CS20006: Software Engineering
Module 05: Software Testing & Maintenance

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

March 24, 25, 26, 31. April 01: 2021

Table of Contents

Why Test?

- **Ariane 5 Flight 501**



  - Un-manned satellite-launching rocket in 1996
  - Self-destructed 37 seconds after launch
  - Conversion from 64-bit floating point to 16-bit signed integer value had caused an exception (re-used from Ariane 4)
    - The floating point number was larger than 32767
    - Efficiency considerations had led to the disabling of the exception handler

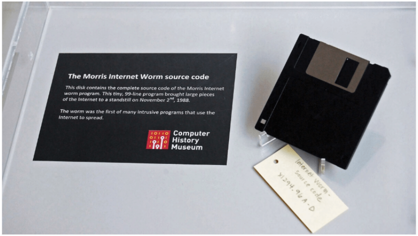
Source: 11 of the most costly software errors in history

Why Test?

- **NASA's Mars Climate Orbiter**
  - Mission to Mars in 1998, $125 Million
  - Lost in space
  - Simple conversion from English units to metric failed
- **EDS Child Support System in 2004**
  - Overpay 1.9 million people
  - Underpay another 700,000
  - US $7 billion in uncollected child support payments
  - Backlog of 239,000 cases
  - 36,000 new cases "stuck" in the system
  - Cost the UK taxpayers over US $1 billion
  - Incompatible software integration
- **Heathrow Terminal 5 Opening**
  - Baggage handling tested for 12,000 test pieces of luggage
  - Missed to test for *removal of baggage*
  - In 10 days some 42,000 bags failed to travel with their owners, and over 500 flights were cancelled

Why Test?

- **The Morris Worm**
  - Developed by a Cornell University student for a harmless experiment
  - Spread wildly and crashing thousands of computers in 1988 because of a coding error
  - It was the first widespread worm attack on the fledgling Internet
  - The graduate student, Robert Tappan Morris, was convicted of a criminal hacking offense and fined $10,000
  - Costs for cleaning up the mess may have gone as high as $100 Million
  - Morris, who co-founded the startup incubator Y Combinator, is now a professor at the Massachusetts Institute of Technology
  - A disk with the worm's source code is now housed at the University of Boston



Why Test?

- **Boeing Crash**
  - On March 10, 2019, Ethiopian Airlines Flight 302 crashed just minutes after takeoff. All 157 people on board the flight died
  - On October of 2018, Lion Air Flight 610 also crashed minutes after taking off
  - Both flights involved Boeing's 737 MAX jet
  - The software overpowered all other flight functions trying to mediate the nose lift
  - Many pilots did not know this system existed - they were not re-trained on 737 MAX Jet

  Source: Boeing Software Scandal Highlights Need for Full Lifecycle Testing

- **Airbus Crash**
  - On May 9, 2015, the Airbus A400M crashed near Seville after a failed emergency landing during its first flight
  - Electronic Control Units (ECU) on board malfunctioned

  Source: Airbus A400M plane crash linked to software fault

Testing a Program

- Input test data to the program
- Observe the output
- Check if the program behaved as expected
- If the program does not behave as expected:
  - Note the conditions under which it failed
  - Debug and correct

What's So Hard About Testing?

- Consider `int proc1(int x, int y)`
- Assuming a 64 bit computer
  - Input space = 2^{128}
- Assuming it takes 10secs to key-in an integer pair
  - It would take about a billion years to enter all possible values!
  - Automatic testing has its own problems!

Testing Facts

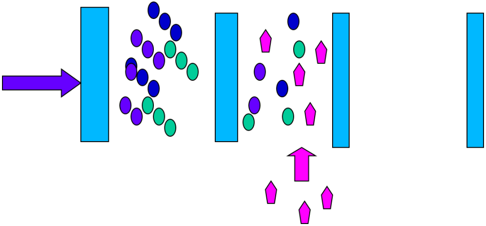
- Consumes largest effort among all phases
    - Largest manpower among all other development roles
    - Implies more job opportunities
- About 50% development effort
    - But 10% of development time?
    - How?
- Testing is getting more complex and sophisticated every year
    - Larger and more complex programs
    - Newer programming paradigms

Overview of Testing

- Testing Activities
  - Test Suite Design
  - Run test cases and observe results to detect failures.
  - Debug to locate errors
  - Correct errors
- Error, Faults, and Failures
  - A failure is a manifestation of an error (also defect or bug)
  - Mere presence of an error may not lead to a failure

Pesticide Effect

- Errors that escape a fault detection technique:
  - Can not be detected by further applications of that technique



- Assume we use 4 fault detection techniques and 1000 bugs:
  - Each detects only 70% bugs
  - How many bugs would remain?
  - $1000 * (0.3)^4 = 81$ bugs

Fault Model

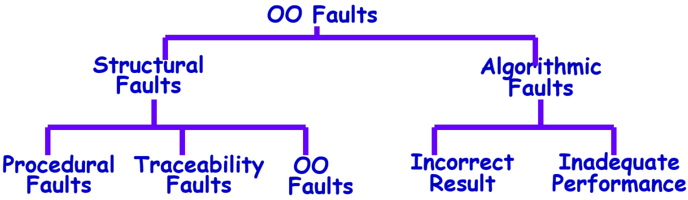
- Types of faults possible in a program
- Some types can be ruled out
  - Concurrency related-problems in a sequential program
  - Consider a singleton in multi-thread

```
#include <iostream>
using namespace std;
class Printer { /* THIS IS A SINGLETON PRINTER -- ONLY ONE INSTANCE */
    bool blackAndWhite_, bothSided_;
    Printer(bool bw = false, bool bs = false) : blackAndWhite_(bw), bothSided_(bs)
    { cout << "Printer constructed" << endl; }
    static Printer *myPrinter_; // Pointer to the Singleton Printer
public:
    ~Printer() { cout << "Printer destructed" << endl; }
    static const Printer& printer(bool bw = false, bool bs = false) {
        if (!myPrinter_) // What happens on multi-thread?
            myPrinter_ = new Printer(bw, bs);
        return *myPrinter_;
    }
    void print(int nP) const { cout << "Printing " << nP << " pages" << endl; }
};
Printer *Printer::myPrinter_ = 0;

int main() {
    Printer::printer().print(10);
    Printer::printer().print(20);
    delete &Printer::printer();
    return 0;
}
```

Fault Model

- Fault Model of an OO Program



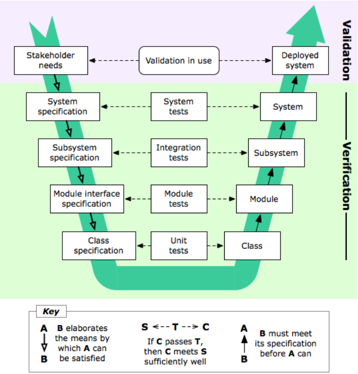
- Hardware Fault-Model
  - Simple:
    - Stuck-at 0
    - Stuck-at 1
    - Open circuit
    - Short circuit
  - Simple ways to test the presence of each
  - Hardware testing is fault-based testing

Test Cases and Test Suites

- Each test case typically tries to establish correct working of some functionality:
  - Executes (covers) some program elements
  - For restricted types of faults, fault-based testing exists
- Test a software using a set of carefully designed test cases:
  - The set of all test cases is called the test suite
- A test case is a triplet [I,S,O]
  - I is the data to be input to the system
  - S is the state of the system at which the data will be input
  - O is the expected output of the system (called *Golden*)

Verification versus Validation

- **Verification** is the process of determining
  - Whether output of one phase conforms to its previous phase
  - If we are building the system correctly
  - *Verification is concerned with phase containment of errors*
- **Validation** is the process of determining
  - Whether a fully developed system conforms to its SRS document
  - If we are building the correct system
  - *Whereas the aim of validation is that the final product be error free*



Source: Software Testing

Design of Test Cases

- Exhaustive testing of any non-trivial system is impractical
    - Input data domain is extremely large
- Design an optimal test suite
    - Of reasonable size and
    - Uncovers as many errors as possible
- If test cases are selected randomly
    - Many test cases would not contribute to the significance of the test suite
    - Would not detect errors not already being detected by other test cases in the suite
- Number of test cases in a randomly selected test suite
    - Not an indication of effectiveness of testing
- Testing a system using a large number of randomly selected test cases
    - Does not mean that many errors in the system will be uncovered

Design of Test Cases

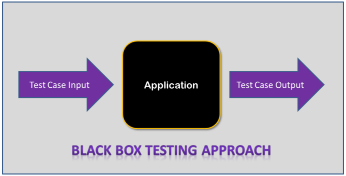
- Consider following example
  - Find the maximum of two integers x and y
- The code has a simple programming error

```
if (x>y)
    max = x;
else
    max = x;
```

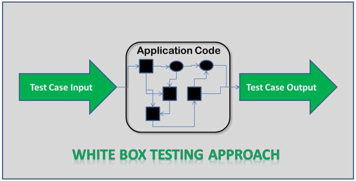
- Test suite {(x=3,y=2); (x=2,y=3)} can detect the error
- A larger test suite {(x=3,y=2); (x=4,y=3); (x=5,y=1)} does not detect the error

Design of Test Cases

- Systematic approaches are required to design an optimal test suite
  - Each test case in the suite should detect different errors
- There are essentially three main approaches to design test cases



  - **Black-box testing** (*Zero Knowledge*)



  - **White-box testing** (*Full Knowledge*)

  - **Grey-box testing** (*Some Knowledge*)

Why Both BB and WB Testing?

Black Box Testing

- Impossible to write a test case for every possible set of inputs and outputs
- Some code parts may not be reachable
- Does not tell if extra functionality has been implemented.

White Box Testing

- Does not address the question of whether or not a program matches the specification
- Does not tell you if all of the functionality has been implemented
- Does not discover missing program logic

Black-Box Testing

- Black-box testing is a method of software testing that examines the functionality of an application without peering into its internal structures or workings
- This method of test can be applied virtually to every level of software testing
  - unit
  - integration
  - system and
  - acceptance
- Test cases are designed using only *functional specification* of the software
  - Without any knowledge of the internal structure of the software
- For this reason, black-box testing is also known as *functional testing* or *specification-based testing*

White-Box Testing

- White-box testing is a method of software testing that tests internal structures or workings of an application, as opposed to its functionality
- In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases
- Designing white-box test cases
  - Requires knowledge about the *internal structure* of software
- White-box testing is also called *structural testing*

Grey-Box Testing

- Grey-box testing is a combination of white-box testing and black-box testing
- The aim of this testing is to search for the defects if any due to improper structure or improper usage of applications

Black-Box Testing

There are essentially two main approaches to design black box test cases

- Equivalence class partitioning
  - Input values to a program are partitioned into equivalence classes
  - Partitioning is done such that
    - Program behaves in similar ways to every input value belonging to an equivalence class
    - Test the code with just one representative value from each equivalence class – As good as testing using any other values from the equivalence classes
- Boundary value analysis
  - Some typical programming errors occur
    - At boundaries of equivalence classes
    - Might be purely due to psychological factors
  - Programmers often fail to see
    - Special processing required at the boundaries of equivalence classes

Equivalence Class Partitioning

How do you determine the equivalence classes?

- Examine the input data – Few general guidelines for determining the equivalence classes can be given
    - If the input data is specified by a range of values
        - For example, numbers between 1 to 5000
        - One valid and two invalid equivalence classes are defined
    - If input is an enumerated set of values
        - For example, { a, b, c }
        - One equivalence class for valid input values
        - Another equivalence class for invalid input values should be defined
    - A program reads an input value in the range of 1 and 5000
        - Computes the square root of the input number
        - One valid and two invalid equivalence classes are defined – The set of negative integers, Set of integers in the range of 1 and 5000, and Integers larger than 5000
        - A possible test suite can be: { -5, 500, 6000 }

Equivalence Class Partitioning

- Max program reads two non-negative integers and spits the larger one

Equivalence Class	Condition	Test Case
EC 1 (Greater)	$x > y$	(5, 2)
EC 2 (Smaller)	$x < y$	(3, 7)
EC 3 (Equal)	$x = y$	(4, 4)

- QES program reads (a, b, c) and solves: $ax^2 + bx + c = 0$

Equivalence Class	Condition	Test Case
Infinite roots	$a = b = c = 0$	(0,0,0)
No root	$a = b = 0; c \neq 0$	(0,0,2)
Single root	$a = 0, b \neq 0$	(0,2,-4)
Repeated roots	$a \neq 0; b * b - 4 * a * c = 0$	(4,4,1)
Distinct roots	$a \neq 0; b * b - 4 * a * c > 0$	(1,-5,6)
Complex roots	$a \neq 0; b * b - 4 * a * c < 0$	(2,3,4)

Boundary Value Analysis

- Some typical programming errors occur
  - At boundaries of equivalence classes
  - Might be purely due to psychological factors
- Programmers often fail to see
  - Special processing required at the boundaries of equivalence classes
- Programmers may improperly use $<$ instead of $\leq$
- Boundary value analysis
  - Select test cases at the boundaries of equivalence classes
- For a function that computes the square root of an integer in the range of 1 and 5000
  - Test cases must include the values: $\{ 0, 1, 5000, 5001 \}$
- QES program reads (a, b, c) and solves: $ax^2 + bx + c = 0$
  - $a = 0$ is a boundary. Check if this test works well
  - $b * b - 4 * a * c = 0$ is a boundary. Check for the test

White-Box Testing Strategies

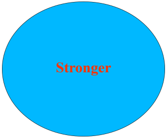
- Coverage-based
  - Design test cases to cover certain program elements
- Fault-based
  - Design test cases to expose some category of faults
- There exist several popular white-box testing methodologies
  - Statement coverage
  - Branch coverage
  - Condition coverage
  - Path coverage
  - MC/DC coverage
  - Mutation testing
  - Data flow-based testing

Coverage-Based Testing Versus Fault-Based Testing

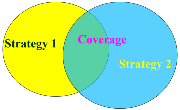
- Idea behind coverage-based testing
  - Design test cases so that certain program elements are executed (or covered)
  - Example: statement coverage, path coverage, etc.
- Idea behind fault-based testing
  - Design test cases that focus on discovering certain types of faults
  - Example: Mutation testing

Stronger, Weaker, and Complementary Testing

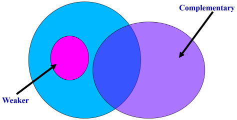
- Stronger and Weaker Testing: Test cases are a super-set of a weaker testing
  - A stronger testing covers at least all the elements of the elements covered by a weaker testing



- Complimentary Testing



- Stronger, Weaker & Complimentary Testing



Statement Coverage

- Statement coverage methodology
  - Design test cases so that every statement in the program is executed at least once
- The principal idea
  - Unless a statement is executed
  - We do not know if an error exists in that statement
- Observe that a statement behaves properly for one i/p
  - No guarantee that it will behave correctly for all i/p values
- Coverage measurement
  - $\frac{\#executed\ \ statements}{\#statements}$
  - Rationale: a fault in a statement can only be revealed by executing the faulty statement.

    Consider Euclid's GCD algorithm:

```
int f1(int x, int y) {
    while (x != y) {
        if (x>y)
            x = x - y;
        else y = y - x;
    }
    return x;
}
```

- By choosing the test set { `(x=3,y=3)`, `(x=4,y=3)`, `(x=3,y=4)` }, all statements are executed at least once

Branch Coverage

- Test cases are designed such that
  - Different branch conditions – are given true and false values in turn
- Branch testing guarantees statement coverage
  - A stronger testing compared to the statement coverage-based testing
  - Why?

```
1: cin >> x;
2: if (0 == x)
3:     x = x + 1;
4: y = 5;
```

Note that, {`(x = 0)`} covers lines {`1, 2, 3, 4`} while {`(x = 1)`} covers only lines {`1, 2, 4`}. So with {`(x = 0)`}, we get 100% statement coverage. But then, did we check for the jump from line 2 to 4 for the false condition? This condition did not get tested. So we need {`(x = 0), (x = 1)`} for 100% branch coverage and it obviously leads to 100% statement coverage.

How do we get 100% branch coverage for:

```
if (true)
    x = x + 1;
y = 5;
```

Branch Coverage

- Example:

```
0: int f1(int x, int y) {
1:     while (x != y) {
2:         if (x > y)
3:             x = x - y;
4:         else y = y - x;
5:     }
6:     return x;
7: }
```

Branches: {1-2, 1-6, 2-3, 2-4, 3-5, 4-5, 5-1}

- Test cases for branch coverage can be
  - (x=3,y=3): {1-6}: {1, 6}
  - (x=4,y=3): {1-2, 2-3, 3-5, 5-1}: {1, 2, 3, 5}
  - (x=3,y=4): {1-2, 2-4, 4-5, 5-1}: {1, 2, 4, 5}
- *Adequacy criterion*: Each branch (in CFG) must be executed at least once
  - Coverage $= \frac{\#executed\_branches}{\#branches}$
- Traversing all edges of a graph causes all nodes to be visited
  - So test suites that satisfy the branch adequacy criterion for a program 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)

All Branches can still miss conditions

- Sample fault: missing operator (negation)

  `digit_high == 1 || digit_low == -1`

- Branch adequacy criterion can be satisfied by varying only `digit_low`
  - The faulty sub-expression might not be tested
  - Even though we test both outcomes of the branch

Condition Coverage

- Test cases are designed such that
  - Each component of a composite conditional expression
    - Given both true and false values
  - Consider the conditional expression
    - `((c1.and.c2).or.c3)`
  - Each of `c1`, `c2`, and `c3` are exercised at least once
    - That is, given true and false values
- Basic condition testing
  - Adequacy criterion: each basic condition must be executed at least once
- Coverage
  - $\frac{\#truth\ \ values\ \ taken\ \ by\ \ all\ \ basic\ \ conditions}{2 * \#basic\ \ conditions}$

Branch Testing

- Branch testing is the simplest condition testing strategy
  - Compound conditions appearing in different branch statements
    - Are given true and false values
  - Condition testing
    - Stronger testing than branch testing
  - Branch testing
    - Stronger than statement coverage testing

Condition Coverage

Software Engineering

Partha P Das

Fundamentals

Verification & Validation

Black Box Testing

White Box Testing

- Consider a boolean expression having n components
  - For condition coverage we require 2^n test cases
- Condition coverage-based testing technique
  - Practical only if n (the number of component conditions) is small
- Commonly known as **Multiple Condition Coverage** (MCC), **Multicondition Coverage** and **Condition Combination Coverage**

Modified Condition / Decision (MC/DC)

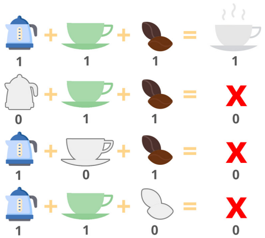
- **Motivation**
  - Effectively test important combinations of conditions, without exponential blowup in test suite size
  - *Important* combinations means: Each basic condition shown to independently affect the outcome of each decision
- **Requires**
  - For each basic condition C, two test cases obtained
  - 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
- **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, that is, the outcome of a decision changes as a result of changing a single condition

MC/DC in action: The Cup of Coffee Example

To make a cup of coffee, we would need ALL of the following: a kettle, a cup and coffee. If any of the components were missing, we would not be able to make our coffee. Or, to express this another way:

```
if (kettle && cup && coffee)
    return cup_of_coffee;
else
    return false;
```

Or to illustrate it visually:



Test	Inputs			Outputs
	Kettle	Mug	Coffee	Result
1	0	0	0	0
2	0	0	1	0
3	0	1	0	0
4	0	1	1	0
5	1	0	0	0
6	1	0	1	0
7	1	1	0	0
8	1	1	1	1

- **Tests 4 & 8 demonstrate that 'kettle' can independently affect the outcome**
- **Tests 6 & 8 demonstrate that 'mug' can independently affect the outcome**
- **Tests 7 & 8 demonstrate that 'coffee' can independently affect the outcome**

Source: What is MC/DC?

MC/DC in action: Flow Check

A sample C/C++ function with a decision composed of OR and AND expressions illustrates the difference between Modified Condition/Decision Coverage and a coverage of all possible combinations as required by MCC:

```
bool isSilent(int *line1, int *line2)
{
    if ((!line1 || *line1 <= 0) && (!line2 || *line2 <= 0))
        return true;
    else
        return false;
}
```

Or to illustrate it visually:

Source: Modified Condition/Decision Coverage (MC/DC)

Modified Condition / Decision (MC/DC)

Cond = `(((a || b) && c) || d) && e`

Condition Coverage Test Cases

Every condition in the decision has taken all possible outcomes at least once

#	a	b	c	d	e	Cond
0:	F	F	F	F	F	F
1:	F	F	F	F	T	F
2:	F	F	F	T	F	F
3:	F	F	F	T	T	T
4:	F	F	T	F	F	F
5:	F	F	T	F	T	F
6:	F	F	T	T	F	F
7:	F	F	T	T	T	T
8:	F	T	F	F	F	F
9:	F	T	F	F	T	F
10:	F	T	F	T	F	F
11:	F	T	F	T	T	T
12:	F	T	T	F	F	F
13:	F	T	T	F	T	T
14:	F	T	T	T	F	F
15:	F	T	T	T	T	T

#	a	b	c	d	e	Cond
16:	T	F	F	F	F	F
17:	T	F	F	F	T	F
18:	T	F	F	T	F	F
19:	T	F	F	T	T	T
20:	T	F	T	F	F	F
21:	T	F	T	F	T	T
22:	T	F	T	T	F	F
23:	T	F	T	T	T	T
24:	T	T	F	F	F	F
25:	T	T	F	F	T	F
26:	T	T	F	T	F	F
27:	T	T	F	T	T	T
28:	T	T	T	F	F	F
29:	T	T	T	F	T	T
30:	T	T	T	T	F	F
31:	T	T	T	T	T	T

MC/DC Coverage Test Cases

Every condition in the decision independently affects the decision's outcome

#	a	b	c	d	e	Cond	Cases
1:	T	X	T	X	T	T	21, 23, 29, 31
2:	F	T	T	X	T	T	13, 15
3:	T	X	F	T	T	T	19, 27
4:	T	X	T	X	F	F	20, 22, 28, 30
5:	T	X	F	F	X	F	16, 17, 24, 25
6:	F	F	X	F	X	F	0, 1, 4, 5

Modified Condition / Decision (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)
- MC/DC code coverage criterion is commonly used software testing. For example, DO-178C software development guidance in the aerospace industry requires MC/DC for the most critical software level (DAL A).
- **MC/DC vs. MCC**
  - MCC testing is characterized as number of tests $= 2^C$. In coffee example we have 3 conditions (kettle, cup and coffee) therefore tests $= 2^3 = 8$
  - MC/DC requires significantly fewer tests (C + 1). In coffee example we have 3 conditions, therefore $3 + 1 = 4$
  - In a real-world setting, most aerospace projects would include some decisions with 16 conditions or more. So the reduction would be from $2^16 = 65{,}536$ to $16 + 1 = 17$. That is, $65{,}519/65{,}536 = 99.97\%$

Source: What is MC/DC?

Different Types of Code Coverage

Coverage Criteria	SC	DC	MC/DC	MCC
Every statement in the program has been invoked at least once	X			
Every point of entry and exit in the program has been invoked at least once		X	X	X
Every control statement (that is, branch-point) in the program has taken all possible outcomes (that is, branches) at least once		X	X	X
Every non-constant Boolean expression in the program has evaluated to both a True and False result		X	X	X
Every non-constant condition in a Boolean expression in the program has evaluated to both a True and False result			X	X
Every non-constant condition in a Boolean expression in the program has been shown to independently affect that expression's outcome			X	X
Every combination of condition outcomes within a decision has been invoked at least once				X

- **SC**: Statement Coverage • **DC**: Decision Coverage
- **MC/DC**: Modified Condition / Decision Coverage • **MCC**: Multiple Condition Coverage

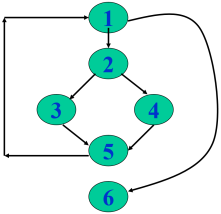
Source: What is MC/DC?

Path Coverage

- Design test cases such that
  - All linearly independent paths in the program are executed at least once
- Defined in terms of
  - Control flow graph (CFG) of a program
- To understand the path coverage-based testing
  - we need to learn how to draw control flow graph of a program
- A control flow graph (CFG) describes
  - The sequence in which different instructions of a program get executed
  - The way control flows through the program
- 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

Path Coverage: CFG

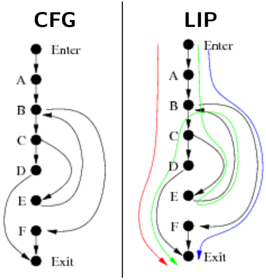
```
int f1(int x,int y) {
1 while (x != y){
2    if (x>y) then
3          x=x-y;
4    else y=y-x;
5 }
6 return x;        }
```



- A path through a program:
  - A node and edge sequence from the starting node to a terminal node of the control flow graph
  - There may be several terminal nodes for program
- Any path through the program that introduces at least one new edge (not included in any other independent path) is a *Linearly Independent Path* (LIP).
- A set of paths are linearly independent if none of them can be created by combining the others in some way.
  - It is straight forward to identify linearly independent paths of simple programs; but not so for complicated programs
- LIP in the above example:
  - 1,6
  - 1,2,3,5,1,6
  - 1,2,4,5,1,6

Path Coverage: LIP

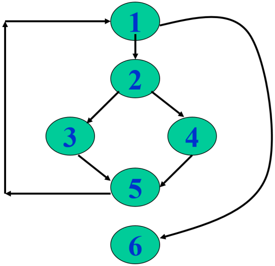
```
    public static boolean isPrime(int n) {
A       int i = 2;
B       while (i < n) {
C           if (n % i == 0) {
D               return false
            }
E           i++;
        }
F       return true;
    }
```



Source: The 'Linearly Independent Paths' Metric for Java

Path Coverage: McCabe's Cyclomatic Metric

- An upper bound for the number of linearly independent paths of a program – a practical way of determining the maximum number of LIP
- Given a control flow graph G, cyclomatic complexity $V(G)$:
  - $V(G) = E - N + 2$
  - N is the number of nodes in G
  - E is the number of edges in G
- Alternately, inspect control flow graph to determine number of bounded areas (any region enclosed by a nodes and edge sequence) in the graph
  $V(G) =$ Total number of bounded areas $+ 1$
- Example: Cyclomatic complexity $= 7 - 6 + 2 = 3 = 2 + 1$



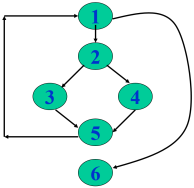
Path Coverage: McCabe's Cyclomatic Metric

- 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
- The first method of computing $V(G)$ is amenable to automation:
  - You can write a program which determines the number of nodes and edges of a graph
  - Applies the formula to find $V(G)$
- 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
- A measure of the number of independent paths in a program
- Provides a lower bound
  - for the number of test cases for path coverage
- 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

Path Coverage: Practical Path Testing

- Tester proposes initial set of test data using her experience & judgement
- A dynamic program analyzer is used to measures which parts of the program have been tested
- Result used to determine when to stop testing
- Derivation of Test Cases
  - Draw control flow graph.
  - Determine V(G).
  - Determine the set of linearly independent paths.
  - Prepare test cases to force execution along each path
- Example: Number of independent paths: 3

```
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;        }
```

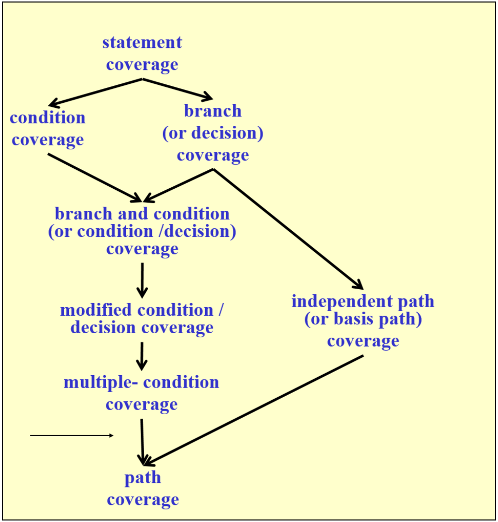


  - 1,6: test case (x=1, y=1)
  - 1,2,3,5,1,6: test case (x=1, y=2)
  - 1,2,4,5,1,6: test case (x=2, y=1)

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

White-Box Testing : Summary



Mutation Testing

- The software is first tested:
  - Using an initial testing method based on white-box strategies we already discussed
- After the initial testing is complete
  - mutation testing is taken up
- The idea behind mutation testing
  - Make a few arbitrary small changes to a program at a time
- Good software development organizations:
  - Restrict cyclomatic complexity of functions to a maximum of ten or so
- Insert faults into a program:
  - Check whether the tests pick them up
  - Either validate or invalidate the tests
  - Example:

```
1: cin >> x;
2: if (0 == x)
3:     x = x + 1;
4: y = 5;
```

Note that, {(x = 0)} covers lines {1, 2, 3, 4} while {(x = 1)} covers only lines {1, 2, 4}. So with {(x = 0)}, we get 100% statement coverage. But then, did we check for the jump from line 2 to 4 for the false condition? This condition did not get tested. So we need {(x = 0), (x = 1)} for 100% branch coverage and it obviously leads to 100% statement coverage.

How do we get 100% branch coverage for:

```
1: if (true)
2:     x = x + 1;
3: y = 5;
```

Mutation Testing

- Insert faults into a program:
  - Check whether the tests pick them up
  - Either validate or invalidate the tests
- Each time the program is changed
  - it is called a **mutated program**
  - the change is called a **mutant**
- A mutated program:
  - Tested against the full test suite of the program
- If there exists at least one test case in the test suite for which:
  - A mutant gives an incorrect result, then the mutant is said to be **dead**
- If a mutant remains **alive**:
  - even after all test cases have been exhausted, the test suite is enhanced to kill the mutant
- The process of generation and killing of mutants
  - can be automated by pre-defining a set of primitive changes that can be applied to the program
- The primitive changes can be
  - Deleting a statement
  - Altering an arithmetic operator
  - Changing the value of a constant
  - Changing a data type, etc.

Data Flow-Based Testing

Data Flow Testing is a type of structural testing. It is a method that is used to find the test paths of a program according to the locations of definitions and uses of variables in the program

- It is concerned with:
  - Statements where variables receive values
  - Statements where these values are used or referenced
- To illustrate the approach of data flow testing, assume that each statement in the program assigned a unique statement number. For a statement number S:
  - DEF(S) = $\{X \mid$ statement S contains a definition of X$\}$
  - USE(S)= $\{X \mid$ statement S contains a use of X$\}$
  - Example: 1: a = b; DEF(1) = $\{a\}$, USE(1) = $\{b\}$
  - Example: 2: a = a + b; DEF(2) = $\{a\}$, USE(2) = $\{a,b\}$
- If a statement is a loop or if condition then its DEF set is empty and USE set is based on the condition of statement s.
- Data Flow Testing uses the control flow graph to find the situations that can interrupt the flow of the program.
- Reference or define anomalies in the flow of the data are detected at the time of associations between values and variables. These anomalies are:
  - A variable is defined but not used or referenced
  - A variable is used but never defined
  - A variable is defined twice before it is used

Data Flow-Based Testing

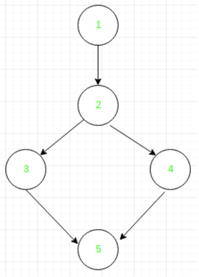
- **Advantages** of Data Flow Testing
  - To find a variable that is used but never defined
  - To find a variable that is defined but never used
  - To find a variable that is defined multiple times before it is use
  - Deallocating a variable before it is used
- **Disadvantages** of Data Flow Testing
  - Time consuming and costly process
  - Requires knowledge of programming languages

Data Flow-Based Testing

- Example:

```
1. read x, y;
2. if (x > y)
3. a = x + 1
   else
4. a = y - 1
5. print a;
```

- CFG



- Define/use of variables

Variable	Defined at node	Used at node
x	1	2, 3
y	1	2, 4
a	3, 4	5

Source: Data Flow Testing

Data Object Categories

- (d) Defined, Created, Initialized. An object (like variable) is defined when it:
  - appears in a data declaration
  - is assigned a new value
  - is a file that has been opened
  - is dynamically allocated
  - ...
- (k) Killed, Undefined, Released
- (u) Used:
  - (c) Used in a calculation
  - (p) Used in a predicate
  - An object is used when it is part of a computation or a predicate
    - A variable is used for a computation (c) when it appears on the RHS (sometimes even the LHS in case of array indices) of an assignment statement
    - A variable is used in a predicate (p) when it appears directly in that predicate

Source: Topics in Software Dynamic White-box Testing: Part 2: Data-flow Testing

Data Flow-Based Testing: Definition and Use

```
1.   read (x, y);
2.   z = x + 2;
3.   if (z < y)
4        w = x + 1;
     else
5.       y = y + 1;
6.   print (x, y, w, z);
```

Def	*C-use*	*P-use*
x, y		
z	x	
		z, y
w	x	
y	y	
	x, y, w, z	

- To find a variable that is used but never defined
- To find a variable that is defined but never used
- To find a variable that is defined multiple times before it is use
- Deallocating a variable before it is used