

Dynamic Testing: Black-Box

Testing Technique

1. Black-box technique is one of the major techniques in dynamic testing for designing effective test cases.
2. This technique considers only the functional requirements of the software or module. In other words, the structure or logic of the software is not considered. Therefore, this is also known as functional testing.
3. Black-box testing attempts to find errors in the following categories:
 1. To test the modules independently
 2. To test the functional validity of the software so that incorrect or missing functions can be recognized
 3. To look for interface errors
 4. To test the system behaviour and check its performance
 5. To test the maximum load or stress on the system
 6. To test the software such that the user/customer accepts the system within defined acceptable limits
4. BVA offers several methods to design test cases. Following are the few methods used:
 - z1. BOUNDARY VALUE CHECKING (BVC)
 - z2. ROBUSTNESS TESTING METHOD
 - z3. WORST-CASE TESTING METHOD
 - z4. ROBUST WORST-CASE TESTING METHOD

4.1.1 BOUNDARY VALUE CHECKING (BVC)

In this method, the test cases are designed by holding one variable at its extreme value and other variables at their nominal values in the input domain.

The variable at its extreme value can be selected at:

- (a) Minimum value (Min)
- (b) Value just above the minimum value (Min+)
- (c) Maximum value (Max)
- (d) Value just below the maximum value (Max-)

Let us take the example of two variables, A and B. If we consider all the above combinations with nominal values, then following test cases (see Fig. 4.3) can be designed:

1. $A_{\text{nom}}, B_{\text{min}}$	2. $A_{\text{nom}}, B_{\text{min}+}$
3. $A_{\text{nom}}, B_{\text{max}}$	4. $A_{\text{nom}}, B_{\text{max}-}$
5. $A_{\text{min}}, B_{\text{nom}}$	6. $A_{\text{min}+}, B_{\text{nom}}$
7. $A_{\text{max}}, B_{\text{nom}}$	8. $A_{\text{max}-}, B_{\text{nom}}$
9. $A_{\text{nom}}, B_{\text{nom}}$	

4.4 Decision Table – Based Testing

- Boundary value analysis and equivalence class partitioning methods do not consider combinations of input conditions.
- There may be some critical behaviour to be tested when some combinations of input conditions are considered.
- Decision table is another useful method to represent the information in a tabular method.
- Decision table has the specialty to consider complex combinations of input conditions and resulting actions.
- Decision tables obtain their power from logical expressions.
- Each operand or variable in a logical expression takes on the value, TRUE or FALSE

This statement means that **Boundary Value Analysis (BVA)** and **Equivalence Class Partitioning (ECP)** focus on testing individual input conditions separately, rather than testing combinations of multiple inputs together.

Explanation:

1. **Equivalence Class Partitioning (ECP)**
 - It divides input values into groups (equivalence classes) where the system is expected to behave similarly.
 - Only one representative value from each class is tested, assuming that if one value works, all others in the class will too.
 - However, ECP does not explicitly consider cases where multiple input conditions interact.
2. **Boundary Value Analysis (BVA)**
 - It focuses on testing values at the edges (boundaries) of input ranges since errors are more likely to occur there.
 - It usually tests minimum, maximum, just inside, and just outside boundary values.
 - Like ECP, BVA tests each input variable individually rather than testing multiple input combinations together.

Why This Matters:

- Combinatorial testing (like decision table testing or pairwise testing) is needed when multiple input conditions interact.
- BVA and ECP alone may not catch bugs that arise from input combinations.

Would you like an example to illustrate this?

4.4.1 FORMATION OF DECISION TABLE

A decision table is formed with the following components (see Table 4.3):

Table 4.3 Decision table structure

40
You said:
yes i need an example to illustrate this

4.4.1 FORMATION OF DECISION TABLE

A decision table is formed with the following components (see Table 4.3):

Table 4.3 Decision table structure

		ENTRY				
Condition Stub		Rule 1	Rule 2	Rule 3	Rule 4	...
	C1	True	True	False	I	
	C2	False	True	False	True	
	C3	True	True	True	I	
Action Stub	A1		X			
	A2	X			X	
	A3			X		

Condition stub It is a list of input conditions for which the complex combination is made.

Action stub It is a list of resulting actions which will be performed if a combination of input condition is satisfied.

Condition entry It is a specific entry in the table corresponding to input conditions mentioned in the condition stub. When we enter TRUE or FALSE

for all input conditions for a particular combination, then it is called a *Rule*. Thus, a rule defines which combination of conditions produces the resulting action. When the condition entry takes only two values—TRUE or FALSE, then it is called *Limited Entry Decision Table*. When the condition entry takes several values, then it is called *Extended Entry Decision Table*. In limited entry decision table, condition entry, which has no effect whether it is True or False, is called a *Don't-Care state* or *immaterial state* (represented by I). The state of a don't-care condition does not affect the resulting action.

Action entry It is the entry in the table for the resulting action to be performed when one rule (which is a combination of input condition) is satisfied. 'X' denotes the action entry in the table.

The guidelines to develop a decision table for a problem are discussed below [7]:

- List all actions that can be associated with a specific procedure (or module).
- List all conditions (or decision made) during execution of the procedure.
- Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
- Define rules by indicating what action occurs for a set of conditions.

40

You said:

yes i need an example to illustrate this

ChatGPT said:

ChatGPT

Example to Illustrate the Statement

Scenario: Login System Validation

Consider a login system where a user must enter:

- Username (1 to 10 characters long)
- Password (8 to 20 characters long)

If we use Equivalence Class Partitioning (ECP) and Boundary Value Analysis (BVA) separately, they would focus on testing individual inputs rather than their combinations.

Equivalence Class Partitioning (ECP)

We divide the inputs into valid and invalid classes:

Input	Invalid Class	Valid Class	Invalid Class
-------	---------------	-------------	---------------

Username Less than 1 character (empty) 1 to 10 characters More than 10 characters

Password Less than 8 characters 8 to 20 characters More than 20 characters

- We select one representative value from each class.

- Example test cases:
 - (Valid Username, Valid Password) → Should Pass
 - (Empty Username, Valid Password) → Should Fail
 - (Valid Username, Short Password) → Should Fail
 - (Long Username, Valid Password) → Should Fail

Boundary Value Analysis (BVA)

We test values at the edges of valid and invalid ranges:

Test Case	Username	Password	Expected Outcome
-----------	----------	----------	------------------

Min Boundary	1 char	8 chars	Pass <input checked="" type="checkbox"/>
--------------	--------	---------	--

Max Boundary	10 chars	20 chars	Pass <input checked="" type="checkbox"/>
--------------	----------	----------	--

Below Min	0 chars	7 chars	Fail <input type="checkbox"/>
-----------	---------	---------	-------------------------------

Above Max	11 chars	21 chars	Fail <input type="checkbox"/>
-----------	----------	----------	-------------------------------

What These Methods DO NOT Consider?

Both ECP and BVA test only one input field at a time, meaning they do not test combinations of values together.

What If the System Fails for Certain Combinations?

Consider a bug where:

- A username of exactly 10 characters and a password of exactly 8 characters together cause a system crash.
- Both values are valid individually, so BVA and ECP wouldn't detect this issue.

To find such issues, we need Combinatorial Testing methods like:

1. Pairwise Testing → Testing different combinations of input values.
2. Decision Table Testing → Identifying rules based on input interactions.

Conclusion:

- BVA and ECP focus on individual inputs and their limits but do not test interactions between them.
- If input conditions interact in unexpected ways, additional testing is required to uncover those bugs.

Would you like an example using another scenario, like form validation or an e-commerce checkout?

4.4.2 TEST CASE DESIGN USING DECISION TABLE

For designing test cases from a decision table, following interpretations should be done:

1. Interpret condition stubs as the inputs for the test case.
2. Interpret action stubs as the expected output for the test case.
3. Rule, which is the combination of input conditions, becomes the test case itself.
4. If there are k rules over n binary conditions, there are at least k test cases and at the most 2^n test cases.

what does this mean explain in simple words with an example

For designing test cases from a decision table, following interpretations should be done: Interpret condition stubs as the inputs for the test case. Interpret action stubs as the expected output for the test case. Rule, which is the combination of input conditions, becomes the test case itself. If there are k rules over n binary conditions, there are at least k test cases and at the most 2^n test cases.

Simple Explanation with an Example

A Decision Table is used to design test cases by organizing different input conditions and their expected outcomes in a structured way.

Here's what the statement means:

1. Condition Stubs = Inputs

- These are the different conditions (inputs) that affect the system's behavior.

2. Action Stubs = Expected Outputs

- These are the expected results (outputs) based on the input conditions.

3. Rule = Test Case

- A rule is a specific combination of input values that defines how the system should behave. Each rule corresponds to one test case.

4. Number of Test Cases

- If there are k rules, we have at least k test cases.
- If there are n binary conditions (each having two possible values: Yes/No or True/False), the maximum number of test cases is 2^n (all possible combinations).

Example: ATM Cash Withdrawal

Let's say an ATM has the following conditions:

- The user has sufficient balance (Yes/No)
- The correct PIN is entered (Yes/No)

Based on these conditions, the ATM performs an action:

- Dispense Cash or Reject Transaction

Decision Table:

Rule	Sufficient Balance	Correct PIN	Expected Action (Output)
1	Yes	Yes	Dispense Cash <input checked="" type="checkbox"/>
2	Yes	No	Reject Transaction <input type="checkbox"/>
3	No	Yes	Reject Transaction <input type="checkbox"/>
4	No	No	Reject Transaction <input type="checkbox"/>

Interpretation:

- Condition Stubs (Inputs): "Sufficient Balance" and "Correct PIN"
- Action Stubs (Outputs): "Dispense Cash" or "Reject Transaction"
- Rules as Test Cases: Each row in the table represents a test case.

Test Cases Generated:

- User has balance + enters correct PIN → Cash dispensed
- User has balance + enters incorrect PIN → Transaction rejected
- User has no balance + enters correct PIN → Transaction rejected
- User has no balance + enters incorrect PIN → Transaction rejected

Number of Test Cases Calculation:

- We have 2 binary conditions (Balance, PIN).
- Maximum test cases = $2^2 = 4$, which matches our table.

Conclusion:

- Each row in the decision table is a test case.
- The number of test cases depends on the number of conditions.
- If a system has n binary conditions, the maximum test cases are 2^n .

Would you like a more complex example, like an e-commerce checkout scenario? 😊

Black Box Testing

- Approaches to design black box test cases
 - Requirements Based Testing
 - Positive and Negative Testing
 - Boundary Value Analysis (BVA)
 - Boundary Value Checking (BVC)
 - Robustness Testing
 - Worst-Case Testing
 - Equivalence Partitioning
 - State Table Based Testing
 - Decision Table Based
 - Cause-Effect Graphing
 - Compatibility Testing
 - User Documentation Testing
 - Domain Testing

Cause-Effect Graphing

- Cause-effect graphing, also known as dependency modeling,
 - focuses on modelling dependency relationships amongst program input conditions, known as causes, and output conditions, known as effects.
- The relationship is expressed visually in terms of a cause-effect graph.
- The graph is a visual representation of a logical relationship amongst inputs and outputs that can be expressed as a Boolean expression.
- The graph allows selection of various combinations of input values as tests.
- The combinatorial explosion in the number of tests is avoided by using certain heuristics during test generation.

Cause-Effect Graphing (Contd.)

- ▶ A cause is any condition in the requirements that may effect the program output.
- ▶ An effect is the response of the program to some combination of input conditions.
 - ▶ For example, it may be
 - ▶ An error message displayed on the screen
 - ▶ A new window displayed
 - ▶ A database updated.
- ▶ An effect need not be an “output” visible to the user of the program.
- ▶ Instead, it could also be an internal *test point* in the program that can be probed during testing to check if some intermediate result is as expected.
 - ▶ For example, the intermediate test point could be at the entrance into a method to indicate that indeed the method has been invoked.

Example

- ▶ Consider the requirement “Dispense food only when the DF switch is ON”
 - ▶ Cause is “DF switch is ON”.
 - ▶ Effect is “Dispense food”.
- ▶ This requirement implies a relationship between the “DF switch is ON” and the effect “Dispense food”.
- ▶ Other requirements might require additional causes for the occurrence of the “Dispense food” effect.

32

Dynamic Testing: White-Box Testing Techniques:

White-box Testing

- Designing white-box test cases:
 - Requires knowledge about the internal structure of software.
 - White-box testing is also called structural testing.
 - In this unit we will study white-box testing.

- It is also known as glass-box testing, as everything that is required to implement the software is visible.
- The entire design, structure, and code of the software have to be studied for this type of testing.
 - It is obvious that the developer is very close to this type of testing.
 - Often, developers use white-box testing techniques to test their own design and code. This testing is also known as structural or development testing.
- The intention is to test this logic so that required results or functionalities can be achieved.
 - Thus, white-box testing ensures that the internal parts of the software are adequately tested.

5.1 NEED OF WHITE-BOX TESTING

1. In fact, white-box testing techniques are used for testing the module for initial stage testing. Black-box testing is the second stage for testing the software. Though test cases for black box can be designed earlier than white-box test cases, they cannot be executed until the code is produced and checked using white-box testing techniques. Thus, white-box testing is not an alternative but an essential stage.
2. Since white-box testing is complementary to black-box testing, there are categories of bugs which can be revealed by white-box testing, but not through black-box testing. There may be portions in the code which are not checked when executing functional test cases, but these will be executed and tested by white-box testing.
3. Errors which have come from the design phase will also be reflected in the code, therefore we must execute white-box test cases for verification of code (unit verification).
4. We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis. White-box testing explores these paths too.
5. Some typographical errors are not observed and go undetected and are not covered by black-box testing techniques. White-box testing techniques help detect these errors.

The statement means that white-box testing and black-box testing are complementary because they each uncover different types of defects (bugs) in software. While black-box testing focuses on validating the software's functionality without looking at its internal code, white-box testing examines the internal structure, logic, and implementation of the code. Since white-box testing analyzes the code itself, it can reveal specific categories of bugs that black-box testing might miss. These include:

Categories of Bugs Revealed by White-Box Testing

1. Logic Errors & Incorrect Assumptions
 - Mistakes in loops, conditionals (if-else statements), or incorrect logical conditions (AND, OR, NOT operations).
 - Example: A developer writes if ($x > 10$) instead of if ($x \geq 10$), causing unexpected behavior for $x = 10$.
2. Unreachable Code (Dead Code)
 - Code that is never executed due to incorrect logic or redundant conditions.
 - Example:

```
cpp
CopyEdit
if (x > 5) {
    return "Valid";
} else if (x < 5) {
    return "Invalid";
} else {
    return "Edge Case"; // This code is unreachable since x == 5 is not handled
}
```

3. Boundary Errors
 - Mistakes in handling edge cases, such as off-by-one errors in loops.
 - Example: Using for ($i = 0; i < N; i++$) instead of for ($i = 0; i \leq N; i++$) could miss the last element in an array.
4. Faulty Control Flow & Path Coverage Issues
 - Execution paths that are not covered or incorrectly handled, leading to unexpected program behavior.
 - Example: A missing break in a switch-case statement that causes unintended fall-through.
5. Security Vulnerabilities
 - Hardcoded credentials, buffer overflows, or improper error handling that can be exploited by attackers.
 - Example:

```
python
CopyEdit
password = input("Enter password: ")
if password == "admin123": # Hardcoded password is a security risk
    print("Access Granted")
```

6. Incorrect Exception Handling
 - Missing or incorrect exception handling that can cause unexpected crashes.
 - Example: Catching a general exception (catch (Exception e)) instead of handling specific errors.
7. Performance Bottlenecks & Inefficient Code
 - Code inefficiencies that increase execution time or memory usage unnecessarily.
 - Example: Using an $O(n^2)$ algorithm when an $O(n \log n)$ solution is available.

Why White-Box Testing is Complementary to Black-Box Testing?

- Black-box testing ensures the software behaves correctly based on specifications but does not analyze the internal implementation.
- White-box testing finds hidden defects in the code structure that black-box testing might overlook.

10.7.1 Basic Concepts

A white-box testing strategy can either be coverage-based or fault-based.

Fault-based testing

A fault-based testing strategy targets to detect certain types of faults. These faults that a test strategy focuses on constitutes the **fault model** of the strategy. An example of a fault-based strategy is mutation testing, which is discussed later in this section.

Coverage-based testing

These faults that a test strategy focuses on constitutes the **fault model** of the strategy. An example of a fault-based strategy is mutation testing, which is discussed later in this section.

Coverage-based testing

A coverage-based testing strategy attempts to execute (or cover) certain elements of a program. Popular examples of coverage-based testing strategies are statement coverage, branch coverage, multiple condition coverage, and path coverage-based testing.

Example: Using an $O(n^2)$ algorithm when an $O(n \log n)$ solution is available.

Why White-Box Testing is Complementary to Black-Box Testing?

- Black-box testing ensures the software behaves correctly based on specifications but **does not analyze the internal implementation**.
- White-box testing finds **hidden defects in the code structure** that black-box testing might overlook.
- Together, they provide a **more thorough software testing strategy**, covering both **functional correctness** (black-box) and **internal code quality** (white-box).

Would you like examples of specific white-box testing techniques, such as statement coverage, branch coverage, or path coverage?

White-Box Testing Methodologies

- There exist several popular white-box testing methodologies:
 - Statement coverage
 - Branch coverage
 - Condition coverage
 - MC/DC coverage
 - Path coverage
 - Data flow-based testing
 - Mutation testing

3

Statement Coverage

- **Statement coverage methodology:**
 - Design test cases so that every statement in the program is executed at least once.

4

Statement Coverage

The principal idea:

- Unless a statement is executed,
- We have no way of knowing if an error exists in that statement.

5

Statement Coverage Criterion

- Observing that a statement behaves properly for one input value:
 - No guarantee that it will behave correctly for all input values.

Statement Testing

- Coverage measurement:
`# executed statements
statements`
- Rationale: a fault in a statement can only be revealed by executing the faulty statement

6

Test case 1: $x = y = n$, where n is any number

Test case 2: $x = n, y = n'$, where n and n' are different numbers.

Test case 1 just skips the while loop and all loop statements are not executed. Considering test case 2, the loop is also executed. However, every statement inside the loop is not executed. So two more cases are designed:

Test case 3: $x > y$

Test case 4: $x < y$

These test cases will cover every statement in the code segment, however statement coverage is a poor criteria for logic coverage. We can see that test case 3 and 4 are sufficient to execute all the statements in the code. But, if we execute only test case 3 and 4, then conditions and paths in test case 1 will never be tested and errors will go undetected. Thus, **statement coverage is a necessary but not a sufficient criteria for logic coverage**.

7

Branch Coverage

- A test suite satisfies branch coverage, if it makes each branch condition in the program to assume true and false values in turn.
- In other words, for branch coverage each branch in the CFG representation of the program must be taken at least once, when the test suite is executed.
- Branch testing is also known as **edge testing**, since in this testing scheme, each edge of a program's control flow graph is traversed at least once.

Example

```
int f1(int x,int y){  
1 while (x != y){  
2   if (x>y) then  
3     x=x-y;  
4   else y=y-x;  
5 }  
6 return x; }
```

Example

- Test cases for branch coverage can be:
- $\{(x=3,y=3), (x=3,y=2), (x=4,y=3), (x=3,y=4)\}$

Branch Testing

- **Adequacy criterion:** Each branch (edge in the CFG) must be executed at least once
- Coverage:
 $\frac{\# \text{ executed branches}}{\# \text{ branches}}$

16

15

Theorem 10.1 Branch coverage-based testing is stronger than statement coverage-based testing.

Proof: We need to show that (a) branch coverage ensures statement coverage, and (b) statement coverage does not ensure branch coverage.

- (a) Branch testing would guarantee statement coverage since every statement must belong to some branch (assuming that there is no unreachable code).
- (b) To show that statement coverage does not ensure branch coverage, it would be sufficient to give an example of a test suite that achieves statement coverage, but does not cover at least one branch. Consider the following code, and the test suite {5}.

```
if (x>2) x+=1;
```

The test suite would achieve statement coverage. However, it does not achieve branch coverage, since the condition ($x > 2$) is not made false by any test case in the suite.

Statements vs Branch Testing

- Traversing all edges of a graph causes all nodes to be visited
 - So test suites that satisfy the branch adequacy criterion for a program P also satisfy the statement adequacy criterion for the same program
- The converse is not true
 - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate)

17

Condition Coverage:

Condition Coverage

Condition coverage states that each condition in a decision takes on all possible outcomes at least once. For example, consider the following statement:

```
while ((I≤5) && (J < COUNT))
```

In this loop statement, two conditions are there. So test cases should be designed such that both the conditions are tested for True and False outcomes. The following test cases are designed:

Test case 1: $I \leq 5, J < COUNT$

Test case 2: $I < 5, J > COUNT$

Branch vs Condition testing

- Branch testing is the simplest condition testing strategy:
 - Compound conditions appearing in different branch statements
 - are given true and false values.

20

10.7.4 Multiple Condition Coverage

Multiple Condition Coverage

- Multiple condition coverage reports whether every possible combination of Boolean sub-expressions occurs.
- The test cases required for full multiple condition coverage of a condition are essentially given by the logical operator truth table for the condition.

- In the multiple condition (MC) coverage-based testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values.
 - For example, consider the composite conditional expression $((c1 \cdot and \cdot c2) \cdot or \cdot c3)$.
 - A test suite would achieve MC coverage, if all the component conditions $c1$, $c2$ and $c3$ are each made to assume both true and false values.
- Branch testing can be considered to be a simplistic condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values.
- It is easy to prove that condition testing is a stronger testing strategy than branch testing.
- For a composite conditional expression of n components, 2^n test cases are required for multiple condition coverage. Thus, for multiple condition coverage, the number of test cases increases exponentially with the number of component conditions.
- Therefore, multiple condition coverage-based testing technique is practical only if n (the number of conditions) is small

21

Multiple Condition Coverage Examples

```
if(A && B) // condition 1
    F1();
else
    F2();
if(C || D) // condition 2
    F3()
else
    F4();
```

Test Cases for MCC:

For condition 1	For condition 2
A=T, B=T	C=T, D=T
A=T, B=F	C=T, D=F
A=F, B=T	C=F, D=T
A=F, B=F	C=F, D=F

24

Multiple condition coverage In case of multiple conditions, even decision/condition coverage fails to exercise all outcomes of all conditions. The reason is that we have considered all possible outcomes of each condition in the decision, but we have not taken all combinations of different multiple conditions. Certain conditions mask other conditions. For example, if an AND condition is False, none of the subsequent conditions in the expression will be evaluated. Similarly, if an OR condition is True, none of the subsequent conditions will be evaluated. Thus, condition coverage and decision/condition coverage need not necessarily uncover all the errors.

Therefore, multiple condition coverage requires that we should write sufficient test cases such that all possible combinations of condition outcomes in each decision and all points of entry are invoked at least once. Thus, as in decision/condition coverage, all possible combinations of multiple conditions should be considered. The following test cases can be there:

Test case 1: A = True, B = True

Test case 2: A = True, B = False

Test case 3: A = False, B = True

Test case 4: A = False, B = False

MCC Testing Adequacy Criterion

- Adequacy criterion:** Each basic condition must be executed at least once
- Coverage:**
 - # truth values taken by all basic conditions
 - $2^{\# \text{ basic conditions}}$

Consider: `if(a || b && c) then ...`

Test	a	b	c
(1)	T	T	T
(2)	T	T	F
(3)	T	F	T
(4)	T	F	F
(5)	F	T	T
(6)	T	T	F
(7)	F	F	T
(8)	F	F	F

MCC

Exponential in
the number of
basic conditions

25

Modified condition decision:

Modified condition/decision (MC/DC)

- Motivation:** Effectively test important combinations of conditions, without exponential blowup in test suite size
 - “Important” combinations means: Each basic condition shown to independently affect the outcome of each decision
- Requires:**
 - For each basic condition C, two test cases,
 - values of all evaluated conditions except C are the same
 - compound condition as a whole evaluates to true for one and false for the other

29

What is MC/DC?

- MC/DC stands for **Modified Condition / Decision Coverage**
- A kind of Predicate Coverage technique
 - Condition:** Leaf level Boolean expression.
 - Decision:** Controls the program flow.
- Main idea:** Each condition must be shown to independently affect the outcome of a decision, i.e. the outcome of a decision changes as a result of changing a single condition.

30

MC/DC: linear complexity

- N+1 test cases for N basic conditions

 $((a \parallel b) \& c) \parallel d) \&& e$

Test Case	a	b	c	d	e	outcome
(1)	true	--	true	--	true	true
(2)	false	true	true	--	true	true
(3)	true	--	false	true	true	true
(6)	true	--	true	--	false	false
(11)	true	--	false	false	--	false
(13)	false	false	--	false	--	false

34

- Underlined values independently affect the output of the decision
- Required by the RTCA/DO-178B standard

Comments on MC/DC

- MC/DC is
 - basic condition coverage (C)
 - branch coverage (DC)
 - plus one additional condition (M): every condition must independently affect the decision's output
- It is subsumed by compound conditions and subsumes all other criteria discussed so far
 - stronger than statement and branch coverage
- A good balance of thoroughness and test size (and therefore widely used)

35

Example cont ...

Number	A	B	Result	A	B
1	T	T	T	3	2
2	T	F	F		1
3	F	T	F	I	
4	F	F	F		

- Show independence of A:
 - Take 1 + 3
- Show independence of B:
 - Take 1 + 2
- Resulting test cases are
 - I + 2 + 3
 - (T, T) + (T, F) + (F, T)

37

More advanced example

IF (A && (B || C)) then...

Number	A	B	C	Result	A	B	C
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

38

More advanced example contd..

Note: We want to determine the MINIMAL set of test cases

Here:

• {2,3,4,6}

• {2,3,4,7}

Non-minimal set is:

• {1,2,3,4,5}

39

Modified Condition/Decision Coverage (MC/DC)

MC/DC is an advanced software testing technique used in safety-critical systems (e.g., aviation, automotive). It ensures that each condition in a decision contributes independently to the outcome of that decision.

Key Requirements of MC/DC

For a decision with multiple conditions, MC/DC requires:

- Each condition is tested as true and false at least once.
- Each condition independently affects the overall decision outcome.
- Every possible branch (true/false outcome) of the decision is executed at least once.

Example

Consider this decision statement:

```
cpp
CopyEdit
if (A || B) {
    // Do something
}
```

Here, the decision depends on two conditions: A and B.

For MC/DC, we need test cases where each condition changes the result independently:

```
| Test Case | A | B | Condition (A || B) | ----- | ----- | ----- | 1 | F | F | F | 2 | T | F | T |
| 3 | F | T | T |
```

Explanation:

- Test Case 1: Both A and B are false, so the decision is false.
- Test Case 2: A is true and B is false, making the decision true. This verifies that A alone can make the decision true.
- Test Case 3: A is false, but B is true, making the decision true. This verifies that B alone can make the decision true.

Since each condition (A and B) is tested independently to show that it alone can control the decision, MC/DC is satisfied.

Why Use MC/DC?

- Required in critical systems (e.g., avionics per DO-178C standard).
- More efficient than full condition coverage while still ensuring correctness.
- Reduces the number of test cases needed compared to full combinatorial testing.

Would you like an example with more conditions (e.g., A && B || C)? ↗

Basis Path Coverage

- Basis path testing is the oldest structural testing technique.
- The technique is based on the control structure of the program.
- Based on the control structure, a flow graph is prepared and all the possible paths can be covered and executed during testing.
- Path coverage is a more general criterion as compared to other coverage criteria and useful for detecting more errors.
- But the problem with path criteria is that programs that contain loops may have an infinite number of possible paths and it is not practical to test all the paths.

The guidelines for effectiveness of path testing are discussed below:

- Path testing is based on control structure of the program for which flow graph is prepared.
- Path testing requires complete knowledge of the program's structure.
- Path testing is closer to the developer and used by him to test his module.
- The effectiveness of path testing gets reduced with the increase in size of software under test [9].
- Choose enough paths in a program such that maximum logic coverage is achieved.

- A test suite achieves path coverage if it executes each linearly independent paths (or basis paths) at least once.
- A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.
- Therefore, to understand path coverage-based testing strategy, we need to first understand how the CFG of a program can be drawn.

A control flow graph describes the sequence in which the different instructions of a program get executed.

How to Draw Control Flow Graph?

Number all statements of a program

Example

int f1(int x, int y){

Example Control Flow Graph

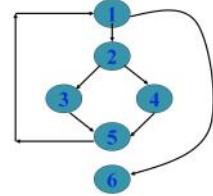
How to Draw Control Flow Graph?

- Number all statements of a program.
- Numbered statements:
 - Represent nodes of control flow graph.
- An edge from one node to another node exists:
 - If execution of the statement representing the first node
 - Can result in transfer of control to the other node.

Example

```
int f1(int x,int y){  
    1 while (x != y){  
        2 if (x>y) then  
            3     x=x-y;  
        4     else y=y-x;  
    5 }  
    6 return x; }
```

Example Control Flow Graph

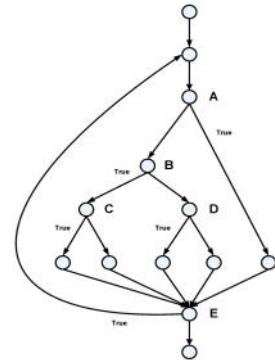


Example Code Fragment

```
Do  
{  
    if (A) then {...};  
    else {  
        if (B) then {  
            if (C) then {...};  
            else {...}  
        }  
        else if (D) then {...};  
        else {...};  
    }  
}  
While (E);
```

10

Example Control Flow Graph



Source: The Art of Software Testing – Glenford Myers

11

Path

A path through a program is any node and edge sequence from the start node to a terminal node of the control flow graph of a program. Please note that a program can have more than one terminal nodes when it contains multiple exit or return type of statements.

Linearly independent set of paths (or basis path set)

- A set of paths for a given program is called linearly independent set of paths (or the set of basis paths or simply the basis set), if each path in the set introduces at least one new edge that is not included in any other path in the set.
- Please note that even if we find that a path has one new node compared to all other linearly independent paths, then this path should also be included in the set of linearly independent paths.

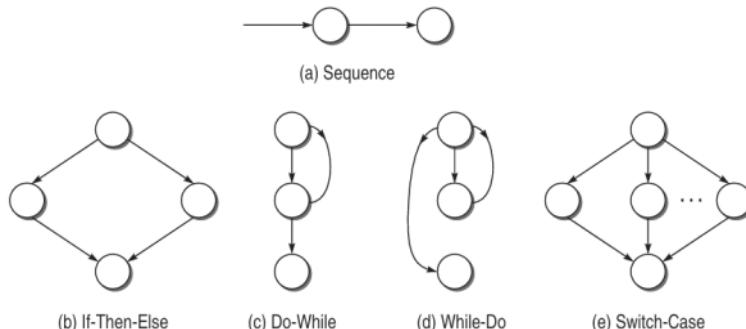


Figure 5.2

10.7.6 McCabe's Cyclomatic Complexity Metric:

- McCabe's cyclomatic complexity defines an upper bound on the number of independent paths in a program.
- We discuss three different ways to compute the cyclomatic complexity.

Method 1: Given a control flow graph G of a program, the cyclomatic complexity $V(G)$ can be computed as:

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

where, N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

For the CFG of example shown in Figure 10.7, $E = 7$ and $N = 6$. Therefore, the value of the Cyclomatic complexity = $7 - 6 + 2 = 3$.

Method 2: An alternate way of computing the cyclomatic complexity of a program is based on a visual inspection of the control flow graph as follows
—In this method, the cyclomatic complexity $V(G)$ for a graph G is given by the following expression:

$$V(G) = \text{Total number of non-overlapping bounded areas} + 1$$

Method 3: The cyclomatic complexity of a program can also be easily computed by computing the number of decision and loop statements of the program. If N is the number of decision and loop statements of a program, then the McCabe's metric is equal to $N + 1$.

- In the program's control flow graph G, any region enclosed by nodes and edges can be called as a bounded area. This is an easy way to determine the McCabe's cyclomatic complexity.
- The number of bounded areas in a CFG increases with the number of decision statements and loops.
- Therefore, the McCabe's metric provides a quantitative measure of testing difficulty and the ultimate reliability of a program.

Cyclomatic Complexity

- McCabe's metric provides:
 - A quantitative measure of testing difficulty and the ultimate reliability
- Intuitively,
 - Number of bounded areas increases with the number of decision nodes and loops.

21

Cyclomatic Complexity

- The cyclomatic complexity of a program provides:
 - A lower bound on the number of test cases to be designed
 - To guarantee coverage of all linearly independent paths.

23

Cyclomatic Complexity

- Knowing the number of test cases required:
 - Does not make it any easier to derive the test cases,
 - Only gives an indication of the minimum number of test cases required.

25

Practical Path Testing

- The tester proposes initial set of test data :
 - Using his experience and judgment.
- A dynamic program analyzer used:
 - Measures which parts of the program have been tested
 - Result used to determine when to stop testing.

26

Steps to carry out path coverage-based testing

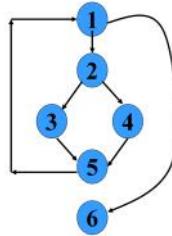
The following is the sequence of steps that need to be undertaken for deriving the path coverage-based test cases for a program:

1. Draw control flow graph for the program.
2. Determine the McCabe's metric $V(G)$.
3. Determine the cyclomatic complexity. This gives the minimum number of test cases required to achieve path coverage.
4. repeat

Example

```
int f1(int x,int y){  
    1 while (x != y){  
        2 if (x>y) then  
            3     x=x-y;  
        4 else y=y-x;  
    5 }  
    6 return x; }
```

Example Control Flow Diagram



Derivation of Test Cases

- Number of independent paths: 3
 - 1,6 test case (x=1, y=1)
 - 1,2,3,5,1,6 test case(x=2, y=1)
 - 1,2,4,5,1,6 test case(x=1, y=2)

An Interesting Application of Cyclomatic Complexity

- Relationship exists between:
 - McCabe's metric
 - The number of errors existing in the code,
 - The time required to find and correct the errors.

31

Cyclomatic Complexity

- Cyclomatic complexity of a program:
 - Also indicates the psychological complexity of a program.
 - Difficulty level of understanding the program.

32

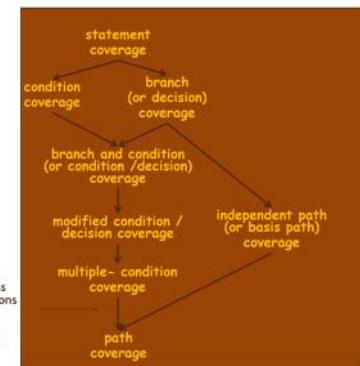
Cyclomatic Complexity

- From maintenance perspective,
 - Limit cyclomatic complexity of modules
 - To some reasonable value.
- Good software development organizations:
 - Restrict cyclomatic complexity of functions to a maximum of ten or so.

33

White-Box Testing : Summary

weakest

only if paths across composite conditions are distinguished
strongest

34







