SOFTWARE TESTING

CO3015 / CO5252

CH3. WHITE-BOX TESTING

Content

- Control flow testing
  - Execution path
  - Control flow graph
  - Control flow testing
- Data flow testing
  - Data flow testing
  - Variable state transition
  - Data flow graph

White-box testing

Old View : Colored Boxes

- Black-box testing : Derive tests from external descriptions of the software, including specifications, requirements, and design
- White-box testing : Derive tests from the source code internals of the software, specifically including branches, individual conditions, and statements
- Model-based testing : Derive tests from a model of the software (such as a UML diagram)

MDTD makes these distinctions less important.

The more general question is:

from what abstraction level do we derive tests?

White-box testing

https://www.guru99.com/white-box-testing.html

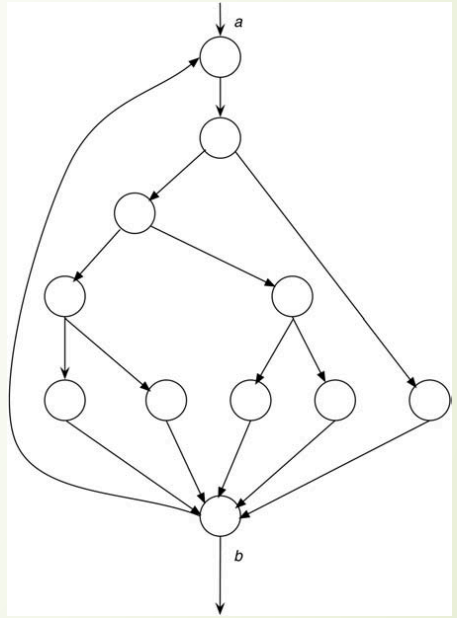
- White Box Testing is software testing technique in which internal structure, design and coding of software are tested to verify flow of input-output and to improve design, usability and security. In white box testing, code is visible to testers so it is also called Clear box testing, Open box testing, Transparent box testing, Code-based testing and Glass box testing.

The foundation of white box techniques is to execute every part of the code of the test object at least once. Flow-oriented test cases are identified, analyzing the program logic, and then they are executed. However, the

3.1. Control flow testing

Execution path

- A ordered list of statements of a unit (piece of code) to be executed from the entry point to the end point of the unit.
- A unit may have (a lot of/huge) execution paths.



- This simple program has 5^1 + ... + 5^19 + 5^20 + ... paths!
  - If we can run a path in a second,

  running (5^1 + ... 5^20) paths needs 3.2 million years!

Testing idea

is usually considered to be *exhaustive path testing*. That is, if you execute, via test cases, all possible paths of control flow through the program, then possibly the program has been completely tested.

Testing idea - problems

- The number of unique logic paths through a program could be astronomically large!
  - Impractical, if not impossible
- Even when every path in a program could be tested, yet the program might still be loaded with errors.
  - No way guarantees that a program matches its specification.
  - A program may be incorrect because of missing paths.
  - Might not uncover data-sensitivity errors.
    - if (a-b<c) vs. if ((a-b)<c)

Control flow graph

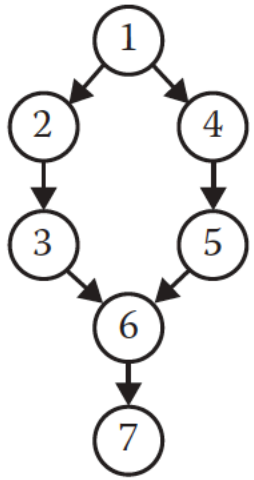
https://en.wikipedia.org/wiki/Control-flow_graph

- In computer science, a control-flow graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution.
- In a control-flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves.[3]
- Because of its construction procedure, in a CFG, every edge A→B has the property that:
  - outdegree(A) > 1 or indegree(B) > 1 (or both).[4]
- The CFG can thus be obtained, at least conceptually, by starting from the program's (full) flow graph—i.e. the graph in which every node represents an individual instruction—and performing an edge contraction for every edge that falsifies the predicate above, i.e. contracting every edge whose source has a single exit and whose destination has a single entry. This contraction-based algorithm is of no practical importance, except as a visualization aid for understanding the CFG construction, because the CFG can be more efficiently constructed directly from the program by scanning it for basic blocks.[4]

Pic from
[4] Jorgensen, Paul C. 2013. Software Testing: a Craftsman's Approach (4th ed.). Software Testing. Hoboken: CRC Press.

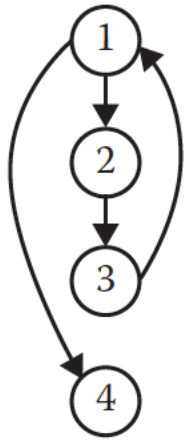
If–Then–Else

```
1  If <condition>
2     Then
3        <then statements>
4     Else
5        <else statements>
6  End If
7  <next statement>
```



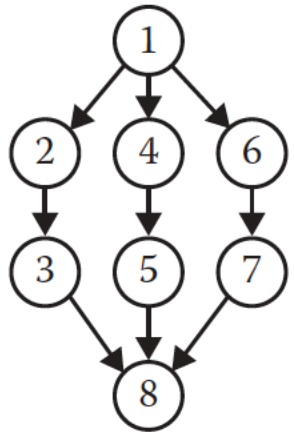
Pretest loop

```
1  While <condition>
2     <repeated body>
3  End While
4  <next statement>
```



Case/Switch

```
1  Case n of 3
2     n=1:
3        <case 1  statements>
4     n=2:
5        <case 2  statements>
6     n=3:
7        <case 3  statements>
8  End Case
```



Posttest loop

```
1  Do
2     <repeated body>
3  Until <condition>
4  <next statement>
```

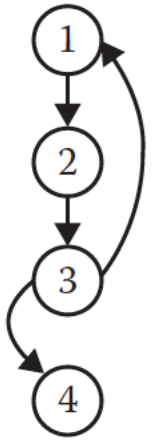


Figure 8.1 Program graphs of four structured programming constructs.

```
1  Program triangle2
2  Dim a,b,c As Integer
3  Dim IsATrinagle As Boolean
4  Output("Enter 3 integers which are sides of a triangle")
5  Input(a,b,c)
6  Output("Side A is", a)
7  Output("Side B is", b)
8  Output("Side C is", c)
9  If (a < b + c) AND (b < a + c) AND (c < a + b)
10    Then IsATriangle = True
11    Else IsATriangle = False
12 EndIf
13 If IsATriangle
14    Then  If (a = b) AND (b = c)
15             Then Output ("Equilateral")
16             Else  If (a≠b) AND (a≠c) AND (b≠c)
17                      Then Output ("Scalene")
18                      Else Output ("Isosceles")
19                EndIf
20             EndIf
21    Else  Output("Nota a Triangle")
22 EndIf
23 End triangle2
```

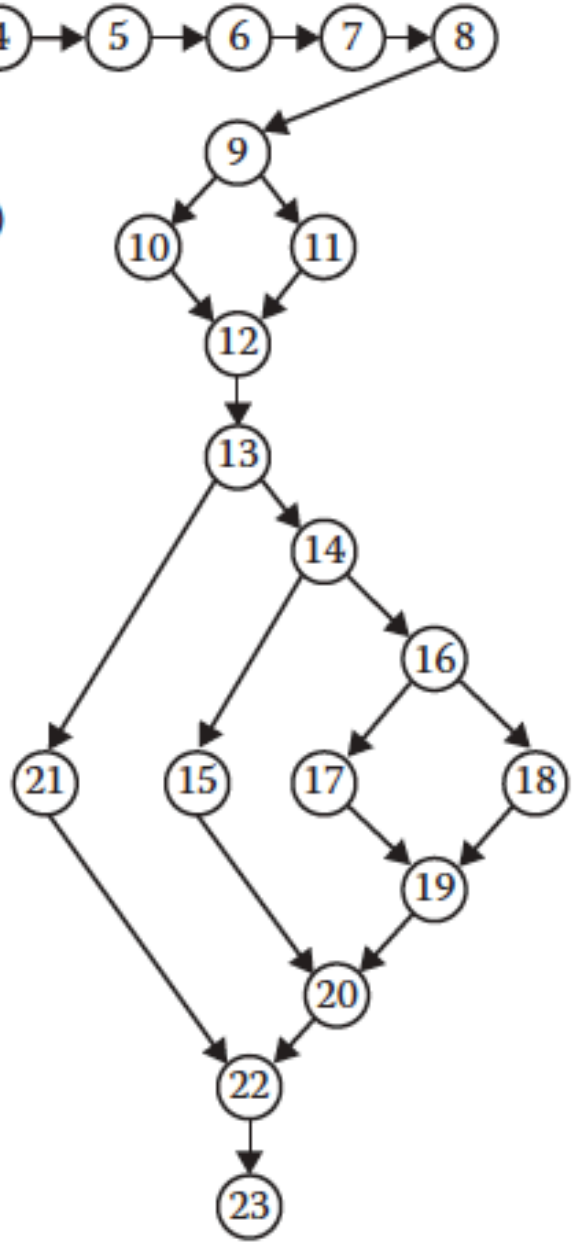


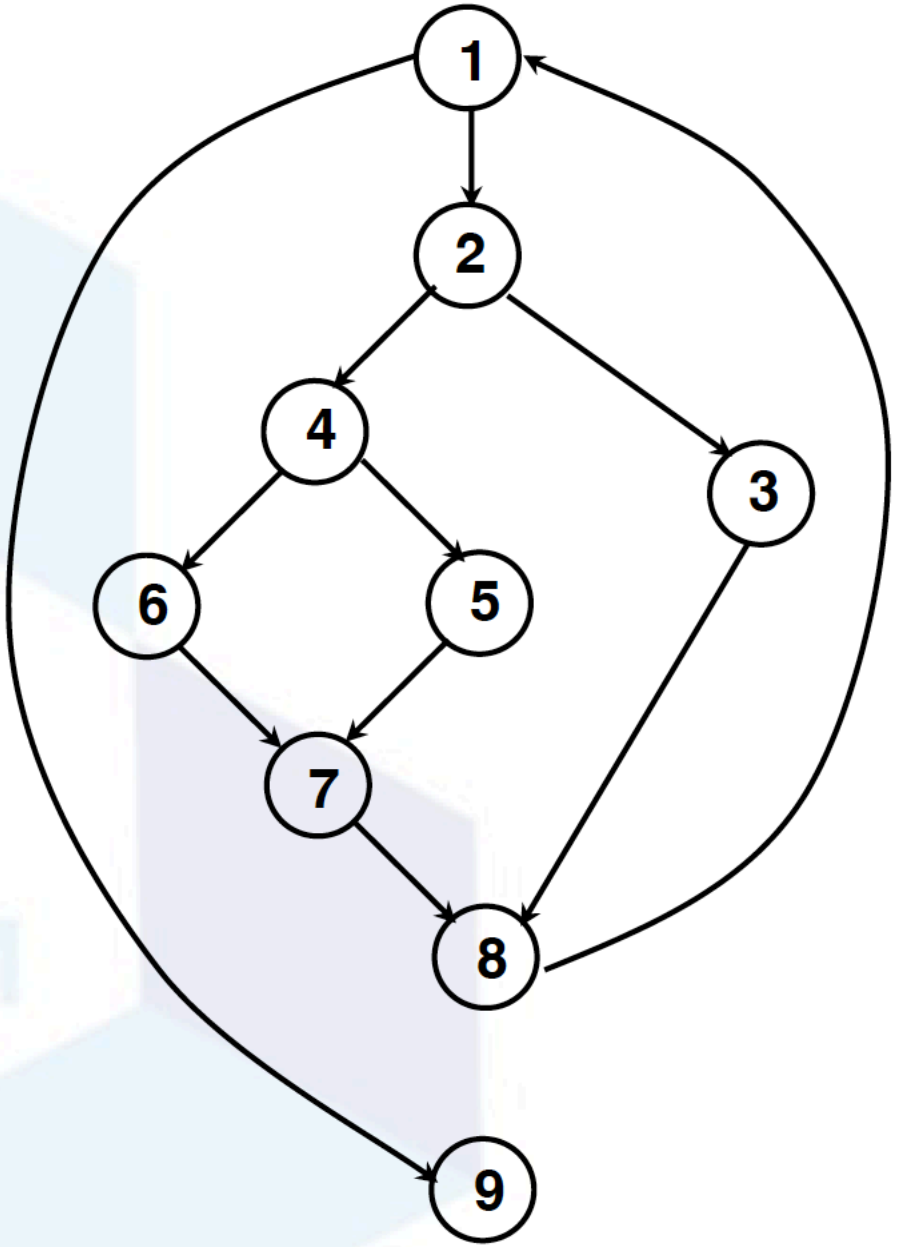
Figure 8.2 Program graph of triangle program.

```
    {
1:    while (x == 0)
      {
2:      if (y == 0)
3:        z = 0;
        else
        {
4:        if (k == 0)
5:          z = 1;
          else
6:          x = 1;
7:      }
8:    }
9: }
```

CFG?

From old slides

```
    {
1:    while (x == 0)
      {
2:      if (y == 0)
3:        z = 0;
        else
        {
4:        if (k == 0)
5:          z = 1;
          else
6:          x = 1;
7:      }
8:    }
9: }
```



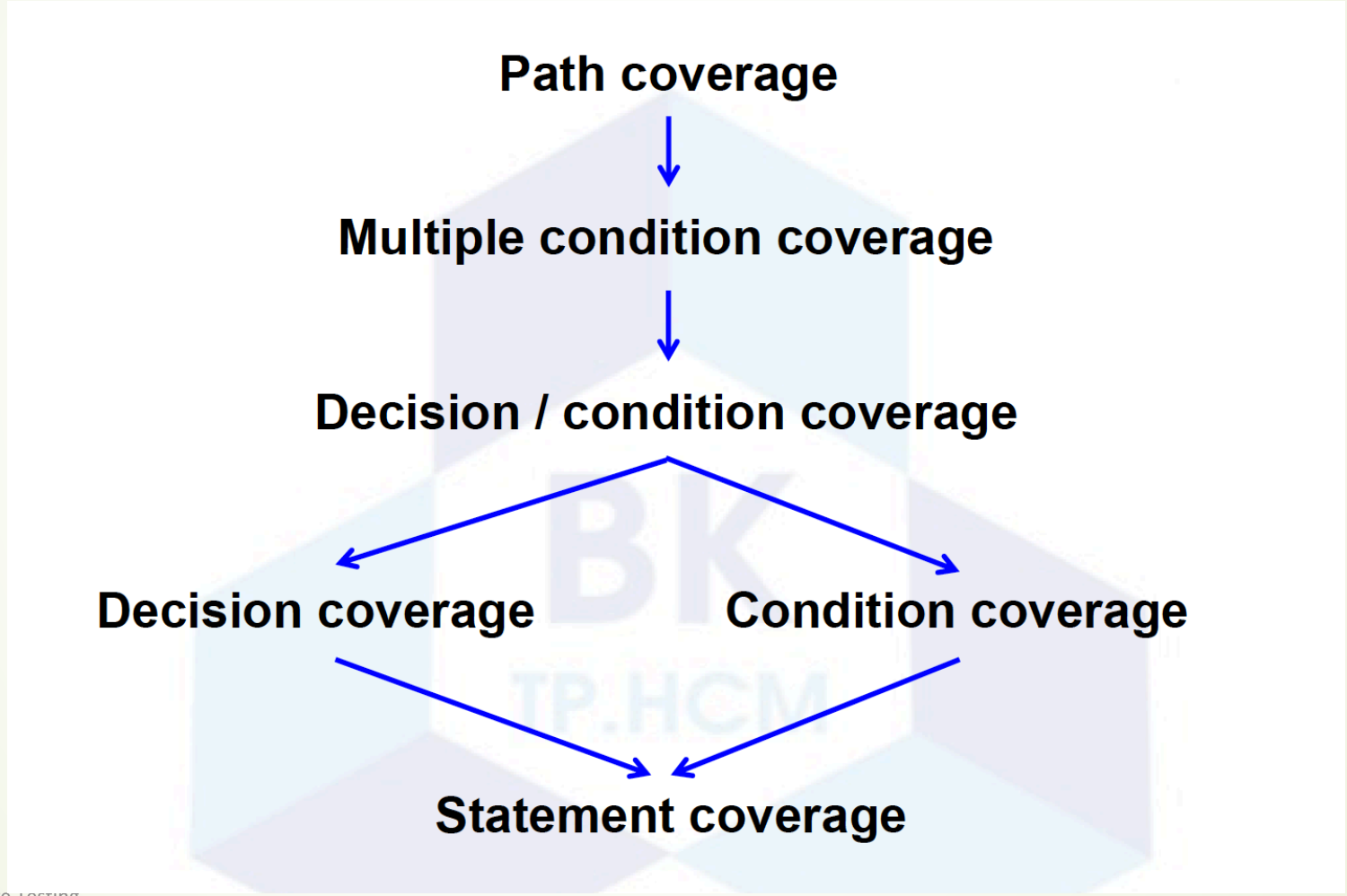
Control flow testing

http://www.mccabe.com/pdf/mccabe-nist235r.pdf

Chapter 8

Path Testing

From old slides



Summary of Control-flow testing

- Execution path:
  - A ordered list of statements of a unit (piece of code) to be executed from the entry point to the end point of the unit.
- Testing idea:
  - If you execute all possible paths of control flow, then possibly the program has been completely tested.
  - Problem:
    - The number of unique logic paths through a program could be astronomically large!
    - Even when every path in a program could be tested, yet the program might still be loaded with errors.
      - No way guarantees that a program matches its specification.
      - A program may be incorrect because of missing paths.
      - Might not uncover data-sensitivity errors.

Summary of Control-flow testing

- DD-Path = “decision to decision” path
- DD-Path Graph = graph of DD-paths
  - Same cyclomatic complexity
- Code-Based Test Coverage Metrics
  - C0: Every statement
  - C1: Every DD-Path
  - C1p: Every predicate outcome
  - C2: C1 coverage + loop coverage
  - Cd: C1 coverage +every pair of dependent DD-Paths
  - CMCC: Multiple condition coverage
  - Cik: Every program path that contains up to k repetitions of a loop (usually k = 2)
  - Cstat: "Statistically significant" fraction of paths
  - C∞: All possible execution paths

Summary of Control-flow testing

- Strategy for Loop Testing
  - 2 tests per loop is sufficient. (Judgment required, based on reality of the code.)
  - For nested loops:
    - Test innermost loop first
    - Then “condense” the loop into a single node (as in condensation graph, see Chapter 4)
    - Work from innermost to outermost loop
  - For concatenated loops: use Huang’s Theorem
  - For knotted loops: Rewrite! (see McCabe’s cyclomatic and essential complexity)

Summary of Control-flow testing

- Multiple Condition Testing
- Modified Condition Decision Coverage (MCDC)
  - Every statement must be executed at least once,
  - Every program entry point and exit point must be invoked at least once,
  - All possible outcomes of every control statement are taken at least once,
  - Every non-constant Boolean expression has been evaluated to both True and False outcomes,
  - Every non-constant condition in a Boolean expression has been evaluated to both True and False outcomes, and
  - Every non-constant condition in a Boolean expression has been shown to independently affect the outcomes (of the expression).
- Two strategies for MCDC
  - Rewrite the code as a decision table
  - Rewrite the code as nested If logic

Summary of Control-flow testing

- Basis Path Testing (McCabe's Baseline Method)
  - V(G) = E - N + P => the number of independent paths that must be tested.
  - Pick a "baseline" path that corresponds to normal execution. (The baseline should have as many decisions as possible.)
  - To get succeeding basis paths, retrace the baseline until you reach a decision node. "Flip" the decision (take another alternative) and continue as much of the baseline as possible.
  - Repeat this until all decisions have been flipped. When you reach V(G) basis paths, you're done.
  - If there aren't enough decisions in the first baseline path, find a second baseline and repeat steps 2 and 3.

3.2. Data flow testing

Data flow testing

- Data-flow testing is a white box testing technique that can be used to detect improper use of data values due to coding errors [5]

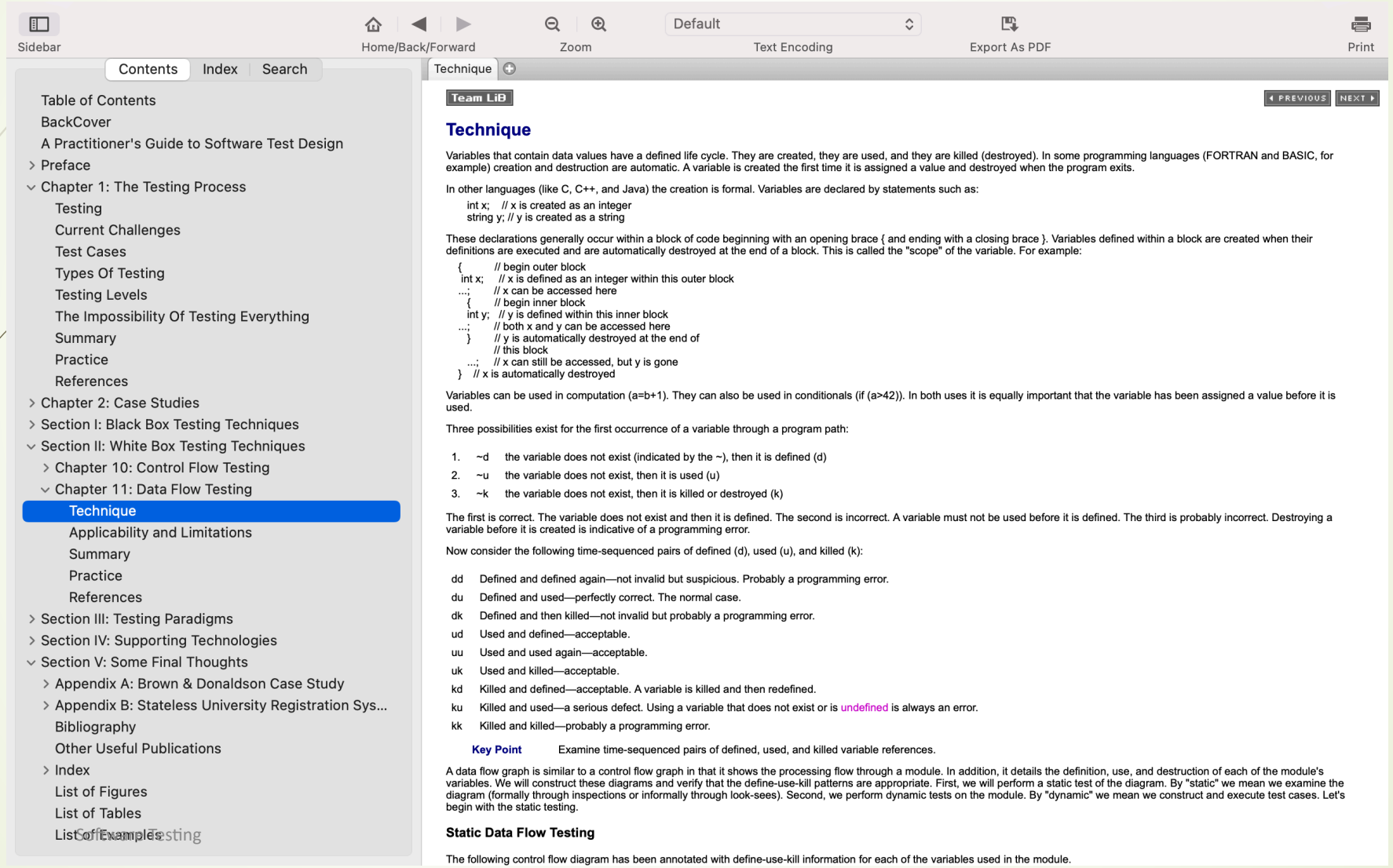
[5] Pezzè, M., & Young, M. (2008). Software testing and analysis: Process, principles, and techniques. Hoboken, N.J.: Wiley.

Data flow testing

Chapter 9

Dataflow and Slice Testing

Variable state transition (reading)



Sidebar | Home/Back/Forward | Zoom | Default (Text Encoding) | Export As PDF | Print

Contents | Index | Search

- Table of Contents
- BackCover
- A Practitioner's Guide to Software Test Design
- Preface
- Chapter 1: The Testing Process
  - Testing
  - Current Challenges
  - Test Cases
  - Types Of Testing
  - Testing Levels
  - The Impossibility Of Testing Everything
  - Summary
  - Practice
  - References
- Chapter 2: Case Studies
- Section I: Black Box Testing Techniques
- Section II: White Box Testing Techniques
  - Chapter 10: Control Flow Testing
  - Chapter 11: Data Flow Testing
    - Technique
    - Applicability and Limitations
    - Summary
    - Practice
    - References
- Section III: Testing Paradigms
- Section IV: Supporting Technologies
- Section V: Some Final Thoughts
  - Appendix A: Brown & Donaldson Case Study
  - Appendix B: Stateless University Registration Sys...
  - Bibliography
  - Other Useful Publications
  - Index
  - List of Figures
  - List of Tables
  - List of Examples

Technique

Team LiB | PREVIOUS | NEXT

Technique

Variables that contain data values have a defined life cycle. They are created, they are used, and they are killed (destroyed). In some programming languages (FORTRAN and BASIC, for example) creation and destruction are automatic. A variable is created the first time it is assigned a value and destroyed when the program exits.

In other languages (like C, C++, and Java) the creation is formal. Variables are declared by statements such as:

```
int x;    // x is created as an integer
string y; // y is created as a string
```

These declarations generally occur within a block of code beginning with an opening brace { and ending with a closing brace }. Variables defined within a block are created when their definitions are executed and are automatically destroyed at the end of a block. This is called the "scope" of the variable. For example:

```
{          // 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
```

Variables can be used in computation (a=b+1). They can also be used in conditionals (if (a>42)). In both uses it is equally important that the variable has been assigned a value before it is used.

Three possibilities exist for the first occurrence of a variable through a program path:

1. ~d the variable does not exist (indicated by the ~), then it is defined (d)
2. ~u the variable does not exist, then it is used (u)
3. ~k the variable does not exist, then it is killed or destroyed (k)

The first is correct. The variable does not exist and then it is defined. The second is incorrect. A variable must not be used before it is defined. The third is probably incorrect. Destroying a variable before it is created is indicative of a programming error.

Now consider the following 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.

Key Point Examine time-sequenced pairs of defined, used, and killed variable references.

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. We will construct these diagrams and verify that the define-use-kill patterns are appropriate. First, we will perform a static test of the diagram. By "static" we mean we examine the diagram (formally through inspections or informally through look-sees). Second, we perform dynamic tests on the module. By "dynamic" we mean we construct and execute test cases. Let's begin with the static testing.

Static Data Flow Testing

The following control flow diagram has been annotated with define-use-kill information for each of the variables used in the module.

Variable state transition (reading)

~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)

Case	Description
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.

Summary of Data-flow testing

- Data-flow testing
  - Main concern: places in a program where data values are defined and used.
  - Static (compile time) vs. dynamic (execution time) versions.
- Static: Define/Reference Anomalies on a variable that
  - is defined but never used (referenced)
  - is used but never defined
  - is defined more than once

Summary of Data-flow testing

- Defs:
  - a defining node of the variable v \in V, written as DEF(v, n), iff the value of the variable v is defined at the statement fragment corresponding to node n.
  - a usage node of the variable v \in V, written as USE(v, n), iff the value of the variable v is used at the statement fragment corresponding to node n.
  - A usage node USE(v, n) is a predicate use (denoted as P-use) iff the statement n is a predicate statement; otherwise, USE(v, n) is a computation use (denoted C-use).
  - A definition-use path with respect to a variable v (denoted du-path) is a path in the set of all paths in P, PATHS(P), such that for some v □ V, there are define and usage nodes DEF(v, m) and USE(v, n) such that m and n are the initial and final nodes of the path.
  - A definition-clear path with respect to a variable v (denoted dc-path) is a definition-use path in PATHS(P) with initial and final nodes DEF (v, m) and USE (v, n) such that no other node in the path is a defining node of v.

Summary of Data-flow testing

- Technique: for a particular variable,
  - find all its definition and usage nodes, then
  - find the du-paths and dc-paths among these.
  - for each path, devise a "suitable" (functional?) set of test cases.

Summary of Data-flow testing

- Coverage Metrics Based on du-paths
  - The set T satisfies the All-Defs criterion for the program P iff for every variable v ∈ V, T contains definition-clear paths from every defining node of v to a use of v.
  - The set T satisfies the All-Uses criterion for the program P iff for every variable v ∈ V, T contains definition-clear paths from every defining node of v to every use of v, and to the successor node of each USE(v, n).
  - The set T satisfies the All-P-Uses/Some C-Uses criterion for the program P iff for every variable v ∈ V, T contains definition-clear paths from every defining node of v to every predicate use of v; if a definition of v has no P-uses, a definition-clear path leads to at least one computation use.
  - The set T satisfies the All-C-Uses/Some P-Uses criterion for the program P iff for every variable v ∈ V, T contains definition-clear paths from every defining node of v to every computation use of v; if a definition of v has no C-uses, a definition-clear path leads to at least one predicate use.
  - The set T satisfies the All-du-paths criterion for the program P iff for every variable v ∈ V, T contains definition-clear paths from every defining node of v to every use of v and to the successor node of each USE(v, n), and that these paths are either single-loop traversals or cycle-free.

Summary of Data-flow testing

- Dataflow Testing Strategies
  - Dataflow testing is indicated in
    - Computation-intensive applications
    - “long” programs
    - Programs with many variables
  - A definition-clear du-path represents a small function that can be tested by itself.
  - If a du-path is not definition-clear, it should be tested for each defining node.

Summary

- Control flow testing
  - Execution path / Control flow graph
  - Control flow testing
    - Code-based coverage
    - Multiple Condition Testing vs Modified Condition Decision Coverage
    - McCabe's Basic Path Testing
- Data flow testing
  - Data flow testing
    - du-path vs. dc-path
    - Coverage based on du-paths
  - Variable state transition (reading)