

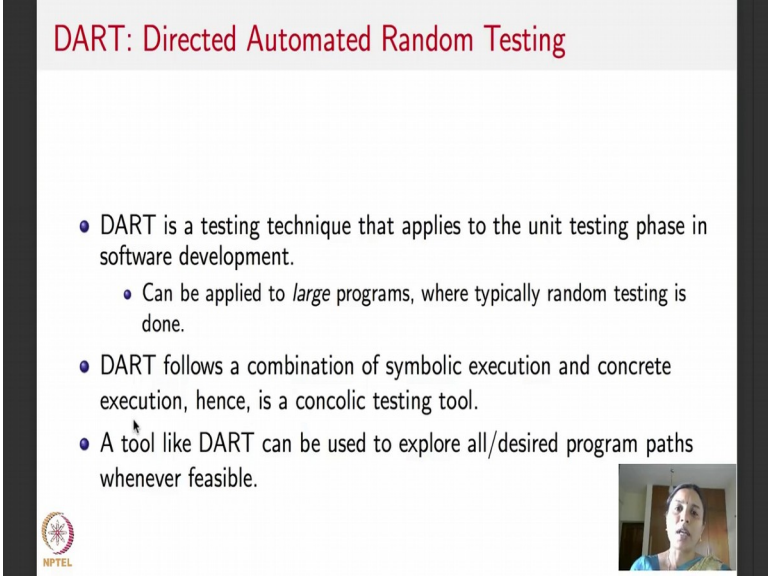
**Software testing**  
**Prof. Meenakshi D'Souza**  
**Department of Computer Science and Engineering**  
**International Institute of Information Technology, Bangalore**

**Lecture – 52**  
**DART: Directed Automated Random Testing**

Hello again, we are going to continue with viewing symbolic testing techniques. Last class I told you about symbolic testing, then we moved on and looked at the disadvantages of symbolic testing. One of the biggest disadvantages is the solving the path constraint. If I collect a set of path constraints, I may not be able to solve it because satisfiability problem is in general un-decidable. The current day solvers that solve these path constraints can handle linear path constraints very well. But let us say if you had something like  $x^2 > 0$  or  $\log^2 x > 0$  which we could encounter while writing code; then the solutions to these become difficult. As a solution, we presented a hybrid symbolic testing technique called concolic testing: concolic, a short form for concrete plus symbolic and as I told you, it does both concrete and symbolic execution together.



So, today what I am going to tell you is one particular concolic testing tool called DART stands for directed automated random testing. We will look at this algorithm that DART deploy is to do test case generation using symbolic testing in detail. We will do it over 3 lectures. In this lecture, I will introduce you to DART. We will get started on what exactly DART does and will look at a mix of concrete and symbolic execution in DART. In the next lecture, I will walk you through the precise algorithm that DART deploy is to instrument a program and to write the test driver. In the last lecture on the DART which will be the third lecture from now, we will look at an example C program where a tool like DART can be applied to do concolic testing.

(Refer Slide Time: 01:59)



**DART: Directed Automated Random Testing**

- DART is a testing technique that applies to the unit testing phase in software development.
  - Can be applied to *large* programs, where typically random testing is done.
- DART follows a combination of symbolic execution and concrete execution, hence, is a concolic testing tool.
- A tool like DART can be used to explore all/desired program paths whenever feasible.

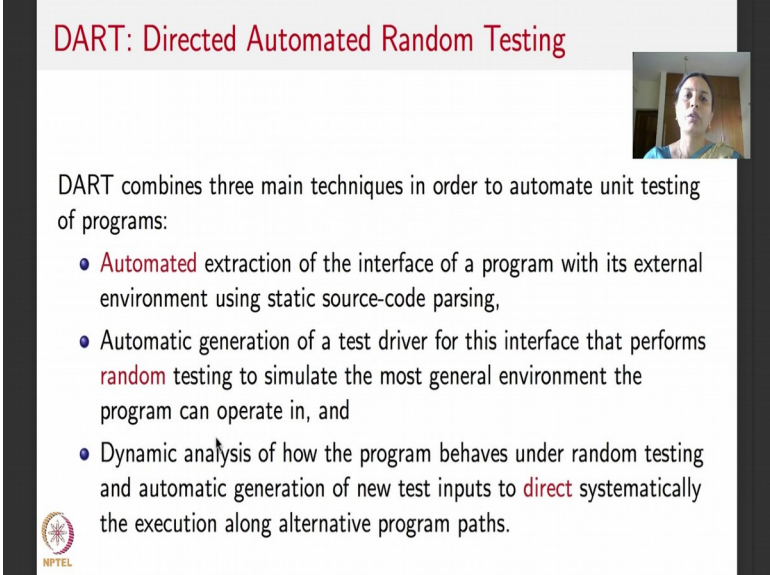
 

So, what is DART? DART is basically a testing technique or an algorithm that applies to unit testing phase. Let us say you have a fairly large piece of code and you are the developer or the programmer and you have been asked to unit test the code. We have seen several testing techniques to do unit test the code, but let us say you are unable to generate graph models or logic models and you have a fairly large piece of code, let us say a few 100-1000 lines of code with you to test. Typically as a unit tester, the burden will be on you to write the test driver to instrument the program for which they may or may not be time left. So, typically unit testers resort to what is called random testing which is they generate a randomly generated test vector a few of them, run the program on it if they are lucky, they might find an error, but if they are not lucky then they may not find an error.

So, what we will see today is a tool called DART which will let you do unit testing in a reasonably automatic way, do not have to write drivers, do not have to instrument the program yourself, DART will do it all on its own. We will see DART as it applies for C, but these days DART like tools are available for Java, for dot net and for several other frame works as I told you. So, what is DART do? DART follows concolic testing which means, it does both symbolic execution and concrete execution and the advantage of DART is that it can be used to explore all feasible program paths, which means it can be done to do what is called exploratory, all combinations paths testing. We have seen when we did graph based testing that all complete path coverage is infeasible several times. So,

DART also may not be able to achieve complete path coverage, but it will do so whenever it can and we will see when it can and when it cannot.


(Refer Slide Time: 03:54)



**DART: Directed Automated Random Testing**

DART combines three main techniques in order to automate unit testing of programs:

- **Automated** extraction of the interface of a program with its external environment using static source-code parsing,
- Automatic generation of a test driver for this interface that performs **random** testing to simulate the most general environment the program can operate in, and
- Dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to **direct** systematically the execution along alternative program paths.



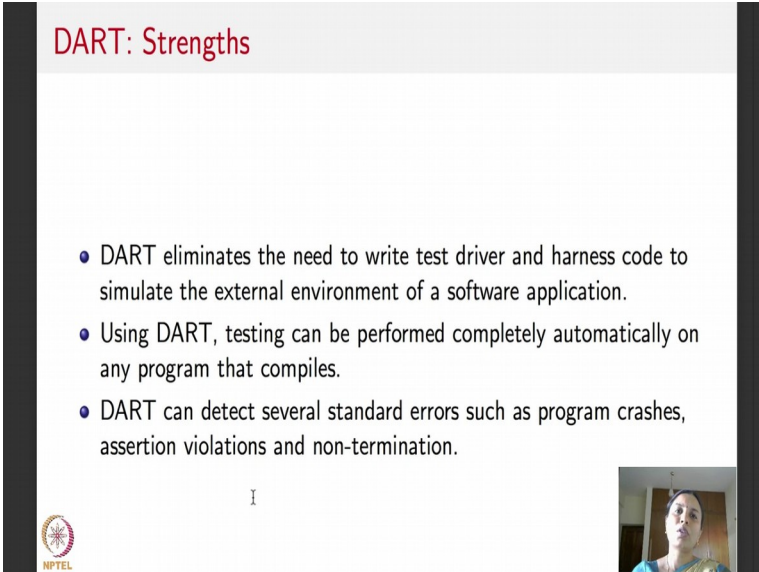
So, what does DART do? DART uses basically 3 main techniques to do its directed automated random testing. So, what are the 3 techniques that DART uses? DART, as I told you, you just give it an executable program, DART will write the driver for your instrument, the program run it and tell you whether it is found an error, explored all paths or it is unable to do either of these. It will do all these automatically. So, to or do all this automatically, DART uses 3 techniques to automate unit testing. First that it does is that it extracts the interface of a program with its external environment automatically. What do we mean by extracting an interface of a program with its external environment? A program might take input from the user, program might send outputs to the console, program might read data from a database.

So, all these things which the program talks to or interfaces which are called external interfaces. DART first does static source code parsing and automatically extracts these interfaces of a program and then what does DART does is it does, writes the test driver. What is a test driver? If you remember from our lessons and integration testing, it is a program that takes and executes your main program under test, DART generates the test driver automatically for this interface which is basically the interface that executes the program, supplies it inputs and what is the main job of the test driver generated by

DART. It is a randomly test a program; randomly test a program means to give random inputs to the program in test. We will see what it means very soon and finally, what is DART do? DART dynamically analyzes how the program behaves under this random testing that was done with to start with in step 2. And then by systematically manipulating the behavior of the program under the random test, DART will generate new inputs to direct the program for specific program paths.

So, let me repeat the 3 steps of DART. The first step is, DART figures out automatically what are the interfaces the program has. After it does that, DART writes a test driver, again automatically that picks up random test case from the interface and starts executing the program on the random test case. As the program executes the random test case, DART collects a few pieces of information. Using that pieces of information, DART will now generate newer and newer inputs that will systematically explore the program on paths, around the paths that the program took on the random test case.

(Refer Slide Time: 06:43)




**DART: Strengths**

- DART eliminates the need to write test driver and harness code to simulate the external environment of a software application.
- Using DART, testing can be performed completely automatically on any program that compiles.
- DART can detect several standard errors such as program crashes, assertion violations and non-termination.

NPTEL

I



So, before we move on and understand DART, what are the strengths of DART, because DART does this interface extraction and driver writing automatically, the user or the tester does not have to write the test driver and the harness code to stimulate the external environment of a software application and because of that testing can be done completely automatically on any program that complies. We will do it for C through an example. DART can detect several standard errors like program crashes that happen because of


errors like division by 0, violations of assertions, presence of non terminating loops and so on.


(Refer Slide Time: 07:21)

### DART through an example

Consider the function h in the file below:

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





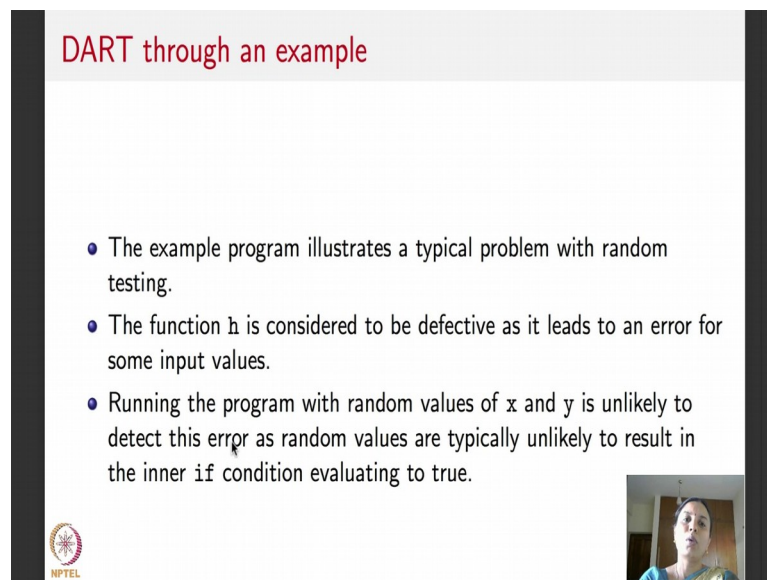
So, before we understand DART, let me revisit the example that I had presented to you towards the end of last lecture. Here is a function h, h takes 2 arguments; integer arguments x and y and returns an integer. It has 2 if statements inside it, first one says if x is not equal to y, you go to the second if statement. Second if statement calls a function f, function f code is given here above function f basically takes an argument x and returns 2 x. So, it says if f of x is equal to x plus 10 which means if 2 x is equal to x plus 10, then read it as that this program for h has reached an error state. So, there is a word called abort and it returns 0.

So, now let us say, this is written by a particular developer and his or her job is to unit test this code. So, typically developer will resort to random testing. Let us assume that because the code is fairly large. In this case obviously, it is not a large example, but let us say that assume that the developer has a large code, he or she will resort to random testing. Random testing means what generate random values of inputs for x and y. Let us say you generate some value, x is 25, y is 35, something like that. Now where is the error in the program? The error in the program is here nested inside the 2 if statements. The conditions for both the if-s should pass for me to be able to reach error. Using a randomly generated input with high probability I will definitely pass the first if

condition, it is unlikely that the 2 randomly generated inputs  $x$  and  $y$  will turn out to be the same.



So, they will most likely be different. So, I will pass the first if condition, but the second if condition is very specific. It says  $f$  of  $x$  which is  $2x$  is  $x$  plus 10, then the second if condition passes. It is not very likely that the randomly generated input will pass the second condition. So, no matter how many times you randomly generate an input and test a program, you might not hit the error statement which is present here in the program. We will see how DART will through one random generation of test case will systematically hit the error statement in the program.

(Refer Slide Time: 09:36)



**DART through an example**


- The example program illustrates a typical problem with random testing.
- The function  $h$  is considered to be defective as it leads to an error for some input values.
- Running the program with random values of  $x$  and  $y$  is unlikely to detect this error as random values are typically unlikely to result in the inner if condition evaluating to true.


So, as I told you, this example program illustrates a typical problem with random testing. The function  $h$  is considered to be defective because it leads to an error for some input values of  $x$  and  $y$ . In particular,  $x$  should not be equal to  $y$  and  $f$  of  $x$  should be equal to  $x$  plus 10; that is when the input it is an error. Running a program with random values for  $x$  and  $y$  is highly unlikely to find the error because the inner if condition is highly unlikely to evaluate true for random values of  $x$  and  $y$ .

(Refer Slide Time: 10:05)

### DART on the example



- DART guesses the value 354 for  $x$  and 34567 for  $y$ .
- As a result, it executes the then-branch of the first if statement, but, fails to execute the then-branch of the second if statement and hence no error is encountered.
- Intertwined with the normal execution, the predicates  $x_0 \neq y_0$  and  $2 \cdot x_0 = x_0 + 10$  are formed on-the-fly.
- $x_0$  and  $y_0$  are symbolic variables that represent the values of the memory locations of the variables  $x$  and  $y$ .
- Note the expression  $2 \cdot x_0$ ; it is formed through an inter-procedural dynamic tracing of symbolic expressions.



So, how will DART now find the error? What are the 3 steps of DART? Figure out the interface of the program begin, generate a random input to the program run the program on it, collect some details, then systematically manipulate the details that you have collected to direct the program on specific program paths. So, DART will begin by generating a random value. Let us say it guesses 354 for  $x$  and some 34567 for  $y$ , some values. Now what will happen? Then in this code, it will obviously, pass the first if statement because  $x$  is not equal to  $y$ , it will go. It will fail the second if statement because  $2x$  is not equal to  $y + 20$ . So, DART will say that the first if statement is passed, the then branch of the second if statement was not taken so, the error is not encountered.

This is a normal execution intertwined with normal execution DART also does extra book keeping, one of the book keeping it does is to collect the predicates that evaluated to true and the predicates that evaluated to false. For this randomly generated test input, the first predicate  $x$  is not equal to  $y$  evaluated to true. So, if DART remembers this, it does not remember it as  $x$  is not equal to  $y$ , it remembers it as  $x_{\text{naught}}$  is not equal to  $y_{\text{naught}}$ . What is this  $x_{\text{naught}}$  and what is this  $y_{\text{naught}}$ ? They are symbolic inputs representing  $x$  and  $y$ . And then the second path constraint is  $2x_{\text{naught}}$  which is false which is actually going to the function  $f$ , says compute  $2x$ , so, you substitute it back here  $x$  is symbolically represented by  $x_{\text{naught}}$ . So, you get  $2x_{\text{naught}}$  is the same as  $x_{\text{naught}}$




plus 10. As I told you  $x$  naught and  $y$  naught are symbolic variables that represent the value of the memory locations corresponding to the values of the variables  $x$  and  $y$ .


So, this will give you a clue of what symbolic representation is. To represent  $x$  symbolically, give a name to the memory location of  $x$ . To represent  $y$  symbolically give a name to the memory location of  $y$ . In this case we have given symbolic value for  $x$  is  $x$  naught, symbolic value for  $y$  is  $y$  naught. Now another thing to note is that how did this path expression come. If you go back to the code this path expression came from if condition. This if condition actually had  $f$  of  $x$ ; that means, that DART had to go to the code corresponding to  $f$  understand that  $f$  of  $x$  is  $2 \star x$ , substitute it back here and then say this is the constraint. So, this comment here just says that. It says that the path expression  $2 \star x$  naught in this predicate is formed through as inter-procedural dynamic tracing of symbolic expression, because from  $h$ , the control gets transferred to the function  $f$  from where you understand that it is  $2 \star x$  and then you substitute it back. DART can do this automatically is what is being said.

(Refer Slide Time: 13:02)

### DART: Exploring different program paths



- DART dynamically gathers knowledge about the execution program, they call it **directed search**.
- Starting with the execution of the program on a random input, DART calculates during each execution, an input vector for the next execution.
- This vector contains values that are the solution of symbolic constraints gathered from predicates in branch statements during the previous execution (called **path constraints**).
- The new vector attempts to force the execution of the program through a new path.
- By repeating this process, a directed search attempts to force the program to sweep through all its feasible execution paths.



Now, DART has done one random generation of a test vector for  $x$  and  $y$ , collected these symbolic constraints. Now DART wants to explore, so, this means what the program for this randomly generated input for  $x$  and  $y$  has taken one particular execution path. Now DART wants to systematically explore the neighborhood of this execution path. So, what it will do is it will do what is called a directed search. So, what it starts with the




execution of the program on the random input and calculates an input vector for next execution. How does it do? As I told you when the program runs on the random input DART collects these path constraints, right. So, what it will do is that it will keep these paths constraints, let us say in a stack. The first constraint is this the second constraint is this; this constraint passed this constraint failed, it will keep that information.


Now, DART wants to explore the neighborhood, for this randomly generated input the second condition failed. So, now, what it will try to do? It will try to negate the second condition that failed, which means it will try to make the second condition pass. So, it will generate a path constraint which will negate the second, this thing, and it will give it to a constraint solver. The constraint solver will give input values for x and y that will satisfy the first constraint and that will satisfy the second constraint.

(Refer Slide Time: 14:30)

### DART on the example

- For the example program, the path constraint  $\langle x_0 \neq y_0, 2 \cdot x_0 \neq x_0 + 10 \rangle$  represents an equivalence class of input vectors containing all input values that drive the program through the path that was just executed.
- To force the program through a different equivalence class, DART-instrumented  $h$  calculates a solution to the path constraint  $\langle x_0 \neq y_0, 2 \cdot x_0 = x_0 + 10 \rangle$  obtained by negating the relevant predicate on the current path constraint.
- A solution to this path constraint (for example,  $(x_0 = 10, y_0 = 4566)$ ) is recorded and when the instrumented  $h$  runs again, it reads the recorded values. Execution with these values reveals the error in the program.

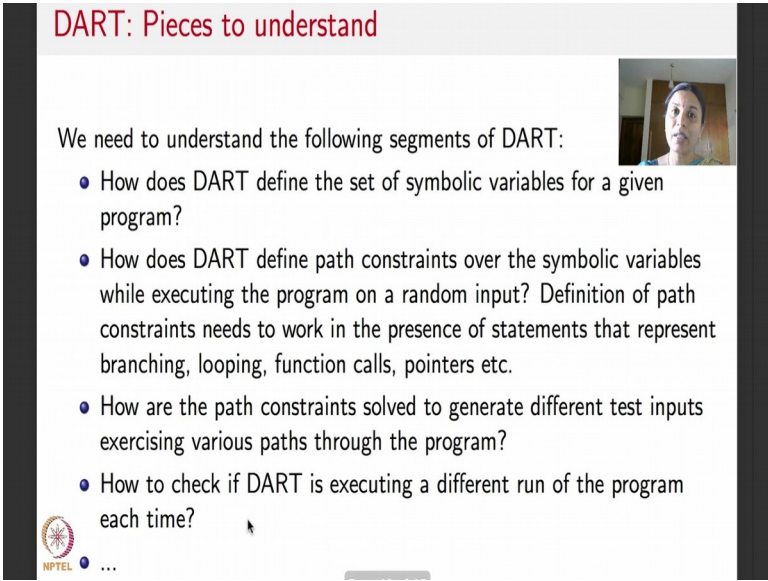




So, that is what is written here. For the example path on the random input these were the constraints that were satisfied:  $x$  naught is not equal to  $y$  naught, I mean not satisfied sorry, generated.  $x$  naught is not equal to  $y$  naught and  $2x$  naught is not equal to  $x$  naught plus 10. Now what DART will do it will take the second path predicate and negate it which means they did it will get  $2x$  naught is equal to  $x$  naught plus 10. What it will do is it will try to give this as a path constraint to a constraint solver and ask the constraint solver to give it values for  $x$  naught and  $y$  naught that will satisfy this. Constraint solver could give a value something like this.

Let us say it will say x naught is 10, y naught is something, and now it will check. X naught is not equal to y naught, correct, 2 x naught is x naught plus 10 both conditions pass. So, which means what both if-s pass the program is reached the error statement. This is how DART goes about systematically exploring neighboring program paths and in the process if it reaches an error statement, it stops and says I have found the error in the program.


(Refer Slide Time: 15:31)




**DART: Pieces to understand**

We need to understand the following segments of DART:

- How does DART define the set of symbolic variables for a given program?
- How does DART define path constraints over the symbolic variables while executing the program on a random input? Definition of path constraints needs to work in the presence of statements that represent branching, looping, function calls, pointers etc.
- How are the path constraints solved to generate different test inputs exercising various paths through the program?
- How to check if DART is executing a different run of the program each time?

 ...

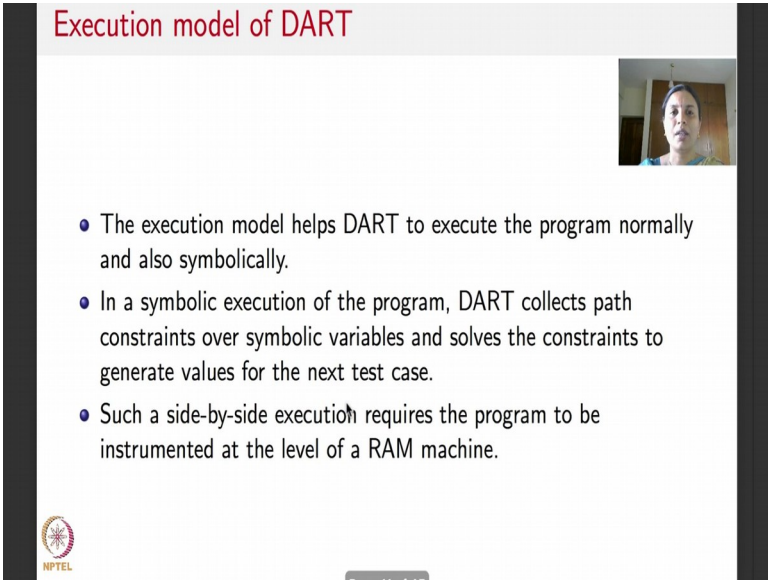


So, that was DART through an example. As I told you our goal is to understand DART in detail. So, what are the pieces that we are going to understand? These are the questions that we will try to answer in this lecture and partly in the next lecture. We will understand how DART defines symbolic variables for a given program. For this example, we will understand how DART came up with x naught, y naught; how did it do it? The second is how does DART define path constraints over the symbolic inputs while it executes the program on a random input, which means what? Here, it executed a program on a random input and it generated these 2 path constraints corresponding to the path taken by the program on the random input. How does DART do that? This toy example did not have much features.

But I my claim is DART will work on arbitrary C programs, which means what they could have branching, looping, they could have function calls which we already saw in this example they could have pointers and so on. So, DART is should be able to generate


these path constraints in the presence of all these entities. The next part is how does it flip one of the conditions in the path constraints? Flipping means, how does it negate one of the conditions in the path constraints. How does it give it to a constraint solver get a solution and then run the program again on that solution and how do you know that DART is actually executing a different run. So, these are the kind of questions that we will understand in detail.

(Refer Slide Time: 17:11)



**Execution model of DART**

- The execution model helps DART to execute the program normally and also symbolically.
- In a symbolic execution of the program, DART collects path constraints over symbolic variables and solves the constraints to generate values for the next test case.
- Such a side-by-side execution requires the program to be instrumented at the level of a RAM machine.




So, before, for us to be able to understand in detail, we need to understand what the execution model of DART is. Which means how does DART go about doing interface extraction, instrumentation, writing a driver. How does DART do this? Remember one thing, DART executes a program both concretely or normally and symbolically. While executing a program symbolically DART collects the symbolic path constraints and solves these constraints by using a constraint solver as a black box. For normal execution, it just records the values of the variables that were defined as the program executes. Now it has to execute both symbolically and normally, this side by side execution of a program demands that DART work, instrument the program at the level of a RAM machine, which means what we are going to assume that the program is actually residing on a particular architecture of a RAM and look at how the program will look like as its residing in memory.

(Refer Slide Time: 18:16)

### 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.




We will understand various bits and pieces of DART. If you focus on the title slide, first thing I am going to understand is symbolic variables. What are symbolic variables in the execution model of DART? We will understand the semantics of a program after that in the execution model of a star DART, followed by understanding of statement labels, concrete semantics, inputs and program execution.

(Refer Slide Time: 18:30)

### Execution model of DART: Semantics of a program

- Typical semantics of a program:
  - A transition system where a state represents the values of all variables and a program counter and transitions represent execution of a program statement resulting in change of state.
  - Each execution of the program results in a path through this transition system.
- For DART, we need to define semantics of a program at memory level. We define it similar to the typical semantics. At memory level, **statements** are specifically tailored abstractions of the machine instructions that are actually executed.



(Refer Slide Time: 18:35)

### Execution model of DART: Statement labels

- For denoting statement labels, we use a set of labels that denote instruction addresses.
- If  $l$  is the address of a statement (other than **abort** or **halt**) then  $l + 1$  is also an address of a statement. There is an initial address, say,  $l_0$ .
- Statements could be of various kinds:
  - A **conditional statement**  $c$  of the form **if**  $e$  **then goto**  $l'$ , where  $e$  is an expression over symbolic variables  $l'$  is a statement label,
  - an **assignment statement**  $a$  of the form  $m \leftarrow e$ , where  $m$  is a memory address,
  - **abort** corresponding to program error, and **halt** corresponding to normal program termination.
- A function **statement** –  $at(l, \mathcal{M})$  specifies the next statement to be executed.

So, these are the bits and pieces with which we will understand the execution model of DART.

(Refer Slide Time: 18:43)

### Execution model of DART: Program Execution

- Let  $\mathbf{C}$  be the set of conditional statements and  $\mathbf{A}$  be the set of assignment statements in  $P$ .
- A **program execution**  $w$  is a finite sequence in  $\mathbf{Execs} := (\mathbf{A} \cup \mathbf{C})^*(\mathbf{abort}|\mathbf{halt})$ .
- 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  $\alpha_i \in \mathbf{A}^*$ ,  $c_i \in \mathbf{C}$ , and  $s \in \{\mathbf{abort}, \mathbf{halt}\}$ .
- An alternate view is to consider  $\mathbf{Execs}(P)$  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 begin with symbolic variables. As I told you, what are symbolic variables? Symbolic variables are place holders for actual values of a variable. In a program as it runs in a real environment which is the best entity to represent a place holder. The best entity to represent a place holder is the actual memory location. Let us say there are 2 variables as we saw in the example,  $x$  and  $y$ . The symbolic representation of  $x$  is the memory location

for  $x$ , the symbolic representation of  $y$  is the memory location for  $y$ . So, first we need to understand how memory looks like. Let us say you have a 32 bit processor or a 64 bit processor or a 128 bit processor then, what is memory? Memory for a 32 bit processor can be thought of as a mapping from memory addresses, call it  $M$ , to 32 bit words. What is  $+$  denote? It denotes that the memory is updated. For example, suppose I write something like this it means that  $M'$  is the same as memory  $M$  except that the memory location small  $m$  is updated with the value for  $V$ .

Otherwise all other memory locations in  $m$  prime  $M'$  are the same as that of  $M$ . What are symbolic variables? As I told you symbolic variables are defined by their addresses. Now in an expression,  $M$  will denote either a memory address or a symbolic variable identified by that address depending on the context as we move on, this will become clear. Now what is a symbolic expression? Symbolic expression is the same as our arithmetic and logical expressions as we saw in the last lecture, but instead of working over normal variables, it works over symbolic variables. So, symbolic expressions could just be variable names which have memory locations for us, it could be a constant which is also another memory location, it could be an arithmetic expression. Please read this star  $e$ ,  $e$  prime within brackets multiplication as any arithmetic expression.

So, it could be plus; it could be minus, it could be slash, it could be mod. For the sake of simplicity, only star is multiplied. But this is to be read and understood as a generic instantiation of any arithmetic expression. Similarly this less than or equal to  $(e, e')$  should be read and understood as a generic instantiation of any relational operator which compares 2 expressions, 2 variables  $e$  and  $e'$ . It could be less than or equal to, greater than or equal to, not equal to, less than, greater than and so on. Again this not  $e$  prime negation of  $e$  prime, is to be interpreted as representing one instances of a logical operator can be substituted with any logical operator and because we are using it for  $C$ , we also work with pointers the star  $e$  prime.

So, this star is a binary star which is multiplication. This star is a unary star which reference to which refers to pointer dereferencing. So, basically symbolic expression is a lot like our arithmetic, logical, relational expression. Instead of working over normal variables, it works over symbolic variables. So, it has constants, it has symbolic variables which are represented by memory location, it has addition, subtraction, multiplication,



division, mod, it has all the relational operators, all the logical operators and operators dealing with pointer arithmetic. We assume that expressions have no side effects.

Now, the semantics of a program. Typically when we look at program, we understand what is the semantics of a program? What is the state of a program? State of a program is the values of all variables in the program plus a location counter which tells you which is the statement that the program is executing. So, that is usually given in this transition system, where a state represents, as I told you a tuple of all the values of a program and variable in a program counter and transition means what. One statement executes and the transition tells you; how one state changes to another state; which value of a variable changes to which other value.

So, each execution of a program results in a path to this through this transition system. But please remember because we are working with DART and DART needs to both concretely and symbolically execute a program, we need to define the semantics of a program at the level of a memory. Which means what, we define it similar to statement, the semantics that is given up here, but we say statements are nothing, but simple machine instructions because that is what happens at the level of a memory.

So, how are statement labels denoted? We take directly the instruction addresses that are available in our computer architecture and say those are the set of statement labels. So, if  $n$  is the address of a statement, let us say a statement label then  $n+1$  is also an address of a statement. As I told you DART is meant to work on our all C programs, but for the purpose of illustrating the algorithm of DART, what the authors of the paper do is consider restricted syntax.

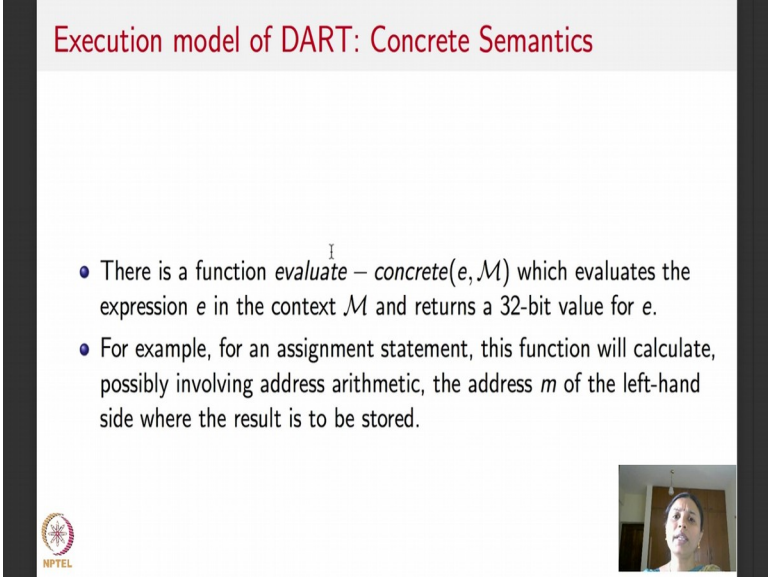
So, what they say is that for now simplicity assume that program has only the following condition kinds of statements, it has conditional statement called C which is like a typical if statement. If an expression evaluates to true then go to a particular statement  $l$  primewhere  $e$  is a symbolic expression and  $l$  is a label of a statement. We also assume that in our restricted syntax a program has only assignment statements which assigns an expression to a memory address and in addition to this, the program has a special statement called abort, which we saw in our example and another statement called halt which corresponds to normal program execution. All other statements that you will



typically find in programming languages like while, anything else, can be expressed using these.

So, for simplicity we assume that as far as understanding the execution of DART is concerned, DART has only, the programming language like C on which DART works, has only condition statements assignment statements and special statements like abort and halt. We also assume that there is a special function that is available to us, let us call it statement at  $l\ m$ , which specifies what is the next statement to be executed.

(Refer Slide Time: 25:12)



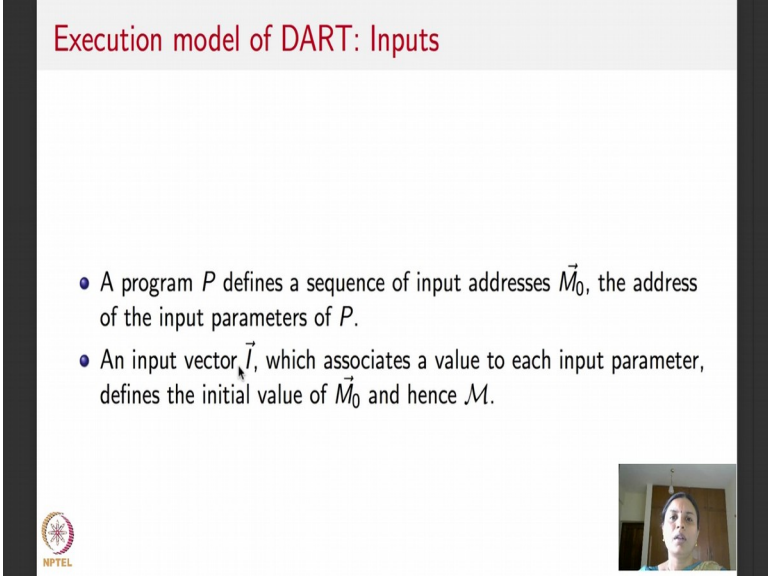
The slide is titled "Execution model of DART: Concrete Semantics" in red text. It contains two bullet points:

- There is a function  $evaluate - concrete(e, \mathcal{M})$  which evaluates the expression  $e$  in the context  $\mathcal{M}$  and returns a 32-bit value for  $e$ .
- For example, for an assignment statement, this function will calculate, possibly involving address arithmetic, the address  $m$  of the left-hand side where the result is to be stored.

In the bottom right corner, there is a small video inset showing a person speaking. In the bottom left corner, there is a logo for NPTEL.

Now, concrete semantics of DART, what will it do? It will directly give you the actual value that is stored in the memory location corresponding to every variable. For example, for an assignment function, this function will calculate, this evaluate concrete which evaluates the concrete semantics, will calculate using some address arithmetic, the address of the left hand side where the result is to be stored.

(Refer Slide Time: 25:33)



The slide is titled "Execution model of DART: Inputs" in red text. It contains two bullet points:

- A program  $P$  defines a sequence of input addresses  $\vec{M}_0$ , the address of the input parameters of  $P$ .
- An input vector  $\vec{I}$ , which associates a value to each input parameter, defines the initial value of  $\vec{M}_0$  and hence  $\mathcal{M}$ .

In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a person speaking.

What are the inputs to DART? Inputs to DART are the memory locations corresponding to the variables symbolic variables that actually correspond to the input. Input vector associates a value to each input parameters hence defines an initial value of memory. So, we assume as I told you for simplicity that a program has only conditional statements and only assignment statements. So, program execution will be a series of assignment statements and conditional statements written as  $A \cup C$ . Any number of them, which is written using the regular expression star and the program can either abnormally halt with an abort, or normally halt with a halt. So, read it like you would read a regular expression, they could be assignments, they could be conditions, star means any combinations of them, any number of them. This represents the program ending, abort represents the program ending abnormally, halt represents the program ending normally.


So, to further simplify and understand our model we can assume without loss of generality that program execution looks like this. It has a series of assignment statements followed by a condition, followed by another series of assignment statements followed by a condition and so on. Basically what it say is assignment statements and conditions could alternate. These  $\alpha_i$ 's are assignment statements they belong to a star they - could be no  $\alpha_i$ 's,  $C_i$ 's are conditionals and then the program always ends with an abort or a halt s belongs to abort or a halt.

So, this is how a program execution looks like and whenever a program executes.

(Refer Slide Time: 27:07)

### Symbolic evaluation of expressions

- DART maintains a **symbolic memory**  $\mathcal{S}$  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,  $\mathcal{S}$  just maps each  $m \in M_0$  to itself.
- The mapping  $\mathcal{S}$  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



There is one path, in the execution tree that the thing takes. So, I will stop here in the lecture. I will continue with how DART evaluates each of these expressions symbolically.

Thank you.