysis of how it happens. So, if it is just a variable, then you directly assign it to the symbolic expression and you return the updated memory. If it is arithmetic expression which is represented by the $*(e, e')$, please read the star as standing for $+$, $-$, $/$, mod , $*$; all the arithmetic operators then it has 2 operands class an expression e and another expression e' , which first have to be evaluated symbolically themselves. So, e is evaluated symbolically, e' is, e' is evaluated symbolically, $e;$ is evaluated symbolically. This is a recursive call, evaluate symbolic calls itself.

But now if let us say this is a multiplication operator, right in this particular case. So, if both x and y are variables; let us f and f' involve variables, let us say f involves $5 \times y$, f' , f' involves $5 \times x$ and f'' involves y , then $5 \times y$ will become a non-linear expression and I just told you the non-linear expressions are difficult to solve by constraints solvers. So, DART has this condition, it says that if one of f and f' is a not a constant, not one of f or f' is a constant. If one of them is not a constant is I just told you now you will get a non-

linear expressions. So, if both are not constants or one of them is not to constant, then you say now I have an expression which is not linear. So, DART keeps a special flag which it calls all linear, sets that flag to 0. If it sets that flag to 0 which means what its intuitively saying the DART is encountered an expression that is non-linear.


So, DART will now do concrete evaluation, which is evaluate concrete of that expression, update the memory and go ahead, won't store the non-linear symbolic expression at all. Why will it not store, because this no point in storing it. DART knows if the constraint solvency that it is going to use cannot handle these expressions. If both f and f' are constants then there is no expression involving variables at all. So, DART does not have a choice, but to directly evaluate it concretely. That is what it does here; updates the memory and goes ahead. If one of them is a constant, then what it will do is it will correctly evaluate that expression and return.

(Refer Slide Time: 07:04)

Symbolic evaluation of expressions

```

case *  $e'$  :
  let  $f^h = \text{evaluate\_symbolic}(e', \mathcal{M}, S)$ ;
  if  $f'$  is a constant  $c$  then
    if *  $c \in \text{domain } S$  then return  $S * c$ 
    else return  $\mathcal{M}(*c)$ 
  else all\_locs\_definite = 0
    return  $\text{evaluate\_concrete}(e, \mathcal{M})$ 
  etc.
  
```



Now, this is the case, the rest of the operators that we saw an expressions where relational and logical operators the only other expressions was pointer related stuff. So, this is the second case, if its pointer, if it is the expression of the form $*e'$, then it has to first symbolically evaluate e' , so, it calls itself recursively. This is a continuation of this code on the slide. If f' is a constant, if it belongs to the domain of S , then it returns whatever value that is. Otherwise it will return point 2 to the memory location. If it cannot determine either of this, then it will set another flag read it has all locations

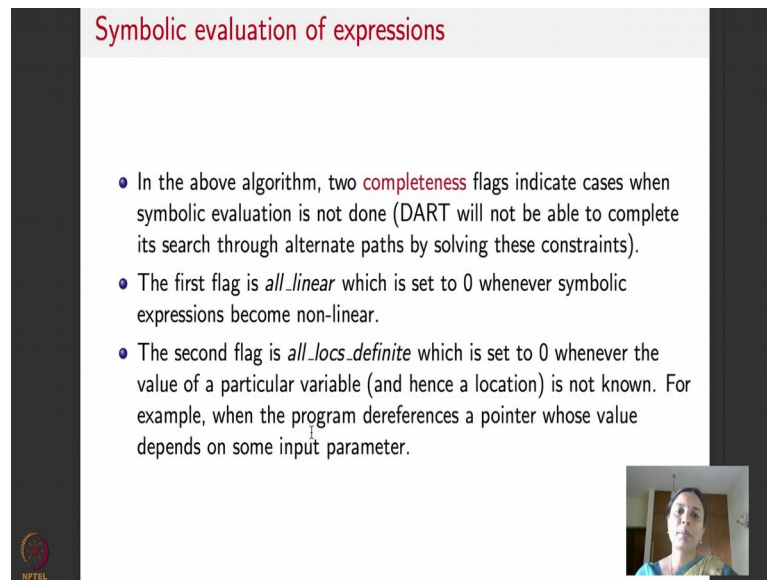
definite, which means it knows the locations of all the variables in the sense that if there is a point of pointer that it does not know, then it will say there is a location that I do not know which means it will set this flag all locations definite is to 0.

So, and then it will evaluate the concrete value and go ahead. So, what is this part then I am describing in. Remember just to recap what DART does? It takes a program, writes a driver, instruments it to first generate one random input runs the program on that random input. As the program is running on the random input, DART symbolically executes the program the symbolically execute the program. It has to symbolically evaluate all the expressions that is the part that I am describing here. So, symbolic evaluation is fairly straight forward it will keep substituting whatever expressions that it got a normal variables with symbolic variables that is what this recursive called does. There is one more thing that this pseudo code establishes. It says that if as I am executing, if I encounter an expression that is likely to be non-linear, that is if and symbolically evaluating e' and e'' and one of them after symbolically evaluating turn out to be, both turn out to be non not constants then I will say I have encountered linear expressions.

So, I will set this flag all near to 0, go back and substitute concrete values, I will not evaluated symbolically; not store it in the path constraint. Otherwise, I will do whatever normally evaluation is. Similarly here, when DART is handling pointer arithmetic, if it encounters a situation where in the memory address is not correctly known, then it will set to set flag which says all locations are not definitely known because here it does not evaluate all of them. So, they could see places where it does not know it, will go ahead and substitute concrete values otherwise it does normal symbolic execution. Why does it do the substitution of concrete values, because by doing this upfront, DART will ensure that if the path constraint that it collect is always solvable by the constraint solver that it is using.



So, this way it overcomes one of the disadvantages of symbolic testing.

(Refer Slide Time: 10:00)



Symbolic evaluation of expressions

- In the above algorithm, two **completeness** flags indicate cases when symbolic evaluation is not done (DART will not be able to complete its search through alternate paths by solving these constraints).
- The first flag is *all_linear* which is set to 0 whenever symbolic expressions become non-linear.
- The second flag is *all_locs_definite* which is set to 0 whenever the value of a particular variable (and hence a location) is not known. For example, when the program dereferences a pointer whose value depends on some input parameter.


 


So, this is the part that I was trying to explain to you. In the algorithm that symbolically evaluate expressions; they were 2 completeness flags the DART kept. One flag was called all linear which is said to 0 whenever symbolic expressions become non-linear. The next flag, all locs definite or all locations definite is said to 0, whenever the value of a particular variable and hence the memory location of the variable is not known. When will such a situation arise? For example, you could consider a program which dereferences a pointer whose value depends on input parameters, then I will definitely not know the value of a particular variable right. So, these situations could quite commonly arise.

(Refer Slide Time: 10:43)

Test driver

```
run_DART() =
  all_linear, all_locs_definite, forcing_ok = 1, 1, 1
  repeat
    stack = {};  $\vec{I}$  = []; directed = 1
    while(directed) do
      try (directed, stack,  $\vec{I}$ ) =
        instrumented_program(stack,  $\vec{I}$ )
      catch any exception  $\rightarrow$ 
        if (forcing_ok)
          print "Bug found"
          exit()
        else forcing_ok = 1
    until all_linear  $\wedge$  all_locs_definite
```





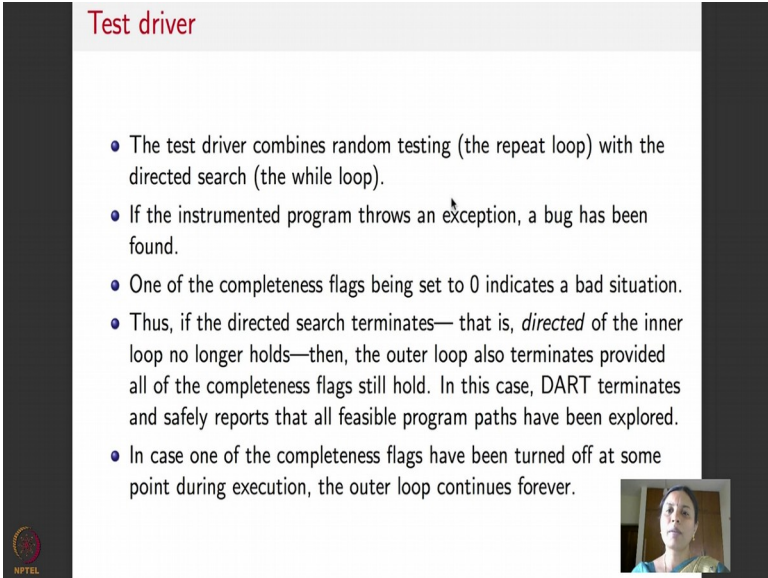
Now, let us look at the main algorithm of DART. How does the test driver of DART look like? So, this is the main program of DART, called the test driver or run DART. So, what does it do? First it initializes all these Boolean flags to 1. Read this as shorthand for saying, all linear, which is a Boolean variable to be 1 or true, all locations definite, which is another Boolean variable is assigned to 1 or true, forcing ok, which is another Boolean variable that I will explain. Now is also assigned to one or true. So, this is shorthand for 3 assignment statements which assigns 3 different Boolean variables each of them to the value true. Then DART goes into which mean loop. What is the main loop? The main loop is a repeat until loop, it is a repeat until this condition is true, which means what both the flags all linear and all locs definite stay as 1, then you keep on repeating this loop. The moment one of them become 0, you exit. Inside the loop what is DART do. DART keeps stack, we will see what the stack contains it initializes the stack to empty, i is the input vector the vector of all inputs it initializes it to empty.

It uses one more Boolean variable called directed which it initializes to one or true and then it enters a while loop. So, while this directed, directed stands for directed search. So, while I am doing directed search it has this try catch exception. So, what is try catch exception do? It at some point, this Boolean flag forcing is ever set to one, then DART is already found an error in the program, it will print this an exit. Otherwise DART, will go on instrumenting the program here. So, it will call this method called instrument program with the stack and i. So, is it clear please? What DART does? DART uses 3 Boolean

flags; all of them set to true and it goes into its main loop until all the path constraints that its encountering is linear and it knows exactly the locations of all the variables inside this loop DART will instrument the program and do a directed search.

At any point and time, if DART is found, the bug then DART will say I have found a bug it will print this message and come out.

(Refer Slide Time: 13:27)



Test driver

- The test driver combines random testing (the repeat loop) with the directed search (the while loop).
- If the instrumented program throws an exception, a bug has been found.
- One of the completeness flags being set to 0 indicates a bad situation.
- Thus, if the directed search terminates— that is, *directed* of the inner loop no longer holds—then, the outer loop also terminates provided all of the completeness flags still hold. In this case, DART terminates and safely reports that all feasible program paths have been explored.
- In case one of the completeness flags have been turned off at some point during execution, the outer loop continues forever.

So, this is how the test driver looks like. It combines random testing which is this repeat loop with directed search which is in a while loop. If the instrumented program throws an exception, a bug has been found. So, DART will print that message and exit you one of the completeness flag is being said to 0 means, what DART is encountered a bad situation, it is encountered a non-linear constraint, it is encountered place where path memory of a particular variable is not known, it is encountered some bad situation.


So, if the directed search terminates then the directed variable, which is this variable of this inner loop no longer holds, then the outer loop will also terminate provided the completeness flag still hold. That is clear from this code, I hope. In this case, DART terminates and safely reports that all the feasible program paths have been explored. But in case, one of the completeness flags have been turned off which is this, all linear or all locations definite, the outer loop will continue forever. So, they could be 3 options; DART will explore all possible program paths successfully, DART will find an error and stop saying that I found an error or DART can run forever. That is not too surprising

because it is a program analysis tools and the problem that is trying to handle which is exploring all program paths is in general and undeniable problem.

(Refer Slide Time: 14:33)

Instrumented Program

```
instrumented_program(stack,  $\vec{I}$ ) =  
// Random initialization of uninitialized i/p parameters in  $\vec{M}_0$   
for each input  $x$  with  $\vec{I}[x]$  undefined do  
     $\vec{I}[x] = \text{random}()$   
Initialize memory  $\mathcal{M}$  from  $\vec{M}_0$  and  $\vec{I}$   
// Set up symbolic memory and prepare execution  
 $S = [m \mapsto m \mid m \in \vec{M}_0]$   
 $l = l_0$  // Initial program counter in  $P$   
 $k = 0$  // No. of conditionals executed  
// Now invoke  $P$  intertwined with symbolic calculations  
 $s = \text{statement\_at}(l, \mathcal{M})$ 
```





So, we will now look at this code. What does the code corresponding to instrumented program in the main test driver of DART looks like. So, here how instrumented program code looks like. It takes a stack and a vector of inputs. So, first it does a random initialization of uninitialized input parameters in the memory locations. So, it says for each input x with a $\vec{I}[x]$ uninitialized do random. This is the initial random text vector that DART generates. It initializes memory to this value that it is found, it sets up the symbolic memory which is S , sets the initial program goes and says, I have not yet encountered any path constraint and begins at the first statement.

(Refer Slide Time: 15:27)

Instrumented Program, contd.

```
while (s  $\notin$  {abort, halt}) do
  match (s)
  case (m  $\leftarrow$  e):
    S = S + [m  $\mapsto$  evaluate_symbolic(e, M, S)]
    v = evaluate_concrete(e, M)
    M = M + [m  $\mapsto$  v];
    l = l + 1
```



And as long as it is not encountering an abort or halt, DART will encounter either an assignment statement or it will encounter a condition statement because we have assumed that program execution is a series of, I will go back to back that slide first.



We have assume that the program execution is a series of assignments and conditions ending with an abort or halt. So, that is what this instrumented program does. So, it sets up the memory, it sets its stat counter to be 0 and then it begins. So, which is as long as statements are not abort or halt, it is either an assignment statement or a condition statement. So, there is a switch case like. So, S is match to an assignment statement, what will it do? It is already has a symbolic memory uploaded. Here is a new assignment statement that it is encountered. For example, this could be the case there which says and $z = x + y$. So, the memory location for z which is M needs to be updated with the expression x0 for $x_0 + y_0$. So, the memory location for M is updated after evaluating symbolically the expression e with the memory location M and with the symbolic set of expressions S.

It also needs to evaluate the expression e concretely because you do not know when in the future it will have to substitute. So, it will evaluate symbolically, it will evaluate concretely. It will update the memory location and it says I finished with this statement. So, we will update the program counter to one.

(Refer Slide Time: 16:56)

Instrumented Program, contd.

```
while (s  $\notin$  {abort, halt}) do
  match (s)
    case (if (e) then goto l') :
      b = evaluate_concrete(e, M)
      c = evaluate_symbolic(e, M, S)
      if b then
        path_constraint = path_constraint ^ (c)
        stack = compare_and_update_stack(1, k, stack)
        l = l'
      else
        path_constraint = path_constraint ^ ( $\neg$ c)
        stack = compare_and_update_stack(0, k, stack)
        l = l + 1
        k = k + 1
      s = statement_at(l, M) // End of while loop
```



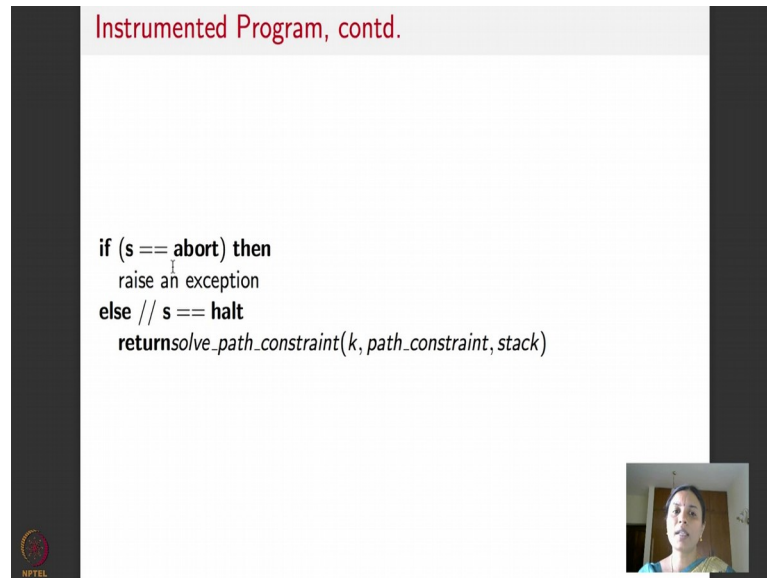
So, let us say DART encounters a condition statement. If the match happens to be a condition. So, it says if e then go to l', then what it will do is that it will evaluate e symbolically and concretely. Let us say b is the symbolic, result of concrete evaluation of e, C is the result of symbolic evaluation of e. If b is true then the then path is taken by the program if b is false, then the else path is taken by the program.

So, if b is true, then it already has a path constraint which it has initialized here it will go ahead and the condition that it got by symbolically evaluating it to the path constraint. Read this as whatever the path constraint that was earlier now C is added to it with an AND and then, it will now put this in this stack saying, I have one more path constraint to update, I found on the way. The program, took a branch, here is a path constraint the program took a then branch, so, I am putting a one here and adding this constraint to the stack and it will change the statement to whichever that it has to go to, l'. But suppose the condition evaluates to falls then to the path constraint it will add negation of that condition. This is path constraint ended with negation of C, it will update it stack by saying the program took the else branch with the 0 and add this to the stack and change the statement number because it is like skipping.

So, statement number now becomes l plus 1 and it will go ahead and do it, is this clear. So, basically this part what it does is that it grows the path constraint one at a time, one at a time and what is the stack do? Stack keeps track of which is the path constraint that

was added the latest because that is the path constraint the DART is going to flip to be able to get the next execution of the program. That is what DART does here right, it will flip it in the in the main thing when it does the next execution of the program.

(Refer Slide Time: 18:55)



Instrumented Program, contd.



```
if (s == abort) then
    raise an exception
else // s == halt
    returnsolve_path_constraint(k, path_constraint, stack)
```

So, if S is an abort or halt, if S is an abort then DART raises an exception, if S is a halt then it just re-directly calls the constraint solver with the path constraint that it is collected as it takes it from the stack. Now in this it used this routine called compare and update stack. What is that do, it tells you how to update the stack here is the code for that.

(Refer Slide Time: 19:06)

Compare and update stack routine

```
compare_and_update_stack(branch, k, stack) =  
if  $k < |stack|$  then  
  if  $stack[k].branch \neq branch$  then  
    forcing_ok = 0  
    raise an exception  
  else if  $k = |stack| - 1$  then  
     $stack[k].branch = branch$   
     $stack[k].done = 1$   
  else  $stack = stack \setminus (branch, 0)$   
return stack
```





So, stack has a maximum capacity this part says as long as if not reached the maximum capacity if the stack branch is not equal to branch then there is a problem four set forcing to 0 and raise an exception. Where is this Boolean variable forcing ok, it was here. So, if forcing say ever set, then you can say bug is found l's is stack. This; like this then you go ahead and normally update the stack branch, you say you to remove it from the stack profit off and then add this. Is this clear? This normally just the normal stack operations that you are doing right solve path constraint which is what it calls here right solve path constraint what is that look like?

(Refer Slide Time: 19:58)

Solve path constraint routine

```
solve_path_constraint( $k_{try}$ , path_constraint, stack) =  
  let  $j$  be the smallest number such that  
  for all  $h$  with  $-1 \leq j < h < k_{try}$ ,  $stack[h].done = 1$   
  if  $j = -1$  then  
    return  $(0, \rightarrow, \rightarrow)$  // Directed search is over  
  else  
     $path\_constraint[j] = \neg(path\_constraint[j])$   
     $stack[j].branch = \neg stack[j].branch$   
    if  $(path\_constraint[0, \dots, j]$  has a solution  $\vec{l'}$ ) then  
      return  $(1, stack[0..j], \vec{l} + \vec{l'})$   
    else  
      solve_path_constraint( $j$ , path_constraint, stack)
```




That will basically call a constraint solver which is third party constraints solver. So, what it does is that it takes the latest path constraint from latest thing from the stack and its says if j is -1 , then I finish my search it terminated normally. Otherwise it says do take this whatever path constraint that you found at the top of the stack negate that path constraint added to the branch now you call it again. So, that it explores the other search, this is the part that forces start to do the directed search. This is an explanation of what the instrumented program does. Each run of the instrumented program, the first one does a random search.

(Refer Slide Time: 20:32)

Instrumented program

- Each run of the instrumented program (except the first) is executed with the help of a record of conditional statements executed in the previous run.
- For each conditional, we record a *branch* value and a *done* value.
- *branch* value is 1 when the **then** branch is taken and 0 otherwise.
- *done* value is 0 only when one branch of the conditional has executed in prior runs (with the same history up to the branch point) and is 1 otherwise.
- This information associated with each conditional statement of the last execution path is stored in a list variable called *stack*, kept in a file between executions. For $i, 0 \leq i < |stack|$, $stack[i] = (stack[i].branch, stack[i].done)$ is thus the record corresponding to the $(i + 1)$ th conditional executed.





So, except for the first one, each run of the instrumented program, how is it executed? You have a stack of all the conditions that you encountered in the path, in the previous run. So, we record a branch value and a done value. Branch value is 1, as I told you when the then part is true or the condition is true and branch value is 0 if the condition is false. Done value 0 only when the branch of the condition is executed in the prior runs which means I have explored this part already, so then you said the fully done with this program constraint. Information associated with each conditional statement of the last execution path is stored in a variable called *stack*, which is what we have been looking at and that is kept in a file that shared between executions and then at any point in time, the whatever is available at the top of the stack is pulled, flipped and that is the alternate path the DART tries to take.

(Refer Slide Time: 21:40)

Instrumented program

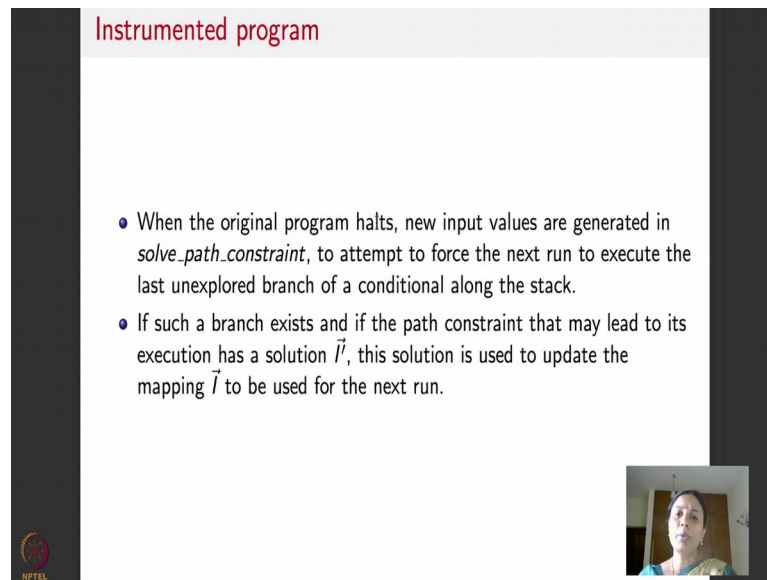
- The instrumented program executes the original program with interleaved gathering of symbolic constraints.
- At each conditional statement, it also checks by calling *compare_and_update_stack*, whether the current execution path matches the one predicted at the end of the previous execution and represented in *stack* passed between runs.
- If it ever happens that a prediction of the outcome of a conditional is not fulfilled, then the flag *forcing_ok* is set to 0 and an exception is raised to restart *run_DART* with a fresh random input vector.
- Note: setting *forcing_ok* to 0 can only be due to a previous incompleteness in DART's directed search, which was then detected and resulted in setting at least one of the completeness flags to 0. In other words, the invariant $all_linear \wedge all_locs_definite \Rightarrow forcing_ok$ holds.



So, if it explored both the options, then it such the done value to 0, removes it from the stack and takes the next available condition. Instrumented program what is it do? It basically executes the original program interleaved with a gathering of symbolic constraints. At each conditional statement, it checks the, it calls this compare and update stack routine, it checks whether the current execution path matches the one that it predicted at the end of the earlier execution and if it is, then he does not repeat, but if it not then it explores this new execution path. But if some problem happens then it sets the flag forcing to 0 and basically then it will restart this run DART which is the main test driver with another fresh randomly generated input.



When will the sourcing be said to 0 that can be only due to a previous incompleteness and darts directed search, it went along a path that it could not complete, so it will start all over again.

(Refer Slide Time: 22:29)



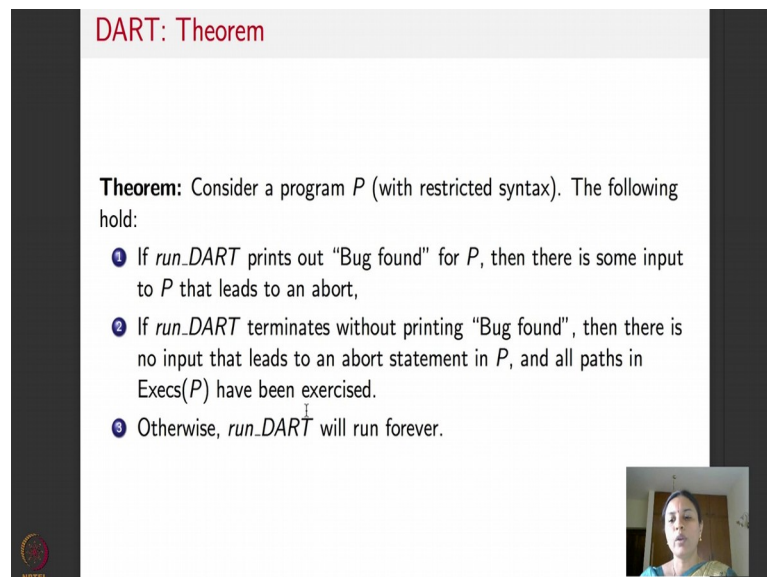
Instrumented program

- When the original program halts, new input values are generated in *solve_path_constraint*, to attempt to force the next run to execute the last unexplored branch of a conditional along the stack.
- If such a branch exists and if the path constraint that may lead to its execution has a solution \vec{I}' , this solution is used to update the mapping \vec{I} to be used for the next run.

So, when the original program halts, new input values are generated from the solve path constraint routine that will attempt to force the next run to execute, the last unexplored branch of the stack such a branch exists, then it is a explored. Otherwise, it removes it from the stack can goes to the next path constraint that it was first.



(Refer Slide Time: 22:48)



DART: Theorem

Theorem: Consider a program P (with restricted syntax). The following hold:

- 1 If *run_DART* prints out "Bug found" for P , then there is some input to P that leads to an abort,
- 2 If *run_DART* terminates without printing "Bug found", then there is no input that leads to an abort statement in P , and all paths in $\text{Execs}(P)$ have been exercised.
- 3 Otherwise, *run_DART* will run forever.

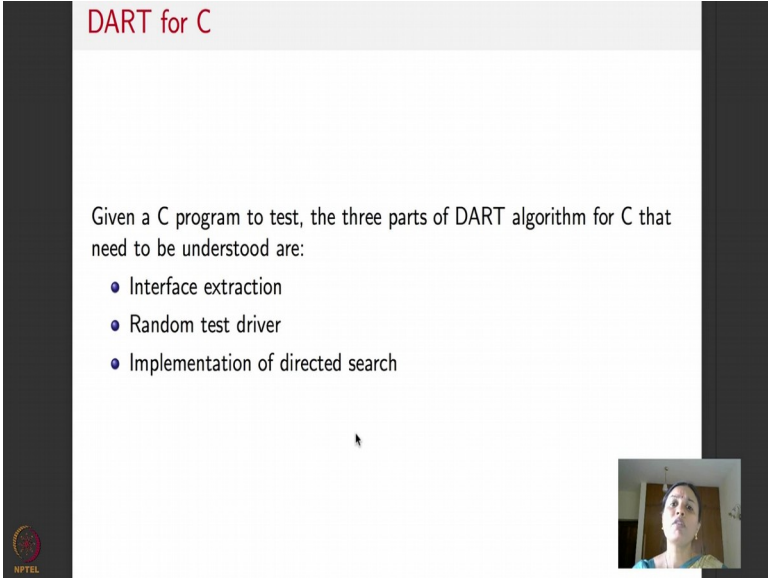
 

So, I hope the code for DART is clear. So, the main code for DART is the test driver which does one random search followed by a program instrumentation which symbolically executes a program, collects the path constraint and does a directed search.

This was the code for that which tells you how to symbolically update and execute a program. It needs a stack to be able to keep track of the path constraints, this is a update, this is the routine method that maintains the stack, this is the routine that calls the constraint solver and DART could as I told you have 3 outcomes.

So, here is the main theorem that talks about the correctness of DART. If DART runs, if DART at any point and time says this print bug found, then there is some input to be that leads to an abort state, it is really found an error. If run DART terminates without printing bug found, then which means or DART is explored all the execution paths of P. Otherwise run DART will run forever somebody has to manually go on about it. The third part is an undesirable behavior of DART, which we cannot avoid because the underlying problem is undesirable

(Refer Slide Time: 23:59)



DART for C

Given a C program to test, the three parts of DART algorithm for C that need to be understood are:

- Interface extraction
- Random test driver
- Implementation of directed search

NPTEL

What will do next time is I will explain how DART works for C by using an example and tell you how these interface extraction test driver extraction undirected search implementation actually happens through an example. So, we will stop with this one now.

Thank you.