

# WHITE AND BLACK BOX TESTING

DEPARTMENT OF COMPUTER SCIENCES  
UNIVERSITY OF KASHMIR  
North Campus

# WHITE BOX AND BLACK BOX TESTING

Dynamic testing is generally divided into the two broad categories depending on a criterion whether we require the knowledge of source code or not for test case design, if it does not require knowledge of the source code, it is known as **black box testing** otherwise it is known as **white box testing**, which correspond with two different starting points for dynamic software testing: the requirements specification and internal structure of the software.

Black box testing is focused on results whereas white box testing is focused on details.

Black box testing is typically used to check the product without examining the code. But we do not know how much of it is being tested. This is where the white-box techniques come in. They allow you to examine the code in detail and be confident that at least you have achieved a certain level of test coverage.

White box testing is in itself insufficient since the software under examination may not perform one of its desired tasks—the function to do this may even be missing—and the examination of the code is unlikely to reveal this. The main purpose of black box testing is to spot such missing functionality.

Considering the purpose of these testing strategies, it appears that both of them should be used in order to test a product fully.

White box strategy is used only at the unit testing and program integration testing stages, whereas black box testing strategy can be used at all testing stages (unit testing to acceptance testing).




# WHITE BOX TESTING

White box testing strategy gives us the internal view of the software and covers implemented behavior of program. It is also known as *open box / clear box testing*.

White box testing strategy deals with the internal logic and structure of the code and depends on the information how software has been designed and coded for test case design.

The test cases designed based on the white box testing strategy incorporate coverage of the code written in terms of branches, paths, statements and internal logic of the code, etc.

White box testing strategy is focused on examining the logic of the program or system, without concerned about the requirements of software which is under test. The expected results are evaluated on a set of coverage criteria.



We need have knowledge of coding and logic of the software to implement white box testing strategy. It checks which statement or segment of code is not working properly. It even checks internal data structures.

This testing methodology enables us to see what is happening inside the software.

White box testing provides a degree of sophistication as we are able to refer to and interact with the objects that comprise a software rather than only having access to the user interface.

An example of white box testing would be an auto-mechanic who looks at the inner-workings of a car to ensure that all of the individual parts are working correctly to ensure the car drives properly.

# STATIC WHITE BOX TESTING

Static white box testing as a process involving a methodical & careful examination of the software architecture, basic design or its code with a view to hunt for faults without executing it. It is called structural analysis as well sometimes.

It is a type of testing in which the program source code is tested without running it. We only need to examine and review the code. It can be thought of as sort of dry run mechanism. We need to find out whether

- a) The code works according to the functional requirements.
- b) The code has been written in accordance with the design developed earlier in the project life cycle.
- c) The code for any functionality has been missed out.
- d) The code handles errors properly.

Static testing can be done by human being ([\*manual testing\*](#)) or with the help of specialized tools ([\*automatic testing\*](#)). Common static white-box techniques include [Formal Review](#), [Desk Checking](#), [Code walkthrough](#) and [Formal Inspections](#).

# DYNAMIC WHITE BOX TESTING

**Dynamic white box testing:** This involves testing a running program. So, now binaries and executable are desired. We try to test the internal logic of the program now. It entails running the actual product against some pre-designed test cases to exercise as much of the code as possible.

Dynamic White-Box testing should tell us exactly what material it covers. It runs program, look inside the box (program code), examine the code.

**Some of the key Dynamic White Box Testing processes are as under:**

## **Unit / Code Functional Testing:**

It is the process of testing in which the developer performs some quick checks prior to subjecting the code to more extensive code coverage testing or code complexity testing. It can be performed in many ways

a) At the initial stages, the developer or tester can perform certain tests based on the input variables and the corresponding expected output variables. This can be a quick test. If we repeat these tests for multiple values of input variables also then the confidence level of the developer to go to the next level increases.

b) For complex modules, the tester can insert some print statements in between to check whether the program control passes through all statements and loops. It is important to remove the intermediate print statements after the defects are fixed.

c) Another method is to run the product under a debugger or an Integrated Development Environment (IDE). These tools involve single stepping of instructions, setting break points at any function or instruction.

All these initial tests, actually fall under "Debugging" category rather than under "Testing" category of activities. These are placed under "White Box Testing" head as all are related to the knowledge of code structure.

### **Code Coverage Testing**

Code coverage testing involves designing and executing test cases and finding out the percentage of code that is covered by testing. It basically finds the extent to which (source) code is executed, i.e. Covered. The percentage of code covered by a test is found by adopting a technique called as the instrumentation of code.

Test coverage attempts to address questions about when to stop testing, or the amount of testing that is enough for a given program.

Ideal testing is to explore exhaustively the entire test domain, which in general (and in practice) is impossible.



Structural testing method, a primary dynamic white box testing method, are used to design test cases based on the internal structure of the component or system; most commonly internal structure is referred to as the structure of the code. These techniques therefore focus on the testing of code and they are primarily used for unit testing and integration testing.

The testing techniques in this category all require that we understand the structure; that also implies that we should have some knowledge of the coding language. That is the reason programmers are mostly preferred to do structural testing. The test basis used is program and design specifications where the details about the program's internal structure are clearly described.

Test cases are designed to get the required coverage for the specified coverage item. The goal of structural testing is to choose a set of inputs according to the structure of the program and aim that all parts (statements, branches or paths) are tested at least once. The structural tests encompass *control flow (program flow)* and *data flow*. Control Flow Testing (CFT) and Data Flow Testing (DFT) help with interactions along the execution path and interactions among data items in execution respectively.

# CONTROL FLOW TESTING

Control flow testing addresses the test case coverage of “execution” paths. Control flow tests check that the internal program logic is working. This family selects a series of paths throughout the program, thereby examining the program control model. The techniques in this family vary as to the rigor with which they cover the code. Table below shows the techniques, giving a brief description of the coverage criterion followed, in ascending order of rigorousness.

TECHNIQUE	DESCRIPTION
<b>Sentence coverage</b>	The test cases are generated so that all the program sentences are executed at least once.
<b>Decision coverage (branch testing)</b>	The test cases are generated so that all the program decisions take the value true or false.
<b>Condition coverage</b>	The test cases are generated so that all the conditions (predicates) that form the logical expression of the decision take the value true or false.
<b>Decision/condition coverage</b>	Decision coverage is not always achieved with condition coverage. Here, the cases generated with condition coverage are supplemented to achieve decision coverage.
<b>Path coverage</b>	Test cases are generated to execute all program paths. This criterion is not workable in practice.

## SENTENCE COVERAGE

Statement coverage is the percentage of executable statements that have been exercised by a test case suite. The test cases are generated so that all the program sentences are executed at least once.

The measure reports on the percentage of executable statements of the code exercised by a set of test cases. Also known as *basic bloc coverage* or *segment coverage*.

The major advantage of this measure is that it can be applied directly to object code and does not require processing source code. Performance profilers commonly implement this measure. The major disadvantage of statement coverage is that it is insensitive to some control structures.

```
if (A) then  
    F1 ();  
    F2 ();
```

Test Case: A=True

Statement Coverage  
achieved

```
int* ptr = NULL;  
if (B)  
    ptr = &variable;  
*ptr = 10;
```

Test Case: B=True, Statement  
Coverage Achieved

Problem : if B is false the code will  
fail with a null pointer exception.


Statement coverage does not report whether loops reach their termination condition - only whether the loop body was executed.

Since do-while loops always execute at least once, statement coverage considers them the same rank as non-branching statements.

Statement coverage is completely insensitive to the logical decisions.

One argument in favor of statement coverage over other measures is that faults are evenly distributed through code; therefore the percentage of executable statements covered reflects the percentage of faults discovered.

Statement coverage requires in most cases very few test cases to achieve. Not acceptable to release code based on statement coverage alone.



## DECISION COVERAGE

Decision coverage (also known as **branch coverage**, **all-edges coverage**, **basis path coverage**, **decision-decision-path testing**) reports whether Boolean expressions in control structures are evaluated to both true and false values by the test cases. The test cases are generated so that all the program decisions are tested.

**Note:** In decision coverage we are not interested in all combinations of True/False values of all program predicates, just that we have exercised all the different values in our test cases. The entire Boolean expression is considered one true-or-false predicate. Additionally, this measure includes coverage of switch-statement cases, exception handlers, and interrupt handlers.

```
if (A)
    F1 ();
else
    F2 ();
if (B)
    F3 ();
else
    F4 ();
```

Test Cases for Decision Coverage:  
A=T, B=T  
A=F, B=F

```
if (A && (B || F1 ()))
    F2 ();
else
    F3 ();
```

Test Cases for Decision Coverage:  
A=T, B=T  
A=F

Problem: F1() never gets called.  
This problem occurs in languages with short circuiting Boolean operators.

This measure has the advantage of simplicity without the problems of statement coverage.

A disadvantage may produce gaps in tested code in programs written in languages that have short-circuit logic operators (C, C++, Java etc.).

Short Circuit Evaluation: Also known as minimal / McCarthy evaluation is the semantics of some Boolean operators in some programming languages in which the second argument is executed or evaluated only if the first does not suffice to determine the value of the expression.

1.  $A \ \&\& \ B \rightarrow$  if A is false, the overall value is false.
2.  $A \ || \ B \rightarrow$  if A is true, the overall value is true.



## CONDITION COVERAGE

The test cases are generated so that all the conditions (predicates) that form the logical expression of the decision take the value true or false. Condition coverage reports the true or false outcome of each Boolean sub-expression. Every statement in the program has been executed at least once, and every condition in each decision has taken all possible outcomes at least once.

Condition coverage measures the sub-expressions independently of each other. This measure is similar to decision coverage but has better sensitivity to the control flow.

```
if (A && B)
    F1 ();
else
    F2 ();
if (C)
    F3 ();
else
    F4 ();
```

Test Cases for Condition Coverage:  
A=T, B=T, C=F  
A=F, B=F, C=T

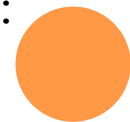
```
if (A && B)
    F1 ();
else
    F2 ();
if (C)
    F3 ();
else
    F4 ();
```

Test Cases for Condition Coverage:

A=T, B=F, C=F

A=F, B=T, C=T

Problem: Not decision coverage achieved



## DECISION/CONDITION COVERAGE

Every statement in the program has been executed at least once, every decision in the program has taken all possible outcomes at least once, and every condition in each decision has taken all possible outcomes at least once.

- To test if (A or B)

A:	T	F	F
B:	F	T	F

- To test if (A and B)

A:	F	T	T
B:	T	F	T

- To test if (A xor B)

A:	T	T	F
B:	T	F	T





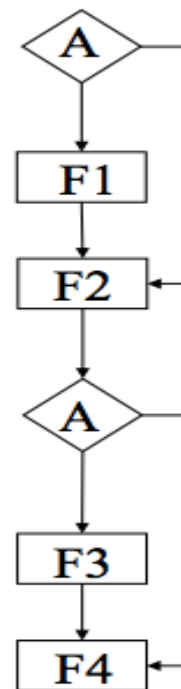
## PATH COVERAGE

Every complete path in the program has been executed at least once. In other words test cases are generated to execute all possible program paths. This criterion is not workable in practice.

This measure reports whether each of the possible paths in each function have been followed. A path is a unique sequence of branches from the function entry to the exit. This measure is also known as predicate coverage.

Since loops introduce an unbounded number of paths, this measure considers only a limited number of looping possibilities.

```
if (A)
    F1 ();
F2 ();
if (A)
    F3 ();
F4 ();
```



### Paths:

A-F1-F2-A-F3-F4  
A-F2-A-F3-F4  
A-F1-F2-A-F4  
A-F2-A-F4

Problem: only two  
are feasible

A=T  
A=F

# DATA FLOW TESTING

Data-flow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects. E.g., Pick enough paths to assure that:

1. Every data object has been initialized prior to its use.
2. All defined objects have been used at least once.

Data flow testing investigates how values of data that are associated with variables can affect the execution of programs.

Data flow tests verify whether the way a variable is defined, used and destroyed is handled correctly.

Data flow testing techniques also require knowledge of source code. The objective of this family is to select program paths to explore sequences of events related to the data state. Again, the techniques in this family vary as to the rigor with which they cover the code variable states.

# DATA OBJECT CATEGORIES

(d) Defined, Created, Initialized

(k) Killed, Undefined, Released

(u) Used:

(c) Used in a calculation

(p) Used in a predicate

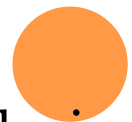
(d) **Defined Objects:** An object (e.g., variable) is defined when it:

- appears in a data declaration / is assigned a new value / is a file that has been opened / is dynamically allocated.

(u) **Used Objects:** An object is used when it is part of a computation or a predicate.

A variable is used for a computation (c) when it appears on the RHS (sometimes even the LHS in case of array indices) of an assignment statement.

A variable is used in a predicate (p) when it appears directly in that predicate.



## EXAMPLE: DEFINITION AND USES

	<i>Def</i>	<i>C-use</i>	<i>P-use</i>
1. read (x, y);	x, y		
2. z = x + 2;	z	x	
3. if (z < y)			z, y
4.     w = x + 1;	w	x	
else			
5.     y = y + 1;	y	y	
6. print (x, y, w, z);		x, y, w, z	



Table reflects the techniques, along with their associated coverage criterion.

TECHNIQUE	DESCRIPTION
<b>All-definitions</b>	Test cases are generated to cover each definition of each variable for at least one use of the variable.
<b>All-c-uses/some-p-uses</b>	Test cases are generated so that there is at least one path of each variable definition to each c-use of the variable. If there are variable definitions that are not covered, use p-uses.
<b>All-p-uses/some-c-uses</b>	Test cases are generated so that there is at least one path of each variable definition to each p-use of the variable. If there are variable definitions that are not covered, use c-uses.
<b>All-c-uses</b>	Test cases are generated so that there is at least one path of each variable definition to each c-use of the variable.
<b>All-p-uses</b>	Test cases are generated so that there is at least one path of each variable definition to each p-use of the variable.
<b>All-uses</b>	Test cases are generated so that there is at least one path of each variable definition to each use of the definition.
<b>All-du-paths</b>	Test cases are generated for all the possible paths of each definition of each variable to each use of the definition.
<b>All-dus</b>	Test cases are generated for all the possible executable paths of each definition of each variable to each use of the definition.

## MUTATION TESTING

Mutation testing is based on creating multiple modified versions of an original program. Each modified program known as *mutant* is same as original except for a small modification created by introducing simple fault.

In mutation testing, a large number of simple faults, called *mutations*, are introduced in an original program one at a time. The resulting changed versions of the test program are called mutants.

Mutation testing techniques are based on modeling typical programming faults by means of what are known as mutation operators (dependent on the programming language).

After the generation of adequate mutants, test cases are applied on the mutants. If the test cases are able to find all the mutants, then the same test cases are used to on the original program to find the faults.

The effectiveness of the test data set is measured by the percentage of mutants killed.

One of the major advantages of mutation testing is that it provides the tester with a target. It is a comprehensive way of testing a program.

A problem with the techniques that belong to this family is scalability. A mutation operator can generate several mutants per line of code. Therefore, there will be a sizeable number of mutants for long programs. Therefore, it is computationally expensive to run the test cases on each and every mutant.

It is also expensive to compile and execute every mutant. Because of this resource intensive process, mutation testing has not been practicable for software industries. The different techniques within this family aim to improve the scalability of standard (or strong) mutation to achieve greater efficiency.

TECHNIQUE	DESCRIPTION
<b>Strong (Standard) mutation</b>	Test cases are generated to cover all the mutants generated by applying all the mutation operators defined for the programming language in question.
<b>Selective ( C o n s t r a i n e d ) mutation</b>	Test cases are generated to cover all the mutants generated by applying some of the mutation operators defined for the programming language. This gives rise to selective mutation variants depending on the selected operators, like, for example, 2, 4 or 6 selective mutation (depending on the number of mutation operators not taken into account) or abs/or mutation, which only uses these two operators.
<b>Weak mutation</b>	Test cases are generated to cover a given percentage of mutants generated by applying all the mutation operators defined for the programming language in question. This gives rise to weak mutation variants, depending on the percentage covered, for example, randomly selected 10% mutation, ex-weak, st-weak, bbweak/ 1, or bb-weak/n.



## BLACK BOX TESTING

In Black box testing strategy we do not need any knowledge of internal design or code etc. for testing a system. *Black box testing* depends on the specification information as it covers specified behavior of the program, without any assumption about what happens in the system. That is why it is also referred to as *behavior based testing / input or output data driven testing*.

This type of testing gives us only the external view (behavior) of the software as it concentrates on what the software does and is not concerned about how it does it.

As a starting point, we often have a specification (requirements) and other requirement document. We ask a system a required question by means of a test case and get an answer in form of a result, and then that result is compared with expected result (test oracle) which determines whether the results are correct or erroneous.

This testing strategy takes into consideration a subset of available inputs and the expected outputs for each input.

It is not concerned with the inner processes involved in achieving a particular output or any other internal aspect of the software that may be involved in the transformation of an input into an output.

Testing techniques under this strategy are totally focused on testing requirements and functionality of the software under test. We need to have thorough knowledge of requirement specification of the system in order to implement the black box testing strategy. In addition we need to know how the system should behave in response to the particular input.

An example of a black-box system would be a search engine. You enter text that you want to search for in the search bar, press “Search” and results are returned to you. In such a case, you do not know or see the specific process that is being employed to obtain your search results, you simply see that you provide an input – a search term – and you receive an output – your search results.

# STATIC BLACK BOX TESTING

Testing the specification is static black-box testing. The specification is a document, not an executing program, so it's considered static.

It's also something that was created using data from many sources usability studies, focus groups, marketing input, and so on.

You don't necessarily need to know how or why that information was obtained or the details of the process used to obtain it, just that it's been boiled down into a product specification.

You can then take that document, perform static black-box testing, and carefully examine it for bugs.

There are usually two techniques used in static Black Box testing:

High Level Specification Testing

Low level Specification Testing




# FUNCTIONAL TESTING

Functional testing techniques are used to design test cases based on the functional requirements of the product. The goal of functional testing is to choose a set of inputs according to the specified functions of the program to test the program so that all functions and sub functions are tested at least once.

The success of the functional testing lies in the selection of appropriate data as per functionality and testing it against the functional specifications in order to check for normal and abnormal behavior of the system. As these techniques focus on the functionality, the test case design is defined by expected output specifications.

If the requirements are not expressed in way corresponding directly to these techniques we will have to put our efforts to do it during requirements documentation or during test design.



These testing techniques can be used at all stages of testing. The techniques can be used as a starting point in low-level tests, component testing and integration testing, where test cases can be designed based on the design and/or the requirements.

The techniques are also very useful in high-level tests like acceptance testing and system testing, where the test cases are designed from the requirements. A major advantage of functional testing is that the tests are geared to what the program or system is supposed to do, which is natural and understood by everyone. The main testing techniques employed in functional testing are shown in table:

TECHNIQUE	DESCRIPTION
<b>Equivalence Partitioning</b>	A test case is generated for each equivalence class found. The test case is selected at random from within the class.
<b>Boundary value analysis</b>	Several test cases are generated for each equivalence class, one that belongs to the inside of the class and as many as necessary to cover the limits (or boundaries) of the class.
<b>Cause Effect Graphing</b>	When the behavior of the unit under test is specified as cause and effect. Design test cases that validate this relationship.

# EQUIVALENCE PARTITIONING

Equivalence partitioning is a testing technique that divides the input and/or output data of a software unit into partitions of data from which test cases can be derived.

Usually it is the input data that is partitioned. However, depending on the software unit to be tested, output data can be partitioned as well.

The equivalence partitions are usually derived from the requirements specification for input attributes that influence the processing of the test object.

Test cases are designed to cover each partition at least once. Each partition shall contain a set or range of values, chosen such that all the values can reasonably be expected to be treated by the component in the same way (i.e. they may be considered 'equivalent')



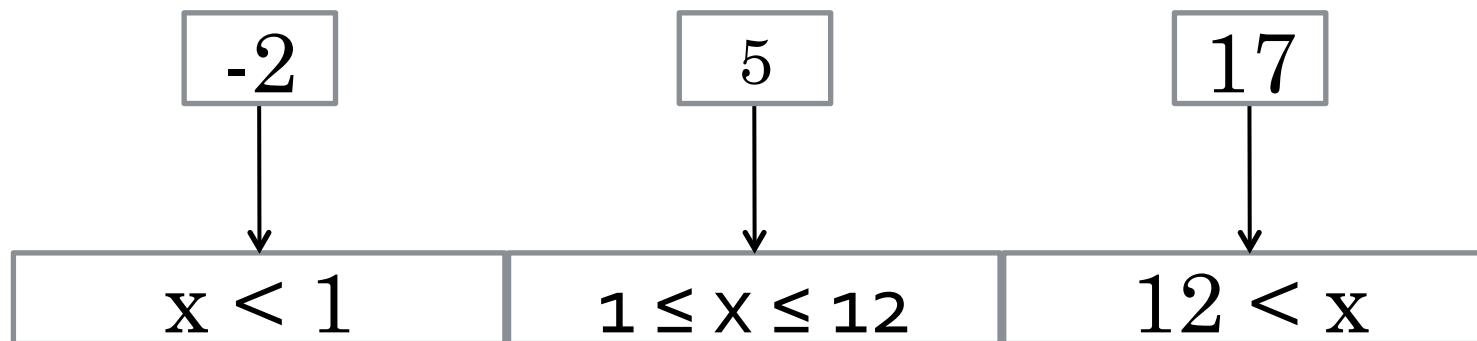
Example of a function which takes a parameter “month”.

The valid range for the month is 1 to 12, representing January to December. This valid range is called a partition.

However, in this case there are two further partitions of invalid ranges.



Test cases are chosen so that each partition would be tested.



**One weakness of boundary-value analysis and equivalence partitioning is that they do not explore *combinations* of input circumstances.**

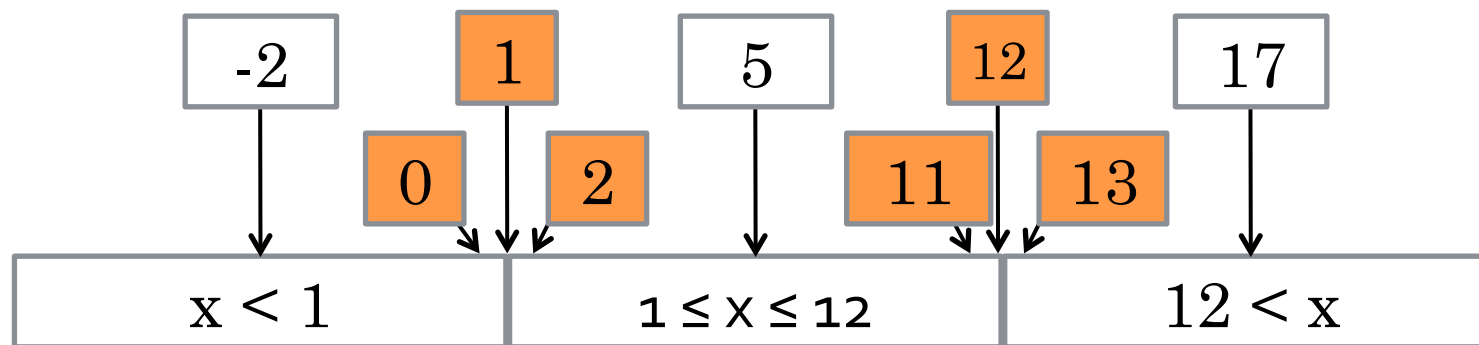


# BOUNDARY VALUE ANALYSIS

Equivalence partitioning is not a stand alone method to determine test cases. It is usually supplemented by ***boundary value analysis***.

***Boundary value analysis*** focuses on values on the edge of an equivalence partition or at the smallest value on either side of an edge.

For previous example, Test cases are supplemented with ***boundary values***.





## CAUSE EFFECT GRAPHING

Cause-effect graph is a graphical way of showing inputs or stimuli (causes) with their associated outputs (effects). The graph is a result of an analysis of requirements. Test cases can be designed from the cause-effect graph.

The technique is a semiformal way of expressing certain requirements, namely requirements that are based on Boolean expressions. The cause-effect graphing technique is used to design test cases for functions that depend on a combination of more input items.

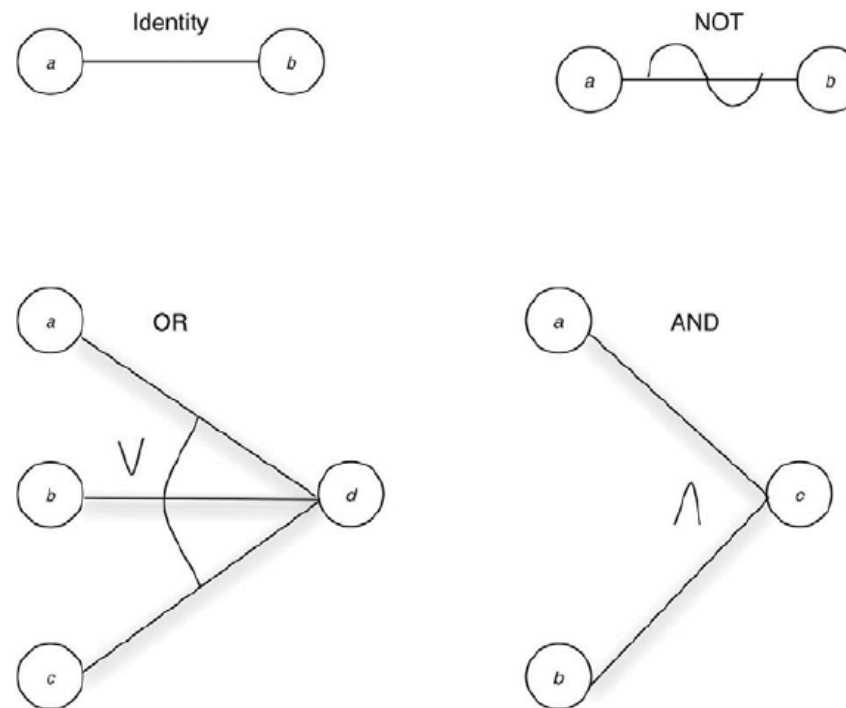
A cause-effect graph is a formal language into which a natural-language specification is translated.



The basic notation for the cause effect graph is shown in Figure below. Think of each node as having the value 0 or 1; 0 represents the “absent” state and 1 represents the “present” state.

- The *identity* function states that if  $a$  is 1,  $b$  is 1; else  $b$  is 0.
- The *not* function states that if  $a$  is 1,  $b$  is 0, else  $b$  is 1.
- The *or* function states that if  $a$  or  $b$  or  $c$  is 1,  $d$  is 1; else  $d$  is 0.
- The *and* function states that if both  $a$  and  $b$  are 1,  $c$  is 1; else  $c$  is 0.

The latter two functions (*or* and *and*) are allowed to have any number of inputs



To illustrate a small graph, consider the following specification:

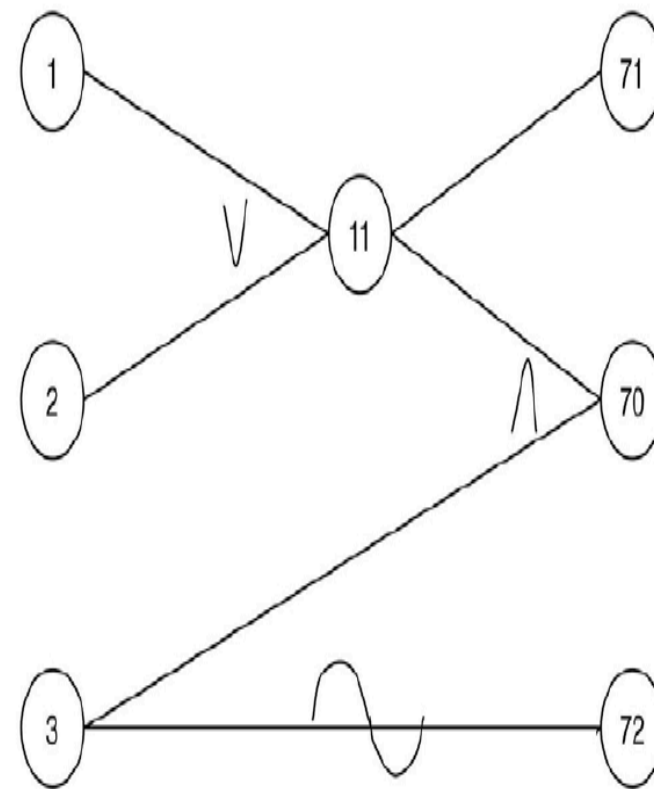
The character in column 1 must be an “A” or a “B.” The character in column 2 must be a digit. In this situation, the file update is made. If the first character is incorrect, message X12 is issued. If the second character is not a digit, message X13 is issued.

The causes are

- 1—character in column 1 is “A”
- 2—character in column 1 is “B”
- 3—character in column 2 is a digit

and the effects are

- 70—update made
- 71—message X12 is issued
- 72—message X13 is issued



11 is intermediate node

## RANDOM TESTING

In random testing, we select inputs from the possible input domain randomly and thus avoid test case designer bias. All inputs are treated as equally valuable. Test cases generated are purely random (not to be confused with statistical testing from the operational profile, where the random generation is biased towards reproducing field usage)

Random testing does not assume any knowledge of the system under test or its internal design. It sometimes even overlooks the requirement specification of the system.

Random testing can be very effective in identifying rarely occurring defects, but is not commonly used since it easily becomes a labor-intensive process.

Random inputs are easier to generate, but less likely to cover parts of the specification or the code. One might think that picking random samples might be a good idea. However, it is not.

For one, we do not care for bias – we specifically want to search where it matters most. Second, random testing is unlikely to uncover specific defects. This technique is insufficient for validating complex, safety-critical or mission-critical software. Random testing consists of following testing techniques:

TECHNIQUE	DESCRIPTION
Pure random	Test cases are generated at random, and generation stops when there appear to be enough.
Guided by the number of cases	Test cases are generated at random, and generation stops when a given number of cases has been reached.
Error guessing	Test cases are generated guided by the subject's knowledge of what typical errors are usually made when programming. It stops when they all appear to have been covered.

# BLACK BOX TESTING AND COTS

- COTS – Commercial-Off-The-Shelf
- COTS describes ready-made products that can easily be obtained.
- It exists a priori
- Used without source code modification
- It is available to the general public
- It can be bought (or leased or licensed).
- Supported and evolved by vendor



## **Unique Challenges of Testing COTS-based Applications**

**Challenge #1 - COTS is a Black Box, no access to source code**

**Challenge #2 - Lack of Functional and Technical Requirements**


**Challenge #3 - The Level of Quality is Unknown**

To understand the behavior of a component, various inputs are executed and outputs are analyzed.

To catch all types of errors all possible combinations of input values should be executed.

To make testing feasible, test cases are selected randomly from test case space.

As source code is unavailable, black box is often the only way to test the COTS; however, the challenges listed ahead can decrease the effectiveness of that process also.



IN CASE OF ANY PROBLEM / CLARIFICATION

PLEASE MAIL ME AT:

SUF[DOT]CS[AT]UOK[DOT]EDU[DOT]IN

