Test Case Design Methods- White Box

What subset of all possible test cases has the highest probability of detecting the most errors?

References

- [4] – Chapter 1, 4
- [5] – Chapter 17
- [6] – Chapter 10, 11

Content

- Overview
- Basis Path Testing
- Control-flow/Coverage Testing
- Loop Testing
- Data Flow Testing
- Limitation

Overview

- White box testing involves looking at the structure of the code
- Why spend time and energy worrying about (and testing) logical minutiae when we might better expend effort ensuring that program requirements have been met?
  - Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed
  - We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis
  - Typographical errors are random.

Overview

- The general white box testing process:
  - The (Software Under Test) SUT's implementation is analyzed.
  - Paths through the SUT are identified.
  - Inputs are chosen to cause the SUT to execute selected paths. This is called path sensitization. Expected results for those inputs are determined.
  - The tests are run.
  - Actual outputs are compared with the expected outputs.
  - A determination is made as to the proper functioning of the SUT

Overview

- White box testing is more than code testing—it is path testing
  - Generally, the paths that are tested are within a module (Unit Testing, Module Testing)
  - Test paths between modules within subsystems, between subsystems within systems, and even between entire systems?

Content

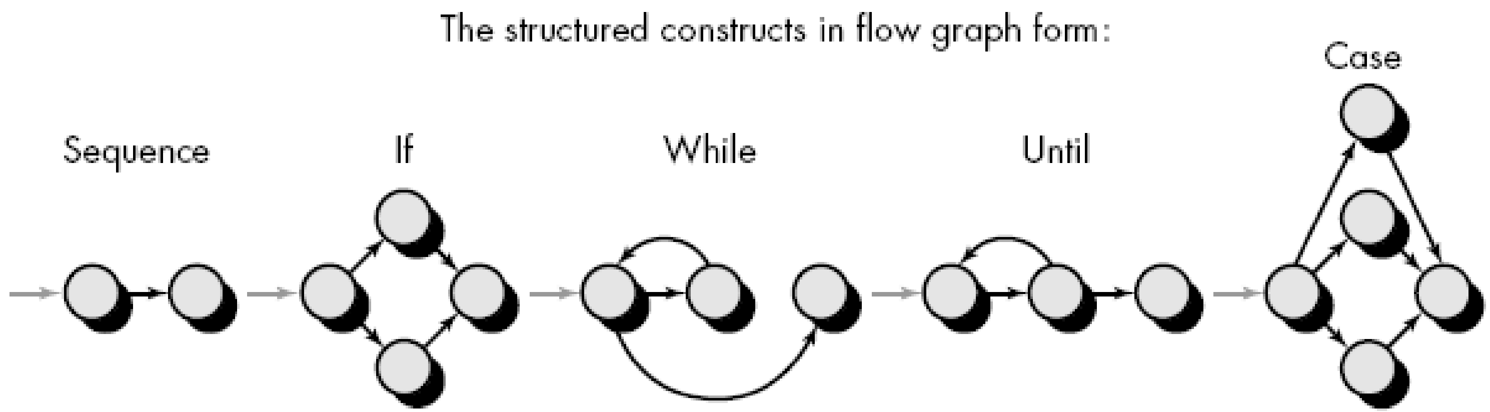
- Overview
- Basis Path Testing
- Control-flow/Coverage Testing
- Loop Testing
- Data Flow Testing
- Limitation

Basis Path Testing

- Flow Graph Notation
- Cyclomatic Complexity
- Deriving Test Cases
- Graph Matrices

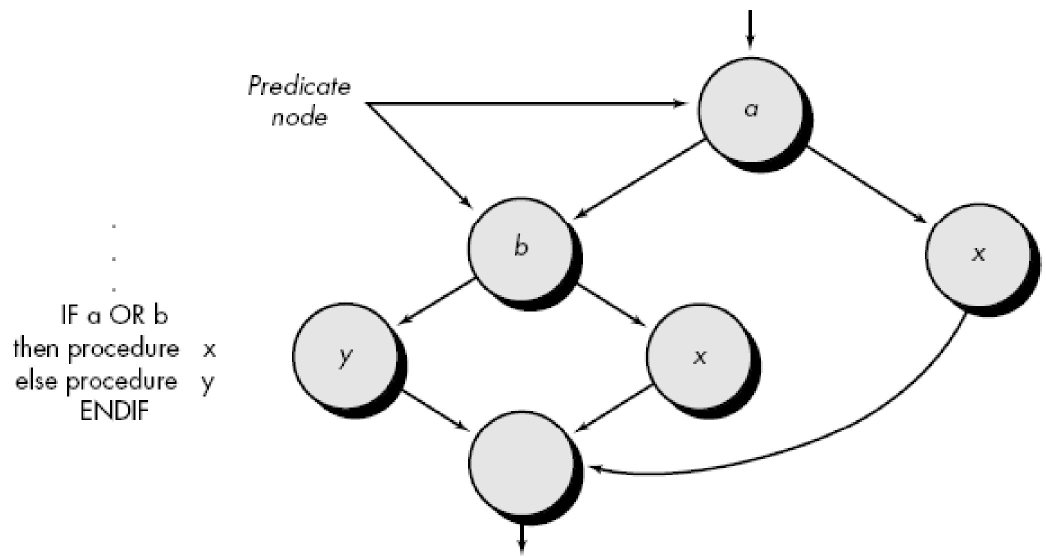
Flow Graph Notation

- Flow graph (or program graph):
  - A simple notation for the representation of control flow.
  - Depicts logical control flow using the following notation



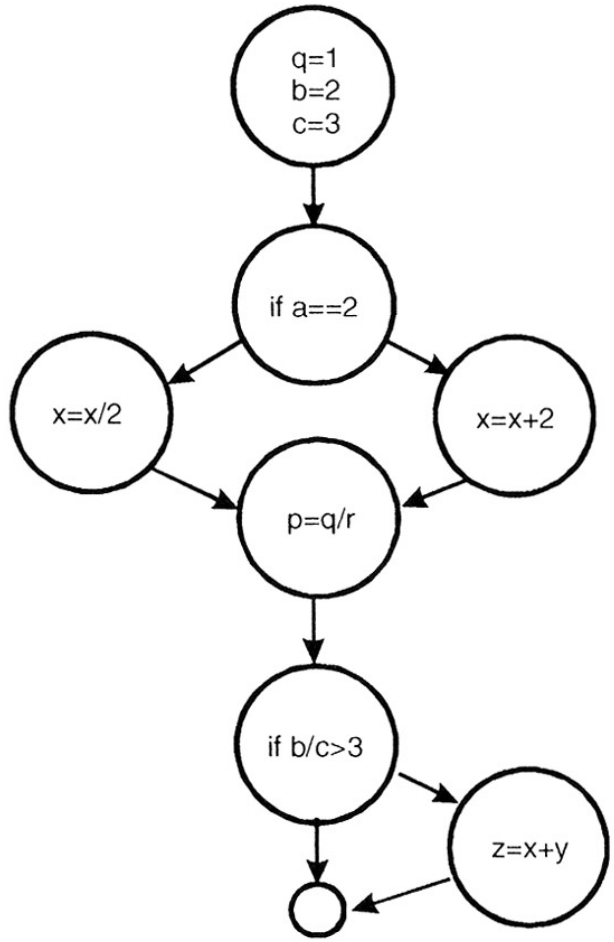
Flow Graph Notation

- Flow graph (or program graph):
  - A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement.
  - Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it



Flow Graph Notation

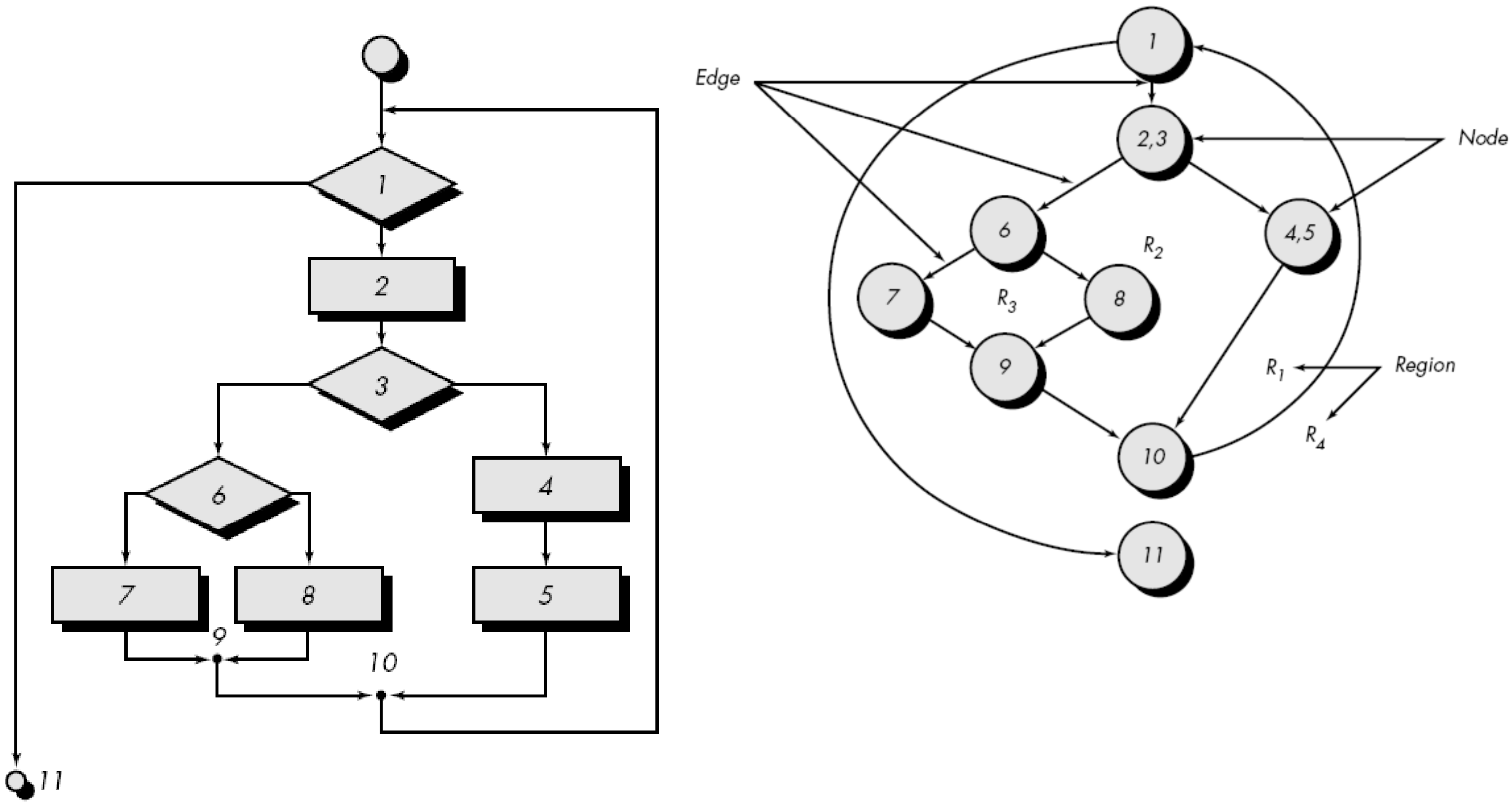
```
q=1;
b=2;
c=3;
if (a==2) {x=x+2;}
else {x=x/2;}
p=q/r;
if (b/c>3) {z=x+y;}
```



Cyclomatic Complexity

- An **independent path**
  - Any path through the program that introduces at least one new set of processing statements or a new condition
  - Must move along at least one edge that has not been traversed before the path is defined

Cyclomatic Complexity



Cyclomatic Complexity

- Independent Paths
  - path 1: 1-11
  - path 2: 1-2-3-4-5-10-1-11
  - path 3: 1-2-3-6-8-9-10-1-11
  - path 4: 1-2-3-6-7-9-10-1-11
- Paths 1, 2, 3, and 4 constitute a basis set for the flow graph
  - If tests can be designed to force execution of these paths (a basis set):
    - Every statement in the program will have been guaranteed to be executed at least one time
    - Every condition will have been executed on its true and false sides

Cyclomatic Complexity

- Cyclomatic complexity, $V(G)$, for a flow graph, G, is defined as $V(G) = E - N + 2$ (E is the number of flow graph edges, N is the number of flow graph nodes)
- If all decisions in the graph are binary,
  - $V(G) = P + 1$ (P is the number of predicate nodes contained in the flow graph G)
- The value computed for cyclomatic complexity
  - Defines the number of independent paths in the basis set of a program
  - Basis path testing provides a minimum, lower-bound on the number of test cases that need to be written

Deriving Test Cases

- Using the design or code as a foundation, draw a corresponding flow graph
- Determine the cyclomatic complexity of the resultant flow graph
- Determine a basis set of linearly independent paths
- Prepare test cases that will force execution of each path in the basis set

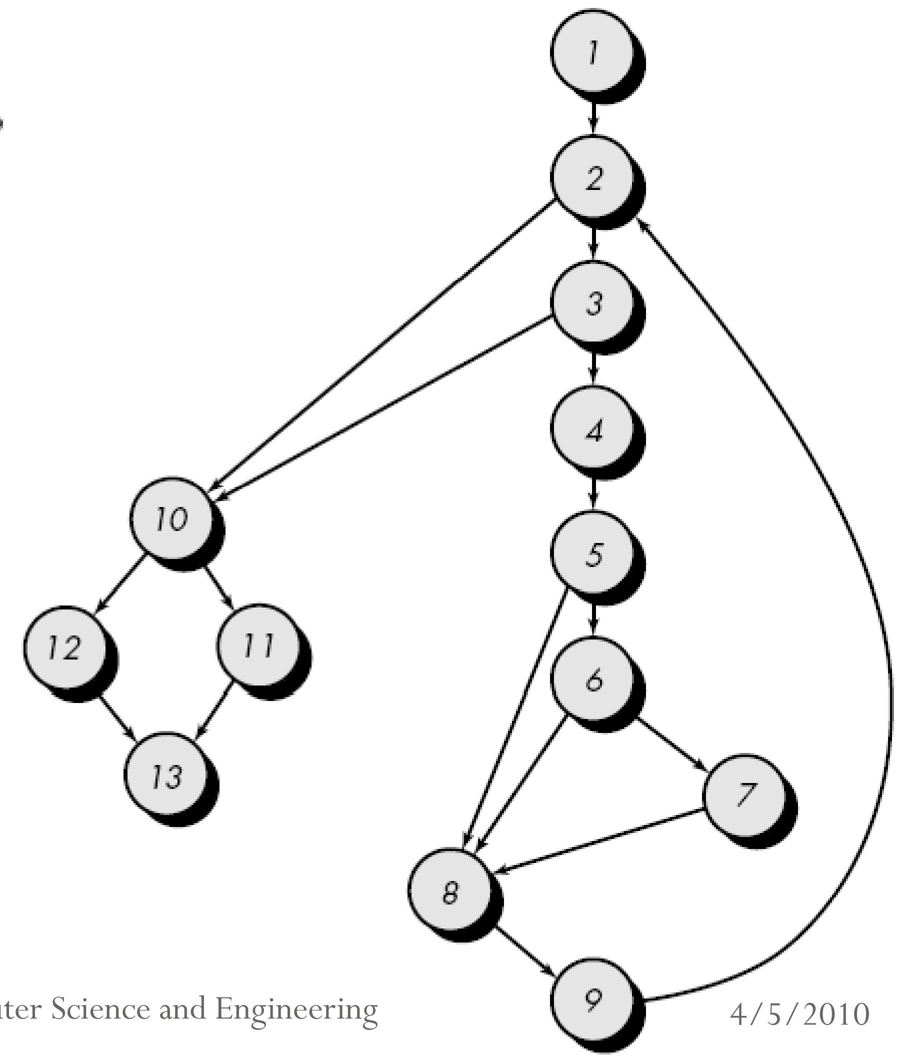
Deriving Test Cases

```
PROCEDURE average;

  *  This procedure computes the average of 100 or fewer
     numbers that lie between bounding values; it also computes the
     sum and the total number valid.

   INTERFACE RETURNS average, total.input, total.valid;
   INTERFACE ACCEPTS value, minimum, maximum;

   TYPE value[1:100] IS SCALAR ARRAY;
   TYPE average, total.input, total.valid;
      minimum, maximum, sum IS SCALAR;
   TYPE i IS INTEGER;
     i = 1;
 1   total.input = total.valid = 0;     2
     sum = 0;
   DO WHILE value[i] <> -999 AND total.input < 100   3
 4   increment total.input by 1;
     IF value[i] > = minimum AND value[i] < = maximum   6
 5       THEN increment total.valid by 1;
   7          sum = s sum + value[i]
         ELSE skip
 8   ENDIF
     increment i by 1;
 9 ENDDO
   IF total.valid > 0   10
 11  THEN average = sum / total.valid;
 12  ELSE average = -999;
 13 ENDIF
END average
```



Deriving Test Cases

V(G) = 5 predicate nodes + 1 = 6

path 1: 1-2-10-11-13

path 2: 1-2-10-12-13

path 3: 1-2-3-10-11-13

path 4: 1-2-3-4-5-8-9-2-. . .

path 5: 1-2-3-4-5-6-8-9-2-. . .

path 6: 1-2-3-4-5-6-7-8-9-2-. . .

Deriving Test Cases

Path 1 test case:

value(k) = *valid input, where* $k < i$ *for* $2 \le i \le 100$

value(i) = *999 where* $2 \le i \le 100$

Expected results: Correct average based on k values and proper totals.

Note: Path 1 cannot be tested stand-alone but must be tested as part of path 4, 5, and 6 tests.

Path 2 test case:

value(1) = 999

Expected results: Average = 999; other totals at initial values.

Path 3 test case:

Attempt to process 101 or more values.

First 100 values should be valid.

Expected results: Same as test case 1.

Path 4 test case:

value(i) = *valid input where* $i < 100$

value(k) < *minimum where* $k < i$

Expected results: Correct average based on k values and proper totals.

Path 5 test case:

value(i) = *valid input where* $i < 100$

value(k) > *maximum where* $k <= i$

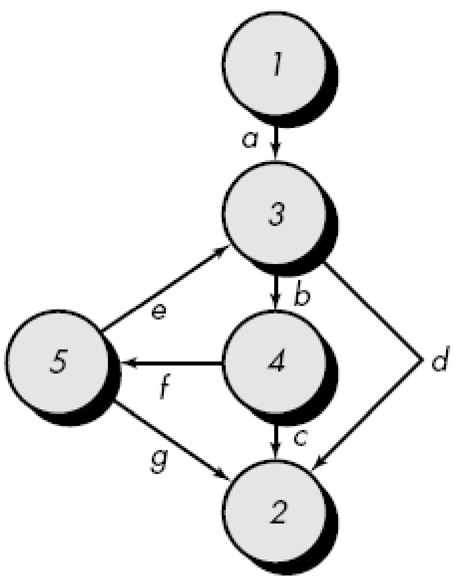
Expected results: Correct average based on n values and proper totals.

Path 6 test case:

value(i) = *valid input where* $i < 100$

Expected results: Correct average based on n values and proper totals.

Graph Matrices



Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Node \ Connected to node	1	2	3	4	5	Connections
1			1			1 – 1= 0
2						
3		1		1		2 – 1 = 1
4		1			1	2 – 1 = 1
5		1	1			2 – 1 = 1

3 + 1 = 4 ← Cyclomatic complexity

Graph matrix

Content

- Overview
- Basis Path Testing
- Control-flow / Coverage Testing
- Loop Testing
- Data Flow Testing
- Limitation

Control-flow/Coverage Testing

- Control flow of the program
- Consider various aspects of this flow graph in order to ensure that we have an adequate set of test cases
- Coverage is a measure of the completeness of the set of test cases
  - Method Coverage
  - Statement Coverage
  - Branch Coverage
  - Condition Coverage

Method Coverage

- A measure of the percentage of methods that have been executed by test cases.
- Objective: 100% method coverage
- Examples
  - Test Case 1: foo(0, 0, 0, 0, 0.), expected return value of 0.

```
int foo (int a, int b, int c, int d, float e)  {
      float e;
      if (a == 0)  {
           return 0;
      }
      int x = 0;
      if ((a==b) OR ((c == d) AND bug(a) )) {
           x=1;
      }
       e = 1/x;
 return e;
  }
```

Statement Coverage

- Statement coverage is a measure of the percentage of statements that have been executed by test cases
- Objective: 100% statement coverage
- Examples:
  - In Test Case 1, the program statements on lines 1-5 out of 12 lines of code are executed => 42% (5/12) statement coverage.
  - Add Test Case 2: the method call foo(1, 1, 1, 1,1.), expected return value of 1 => 100% statement coverage

```
1    int foo (int a, int b, int c, int d, float e)  {
2          float e;
3          if (a == 0)  {
4               return 0;
5          }
6          int x = 0;
7          if ((a==b) OR ((c == d) AND bug(a) )) {
8               x=1;
9          }
10          e = 1/x;
11    return e;
12     }
```

Branch Coverage

- Branch coverage is a measure of the percentage of the decision points (Boolean expressions) of the program have been evaluated as both true and false in test cases
- For decision/branch coverage, we evaluate an entire Boolean expression as one true-or-false predicate even if it contains multiple logical-and or logical-or operators
- Objective:
  - 100% Branch Coverage
  - Only 50% branch coverage is practical in very large systems of 10 million source lines of code or more (Beizer, 1990).
- Examples:

```
3     if (a == 0) {
7     if ((a==b) OR ((c == d) AND bug(a) )) {
```

Branch Coverage

Line	Predicate	True	False
3	(a == 0)	Test Case 1 foo(0, 0, 0, 0, 0) return 0	Test Case 2 foo(1, 1, 1, 1, 1) return 1
7	((a==b) OR ((c == d) AND bug(a)))	Test Case 2 foo(1, 1, 1, 1, 1) return 1	

- Branch Coverage: 75%
- Test Case 3: foo(1, 2, 1, 2, 1) => 100% Branch Coverage (uncovers a previously undetected division-by-zero problem on line 10!)

Condition Coverage

- Condition coverage is a measure of percentage of Boolean sub-expressions of the program that have been evaluated as both true or false outcome [applies to compound predicate] in test cases
- There are no industry standard objectives for condition coverage
- Example:

```
7     if ((a==b) OR ((c == d) AND bug(a) )) {
```

Condition Coverage

Predicate	True	False
(a==b)	Test Case 2 foo(1, 1, x, x, 1) return value 0	Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
(c==d)		Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
bug(a)		

- Condition Coverage: 50%
  - (c==d) -> TRUE: never been tested
  - short-circuit Boolean has prevented the method bug(int) from ever being executed

Condition Coverage

Predicate	True	False
(a==b)	Test Case 2 foo(1, 1, x, x, 1) return value 0	Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
(c==d)	Test Case 4 foo(1, 2, 1, 1, 1) return value 1	Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
bug(a)	Test Case 4 foo(1, 2, 1, 1, 1) return value 1	Test Case 5 foo(3, 2, 1, 1, 1) division by zero!

- bug(a)
  - If a > 1 return FALSE
  - Else return TRUE

Content

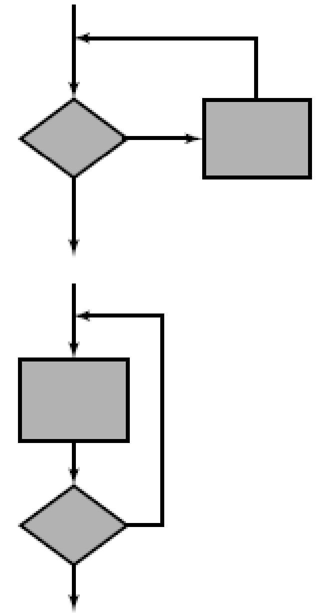
- Overview
- Basis Path Testing
- Control-flow/Coverage Testing
- Loop Testing
- Data Flow Testing
- Limitation

Loop Testing

- Focuses exclusively on the validity of loop constructs.
- Types of Loop
  - Simple loops
  - Nested Loops
  - Concatenated Loops
  - Unstructured Loops

Simple loops

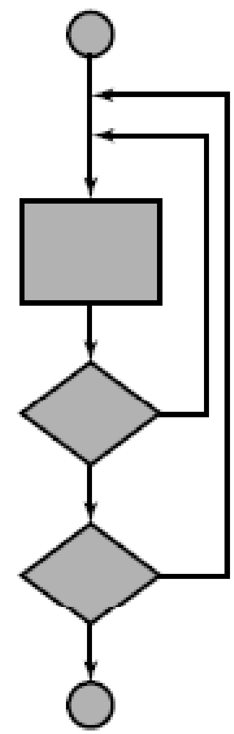
- The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop
  - Skip the loop entirely.
  - Only one pass through the loop.
  - Two passes through the loop.
  - m passes through the loop where $m < n$.
  - $n - 1$, n, $n + 1$ passes through the loop



Simple loops

Nested Loops

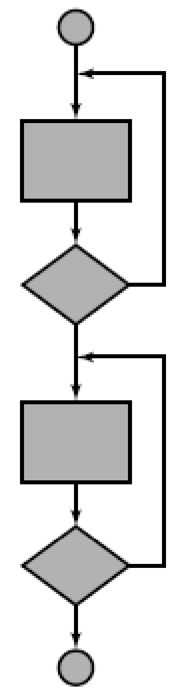
- Start at the innermost loop. Set all other loops to minimum values.
- Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
- Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
- Continue until all loops have been tested.



Nested loops

Concatenated Loops

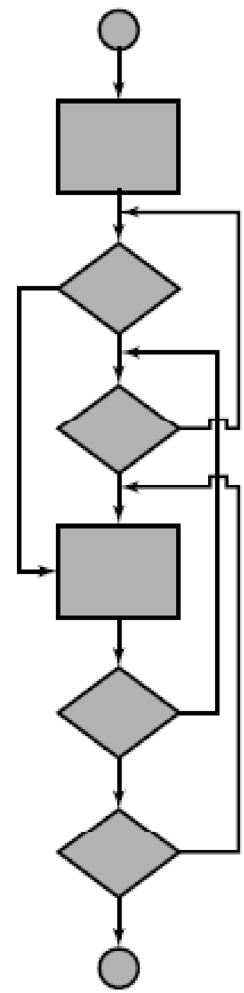
- If each of the loops is independent of the other:
  - Concatenated loops can be tested using the approach defined for simple loops,
- Else
  - the approach applied to nested loops is recommended.



Concatenated loops

Unstructured Loops

- Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs



Unstructured loops

Content

- Overview
- Basis Path Testing
- Control-flow/Coverage Testing
- Loop Testing
- Data Flow Testing
- Limitation

Data Flow Testing

- The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program
- Data flow testing is a powerful tool to detect improper use of data values due to coding errors
  - Incorrect assignment or input statement
  - Definition is missing (use of null definition)
  - Predicate is faulty (incorrect path is taken which leads to incorrect definition)

Data Flow Testing

- Variables that contain data values have a defined life cycle: created, used, killed (destroyed).
- The "scope" of the variable

```
{         // begin outer block
  int x;    // x is defined as an integer within this outer block
  ...;      // x can be accessed here
   {        // begin inner block
   int y;   // y is defined within this inner block
  ...;      // both x and y can be accessed here
   }        // y is automatically destroyed at the end of
            // this block
   ...;     // x can still be accessed, but y is gone
  }   // x is automatically destroyed
```

Data Flow Testing

- Three possibilities exist for the first occurrence of a variable through a program path:
  - ~d - the variable does not exist (indicated by the ~), then it is defined (d)
  - ~u - the variable does not exist, then it is used (u)
  - ~k - the variable does not exist, then it is killed or destroyed (k)

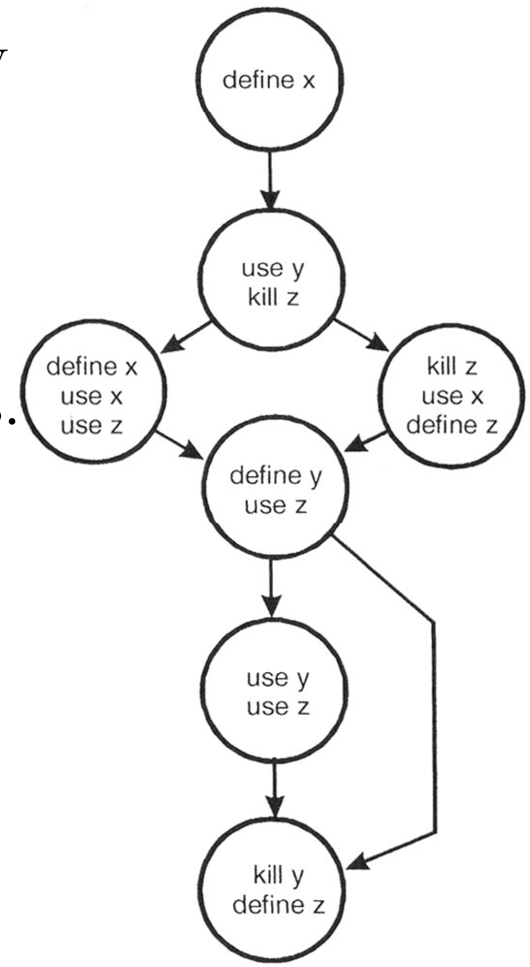
Data Flow Testing

- Time-sequenced pairs of defined (d), used (u), and killed (k):
  - dd - Defined and defined again—not invalid but suspicious. Probably a programming error.
  - du - Defined and used—perfectly correct. The normal case.
  - dk - Defined and then killed—not invalid but probably a programming error.
  - ud - Used and defined—acceptable.
  - uu - Used and used again—acceptable.
  - uk - Used and killed—acceptable.
  - kd - Killed and defined—acceptable. A variable is killed and then redefined.
  - ku - Killed and used—a serious defect. Using a variable that does not exist or is undefined is always an error.
  - kk - Killed and killed—probably a programming error

Data Flow Testing

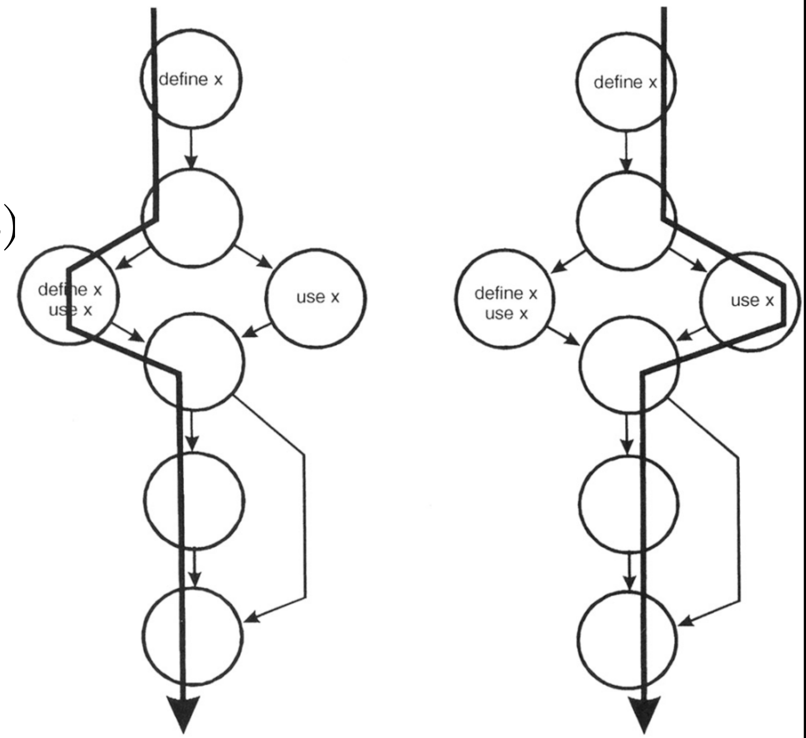
- A data flow graph is similar to a control flow graph in that it shows the processing flow through a module.

  In addition, it details the definition, use, and destruction of each of the module's variables.
- Technique
  - Construct diagrams
  - Perform a static test of the diagram
  - Perform dynamic tests on the module



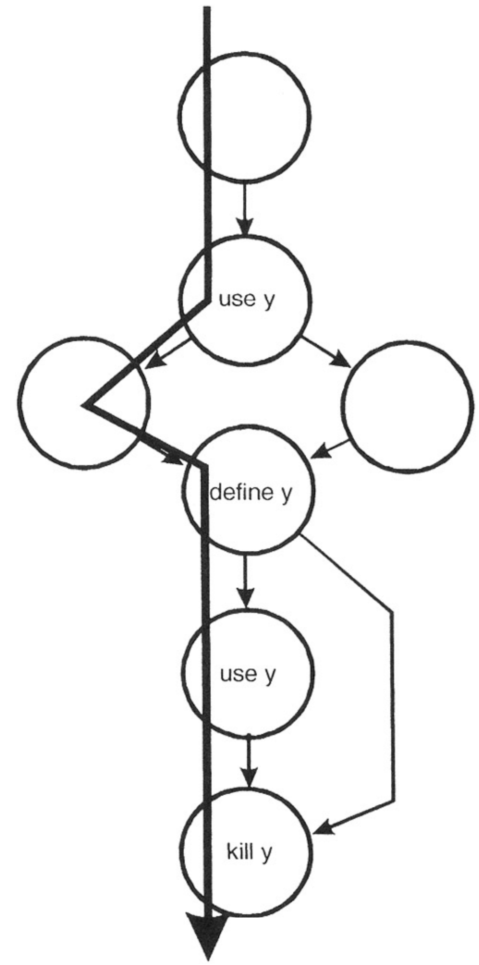
Data Flow Testing

- Perform a static test of the diagram
  - For each variable within the module we will examine define-use-kill patterns along the control flow paths
  - The define-use-kill patterns for x (taken in pairs as we follow the paths) are:
    - ~define - correct, the normal case
    - define-define - suspicious, perhaps a programming error
    - define-use - correct, the normal case
  -



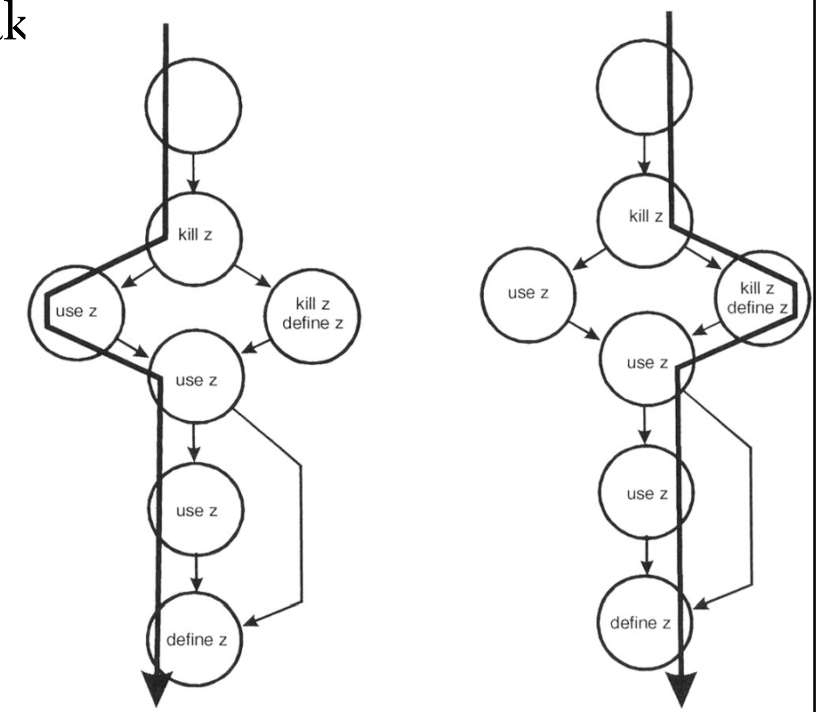
Data Flow Testing

- Perform a static test of the diagram
  - The define-use-kill patterns for y (taken in pairs as we follow the paths) are:
    - ~use - major blunder
    - use-define - acceptable
    - define-use - correct, the normal case
    - use-kill - acceptable
    - define-kill - probable programming error



Data Flow Testing

- Perform a static test of the diagram
  - The define-use-kill patterns for z (tak in pairs as we follow the paths) are:
    - ~kill - programming error
    - kill-use - major blunder
    - use-use - correct, the normal case
    - use-define - acceptable
    - kill-kill - probably a programming error
    - kill-define - acceptable
    - define-use - correct, the normal case



Data Flow Testing

- In performing a static analysis on this data flow model the following problems have been discovered:
  - x: define-define
  - y: ~use
  - y: define-kill
  - z: ~kill
  - z: kill-use
  - z: kill-kill

Data Flow Testing

- While static testing can detect many data flow errors, it cannot find them all => Dynamic Data Flow Testing
- Dynamic Data Flow Testing
  - Every "define" is traced to each of its "uses"
  - Every "use" is traced from its corresponding "define“
  - Steps
    - Enumerate the paths through the module
    - For every variable, create at least one test case to cover every define-use pair.

Limitation

- The tester must have sufficient programming skill to understand the code and its control flow.
- Can be very time consuming because of all the modules and basis paths that comprise a system

Q&A