

Software Testing & Audit

Unit-2

Jayash Kumar Sharma
Department of Computer Science & Engineering
Anand Engineering College
Mob: +91-9639325975
Email: jayash.sharma@gmail.com

Functional / Black Box Testing

- Functional testing techniques attempt to design those test cases which have a higher probability of making a software fail.
- The test cases are designed on the basis of user requirements without considering the internal structure of the program.
- Black box knowledge is sufficient to design a good number of test cases.
- In functional testing techniques, execution of a program is essential →
Validation

Functional / Black Box Testing

- In these techniques both valid and invalid inputs are chosen to see the observed behaviour of the program.
- These techniques can be used at all levels of software testing like unit, integration, system and acceptance testing.
- Help the tester to design efficient and effective test cases to find faults in the software.

Functional / Black Box Testing - Methods

- Boundary Value Analysis
- Equivalence Class Testing
- Decision Table based Testing
- Cause Effect Graphing Technique

Boundary Value Analysis

- Simple but popular functional testing technique approach.
- We concentrate on input values and design test cases with input values that are on or close to boundary values (**weak & Critical Point in Program**).
- Experience has shown that such test cases have a higher probability of detecting a fault in the software.

Boundary Value Analysis – Example-1

Consider a program 'Square'

Input → 'x' (Range is from 1 to 100)

Output → prints the square of 'x'

Exhaustive Approach:

- Execute this program 100 times to check every input value.
- Give all values from 1 to 100 one by one to the program and see the observed behaviour.

Boundary Value Analysis – Example-1

Boundary Value Analysis Approach:

- Select values on or close to boundaries.
- Input values may have one of the following:
 - Minimum value (1)
 - Just above minimum value (2)
 - Maximum value (100)
 - Just below maximum value (99)
 - Nominal (Average) value (50)

Boundary Value Analysis – Example-1

The number of inputs selected by this technique is $4n + 1$ where 'n' is the number of inputs.

Test Case	Input (x)	Output (Square)
1.	1	1
2.	2	4
3.	100	10000
4.	99	9801
5.	50	2500

Boundary Value Analysis – Example-2

Consider a program 'Addition'

$$100 \leq x \leq 300$$

Input \rightarrow x & y

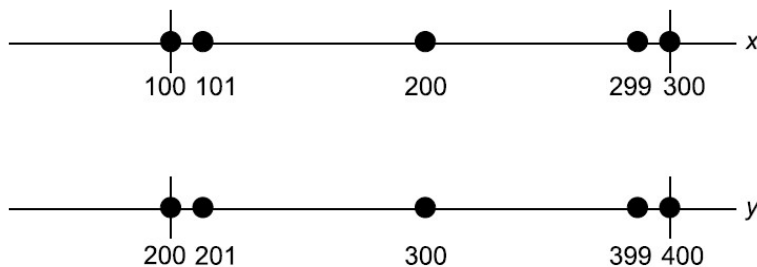
$$200 \leq y \leq 400$$

Output \rightarrow prints the addition of x and y

Write test cases for this program using boundary Value Analysis Approach.

Boundary Value Analysis – Example-2

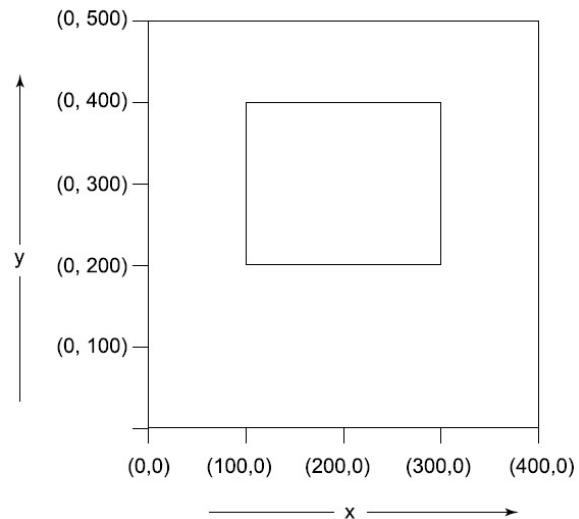
Selected values for x and y :



Boundary Value Analysis – Example-2

Input Domain of the Program :

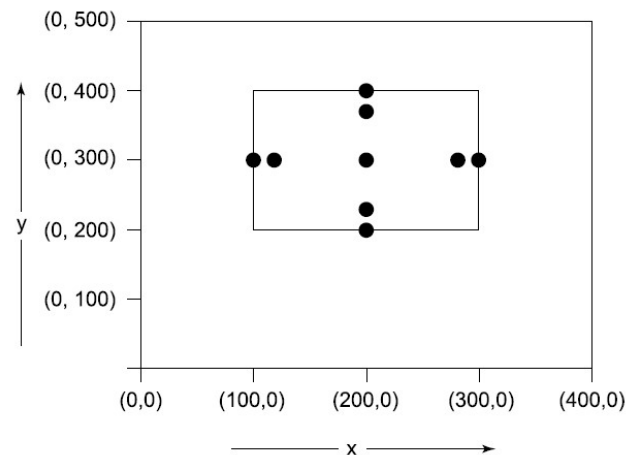
Any point within the inner rectangle is a legitimate input to the program.



Boundary Value Analysis – Example-2

we select one input value:

- On boundary (minimum)
- just above boundary (minimum +)
- Just below boundary (maximum -)
- On boundary (maximum)
- Nominal (average) and
- other n-1 input values as nominal values.



Boundary Value Analysis – Example-2

Test Case	Input (x)	Input (y)	Output (Addition)
1.	100	300	400
2.	101	300	401
3.	200	300	500
4.	299	300	599
5.	300	300	600
6.	200	200	400
7.	200	201	401
8.	200	399	599
9.	200	400	600
Total $4n+1$ test cases $\rightarrow 4 * 2 + 1 \rightarrow 9$ test cases			

Boundary Value Analysis – Example-3

Consider a program for the determination of the largest amongst three numbers.

Input \rightarrow Triple of positive integers (x, y and z)
values are from interval [1, 300].

Output \rightarrow Print largest number

Design the boundary value test cases.

Boundary Value Analysis – Example-3

Test Case	X	Y	Z	Output
1.	1	150	150	150
2.	2	150	150	150
3.	150	150	150	150
4.	299	150	150	299
5.	300	150	150	300
6.	150	1	150	150
7.	150	2	150	150

Boundary Value Analysis – Example-3

Test Case	X	Y	Z	Output
8.	150	299	150	299
9.	150	300	150	300
10.	150	150	1	150
11.	150	150	2	150
12.	150	150	299	299
13.	150	150	300	300

Boundary Value Analysis – Example-4

Consider a program for the **determination of division** of a student based on the marks in three subjects.

Input: Triple of positive integers (say mark1, mark2, and mark3) and values are from interval [0, 100].

Output: Division based on $\text{Average} = (\text{mark1} + \text{mark2} + \text{mark3}) / 3$

75 – 100	→	First Division with distinction
60 – 74	→	First division
50 – 59	→	Second division
40 – 49	→	Third division
0 – 39	→	Fail

Boundary Value Analysis – Example-4

Test Case	Marks-1	Marks-2	Marks-3	Output
1.	0	50	50	Fail
2.	1	50	50	Fail
3.	50	50	50	Second Division
4.	99	50	50	First Division
5.	100	50	50	First Division
6.	50	0	50	Fail
7.	50	1	50	Fail

Boundary Value Analysis – Example-4

Test Case	Marks-1	Marks-2	Marks-3	Output
8.	50	99	50	First Division
9.	50	100	50	First Division
10	50	50	0	Fail
11	50	50	1	Fail
12	50	50	99	First Division
13	50	50	100	First Division

Robustness Testing

- Extension of Boundary Value Analysis.
- Here, we also select invalid values and see the responses of the program.
- Invalid values are also important to check the behavior of the program.
- Two additional states are added
 - Just below minimum value (minimum value -)
 - Just above maximum value (maximum value +).

Robustness Testing

- The total test cases in robustness testing are **$6n + 1$** , where 'n' is the number of input values.
- All input values may have one of the following values:
 - Minimum value
 - Just above minimum value
 - Just below minimum value
 - Just above maximum value
 - Just below maximum value
 - Maximum value
 - Nominal (Average) value

Robustness Testing – Example-1

Consider a program 'Addition'

Input → x & y

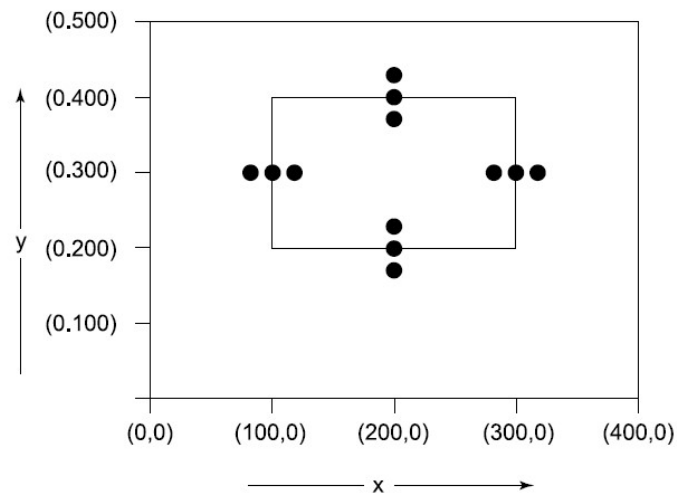
$$100 \leq x \leq 300$$

$$200 \leq y \leq 400$$

Output → prints the addition of x and y

Write test cases for this program using **Robustness Testing**.

Robustness Testing – Example-1



Robustness Testing – Example-1

Test Case	Input (x)	Input (y)	Output (Addition)
1.	100	300	400
2.	101	300	401
3.	200	300	500
4.	299	300	599
5.	300	300	600
6.	200	200	400
7.	200	201	401
8.	200	399	599
9.	200	400	600

Robustness Testing – Example-1

Test Case	Input (x)	Input (y)	Output (Addition)
10.	99	300	Invalid Input
11	301	300	Invalid Input
12.	200	199	Invalid Input
13.	200	401	Invalid Input
Total $6n+1$ test cases $\rightarrow 6 * 2 + 1 \rightarrow 13$ test cases			

Worst Case Testing

- Special form of boundary value analysis where we don't consider the 'single fault' assumption theory of reliability.
- Now, failures are also due to occurrence of more than one fault simultaneously.
- Single Fault Assumption Theory of Reliability : Failures are rarely the result of the simultaneous occurrence of two (or more) faults. Normally, one fault is responsible for one failure.
- All previous methods were based on Single Fault Assumption Theory of Reliability.

Worst Case Testing

- In this method, all input values may have one of the following:
 - Minimum value
 - Just above minimum value
 - Just below maximum value
 - Maximum value
 - Nominal (Average) value

Worst Case Testing

- The restriction of one input value at any of the above mentioned values and other input values must be at nominal is not valid in worst-case testing.
- This approach increases the number of test cases from $4n + 1$ test cases to 5^n test cases, where 'n' is the number of input values.
- This requires more effort and is recommended in situations where failure of the program is extremely critical and costly

Worst Case Testing – Example-1

Consider a program 'Addition'

Input → x & y

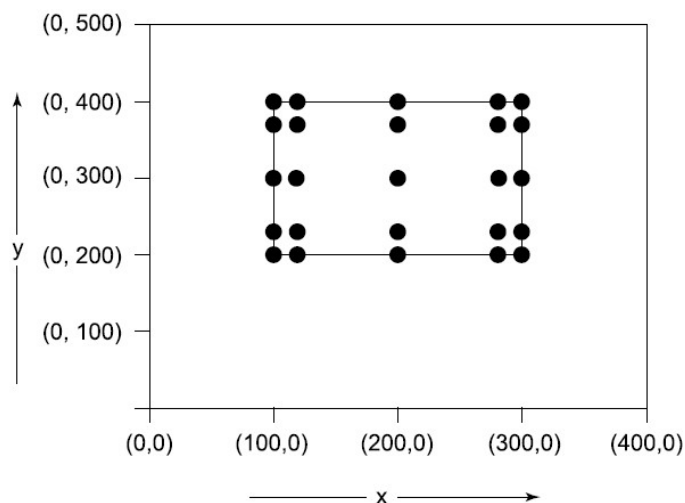
$$100 \leq x \leq 300$$

$$200 \leq y \leq 400$$

Output → prints the addition of x and y

Write test cases for this program using **Worst Case Testing**.

Worst Case Testing – Example-1



Total test cases = 5^n

$$n = 2$$

$$\text{Total test case} = 5^2 = 25$$

Worst Case Testing – Example-1

Test Case	x	y	Expected Output
1.	100	200	300
2.	100	201	301
3.	100	300	400
4.	100	399	499
5.	100	400	500
6.	101	200	301
7.	101	201	302
8.	101	300	401
9.	101	399	500
10.	101	400	501
11.	200	200	400
12.	200	201	401
13.	200	300	500
14.	200	399	599

Worst Case Testing – Example-1

Test Case	x	y	Expected Output
15.	200	400	600
16.	299	200	499
17.	299	201	500
18.	299	300	599
19.	299	399	698
20.	299	400	699
21.	300	200	500
22.	300	201	501
23.	300	300	600
24.	300	399	699
25.	300	400	700

Robust Worst Case Testing

- Combination of Robustness & Worst-case testing.
- In robustness testing, two more states are added.
 - Just below minimum value (minimum value-)
 - Just above maximum value (maximum value+). We also give invalid inputs and observe
- Total of 7^n test cases are generated.
- Largest set of test cases and requires the maximum effort to generate such test cases.

Robust Worst Case Testing

- There are seven states :

- Minimum -
- Minimum
- Minimum +
- Nominal
- Maximum -
- Maximum
- Maximum +

Robust Worst Case Testing – Example-1

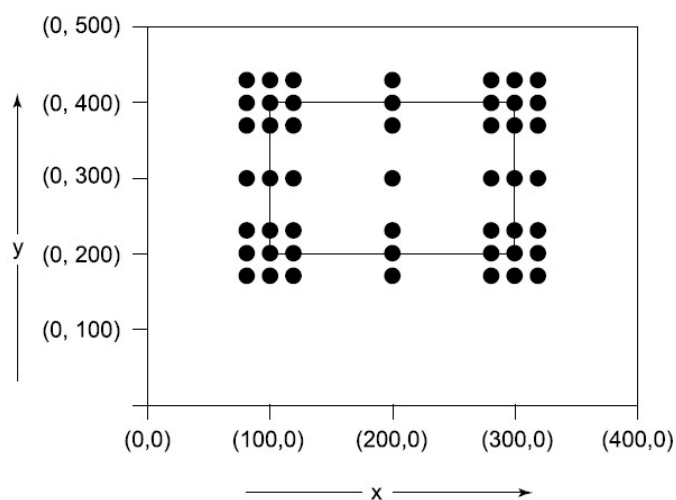
Consider a program 'Addition'

Input \rightarrow x & y

Output \rightarrow prints the addition of x and y

Write test cases for this program using **Robust Worst Case Testing**.

Robust Worst Case Testing – Example-1



Total test cases = 7^n

$n = 2$

Total test case = $7^2 = 49$

Applicability of BVA

- Boundary value analysis is a simple and effective when used correctly.
- Input values should be independent which restricts its applicability in many programs.
- This technique does not make sense for Boolean variables where input values are TRUE and FALSE only. (no choice is available for nominal values, just above boundary values, just below boundary values, etc.)

Applicability of BVA

- This technique can reduce the number of test cases
- Suited to programs in which input values are within ranges or within sets.
- This testing is equally applicable at the unit, integration, system and acceptance test levels.

Equivalence Partitioning / Class Testing

- A large number of test cases can be generated for any program.
- It is neither feasible nor desirable to execute all such test cases.
- We want to select a few test cases and still wish to achieve a reasonable level of coverage.
- Many test cases do not test any new thing and they just execute the same lines of source code again and again.

Example: Traditional Boundary Value analysis method

Equivalence Partitioning / Class Testing

- Divide input domain into various categories with some relationship and expect that every test case from a category exhibits the same behaviour.
- It is assumed that if one representative test case works correctly, others may also give the same results.
- This assumption allows us to select exactly one test case from each category.
- Each category is called an **Equivalence Class** and this type of testing is known as **Equivalence Class Testing**.

Equivalence Class Testing – Class Creation

- The entire input domain can be divided into at least two equivalence classes:
 - ❑ One containing all valid inputs
 - ❑ Other containing all invalid inputs.
- Each equivalence class can further be sub-divided into equivalence classes on which the program is required to behave differently.

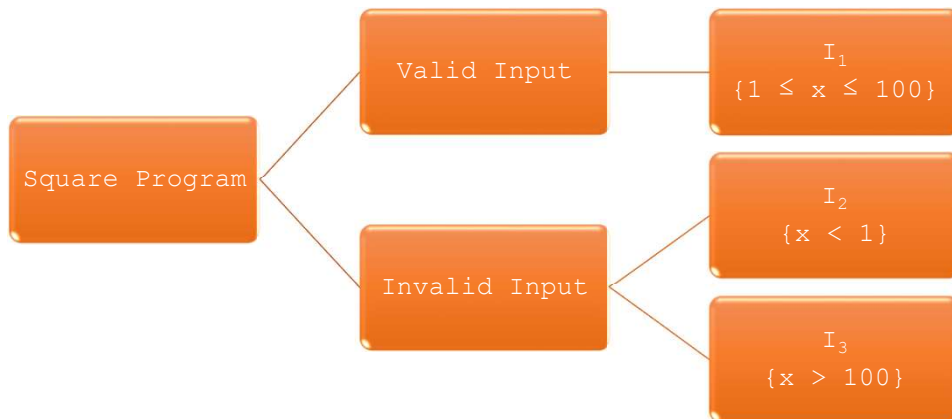
Equivalence Class Testing – Example-1

Consider a program 'Square'

Input → 'x' (Range is from 1 to 100)
Output → prints the square of 'x'

- Identify Equivalence Classes and write test cases accordingly.

Equivalence Class Testing - Class Creation



Equivalence Class Testing - Example-1

Test Case	Input (x)	Output (Square)
1.	50	2500
2.	0	Invalid Input
3.	101	Invalid Input

Equivalence Class Testing – Example-2

Consider a program 'Addition'

$$100 \leq x \leq 300$$

Input \rightarrow X & Y

$$200 \leq y \leq 400$$

Output \rightarrow prints the addition of X and Y

Write test cases for this program using Equivalence Class Testing.

Equivalence Class Testing – Example-2

Class Name	X	Y	Remarks
I ₁	$100 \leq X \leq 300$	$200 \leq Y \leq 400$	X-Valid, Y-Valid
I ₂	$100 \leq X \leq 300$	$Y < 200$	X-Valid, Y-Invalid
I ₃	$100 \leq X \leq 300$	$Y > 400$	X-Valid, Y-Invalid
I ₄	$X < 100$	$200 \leq Y \leq 400$	X-Invalid, Y-Valid
I ₅	$X > 300$	$200 \leq Y \leq 400$	X-Invalid, Y-Valid
I ₆	$X < 100$	$Y < 200$	X-Invalid, Y-Invalid
I ₇	$X < 100$	$Y > 400$	X-Invalid, Y-Invalid
I ₈	$X > 300$	$Y < 200$	X-Invalid, Y-Invalid
I ₉	$X > 300$	$Y > 400$	X-Invalid, Y-Invalid

Equivalence Class Testing – Example-2

Test Case	X	Y	Output
1.	200	300	500
2.	200	199	Invalid Input
3.	200	401	Invalid Input
4.	99	300	Invalid Input
5.	301	300	Invalid Input
6.	99	199	Invalid Input
7.	99	401	Invalid Input
8.	301	199	Invalid Input
9.	301	401	Invalid Input

Decision Table based Testing

- The techniques of equivalence partitioning and boundary value analysis are often applied to specific situations or inputs.
- If different combinations of inputs result in different actions being taken, this can be more difficult to show using equivalence partitioning and boundary value analysis.
- In this case, Decision table based testing is useful.

Decision Table based Testing

- Decision table gives a pictorial view of various combinations of input conditions.
- The decision table provides a set of conditions and their corresponding actions.
- There are four portions of the decision table:

Decision Table based Testing

Table 2.30. Decision table												
	Stubs Entries											
Condition	<table border="1"> <tr> <td>c₁</td><td rowspan="3"></td></tr> <tr> <td>c₂</td></tr> <tr> <td>c₃</td></tr> <tr> <td>Action</td><td> <table border="1"> <tr> <td>a₁</td><td rowspan="4"></td></tr> <tr> <td>a₂</td></tr> <tr> <td>a₃</td></tr> <tr> <td>a₄</td></tr> </table> </td></tr> </table>	c ₁		c ₂	c ₃	Action	<table border="1"> <tr> <td>a₁</td><td rowspan="4"></td></tr> <tr> <td>a₂</td></tr> <tr> <td>a₃</td></tr> <tr> <td>a₄</td></tr> </table>	a ₁		a ₂	a ₃	a ₄
c ₁												
c ₂												
c ₃												
Action	<table border="1"> <tr> <td>a₁</td><td rowspan="4"></td></tr> <tr> <td>a₂</td></tr> <tr> <td>a₃</td></tr> <tr> <td>a₄</td></tr> </table>	a ₁		a ₂	a ₃	a ₄						
a ₁												
a ₂												
a ₃												
a ₄												

1. Condition Stub
2. Condition Entries
3. Action Stub
4. Action Entries

Decision Table based Testing - Parts

Condition Stubs:

- All the conditions are represented in this upper left section of the decision table.
- These conditions are used to determine a particular action or set of actions.

Action Stubs:

- All possible actions are listed in this lower left portion of the decision table.

Decision Table based Testing - Parts

Condition Entries:

- In the portion, we have a number of columns and each column represents a rule.
- Values entered in this upper right portion of the table are known as inputs.

Decision Table based Testing - Parts

Action Entries:

- Each entry in the action entries portion has some associated action or set of actions in this lower right portion of the table.
- These values are known as outputs and are dependent upon the functionality of the program.

Decision Table based Testing - Parts

- Decision table testing technique is used to design a complete set of test cases without using the internal structure of the program.
- Every column is associated with a rule and generates a test case.

Table 2.31. Typical structure of a decision table				
Stubs	R ₁	R ₂	R ₃	R ₄
c ₁	F	T	T	T
c ₂	-	F	T	T
c ₃	-	-	F	T
a ₁	X	X		X
a ₂			X	
a ₃	X			

Decision Table based Testing - Parts

- The decision tables which use only binary conditions (True / False) as input values are known as **Limited Entry Decision Tables**.
- The decision tables which use multiple conditions where a condition may have many possibilities instead of only 'true' and 'false' are known as **Extended Entry Decision Tables**.

Decision Table based Testing – How to use

- Identify a suitable function or subsystem which reacts according to a combination of inputs or events.
- The system should not contain too many inputs otherwise the number of combinations will become unmanageable.
- It is better to deal with large numbers of conditions by dividing them into subsets and dealing with the subsets one at a time.
- Once you have identified the aspects that need to be combined, put them into a table listing all the combinations of True and False for each of the aspects.

Decision Table based Testing – Example

- Let us consider an example of a loan application, where you can enter the amount of the monthly repayment or the number of years you want to take to pay it back (the term of the loan). If you enter both, the system will make a compromise between the two if they conflict. The two conditions are the loan amount and the term, so we put them in a table

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
<i>Repayment amount has been entered:</i>				
<i>Term of loan has been Entered:</i>				

Decision Table based Testing – Example

- Identify all of the combinations of True and False.
- With two conditions, each of which can be True or False, we will have four combinations (two to the power of the number of things to be combined).

Conditions	Rule 1	Rule 2	Rule 3	Rule
4				
<i>Repayment amount has been entered:</i>	T	T	F	F
<i>Term of loan has been entered:</i>	T	F	T	F

Decision Table based Testing – Example

- Identify the correct outcome for each combination.
- Each combination is sometimes referred to as a rule.

Conditions	Rule 1	Rule 2	Rule 3	Rule
4				
<i>Repayment amount has been entered:</i>	T	T	F	F
<i>Term of loan has been entered:</i>	T	F	T	F
Actions/Outcomes				
<i>Process loan amount:</i>	Y	Y		
<i>Process term:</i>	Y		Y	


 ?

Decision Table based Testing – Example

- We could assume that this combination should result in an error message, so we need to add another action.

Conditions	Rule 1	Rule 2	Rule 3	Rule
4				
<i>Repayment amount has been entered:</i>	T	T	F	F
<i>Term of loan has been entered:</i>	T	F	T	F
Actions/Outcomes				
<i>Process loan amount:</i>	Y	Y		
<i>Process term:</i>	Y		Y	
<i>Error message:</i>				Y

Decision Table based Testing – Example

- we make slight change in this example, so that the customer is not allowed to enter both repayment and term.
- Now the outcome of our table will change, because there should also be an error message if both are entered.

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
<i>Repayment amount has been entered:</i>	T	T	F	F
<i>Term of loan has been entered:</i>	T	F	T	F
Actions/Outcomes				
<i>Process loan amount:</i>		Y		
<i>Process term:</i>			Y	
<i>Error message:</i>	Y			Y

Mutually Exclusive

Decision Table based Testing – Example

Mutually Exclusive

- Our actions are mutually exclusive – only one action occurs for each combination of conditions.

Don't Care Condition (How to Simplify Decision Table)

- The 'do not care' conditions are represented by the '-' sign.
- A 'do not care' condition has no effect on the output.

Decision Table based Testing – Example

Video Tutorial

[Decision Table Testing tutorial with examples - YouTube \[720p\].mp4](#)

Decision Table based Testing - Assignment

Assignment-4 (Credit Card Example):

Let's take another example. If you are a new customer and you want to open a credit card account then there are three conditions first you will get a 15% discount on all your purchases today, second if you are an existing customer and you hold a loyalty card, you get a 10% discount and third if you have a coupon, you can get 20% off today (but it can't be used with the 'new customer' discount). Discount amounts are added, if applicable.

Prepare Decision Table for given requirement.

Decision Table based Testing - Assignment

Assignment-4 (Credit Card Example - Solution):

Conditions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
New customer (15%)	T	T	T	T	F	F	F	F
Loyalty card (10%)	T	T	F	F	T	T	F	F
Coupon (20%)	T	F	T	F	T	F	T	F
Actions								
Discount (%)	X	X	20	15	30	10	20	0

Decision Table based Testing – Applicability

- Decision tables are used where an output is dependent on many conditions and a large number of decisions are required to be taken.
- Decision tables are also known as [Cause-Effect Tables](#).
- May incorporate complex business rules and use them to design test cases.
- Every column of the decision table generates a test case.

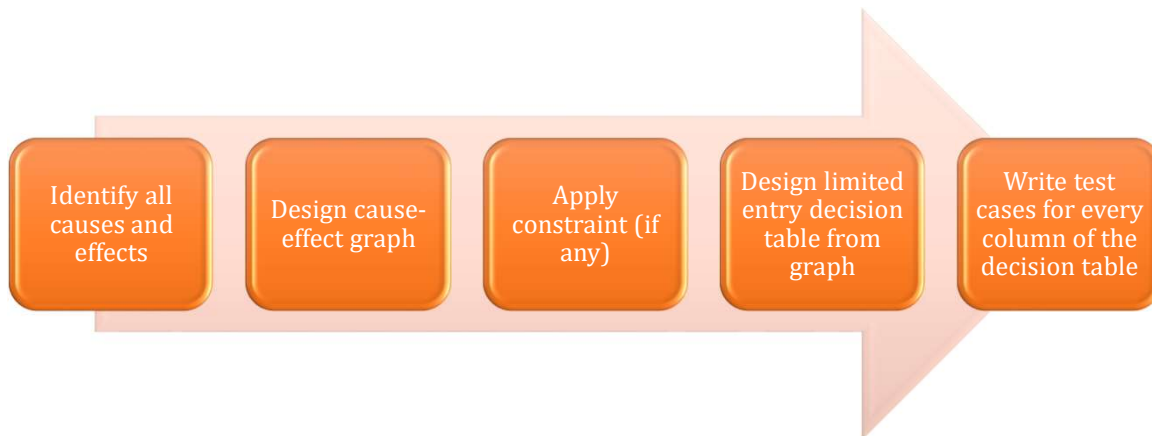
Decision Table based Testing – Applicability

- As the size of the program increases, handling of decision tables becomes difficult and cumbersome.
- Can be applied easily at Unit Level only.
- System testing and integration testing may not find its effective applications.

Cause-Effect Graphing Technique

- A popular technique for small programs
- Two new terms are used in this technique “causes” and “effects”, which are nothing but inputs and outputs respectively.

Cause-Effect Graphing Technique - Steps



Cause-Effect Graphing Technique

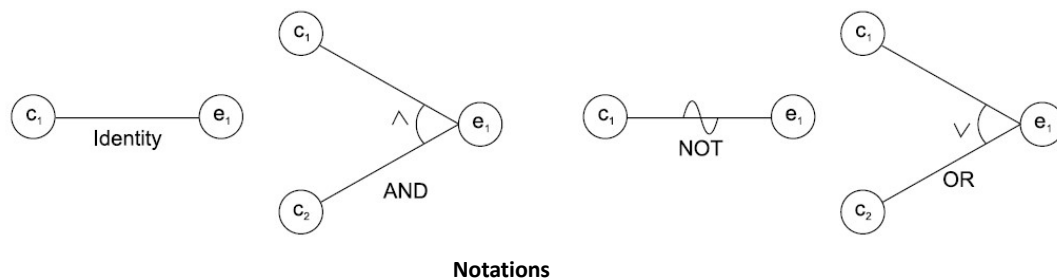
Identify causes and effects

- SRS document is used for the identification of causes and effects.
- Causes (inputs to the program) and effects (outputs of the program) can easily be identified after reading the SRS document.
- A list is prepared for all causes and effects.

Cause-Effect Graphing Technique

Design Cause-Effect Graph

- The relationship amongst causes and effects are established using cause-effect graph.



Cause-Effect Graphing Technique

Design Cause-Effect Graph

- Each node represents either TRUE (present) or FALSE (absent) state and may be assigned 1 and 0 value respectively.
- The purpose of four functions is given as:
 - Identity:** If c_1 is 1, then e_1 is 1; else e_1 is 0.
 - NOT:** If c_1 is 1, then e_1 is 0; else e_1 is 1.
 - AND:** If both c_1 and c_2 are 1, then e_1 is 1; else e_1 is 0.
 - OR:** If either c_1 or c_2 is 1, then e_1 is 1; else e_1 is 0.

Cause-Effect Graphing Technique

Use Constraints (If any)

- There may be a number of causes (inputs) in any program and relationships amongst the causes can be explored.
- This process may lead to some impossible combinations of causes.
- Such impossible combinations or situations are represented by constraint symbols.

Cause-Effect Graphing Technique

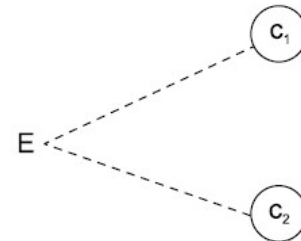
Use Constraints (If any)

- There are five constraint symbols:
 1. Exclusive
 2. Inclusive
 3. One and Only One
 4. Requires
 5. Mask

Cause-Effect Graphing Technique

Use Constraints (If any) – Exclusive Constraint

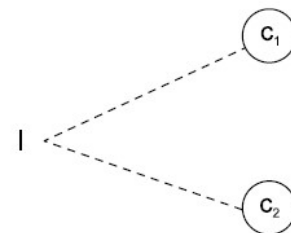
- The Exclusive (E) constraint states that at most one of c_1 or c_2 can be 1.
- c_1 or c_2 cannot be 1 simultaneously.
- However, both c_1 and c_2 can be 0 simultaneously.



Cause-Effect Graphing Technique

Use Constraints (If any) – Inclusive Constraint

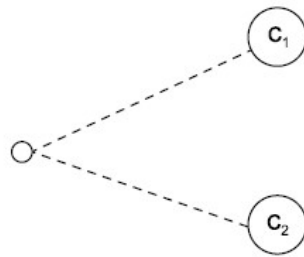
- The Inclusive (I) constraint states that at least one of c_1 or c_2 must always be 1.
- Hence, both cannot be 0 simultaneously. However, both can be 1.



Cause-Effect Graphing Technique

Use Constraints (If any) – One and Only One Constraint

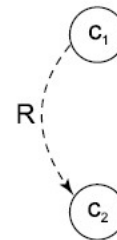
- The one and only one (O) constraint states that one and only one of c_1 and c_2 must be 1.



Cause-Effect Graphing Technique

Use Constraints (If any) – Requires Constraint

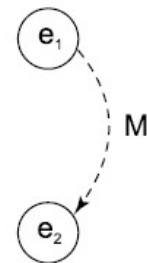
- The requires (R) constraint states that for c_1 to be 1, c_2 must be 1;
- It is impossible for c_1 to be 1 if c_2 is 0.



Cause-Effect Graphing Technique

Use Constraints (If any) – Mask Constraint

- This constraint is applicable at the effect side of the cause-effect graph.
- This states that if effect e1 is 1, effect e2 is forced to be 0.



Cause-Effect Graphing Technique – Example-1

Proposed system maintains the record of marital status and number of children of a citizen. The value of marital status must be 'U' or 'M'. The value of the number of children must be digit or null in case a citizen is unmarried. If the information entered by the user is correct then an update is made. If the value of marital status of the citizen is incorrect, then the error message 1 is issued. Similarly, if the value of number of children is incorrect, then the error message 2 is issued.

Cause-Effect Graphing Technique – Example-1

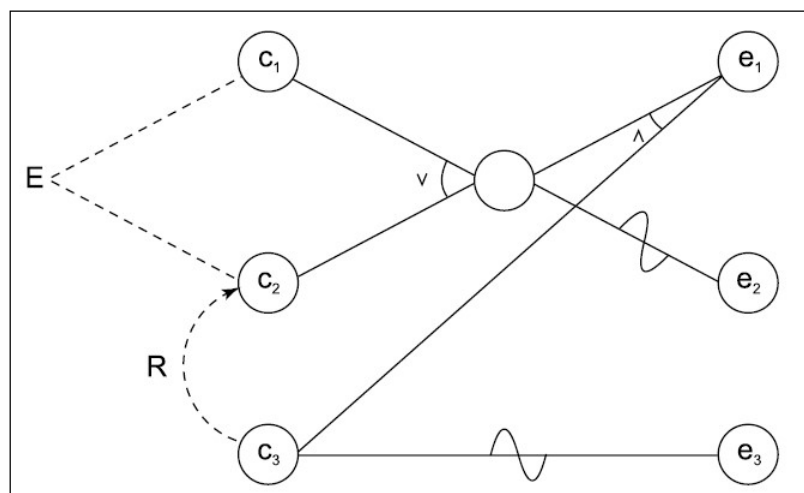
The causes are:

- c1: marital status is 'U'
- c2: marital status is 'M'
- c3: number of children is a digit

The effects are:

- e1: Update made
- e2: error message 1 is issued
- e3: error message 2 is issued

Cause-Effect Graphing Technique – Example-1



Cause-Effect Graphing Technique – Example-2

A tourist of age greater than 21 years and having a clean driving record is supplied a rental car. A premium amount is also charged if the tourist is on business, otherwise it is not charged. If the tourist is less than 21 year old, or does not have a clean driving record, the system will display the following message: *"Car cannot be supplied"*

Draw the cause-effect graph and generate test cases.

Cause-Effect Graphing Technique – Example-2

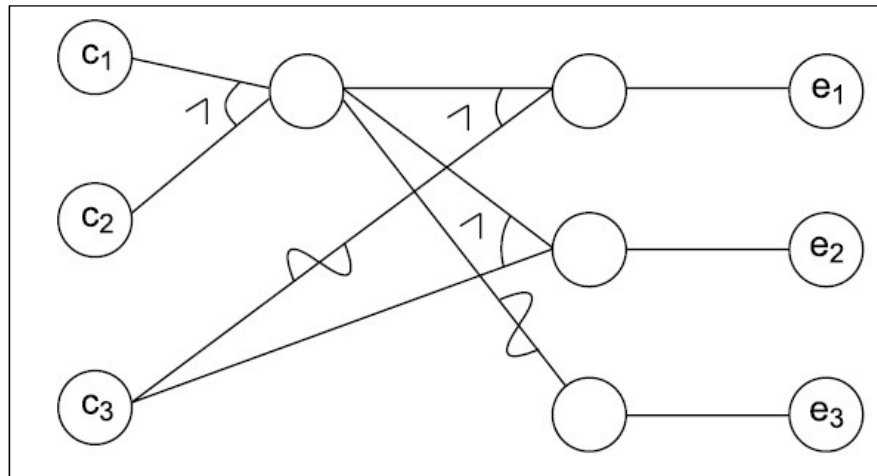
The causes are:

- c1: Age is greater than 21
- c2: Driving record is clean
- c3: Tourist on business

The effects are:

- e1: Supply car without premium amount
- e2: Supply car with premium amount
- e3: *Car cannot be supplied*

Cause-Effect Graphing Technique - Example-2



Cause-Effect Graphing Technique - Example-2

Conditions	R1	R2	R3	R4	R5	R6	R7	R8
c1	F	F	F	F	T	T	T	T
c2	F	F	T	T	F	F	T	T
c3	F	T	F	T	F	T	F	T
Action								
e1							√	
e2								√
e3	√	√	√	√	√	√		

Cause-Effect Graphing Technique – Example-2

Conditions	R1	R3	R4	R5	R6	R7	R8
c1	F	F	F	T	T	T	T
c2	F	T	T	F	F	T	T
c3	-	F	T	F	T	F	T
Action							
e1						√	
e2							√
e3	√	√	√	√	√		

Cause-Effect Graphing Technique – Example-2

Conditions	R1	R3	R5	R6	R7	R8
c1	F	F	T	T	T	T
c2	F	T	F	F	T	T
c3	-	-	F	T	F	T
Action						
e1					√	
e2						√
e3	√	√	√	√		

Cause-Effect Graphing Technique – Example-2

Conditions	R1	R3	R5	R7	R8
c1	F	F	T	T	T
c2	F	T	F	T	T
c3	-	-	-	F	T
Action					
e1				√	
e2					√
e3	√	√	√		

Cause-Effect Graphing Technique – Example-2

Conditions	R1	R5	R7	R8
c1	F	T	T	T
c2	-	F	T	T
c3	-	-	F	T
Action				
e1			√	
e2				√
e3	√	√		

Rename rules

Cause-Effect Graphing Technique – Example-2

Conditions	R1	R2	R3	R4
c1	F	T	T	T
c2	-	F	T	T
c3	-	-	F	T
Action				
e1			√	
e2				√
e3	√	√		

Cause-Effect Graphing Technique – Example-2

Test Case	Age	Driving Record Clean?	On Business?	Output
1.	20	Yes	Yes	Car can not be supplied
2.	26	No	Yes	Car can not be supplied
3.	50	Yes	No	Supply a rental car without premium charge
4.	50	Yes	Yes	Supply a rental car with premium charge

Structural Testing

- Structural testing is considered more technical than functional testing.
- Attempts to design test cases from the source code and not from the specifications.
- The source code becomes the base document which is examined thoroughly in order to understand the internal structure and other implementation details.

Structural Testing

- Structural testing techniques are also known as **White Box Testing** techniques due to consideration of internal structure and other implementation details of the program.
- Types:
 1. Control Flow Testing
 2. Data Flow Testing
 3. Mutation Testing

Control Flow Testing

- Paths of the program are identified and write test cases to execute those paths.
- Path → a sequence of statements that begins at an entry and ends at an exit.
- There may be too many paths in a program and it may not be feasible to execute all of them.
- As the number of decisions increase in the program, the number of paths also increase accordingly.

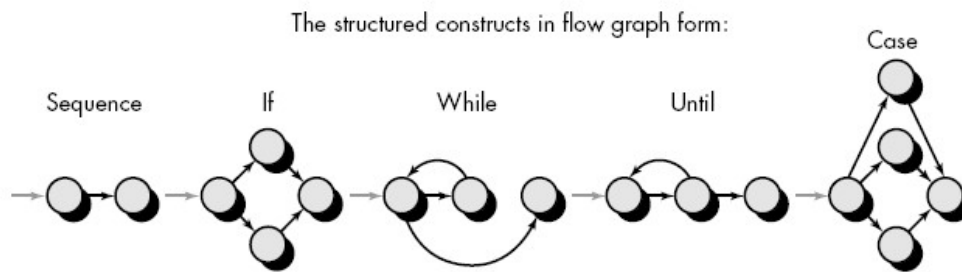
Control Flow Testing

- Every path covers a portion of the program.
- Coverage → percentage of source code that has been tested with respect to the total source code available for testing.
- We may like to achieve a reasonable level of coverage using control flow testing.
- The most reasonable level may be to test every statement of a program at least once before the completion of testing.

Control Flow Testing – Statement Coverage

Objective: To execute every statement of the program in order to achieve 100% statement coverage.

- Use Flow Graph Notation



Control Flow Testing – Statement Coverage

Flow Graph Notation: Terminology

Flow Graph Node: Each circle in flow graph

Edge / Link: Arrows on the flow graph

Regions: Areas bounded by edges and regions. Area outside the region is considered as a region.

Predicate Node: Each node that contains a condition.

Control Flow Testing – Statement Coverage

Cyclomatic complexity / Number of Independent Path $V(G)$

$V(G)$ = Number of regions

$V(G) = E - N + 2$

$V(G) = P + 1$

E = Number of flow graph edges

N = Number of flow graph nodes

P = Number of predicate nodes in flow graph G

Control Flow Testing – Statement Coverage

Example:

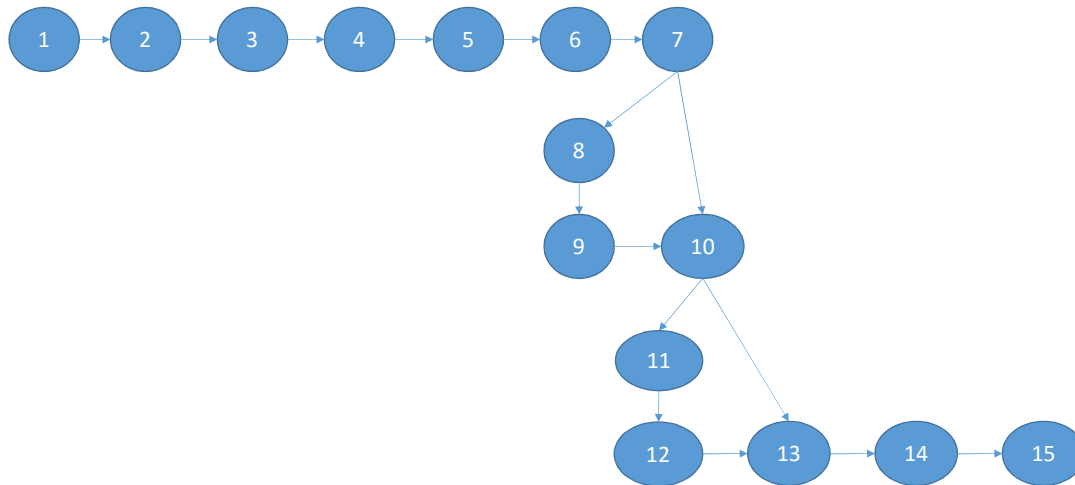
```

1.  void main()
2.  {
3.    int a,b,c,x=0,y=0;
4.    clrscr();
5.    printf("Enter three numbers:");
6.    scanf("%d %d %d",&a,&b,&c);
7.    if((a>b)&&(a>c)){
8.        x=a*a+b*b;
9.    }
```

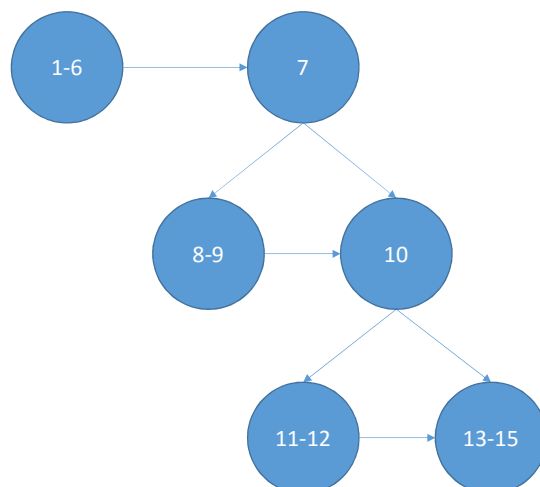
```

10.  if(b>c){
11.      y=a*a-b*b;
12.  }
13.  printf("x= %d y= %d",x,y);
14.  getch();
15.  }
```

Control Flow Testing - Statement Coverage



Control Flow Testing - Statement Coverage



Control Flow Testing – Statement Coverage

E = Number of flow graph edge = **7**

N = Number of flow graph nodes = **6**

P = Number of predicate nodes in flow graph G = **2**

$$\begin{aligned} V(G) &= E - N + 2 \\ &= 7 - 6 + 2 \\ &= 3 \end{aligned}$$

Control Flow Testing – Statement Coverage

E = Number of flow graph edge = **7**

N = Number of flow graph nodes = **6**

P = Number of predicate nodes in flow graph G = **2**

$$V(G) = \text{Number of regions} = 3$$

$$\begin{aligned} V(G) &= P + 1 \\ &= 2 + 1 \\ &= 3 \end{aligned}$$

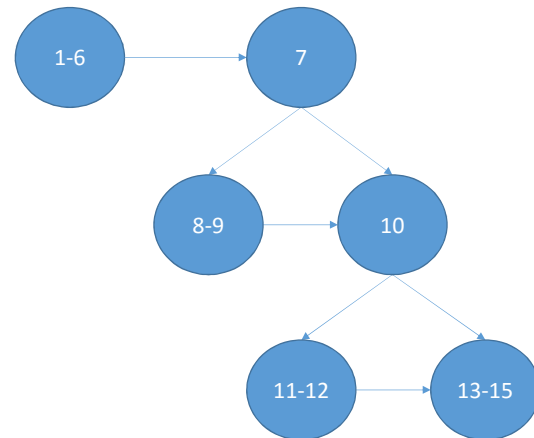
Control Flow Testing – Statement Coverage

Total paths are:

- 1-6 → 7 → 8-9 → 10 → 11-12 → 13-15
- 1-6 → 7 → 8-9 → 10 → 13-15
- 1-6 → 7 → 10 → 11-12 → 13-15
- 1-6 → 7 → 10 → 13-15

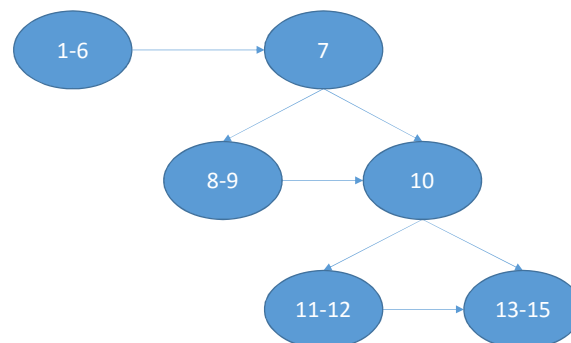
Independent paths are:

- 1-6 → 7 → 8-9 → 10 → 13-15
- 1-6 → 7 → 10 → 11-12 → 13-15
- 1-6 → 7 → 10 → 13-15



Control Flow Testing – Branch Coverage

- Objective: To test every branch of the program with 'True' and 'False' condition of the program.



Control Flow Testing – Branch Coverage

Test Case	A	B	C	Remark
1.	9	8	7	100% statement coverage. All branch coverage with TRUE condition Path Followed: 1-6 → 7 → 8-9 → 10 → 11-12 → 13-15
2	7	8	9	Branch coverage with FALSE conditions Path Followed: 1-6 → 7 → 10 → 13-15

Control Flow Testing – Branch Coverage

- These two test cases are sufficient to guarantee 100% branch coverage.
- The branch coverage does not guarantee 100% path coverage but it does guarantee 100% statement coverage.

Control Flow Testing – Condition Coverage

- Condition coverage is better than branch coverage because we want to test every condition at least once.
- Branch coverage can be achieved without testing every condition.

```

1.  void main()
2.  {
3.  int a,b,c,x=0,y=0;
4.  clrscr();
5.  printf("Enter three numbers:");
6.  scanf("%d %d %d",&a,&b,&c);
7.  if((a>b)&&(a>c)){ ←
8.      x=a*a+b*b;
9.  }
```

Control Flow Testing – Condition Coverage

- Statement-7 has two conditions (a>b) and (a>c).
- There are four possibilities namely:
 - Both are true
 - First is true, second is false
 - First is false, second is true
 - Both are false

```

1.  void main()
2.  {
3.  int a,b,c,x=0,y=0;
4.  clrscr();
5.  printf("Enter three numbers:");
6.  scanf("%d %d %d",&a,&b,&c);
7.  if((a>b)&&(a>c)){ ←
8.      x=a*a+b*b;
9.  }
```

Control Flow Testing – Condition Coverage

Condition-1	Condition-2	Statement-7 (True / False)
T	T	T
T	F	F
F	T	F
F	F	F

- Test cases must be written according to conditions.

Test Case	a	b	c	Statement-7 (True / False)
1.	9	8	7	T (1)
2.	9	8	10	F (2)
3.	7	8	9	F (3 & 4)

- 3 test cases are sufficient to ensure execution of every condition.

Control Flow Testing – Path Coverage

- Objective: To test every path of the program.
- There are too many paths in any program due to loops and feedback connections.
- It may not be possible to execute all paths in many programs.
- We should do enough testing to achieve a reasonable level of coverage.

Control Flow Testing – Path Coverage

- At least (minimum level) all independent paths must be executed which are also referred to as basis paths to achieve reasonable coverage.
- These paths can be found using any method of Cyclomatic Complexity.
- Path testing guarantee statement coverage, branch coverage and condition coverage.

Data Flow Testing

Find output of the following program:

```
1. # include < stdio.h>
2. void main ()
3. {
4.   int a, b, c;
5.   a = b + c;
6.   printf ("%d", a);
7. }
```

Data Flow Testing

- Output may be Garbage Value
- The mistake is in the usage (reference) of the variables without first assigning a value to it.
- We may assume that all variables are automatically assigned to zero initially.
- This does not happen always.

Data Flow Testing

- Data flow testing may help us to minimize such mistakes.
- It is based on variables, their usage and their definition(s) (assignment) in the program.
- The main points of concern are:
 - Statements where variables receive values (definition).
 - Statements where these values are used (referenced).

Data Flow Testing – Define/Reference Anomalies

- May be identified by static analysis of the program.
- Some of the define / reference anomalies are given as:
 - A variable is defined but never used / referenced.
 - A variable is used but never defined.
 - A variable is defined twice before it is used.
 - A variable is used before even first-definition.
- This technique uses the program graphs to understand the 'define / use' conditions of all variables.

Data Flow Testing – Definitions

- A program is first converted into a program graph.
- There may be many paths in the program graph.
 - Defining Node
 - Usage Node
 - Definition Use Path
 - Definition Clear Path

Data Flow Testing – Defining Node

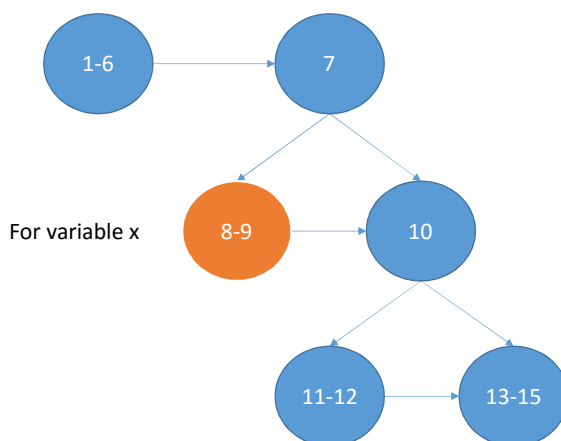
- A node of a program graph is a defining node for a variable v , if and only if, the value of the variable is defined in the statement corresponding to that node.
- It is represented as $DEF(v, n)$ where v is the variable and n is the node corresponding to the statement in which is defined.

Data Flow Testing – Defining Node

```

1.  void main()
2.  {
3.  int a,b,c,x=0,y=0;
4.  clrscr();
5.  printf("Enter three numbers:");
6.  scanf("%d %d %d",&a,&b,&c);
7.  if((a>b)&&(a>c)){
8.      x=a*a+b*b;
9.  }
10. if(b>c){
11.     y=a*a-b*b;
12. }
13. printf("x= %d y= %d",x,y);
14. getch();
15. }

```



Data Flow Testing – Usage Node

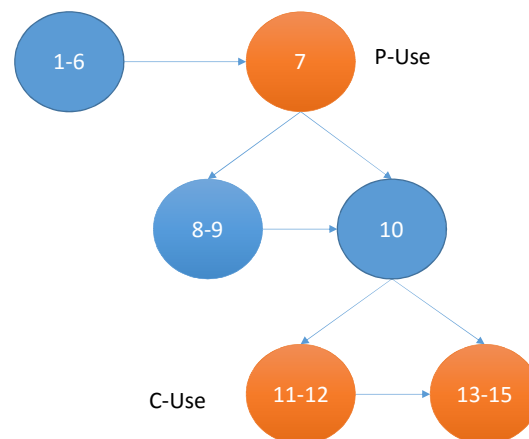
- A node of a program graph is a usage node for a variable v , if and only if, the value of the variable is used in the statement corresponding to that node.
- It is represented as $USE(v, n)$, where ' v ' is the variable and ' n ' in the node corresponding to the statement in which ' v ' is used.
- A usage node $USE(v, n)$ is a predicate use node (denoted as P-use), if and only if, the statement corresponding to node ' n ' is a predicate statement otherwise $USE(v, n)$ is a computation use node (denoted as C-use).

Data Flow Testing – Usage Node

```

1.  void main()
2.  {
3.  int a,b,c,x=0,y=0;
4.  clrscr();
5.  printf("Enter three numbers:");
6.  scanf("%d %d %d",&a,&b,&c);
7.  if((a>b)&&(a>c)){
8.      x=a*a+b*b;
9.  }
10. if(b>c){
11.     y=a*a-b*b;
12. }
13. printf("x= %d y= %d",x,y);
14. getch();
15. }

```



Data Flow Testing – Definition Use Path

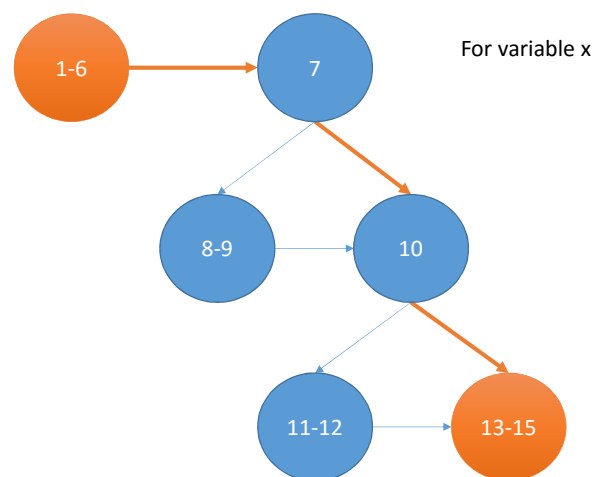
- A definition use path (denoted as du-path) for a variable 'v' is a path between two nodes 'm' and 'n'.
- 'm' is the initial node in the path but the defining node for variable 'v' (denoted as DEF (v, m))
- 'n' is the final node in the path but usage node for variable 'v' (denoted as USE (v, n)).
- A du-path for a variable 'v' may have many redefinitions of variable 'v' between initial node (DEF (v, m)) and final node (USE (v, n)).

Data Flow Testing – Definition Use Path

```

1.  void main()
2.  {
3.  int a,b,c,x=0,y=0;
4.  clrscr();
5.  printf("Enter three numbers:");
6.  scanf("%d %d %d",&a,&b,&c);
7.  if((a>b)&&(a>c)){
8.      x=a*a+b*b;
9.  }
10. if(b>c){
11.     y=a*a-b*b;
12. }
13. printf("x= %d y= %d",x,y);
14. getch();
15. }

```



Data Flow Testing – Definition Clear Path

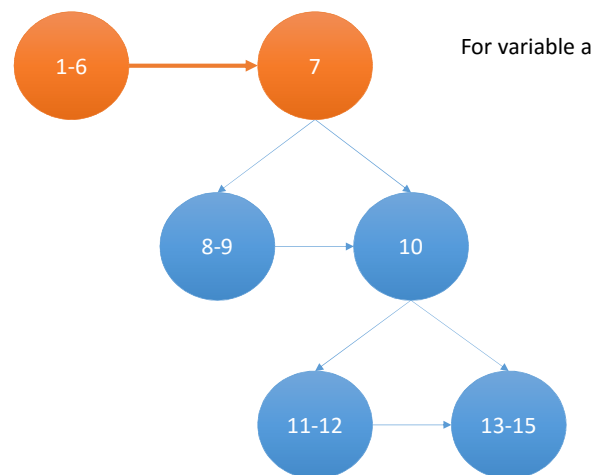
- A definition clear path (denoted as dc-path) for a variable 'v' is a definition use path with initial and final nodes DEF (v, m) and USE (v, n) such that no other node in the path is a defining node of variable 'v'.
- A dc-path for a variable 'v' will not have any definition of variable 'v' between initial node (DEF (v, m)) and final node (USE (v, n)).
- The du-paths that are not definition clear paths are potential troublesome paths.
- They should be identified and tested on topmost priority.

Data Flow Testing – Definition Clear Path

```

1.  void main()
2.  {
3.  int a,b,c,x=0,y=0;
4.  clrscr();
5.  printf("Enter three numbers:");
6.  scanf("%d %d %d",&a,&b,&c);
7.  if((a>b)&&(a>c)){
8.      x=a*a+b*b;
9.  }
10. if(b>c){
11.     y=a*a-b*b;
12. }
13. printf("x= %d y= %d",x,y);
14. getch();
15. }

```



Data Flow Testing – Identify du / dc Paths

1. Draw the program graph of the program.
2. Find all variables of the program and prepare a table for define / use status of all variables using the following format:

Sr. No.	Variables	Defined at Node	Used at Node

Data Flow Testing – Identify du / dc Paths

3. Generate all du-paths from define/use variable table using the following format:

Sr. No.	Variables	du-path (Begin, End)

4. Identify those du-paths which are not dc-paths.

Data Flow Testing – Testing Strategy using du-path

Test all du-paths

- All du-paths generated for all variables are tested.
- This is the strongest data flow testing strategy covering all possible du-paths.
- Ensures that each definition reaches all possible uses through all possible du-paths

Data Flow Testing – Testing Strategy using du-path

Test all uses

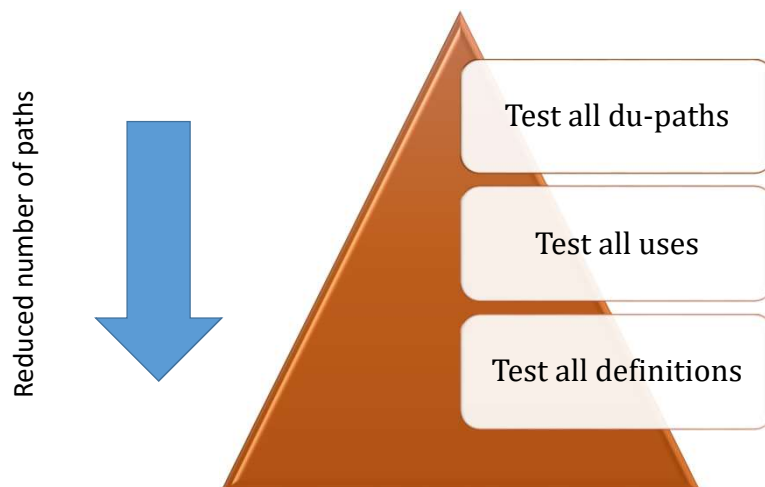
- Find at least one path from every definition of every variable to every use of that variable which can be reached by that definition.
- For every use of a variable, there is a path from the definition of that variable to the use of that variable.
- Ensures that each definition reaches all possible uses.

Data Flow Testing – Testing Strategy using du-path

Test all definitions

- Find paths from every definition of every variable to at least one use of that variable;
- Ensures that each definition reaches at least one use.

Data Flow Testing – Testing Strategy using du-path



Data Flow Testing – Generation of Test Cases

- Consider the program to find the largest number amongst three numbers.

```

1.         #include<stdio.h>
2.         #include<conio.h>
3.         void main()
4.         {
5.         float A,B,C;
6.         clrscr();
7.         printf("Enter number 1:\n");
8.         scanf("%f", &A);
9.         printf("Enter number 2:\n");
10.        scanf("%f", &B);
11.        printf("Enter number 3:\n");
12.        scanf("%f", &C);

```

Data Flow Testing – Generation of Test Cases

```

11.        /*Check for greatest of three numbers*/
12.        if(A>B) {
13.        if(A>C) {
14.            printf("The largest number is: %f\n",A);
15.        }
16.        else {
17.            printf("The largest number is: %f\n",C);
18.        }
19.        else {
20.        if(C>B) {
21.            printf("The largest number is: %f\n",C);
22.        }

```

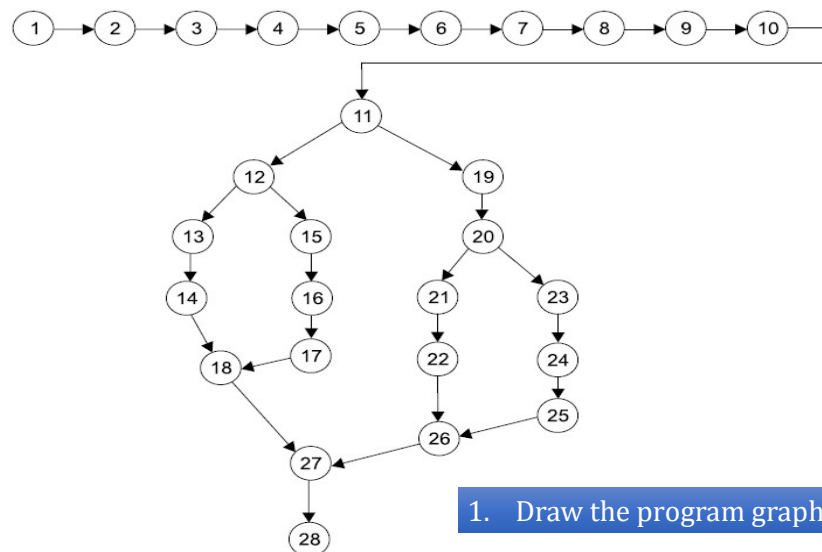
Data Flow Testing – Generation of Test Cases

```

23.      else {
24.          printf("The largest number is: %f\n",B);
25.      }
26.  }
27.  getch();
28.  }

```

Data Flow Testing – Generation of Test Cases



1. Draw the program graph of the program.

Data Flow Testing – Generation of Test Cases

2. Find all variables of the program and prepare a table for define / use status of all variables.

- There are three variables: A, B, C.
- Define /use nodes for all these variables.

Sr. No.	Variables	Defined at Node	Used at Node
1.	A	6	11, 12, 13
2.	B	8	11, 20, 24
3.	C	10	12, 16, 20, 21

Data Flow Testing – Generation of Test Cases

3. Generate all du-paths from define/use variable table.

- The du-paths with beginning node and end node are given as:

Sr. No.	Variables	du-path (Begin, End)
1.	A	6, 11 6, 12 6, 13
2.	B	8, 11 8, 20 8, 24

Data Flow Testing – Generation of Test Cases

3. Generate all du-paths from define/use variable table.

Sr. No.	Variables	du-path (Begin, End)
3.	C	10, 12 10, 16 10, 20 10, 21

Data Flow Testing – Generation of Test Cases

4. Choose Test Strategy : (Test all du-paths)

	Path	Definition Clear?
All du-path	6-11	YES
	6-12	YES
	6-13	YES
	8-11	YES
	8-11, 19, 20	YES
	8-11, 19, 20, 23, 24	YES
	10-12	YES
	10-12, 15, 16	YES
	10, 11, 19, 20	YES
	10, 11, 19-21	YES

Data Flow Testing – Generation of Test Cases

4. Choose Test Strategy : (Test all du-paths)

	Path	Definition Clear?
All definitions	6-11	YES
	8-11	YES
	10-12	YES

Data Flow Testing – Generation of Test Cases

5. Write Test Cases (All du-path):

Sr. No.	A	B	C	Expected Output	Remark
1.	9	8	7	9	6-11
2.	9	8	7	9	6-12
3.	9	8	7	9	6-13
4.	7	9	8	9	8-11
5.	7	9	8	9	8-11, 19, 20
6.	7	9	8	9	8-11, 19, 20, 23, 24
7.	8	7	9	9	10-12
8.	8	7	9	9	10-12, 15, 16
9.	7	8	9	9	10, 11, 19, 20
10.	7	8	9	9	10, 11, 19-21

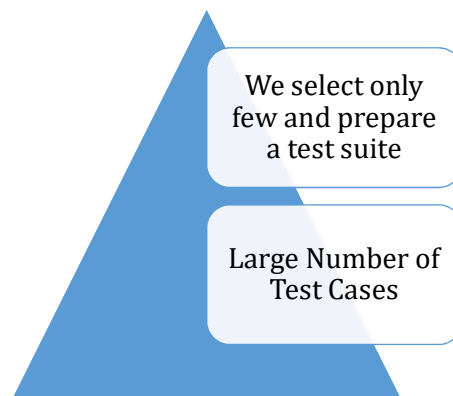
Data Flow Testing – Generation of Test Cases

5. Write Test Cases (All Definition):

Sr. No.	A	B	C	Expected Output	Remark
1.	9	8	7	9	6-11
2.	7	9	8	9	8-11
3.	8	7	9	9	10-12

Mutation Testing

- Technique to test effectiveness of Test Suite.



- How to test effectiveness?
- Is this suite adequate?

Not possible to execute all because of limited time and resources.

Mutation Testing

- Test suite may not be able to make program fail (no errors are found).

Reasons???

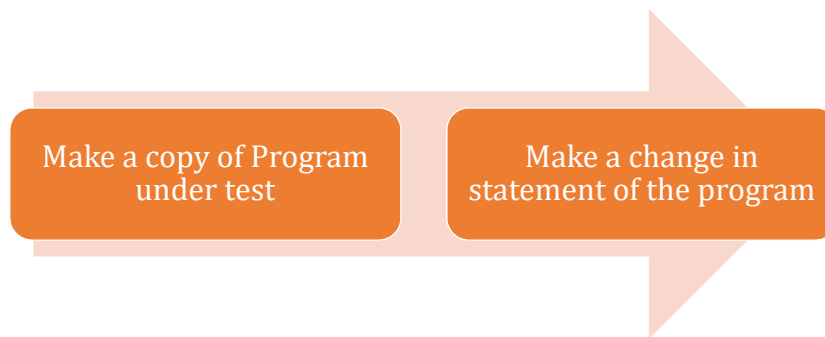
1. The test suite is effective but hardly any errors are there in the program.
How will a test suite detect errors when they are not there?
2. The test suite is not effective and could not find any errors. Although there may be errors, they could not be detected due to poor selection of test suite.

Mutation Testing

- When the test suite is not able to detect errors, how do we know whether the test suite is not effective or the program quality is good?
- Assessing the effectiveness and quality of a test suite is very important.
- Mutation testing may help us to assess the effectiveness of a test suite.
- Mutation testing may also help in enhancing the test suite, if it is not adequate for a program.

Mutation Testing – Mutation and Mutants

- The process of changing a program is known as **Mutation**.
- This change may be limited to one, two or very few changes in the program.



Mutation Testing – Mutation and Mutants

- This changed version of the program is known as a **Mutant** of the original program.
- The behavior of the mutant may be different from the original program due to the introduction of a change.
- However, the original program and mutant are syntactically correct and should compile correctly.
- To mutate a program means to change a program.

Mutation Testing – Mutation and Mutants

- We generally make only one or two changes in order to assess the effectiveness of the selected test suite.
- We may make many mutants of a program by making small changes in the program.
- Every mutant will have a different change in a program.

Mutation Testing – Mutation and Mutants

- Consider a program to find the largest amongst three numbers

```

1.      #include<stdio.h>
2.      #include<conio.h>
3.      void main()
4.      {
5.          float A,B,C;
6.          clrscr();
7.          printf("Enter number 1:\n");
8.          scanf("%f", &A);
9.          printf("Enter number 2:\n");
10.         scanf("%f", &B);
11.         printf("Enter number 3:\n");
12.         scanf("%f", &C);

```

Mutation Testing – Mutation and Mutants

```
/*Check for greatest of three numbers*/
11.      if(A>B) {
12.      if(A>C) {
13.          printf("The largest number is: %f\n",A);
14.      }
15.      else {
16.          printf("The largest number is: %f\n",C);
17.      }
18.      }
19.      else {
20.      if(C>B) {
21.          printf("The largest number is: %f\n",C);
22.      }
```

Mutation Testing – Mutation and Mutants

```
23.      else {
24.          printf("The largest number is: %f\n",B);
25.      }
26.      }
27.      getch();
28.      }
```

Mutation Testing – Mutation and Mutants

Mutant – 1

```

10.      scanf("%f", &C);
        /*Check for greatest of three numbers*/
11.      if(A>B){ ← if(A=B) { mutated statement ('>' is replaced by '=')
12.      if(A>C) {
13.          printf("The largest number is: %f\n",A);
14.      }

```

Mutant M1 is obtained by replacing the operator '>' of line number 11 by the operator '='.

Mutation Testing – Mutation and Mutants

Mutant – 2

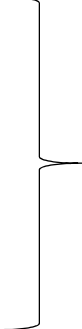
```

19.      else {
20.      if(C>B) { ← if(C<B) { mutated statement ('>' is replaced by '<')
21.          printf("The largest number is: %f\n",C);
22.      }
23.      else {

```

Mutant M2 is obtained by replacing the operator '>' of line number 11 by the operator '<'.

Mutation Testing – Mutation and Mutants

- The mutants generated by making only one change are known as **First Order Mutants**.
 - **Second Order Mutants** can be obtained by making two simple changes in the program
 - **Third Order Mutants** are obtained by making three simple changes, and so on.
- 
- ✓ Higher Order Mutants
 - ✓ Generally first order mutants are preferred in order to simplify the process of mutation

Mutation Testing – Mutation Operator

- Mutants are produced by applying mutant operators.
- **Operator** : A grammatical rule that changes a single expression to another expression.
- The changed expression should be grammatically correct as per the used language.
- If one or more mutant operators are applied to all expressions of a program, we may be able to generate a large set of mutants.

Mutation Testing – Mutation Operator

- We should measure the degree to which the program is changed.
- Example: If the original expression is $x + 1$, and the mutant for that expression is $x + 2$, that is considered as a lesser change.
- Example: If the original expression is $x + 1$ and changed expression is $(y * 2)$, It become more complex by changing both operands and the operator.
- Higher order mutants become difficult to manage, control and trace. They are not popular in practice and first order mutants are recommended to be used.

Mutation Testing – Mutation Operator

- To kill a mutant, we should be able to execute the changed statement of the program otherwise the fault will not be detected.

Some of the mutant operators	
Changing the access modifier, like public to private	Static modifier change
Argument order change	Super Keyword change
Operator change	Any operand change by a numeric value

Mutation Testing – Mutation Score

- **Killed Mutant:** The results of the program are affected by the change and any test case of the test suite detects it. If this happens, then the mutant is called a killed mutant.
- **Live Mutant:** The results of the program are not affected by the change and any test case of the test suite does not detect the mutation. The mutant is called a live mutant.

$$\text{Mutation Score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$

Killed Mutants
+
Live Mutants

Mutation Testing – Mutation Score

- The Mutation Score measures how sensitive the program is to the changes and how accurate the test suite is.
- A mutation score is always between 0 and 1.
- A higher value of mutation score indicates the effectiveness of the test suite although effectiveness also depends on the types of faults that the mutation operators are designed to represent.
- The live mutants are important for us and should be analyzed thoroughly.

Mutation Testing – Mutation Score

- Why any test case of the test suite was not able to detect the changed behavior of the program?

Reasons:

1. Changed statement was not executed by these test cases.
2. If executed, then also it has no effect on the behavior of the program.

Solution: Write new test cases to kill all Live mutants.

Mutation Testing – Mutation Score

- Such test cases should be preserved and transferred to the original test suite in order to enhance the capability of the test suite.
- Hence, the purpose of mutation testing is not only to assess the capability of a test suite but also to enhance the test suite.

Some Mutation Testing Tools

- Insure++
- Jester for Java (open source)
- Nester for C++ (open source).

Mutation Testing - Example

- Consider the program to find the largest of three numbers:

Sample Test Suite				
Sr. No.	A	B	C	Expected Output
1.	6	10	2	10
2.	10	6	2	10
3.	6	2	10	10
4.	6	10	20	20

- Generate 5 mutants (M1 - M5) and calculate Mutation Score of given test suite.

Mutation Testing - Example

Mutants

Mutant No.	Line no.	Original line	Modified Line
M ₁	11	if(A>B)	if (A<B)
M ₂	11	if(A>B)	if(A>(B+C))
M ₃	12	if(A>C)	if(A<C)
M ₄	20	if(C>B)	if(C=B)
M ₅	16	printf("The Largest number is:%f\n",C);	printf("The Largest number is:%f\n",B);

Mutation Testing - Example

Actual Output of Mutant -1 (Killed)					
Sr. No.	A	B	C	Expected Output	Actual Output
1.	6	10	2	10	6
2.	10	6	2	10	6
3.	6	2	10	10	10
4.	6	10	20	20	20

Mutation Testing - Example

Actual Output of Mutant -2 (Live)					
Sr. No.	A	B	C	Expected Output	Actual Output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	10
4.	6	10	20	20	20

Mutation Testing - Example

Actual Output of Mutant -3 (Killed)					
Sr. No.	A	B	C	Expected Output	Actual Output
1.	6	10	2	10	10
2.	10	6	2	10	2
3.	6	2	10	10	6
4.	6	10	20	20	20

Mutation Testing - Example

Actual Output of Mutant -4 (Killed)					
Sr. No.	A	B	C	Expected Output	Actual Output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	10
4.	6	10	20	20	10

Mutation Testing - Example

Actual Output of Mutant -5 (Killed)					
Sr. No.	A	B	C	Expected Output	Actual Output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	2
4.	6	10	20	20	20

Mutation Testing - Example

- Killed Mutants = 4
- Live Mutants = 1
- Total Mutants = 5
- Mutation Score = $4/5 = 0.8$
- M2 is Live. Write additional test case for M2.

Mutation Testing - Example

Additional Test Case				
Sr. No.	A	B	C	Expected Output
5.	10	5	6	10

Actual Output of Added Test Case for Mutant -2 (Killed)					
Sr. No.	A	B	C	Expected Output	Actual Output
5.	10	5	6	10	6

Mutation Testing - Example

Revised Test Suite				
Sr. No.	A	B	C	Expected Output
1.	6	10	2	10
2.	10	6	2	10
3.	6	2	10	10
4.	6	10	20	20
5.	10	5	6	10