**1. VERIFICATION AND VALIDATION**

* Software testing is one element of a broader topic that is often referred to **as verification and validation (V&V).**
* **Verification**refers to the set of activities that ensure that software correctly implements a specific function.
* Verification*:* "Are we building the product right?"
* **Validation**refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.
* *Validation:* "Are we building the right product?"

**1.1**   **STATIC AND DYNAMIC TESTING**

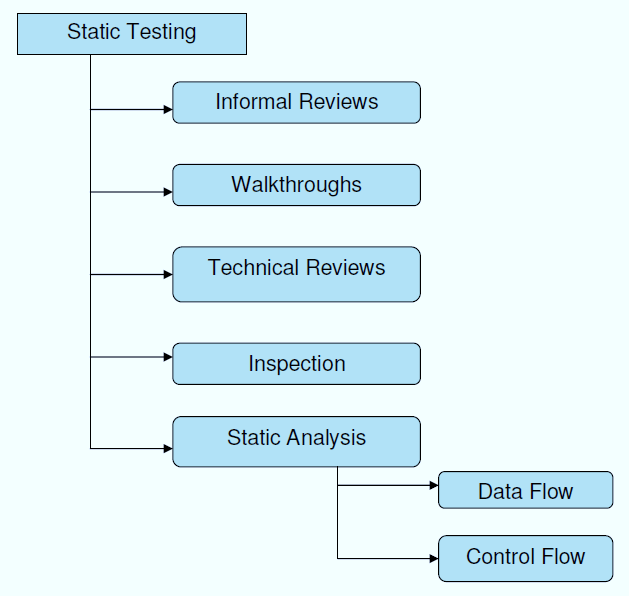
**1.1.1 STATIC TESTING**

Main objective of this testing is to improve the quality of software products by finding errors in early stages of the development cycle. This testing is also called as Non-execution technique or verification testing. Static testing involves manual or automated reviews of the documents.It examines work documents and code and provides review comments. Work document and code can be of following:

1. Requirement specifications
2. Design document
3. Source Code
4. Test Plans
5. Test Cases
6. Test Scripts
7. Help or User document
8. Web Page content

**1.1.1.1 Testing Techniques used for Static Testing:**

* **Informal Reviews:**This is one of the type of review which doesn't follow any process to find errors in the document. Under this technique , you just review the document and give informal comments on it.
* **Technical Reviews:**A team consisting of your  peers,   review the technical specification of the software product and checks whether it is suitable for the project. They try to  find any discrepancies in the specifications and standards followed. This review concentrates mainly on the technical document related to the software such as Test Strategy, Test Plan and requirement specification documents.
* **Walkthrough:**The author of the work product explains the product to his team. Participants can ask questions if any.  Meeting is led by the author*.* Scribe makes note of review comments
* **Inspection:**The main purpose is to find defects and meeting is led by trained moderator. This review is a formal type of review where it follows strict process to find the defects. Reviewers have checklist to review the work products .They record the defect and inform the participants to rectify those errors.
* **Static code Review:** This is systematic review of the software source code without executing the code. It checks the syntax of the code, coding standards, code optimization, etc. This is also termed as white box testing .This review can be done at any point during development.



**Figure representing Static Testing**

**1.1.2 DYNAMIC TESTING**

Main objective of this testing is to confirm that the software product works in conformance with  the business requirements. This testing is also called as Execution technique or validation testing.Dynamic testing executes the software and validates the output with the expected outcome. Dynamic testing is performed at all levels of testing and it can be either black or white box testing.

## 1.1.2.1 Testing Techniques used for Dynamic Testing:

* **Unit Testing:**

Under unit testing ,  individual units or modules is  tested by the developers. It  involves testing of source code by developers. It forms the basis for component testing.

* **Integration Testing:**

Individual modules are grouped together and tested by the developers. The purpose is to determine that modules are working as expected once they are integrated. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing. The purpose of integration testing is to verify functional, performance, and reliability requirements placed on major design items.

* **System Testing**: System testing is performed on the whole system by checking whether the system or application meets the requirement specification document.
* **Acceptance testing:**Formal testing with respect to user needs, requirements, and business processes conducted to determine  whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system. Usually, [Black Box Testing](http://softwaretestingfundamentals.com/black-box-testing/) method is used in Acceptance Testing.
* **Functional Testing:**Functional Testing is a testing technique that is used to test the functionality of the system or Software, should cover all the scenarios including failure paths and boundary cases.It falls under the class of black box testing.
* **Stress Testing:**

Testing used to determine the stability of a given system or entity. It involves testing beyond normal operational capacity, often to a breaking point, in order to observe the results.Testing is conducted at beyond limits of the specifications.

* **Performance Testing:**

Performance testing is the process of determining the speed or effectiveness of a computer, network, software program or device. This process can involve quantitative tests done in a lab, such as measuring the response time at which a system functions.

* **Usability Testing:**

Usability testing is a technique used in user-centered interaction design to evaluate a product by testing it on users. Usability testing focuses on measuring a human-made product's capacity to meet its intended purpose.The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

* **Regression Testing:**

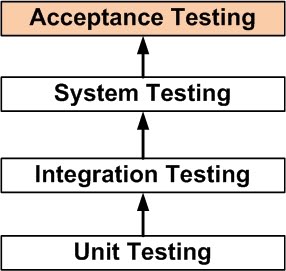
Regression testing is the testing after modification of a system or a component, or a group of related units to ensure that the modification is working correctly and is not damaging or imposing other modules to produce unexpected results. It falls under the class of black box testing.

* **Alpha Testing:**

Alpha testing is testing of an application when development is about to complete. Minor design changes can still be made as a result of alpha testing.Alpha testing is typically performed by a group that is independent of the design team, but still within the company.Alpha testing is often employed for off-the- shelf software as a form of internal acceptance testing, before the software goes to beta testing.

* **Beta Testing:**

Beta testing is the testing which is done by end users, a team outside development, or publicly releasing full pre-version of the product which is known as beta version.The aim of beta testing is to cover unexpected errors. It falls under the class of black box testing.



**Figure representing dynamic testing**

.

**To understand the concept of code coverage, the different phases are needed to be studied under SDLC to know how code is generated,on what basis it was generated and how can it be modified** .

**2 SDLC(SOFTWARE DEVELOPEMENT LIFE CYCLE) PHASES**

Software life cycle models describe phases of the software cycle and the order in which those phases are executed. Each phase produces deliverables required by the next phase in the life cycle. Requirements are translated into design**. Code** is produced according to the design which is called development phase. After the coding and development the testing verifies the deliverable of the implementation phase against requirements.

Six phases in sdlc are:

1) Requirement gathering Analysis

2) Design

3) Implementation

4) Testing

5) Deployment

6) Maintenance.

**1)Requirement gathering Analysis:**

Business requirements are gathered in this phase. This phase is the main focus of the project managers and stake holders and users are held in order to determine the requirements .After the requirements gathering these requirements are analyzed for their validity and the possibility of incorporating the requirements in the system to be development is also studied. Finally SRS is prepared .

**2)  Design:**

In this phase the system and software design is prepared from the requirement specifications which were studied in the first phase. System Design helps in specifying hardware and software requirements and also helps in defining overall system architecture. The system design specifications serve as input for the next phase of the model.

**3)  Implementation / Coding:**

  On receiving system design documents, the work is divided in modules/units and actual coding is started. Since, in this phase the code is produced so it is the main focus for the developer. This is the longest phase of the software development life cycle.

**4)**[**Testing**](http://istqbexamcertification.com/what-is-a-software-testing/)**:**

  After the code is developed it is tested against the requirements to make sure that the product is actually solving the needs addressed during the requirements phase.

**5) Deployment:** After successful testing the product is delivered / deployed to the customer for their use.As soon as the product is given to the customers they will first do the beta testing. If any changes are required or if any bugs are caught, then they will report it to the engineering team. Once those changes are made or the bugs are fixed then the final deployment will happen.

**6) Maintenance:**

 Maintenance needed due to a change in the environment or the requirements of the system.  maintenance resolves around understanding the existing software and spares most of their time trying to understand the software that they have to modify. Understanding the software involves not only understanding the code, but also the related documents. During the modification of the software, the effects of the change have to be clearly understood by the maintainer since introducing undesired side effects in the system during modification is easier.

**Requirement gathering Analysis**

**Design**

**Implementation / Coding**

[**Testing**](http://istqbexamcertification.com/what-is-a-software-testing/)

**Deployment**

**Maintenance**

**Figure representing phases of SDLC**

**3 CODE COVERAGE**

**3.1 What is code coverage** ?

Code coverageis the degree of code which has been tested during testing or how much code has been covered during testing. Code coverage measures if all the possible branches have been exercised by your test suite.

It also lets you find hotspots. Code coverage should be part of continuous integration.

**Cyclomatic Complexity** is a metric of complexity that counts the number of independent paths through the source code, and assigns a single numerical score for each method. This provides information on code coverage.

**3.2 Code coverage tool :**

No matter how good methodologies are, and how diligently they're followed in an organization, it isn't possible to ensure that software testing is as comprehensive as it could be. That's where tools can help.

**3.3 How much coverage is required?**

If we can reach 100 % coverage, it would be great but it will require a considerable time, effort and resources to be there. The norm is that 80-90% of coverage is good. But anything below 60% is unacceptable.

**3.4 TYPES OF CODE COVERAGE**

**3.4.1)Statement coverage** - Has each [statement](https://en.wikipedia.org/wiki/Statement_(computer_science)) in the program been executed?

For example, consider the following C function:

int foo (int x, int y)

{

int z = 0;

**if** ((x>0) && (y>0))

{

z = x;

}

**return** z;

}

Assume this function is a part of some bigger program and this program was run with some test suite.

Statementcoverage for this function will be satisfied if it was called e.g. as foo(1,1), as in this case, every line in the function is executed including z = x;.

**3.4.2)Branch coverage** - Has each branch of each control structure (such as in [if and case statements](https://en.wikipedia.org/wiki/Conditional_(programming))) been executed? For example, given an *if* statement, have both the true and false branches been executed? Another way of saying this is, has every [edge](https://en.wikipedia.org/wiki/Graph_theory) in the program been executed?

In the above example Tests calling foo(1,1) and foo(0,1) will satisfy branchcoverage because, in the first case, the 2 if conditions are met and z = x; is executed, while in the second case, the first condition (x>0) is not satisfied, which prevents executing z = x;.

**3.4.3)Condition coverage** (or predicate coverage) - Has each Boolean sub-expression evaluated both to true and false?

Conditioncoverage can be satisfied with tests that call foo(1,1), foo(1,0) and foo(0,0). These are necessary because in the first two cases, (x>0) evaluates to true, while in the third, it evaluates false. At the same time, the first case makes (y>0) true, while the second and third make it false.

Condition coverage does not necessarily imply branch coverage. For example, consider the following fragment of code:

**If a and b then**

Condition coverage can be satisfied by two tests

* a=true, b=false
* a=false, b=true

However, this set of tests does not satisfy branch coverage since neither case will meet the if condition.

[Fault injection](https://en.wikipedia.org/wiki/Fault_injection) may be necessary to ensure that all conditions and branches of [exception handling](https://en.wikipedia.org/wiki/Exception_handling) code have adequate coverage during testing.

**3.4.4)Modified condition/decision coverag**e:

A combination of “function coverage and branch coverage is sometimes also called **decision coverage”**. This criterion requires that every point of entry and exit in the program have been invoked at least once, and every decision in the program have taken on all possible outcomes at least once. In this context the decision is a boolean expression composed of conditions and zero or more boolean operators. This definition is not the same as branch coverage,however, some do use the term decisioncoverage as a synonym for branchcoverage.

Condition**/**decisioncoverage requires that both decision and condition coverage be satisfied. However, for [safety-critical](https://en.wikipedia.org/wiki/Safety-critical) applications (e.g., for avionics software) it is often required that modifiedcondition**/**decisioncoverage **(**MC**/**DC**)** be satisfied. This criterion extends condition/decision criteria with requirements that each condition should affect the decision outcome independently. For example, consider the following code:

**If ( a or b) and c then**

The condition/decision criteria will be satisfied by the following set of tests:

* a=true, b=true, c=true
* a=false, b=false, c=false

However, the above tests set will not satisfy modified condition/decision coverage, since in the first test, the value of 'b' and in the second test the value of 'c' would not influence the output. So, the following test set is needed to satisfy MC/DC:

* a=false, b=false, c=true
* a=true, b=false, c=true
* a=false, b=true, c=true
* a=false, b=true, c=false

**3.5 Tools** :

Some of the code coverage tools are**:**

* GCOV
* Cobertura
* EMMA
* Clover
* JCoverage

**GCOV:**

* Gcov generates exact counts of the number of times each statement in a program is executed and annotates [source code](https://en.wikipedia.org/wiki/Source_code) to add instrumentation.
* To enable coverage testing the program must be compiled with the following options:

**gcc -Wall -fprofile-arcs -ftest-coverage cov.c**

* gcov creates a logfile called sourcefile*.*gcov which indicates how many times each line of a source file sourcefile*.c* has executed.

**Cobertura :**

* Standard execution environment.
* Ant/maven Integration .
* XML/HTML Reports including code complexity
* Support by several continuous integration tools.

**Emma:**

* Byte code instrumentation
* Standard execution environment
* Ant/maven Integration
* XML/HTML Reports doesn't support branch or path coverage
* No support for eclipse

**Clover *:***

* Source code instrumentation
* Standard execution environment
* Ant/maven Integration
* XML/HTML Reports
* Commercial product from Atlassian

**JCoverage:**

* Commercial - cobertura was forked from the JCoverage code base.

**3.5 Conclusion :**

Code coverage report is not silver bullet for code quality. Lines of code can be misleading. Coverage - all it means is more lines of code exercised. Pick tools that best suit your needs:

Scenario tests coupled with code coverage could workout. Insist on code quality reports with all code drops and releases. Bug detection tools come handy . Ex : Find Bugs.

**4 CONCOLIC TESTING**

**4.1 Concrete Execution:**

Concrete Execution is nothing but execution of random input data.

Void f(int x)

{

If(x==1451)

{

Printf (“x=%d”, x);

}

Else

{

//error

}

}

The values are from **-32768 to 32767** .So the searching of the “x” value will take lot of time. If the value of x becomes 1451 then it becomes true otherwise it will go the else block.

**Drawback**: It takes more time for searching of variables.

**4.2 Symbolic Execution:**

Symbolic execution allows us to execute a program through all possible execution paths, thus achieving all possible path conditions.

* automic test case generation.
* high code coverage.

**4.2.1 Execute the program in symbolic domain :**

* Explore all possible execution paths.
* for each path the constraints of the branching points are collected
* generate test input based on the constraints.

Consider the below example.

y=a , z=2\*a;

1. int f() {

2 ...

3 y = read();

4 z = y \* 2;

5 **if** (z == 12) {

6 fail();

7 } **else** {

8 printf("OK");

9 }

10 }

* During a normal execution , the program would read a concrete input value and assign it to y. Execution would then proceed with the multiplication and the conditional branch, which would evaluate to false and print OK.
* During symbolic execution, the program reads a symbolic value (e.g., a) and assigns it to y. The program would then proceed with the multiplication and assign  **a\* 2 to z.**
* When reaching the if  statement, it would evaluate **a\* 2 == 12**. At this point of the program, “a” could **take any value**, and symbolic execution can therefore proceed along both branches, by **"forking"** two paths.
* Each path get assigned a copy of the program state at the branch instruction as well as a path constraint. In this example, the path constraint is **a \* 2 == 12 for the if branch** and **a\* 2 != 12 for the  else  branch.**
* Both paths can be symbolically executed independently. When paths terminate (e.g., as a result of executing fail() or simply exiting), symbolic execution computes a concrete value for “ a” by solving the accumulated path constraints on each path.
* These concrete values can be thought of as concrete test cases that can, e.g., help developers reproduce bugs. In this example, the constraint solver would determine that in order to reach the fail()statement, “a” would need to equal 6.

**4.2.2 Limitations:**

1. **Path Explosion:**

* Symbolically, executing all feasible program paths does not scale to large programs.
* The number of feasible paths in a program grows exponentially with an increase in program size and can even be infinite in the case of programs with unbounded loop iterations.
* Solutions to the ***path explosion***problem generally use either heuristics for path-finding to increase code coverage, reduce execution time by parallelizing independent paths, or by merging similar paths.

1. **Program-Dependent Efficacy**:

* Symbolic execution is used to reason about a program path-by-path which is an advantage over reasoning about a program input-by-input as other testing paradigms use .
* However, if few inputs take the same path through the program, there is little savings over testing each of the inputs separately.

**3. Modeling the environment.:**

 In this case, the engine instruments the system calls with a model that simulates their effects and that keeps all the side effects in per-state storage. The advantage is that one would get correct results when symbolically executing programs that interact with the environment. The disadvantage is that one needs to implement and maintain many potentially complex models of system calls. Tools such as KLEEand Cloud9 take this approach by implementing models for file system operations, sockets, IPC, etc.

**4.3 Concolic Testing:**

In **unit testing** as discussed earlier, a program is decomposed into units which are collections of functions. A part of unit can be tested by generating inputs for a single entry function.

* Concolic testing is also known as Symbolic Execution. It is the only method for test input generation where a given program is executed both concretely and symbolically at the same time. The main idea behind this is to at runtime collect symbolic constraints on inputs to the system that specify the possible input values to the specific execution path.
* In concolic testing the program under the test is first executed with concrete random input values. Symbolic execution is used in conjunction with an automated theorem prover based on constraint logic programming to generate new concrete inputs with the aim of maximizing code coverage. The concrete execution in concolic testing brings the benefit that it makes available accurate information about the program state which might not be easily accessible when using only static analysis.
* The concolic approach is also applicable for model checking. In a concolic model checker, the model checker traverses states of the model representing the software being checked, while storing both a concrete state and a symbolic state. The symbolic state is used for checking properties on the software, while the concrete state is used to avoid reaching unreachable state.
* Concolic execution is a mix between **CONC**rete execution and symb**OLIC** execution, with the purpose of feasibility.
* Symbolic execution allows us to execute a program through all possible execution paths, thus achieving all possible path conditions (path condition = the set of logical constraints that takes us to a specific point in the execution). The problem is that, except for micro benchmarks, the cost of executing a program through all possible execution paths is exponentially large, thus prohibitive.
* On the other hand, if we provide the symbolic execution with concrete values, you can guide it through a specific execution path (without traversing all of them) and achieve the respective path condition. This is feasible.

**4.3.1 Algorithm:**

Some steps are considered while executing the given system in concolic testing, they are listed below.

1.classify a particular set of variables as *input variables*. these variables will be treated as symbolic variables during symbolic execution. all other variables will be treated as concrete values.

2.Instrument the program so that each operation which may affect a symbolic variable value or a path condition is logged to a trace file, as well as any error that occurs.

3.Choose an arbitrary input to begin with.

4.Execute the program.

5.Symbolically re-execute the program on the trace, generating a set of symbolic constraints (including path conditions).

6.Negate the last path condition not already negated in order to visit a new execution path. If there is no such path condition, the algorithm terminates.

7.Invoke an automated theorem prover to generate a new input. If there is no input satisfying the constraints, return to step 6 to try the next execution path

8.Return to step 4.

**4.3.2 Concolic testing example :**

1.void f(int x, int y) {

2 int z = 2\*y;

3 **if** (x = = 100000) {

4 **if** (x < z) {

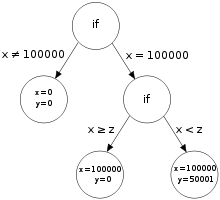
5 assert(0); */\* error \*/*

6 }

7 }

8 }

* .Simple random testing, trying random values of *x* and *y*, would require an impractically large number of tests to reproduce the fails.
* We begin with an arbitrary choice for *x* and *y*, for example ***x* = *y* = 1**. In the concrete execution, line **2 sets *z* to 2,** and the test in **line 3 fails since 1 ≠ 100000**.
* Concurrently, the symbolic execution follows the same path but treats *x* and *y* as symbolic variables. It sets *z* to the expression 2*y* and notes that, because the test in line 3 failed, *x* ≠ 100000.
* This inequality is called a ***path condition*** and must be true for all executions following the same execution path as the current one.
* Since we'd like the program to follow a different execution path on the next run, we take the last path condition encountered, *x* ≠ 100000, and **negate it, giving *x* = 100000**.
* An **automated theorem prover** is then invoked to find values for the input variables *x* and *y* given the complete set of symbolic variable values and path conditions constructed during symbolic execution. In this case, a valid response from the theorem prover might be ***x* = 100000, *y* = 0**.
* Running the program on this input allows it to reach the inner branch on line 4, which is **not taken** since **100000 (*x*) is not less than 0 (*z* = 2*y*).**
* The path conditions are ***x* = 100000** and ***x* ≥ *z*.** The latter is **negated,** giving ***x* < *z*.** The theorem prover then looks for *x*, *y* **satisfying *x* = 100000, *x* < *z***, and ***z* = 2*y***; for example, ***x* = 100000, *y* = 50001**. This input reaches the error.



**Figure representing the branched structure of above example code**

**4.3.2 Another example representing the approach of concolic testing:**

int double (int v) {

return 2\*v;

}

void testme (int x, int y) {

z = double (y);

if (z == x) {

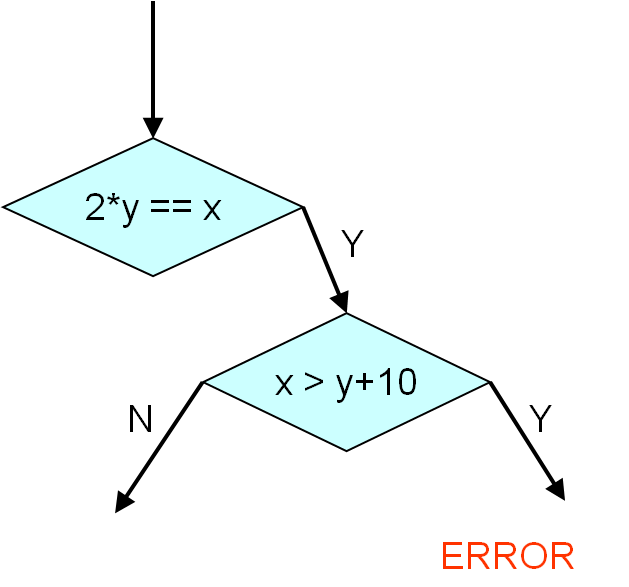
if (x > y+10) {

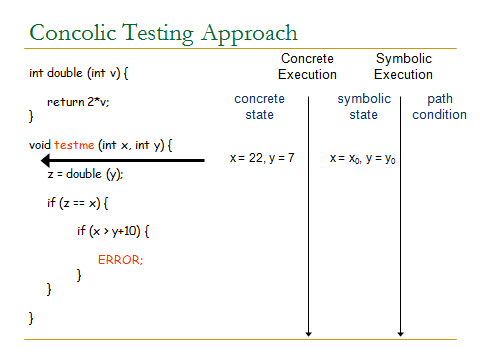
ERROR;

}

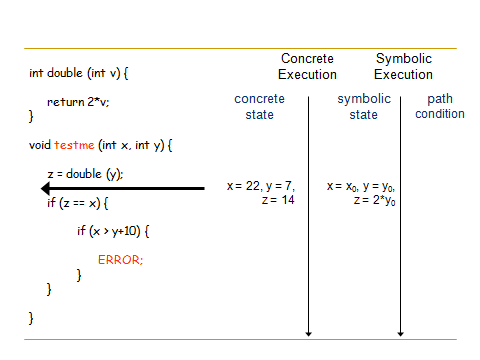
}

}

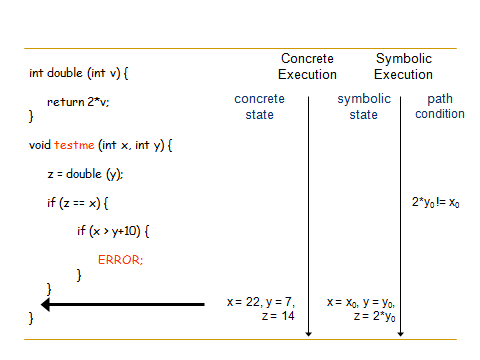
**  
Figure representing the above example code**

****

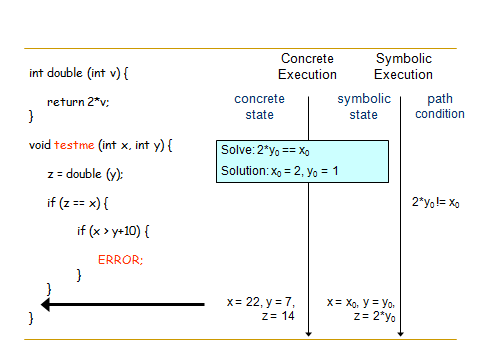
1. Now concrete values that are random are substituted and z value is obtained.

****

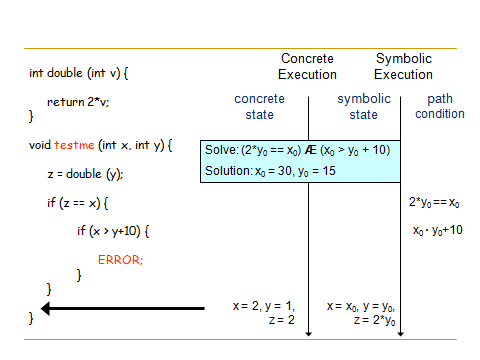
1. Equation for z is obtained by symbolic execution

****

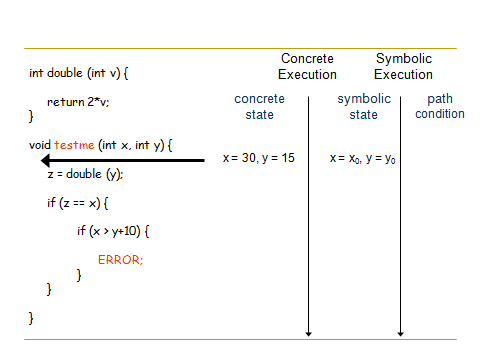
1. Equation is solved and solution ontained and process continued till all branches are covered in the code.

****

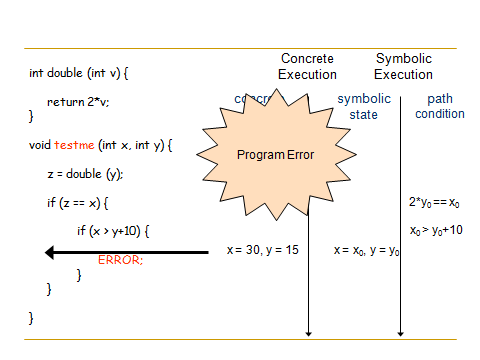
1. Finally,a from above example the inputs are obtained by solving that shows error in code.

****

1. These values are substituted again and processed till condition for error is found.

****

1. Condition in if statement is satisfied and therfore program error found and needed to be modified.

****

* The above example approach shows how both the concrete execution and symbolic execution are done together to identify the error in the code.
* So finally,the values of x and y taken after solving using both executions **x=30** and **y=15** shows the error since the conditon **x>y+10**which gives **30>25** is true.

**5 TOOLS FOR CONCOLIC TESTING**

**5.1** **CREST: An Automatic generation Tool For C**

**5.1.1 Introduction :**

CREST is an automatic test generation tool for C.

It works by inserting instrumentation code into a target program to perform symbolic execution concurrently with the concrete execution.

The generated symbolic constraints are solved using Yices to generate input that drive the test execution down new, unexpected program paths.

CREST currently reasons symbolically only about linear, integer arithmetic. CREST should be usable on any modern Linux system. It is usable on recent Mac OS X versions, as well, although some small modifications are needed for the code to build.

* CREST is a concolic testing tool for C programs
* Generates test inputs automatically
* Execute target under test on generated test inputs
* Explore all possible execution paths of a target systematically
* CREST is a open source reimplantation of CUTE
* CREST's instrumentation is implemented as a module of CIL ( C Intermediate language) written in ocaml.

**5.1.2 4 Main steps of Concolic Testing**

1. Instrumentation of a target program

* To insert probes to build symbolic path formula

2. Transform a constructed symbolic path formula to SMT- compatible format

* SMT solvers can solve simple formula only

3. Select one branch condition to negate

* Core technique impacting both effectiveness and efficiency

4. Invoking SMT- solvers on the SPF SMT- formula

* Selection of a SMT- solver and proper configuration parameters

**5.1.3 4 main tasks of human engineers**

1.Adding proper assert() statements

2. Selection of symbolic variables in a target program

* Identify which parts of a target program are most important

3. Construction of symbolic external environment

* To detect real bugs

4. Performance tuning and debugging

* To obtain better concolic testing results

**5.1.4 CREST** **commands**

1. CREST <filename>.c

**5.1.5 Output :**

* <filename>.cil.c //instrumented c file
* branches //lists of paired branches
* <filename> //executable file
* <n> # of iterations
* *dfd* : depth first search (in reverse order)
* *cfg* : uncovered branch first
* *random* : negated branch is randomly selected
* *random*\_input : pure random input
* *hybrid* : combination of dfd and random