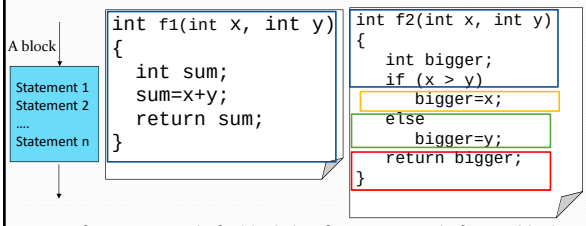
White-box Testing Techniques

Software Testing and Quality Assurance
Joe Timoney

CFG - Definition: Basic Block

Sequence of statements, such that the only entrance to the block is through the first statement, and the only exit from the block is through the last statement



Function f1 is composed of 1 block, but f2 is composed of many blocks

Basic principles

- Based on the analysis of the software under test (SUT) 's implementation (unit, module, system)
- Test against the implementation
- Develop test cases derived from the implementation
- Exercise the implementation
- White-Box testing is usually used after Black-box testing in order to improve coverage of the internal components that form the program

Definition: Control Flow Graph

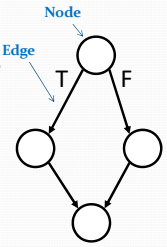
- Represent the flow of control
- Directed graph G(V, E)
  - V is set of vertices
  - E is set of edges, E = V*V
- The granularity of the vertices can be an operation (e.g. assign), a statement or a basic block
- The edges are directed; direction indicates flow of control from one vertex to another

Measurement of Coverage

- Using tools analyze the code execution
- Recording instructions have been executed (coverage): percentage of lines executed or branches taken during the tests
- These measurement tools can be used to measure the coverage provided by Black-Box testing, and white-box testing can then be used to augment the coverage of the software by the tests.

Control Flow Graphs

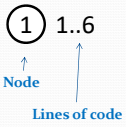
- Each node represents a block (i.e. one or more statement/lines of code)
- Each edge represents a 'jump'
- Two exits ~ a decision (True or False)



Example of CFG

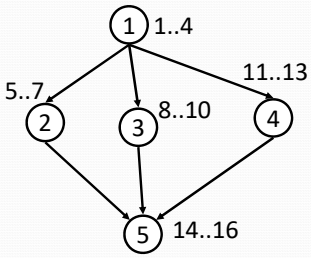
CFG/Sequence

```
1 int f(int x, int y)
2 {
3   int sum;
4   sum=x+y;
5   return sum;
6 }
```



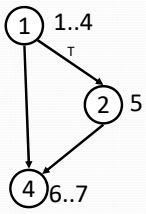
CFG/Selection (switch)

```
01 int f(int a)
02 {
03   int x;
04   switch (a) {
05     case 0:
06       x=0;
07       break;
08     case 1:
09       x=1;
10       break;
11     otherwise:
12       x=2;
13       break;
14   }
15   return x;
16 }
```



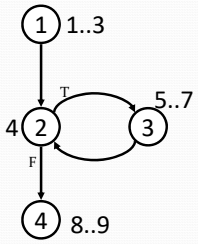
CFG/Selection (if-then)

```
1 int f(int a)
2 {
3   int x=0;
4   if (a>10)
5     x=a;
6   return x;
7 }
```



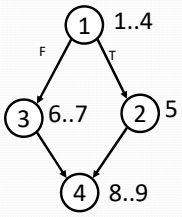
CFG/Iteration (while)

```
1 int f(int a)
2 {
3   int x=0;
4   while (a>0){
5     x++;
6     a--;
7   }
8   return x;
9 }
```



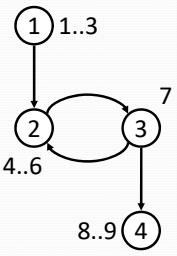
CFG/Selection (if-then-else)

```
1 int f(int a)
2 {
3   int x;
4   if (a>10)
5     x=a;
6   else
7     x=10;
8   return x;
9 }
```



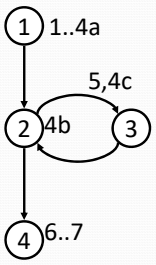
CFG/Iteration (do-while)

```
1 int f(int a)
2 {
3   int x=0;
4   do {
5     x++;
6     a--;
7   } while (a>0);
8   return x;
9 }
```



CFG/Iteration (for)

```
01 int f(int a)
02 {
03   int x=0,y=0;
04   for (x=0; x<a; x++)
05     y=y+x;
06    return y;
07  }
```



Example Code 2

A pump storage power station should start generating if
Capacity > 75 and demand>=trigger, or
Capacity > 75 and demand>90, or
Capacity >50 and demand>95 , or
Capacity>80 and demand>current

Tips

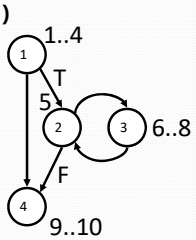
- Identify all the "jump" points (decisions):
  if, while, switch/case, for
- Start at the top of the code
- Work your way down to the next jump point
- Create a new node
- For each decision identify the destination node if (a) True and (b) false
- Connect the nodes

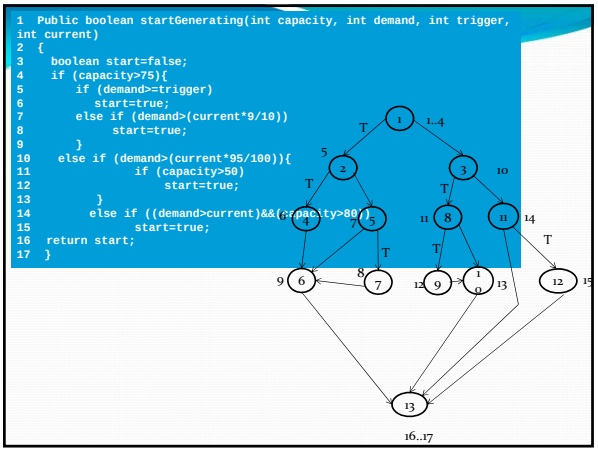
Example Code 2

```
1  Public boolean startGenerating(int capacity, int
demand, int trigger, int current)
2  {
3    boolean start=false;
4    if (capacity>75){
5      if (demand>=trigger)
6         start=true;
7      else if (demand>(current*9/10))
8         start=true;
9     }
10    else if (demand>(current*95/100)){
11            if (capacity>50)
12               start=true;
13         }
14         else if ((demand>current)&&(capacity>80))
15               start=true;
16   return start;
17  }
```

Example Code 1

```
1  int multiply(int a, int b)
2  {
3    int v=0;
4    if ((a>0)&&(b>0))
5      while (a>0) {
6        v=v+b;
7        a--;
8      }
9    return v;
10 }
```





Data Flow Testing

- All the previous analysis techniques are concerned with the flow of control through the program.
- An alternative is to analyse the flow of data through a program.
- A program accepts inputs, performs computations, assigns new values to variables, and returns results.
- Data values "flow" from one statement to another: a data value produced in one statement will be used by a later statement.

Data Flow Anomaly

Example 1:

…

x = f1(y);

x = f2(z);

…

- Type 1: Defined and then defined again. Four interpretations of Example 1
  - a. The first statement is redundant
  - b. The first statement has a fault -- the intended one might be: w = f1(y).
  - c. The second statement has a fault – the intended one might be: v = f2(z).
  - d. There is a missing statement in between the two: v = f3(x).
  - Note: It is for the programmer to make the desired interpretation.

Data Flow Testing

- There are two possible uses for variables (parameters, local variables, and attributes) in a program:
  - they can be assigned a new value (*definition*),
  - they can be referred to in an expression (*use*).
- Each possible path from the definition of a variable to its subsequent use is referred to as a d-u pair.
- Not all paths from every definition to every use are possible, so it is useful to refer to the pairs as *candidate* d-u pairs until this has been confirmed.

Data Flow Anomaly

- Type 2: Undefined but referenced
  - Example: x = x – y – w;

    /* w has not been defined by the programmer. */
  - Two interpretations
    - The programmer made a mistake in using w.
    - The programmer wants to use the compiler assigned value of w.
- Type 3: Defined but not referenced
  - Example: Consider x = f(a, b). If x is not used subsequently, we have a Type 3 anomaly.

Data Flow Anomalies

- Analysis of d-u pairs can identify problems with data referred to as *Data Flow Anomalies*. There are three types of anomaly:
  - 1. Defined and then re-defined before use.
  - 2. Used before defined.
  - 3. Defined but not used subsequently

Data Flow Testing

- Data Flow Testing is just below Path testing and above Branch testing in terms of its coverage strength relative to these other methods

Data Flow Increment & Decrement

Note: a **definition** is the assignment of a value to a variable, including assignment at function entry. A **use** is the reading of the value from a variable.

Increment and decrement operations cause a **use** followed by a **definition**.

Example

```
(1) int factorial(int n) {
(2)    int result=1;
(3)     for (i=2; i<=n; i++) {
(4)       result = result * i;
(5)    }
(6)    return result;
(7) }
```

D-u pairs of result

d-u pair	d	u
1	2	4
2	2	6
3	4	4
4	4	6

Example

```
(1) int factorial(int n) {
(2)    int result=1;
(3)    for (i=2; i<=n; i++) {
(4)       result = result * i;
(5)    }
(6)    return result;
(7) }
```

Strengths and Weaknesses of White Box Testing Techniques

Example

- Concentrating on the variable *result*
- It is defined on line 2
- It is defined and used on line 4, and used on line 6
- The definition on line 4 takes place on the left-hand side of the assignment statement and this definition is used on the right-hand side *only after one iteration of the loop*- it is used some time after it has been defined

Statement Testing

- The goal of Statement Coverage is to make sure that every line of code has been executed during testing. The ideal test completion criteria is 100% statement coverage.
- First of all, draw a CFG for the program. Each node on the CFG, representing a sequence of indivisible source code statements, is a test case. Then working from the CFG, and the source code, work out input values required to ensure that every source code statement is executed.
- Finally, using the specification-NOT the source code-work out the correct output for each set of input values.

Statement Testing Comment

- The level of statement coverage achieved during testing is usually shown as a percentage-for example, 100% statement coverage means that every statement has been executed, 50% means that only half the statements have been covered.
- Normally, each line of code is regarded as a statement.
- Statement Coverage is the weakest form of white-box testing. 100% statement coverage will probably not cover all the logic of a program, or cause all the possible output values to be generated.
- Note that the analysis phase of Statement Coverage testing allows unreachable code to be identified. (this is code that cannot be executed by any combination of input data and is a sign of a design error or poor maintenance)

Statement Testing Weaknesses

▪Does force the execution of every statement but is not a very demanding technique, particularly for compound logical conditions;

▪does not provide coverage for the "NULL else"

```
if (number < 3) ++number;
```

Here Statement Coverage does not force a test case for number >=3

Statement Testing Strengths

▪ Statement testing achieves the minimum level of code coverage, it is recommended that at least every statement should be executed once before software is released

▪ Statement Coverage can generally be achieved using only a small number of tests.

Statement Testing Weaknesses

- Another example where statement testing would not uncover a fault is

  if (a>1) || (b=0) **then** x = x/a;

- The test case (a=2, b=0) covers both the **if** and the following assignment

- However, in this case a fault in the program logic, such as an 'and' being used instead of 'or', would not be detected

Statement Testing Weaknesses

▪However, it may be impossible to achieve 100% statement coverage. This is usually due to the presence of code that can only be executed in exceptional or dangerous circumstances that are hard to reproduce. Do inspection instead.

▪ Does not specify how to calculate the input values

Branch Testing

▪Generate test data to exercise the true and false outcome of every decision

▪ A Control flow graph (cfg) is helpful in representing the program.

Branch Testing Comment

- Branch coverage ensures that each output path from each "decision" is tested at least once.
- Note that 100% branch coverage guarantees 100% statement coverage-but the test data is harder to generate.
- Branch Coverage is a stronger form of testing than Statement Coverage, but it still does not exercise either all the reasons for taking each branch, or combinations of different branches taken.

Path Coverage

▪ Generate test data to exercise all the possible paths from entry to exit in a program. This is called “path coverage”

▪The goal is to achieve 100% coverage of every start-to-finish path in the code.

Branch Testing Strengths/Weaknesses

▪Branch testing is one level up from statement testing in the degree of coverage

▪Same problems in achieving 100% coverage

▪Resolves the “NULL else” problem

▪Undemanding of compound conditions:

To achieve a **true** outcome all is needed is to give ch a value of x, Ignoring the rest of the condition

```
if ( ch == 'x' || (b < 0.0012 && c))
      <some code>
else
      <some code>
```

Path Coverage Test Cases and Data

- Each unique path from start to finish is a test case.
- Test data is selected to ensure that every path is followed. This selection requires the tester to review the code carefully, and select input parameter values that cause these paths to be executed.

Branch Testing Strengths/Weaknesses

- This measure will include coverage of switch-statement cases, and exception handlers if they exist in the program.

Path Coverage Strengths/Weaknesses

▪Strength: Path testing can be difficult to realize for complex programs-some paths may not be possible.

▪ Strength: combinations of paths are exercised that other methods do not achieve

▪ Weakness: the conditions within the decisions are not explicitly exercised

▪ Weakness -- lots of computation if the program is complex

▪ Weakness: technique is not readily applicable to unstructured programs

Data Flow Coverage

▪Generate test data that follows the pattern of **data definition** & **use** through the program.

▪Originally used to statically detect anomalies in the code; (e.g., referencing an undefined variable)

▪The goal is to achieve 100% coverage of every possible du-pair in the code.

DU-Pairs Strengths

▪ Comment: The du pair testing technique provides comprehensive testing of all the Definition-Use Paths in a program, but generating the test data is a very time consuming exercise.

▪ Strength: strong form of testing

▪ Strength: generates test data in the pattern that data is manipulated in the program rather than following "artificial" branches

Data Flow Coverage Test Cases

- Every possible du pair is a test case. Some du pairs will be impossible to execute-these are best handled as "candidate" du pairs, and not assigned a test case.
- Each test case should be given a unique identifier (for example: DU-1-1 for pair 1 for variable 1, or DU-x-1 where x is the name of the variable).
- It is essential to list the candidate du pairs in the code, and then give test case identifiers to the possible ones. As the test data is worked out, this table may need updating.

DU-Pairs Weakness

- Number of test cases can be very large. It can be up to

$$\sum_{i=1}^{N} d_i u_i$$

where d_i is the number of definitions for variable i and u_i is its number of uses, and N the number of variables.

- With object references or pointer variables (in C) it can be difficult to determine which variable is being referenced

Data Flow Coverage Test Data

- Test data is selected to ensure that every du pair is executed.
- This selection requires the tester to review the code carefully, and select input parameter values that cause the path from every definition to every use to be followed

Weakness: DU- Pairs for Array Elements

▪ An Array is a problem since it's difficult to determine which element in the array is being used.

▪For example, its index can be changing dynamically as the program executes.

▪Solution: treat the entire array as one variable ==> one du-pair

Test Ranking