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

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

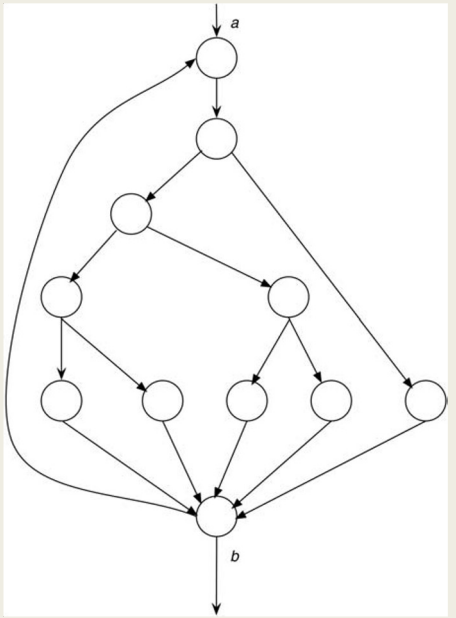
- White Box Testing is software testing technique in which internal structure, design and coding of software are tested to verify flow of input-output and to improve design, usability and security. In white box testing, code is visible to testers so it is also called Clear box testing, Open box testing,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

- 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 CFG:
  - *Node: 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: jumps in the control flow*
  - *Two specially designated blocks: entry and exit blocks*

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

Control flow graph

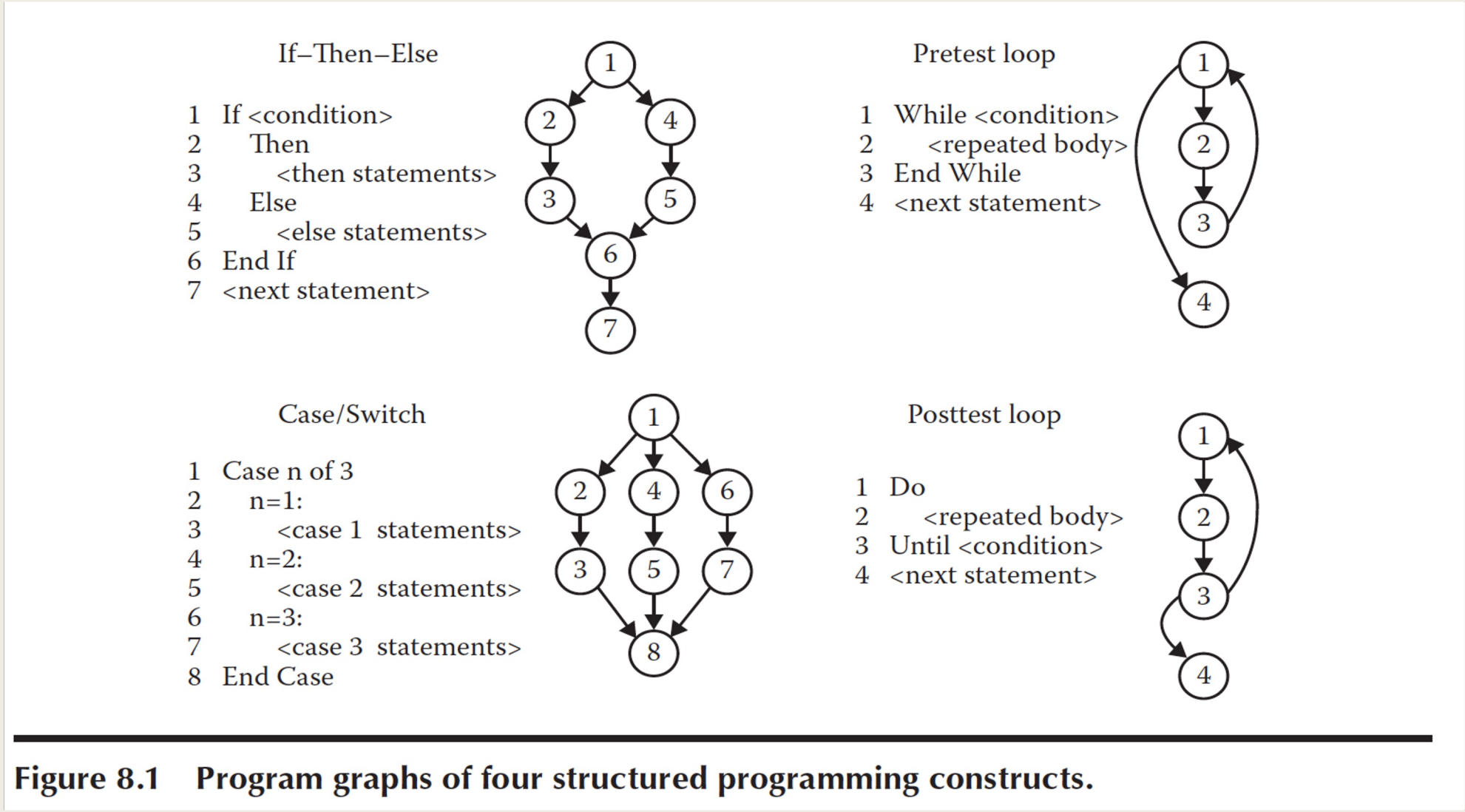


Figure 8.1 Program graphs of four structured programming constructs.

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

Control flow graph

```
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
```

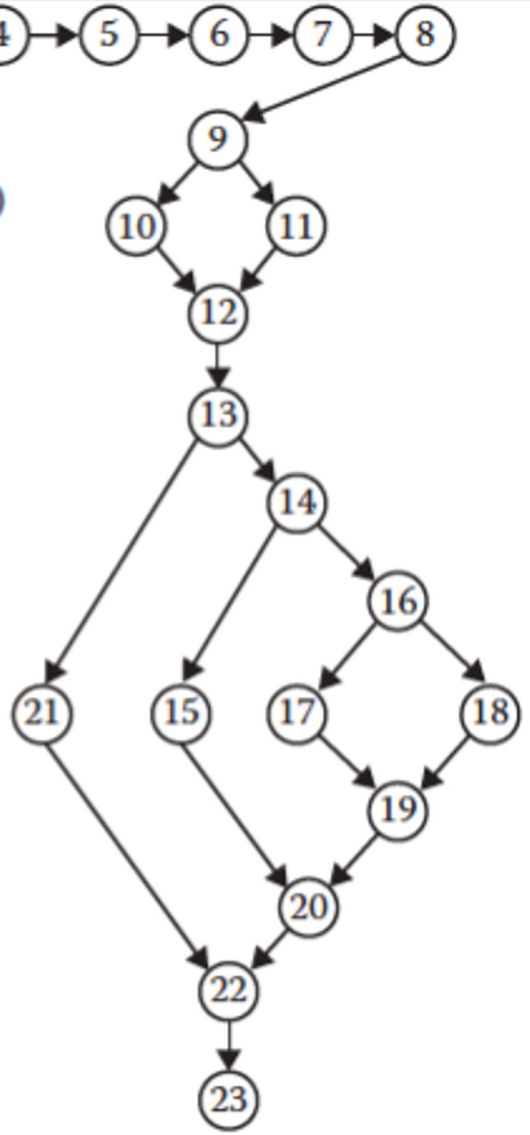
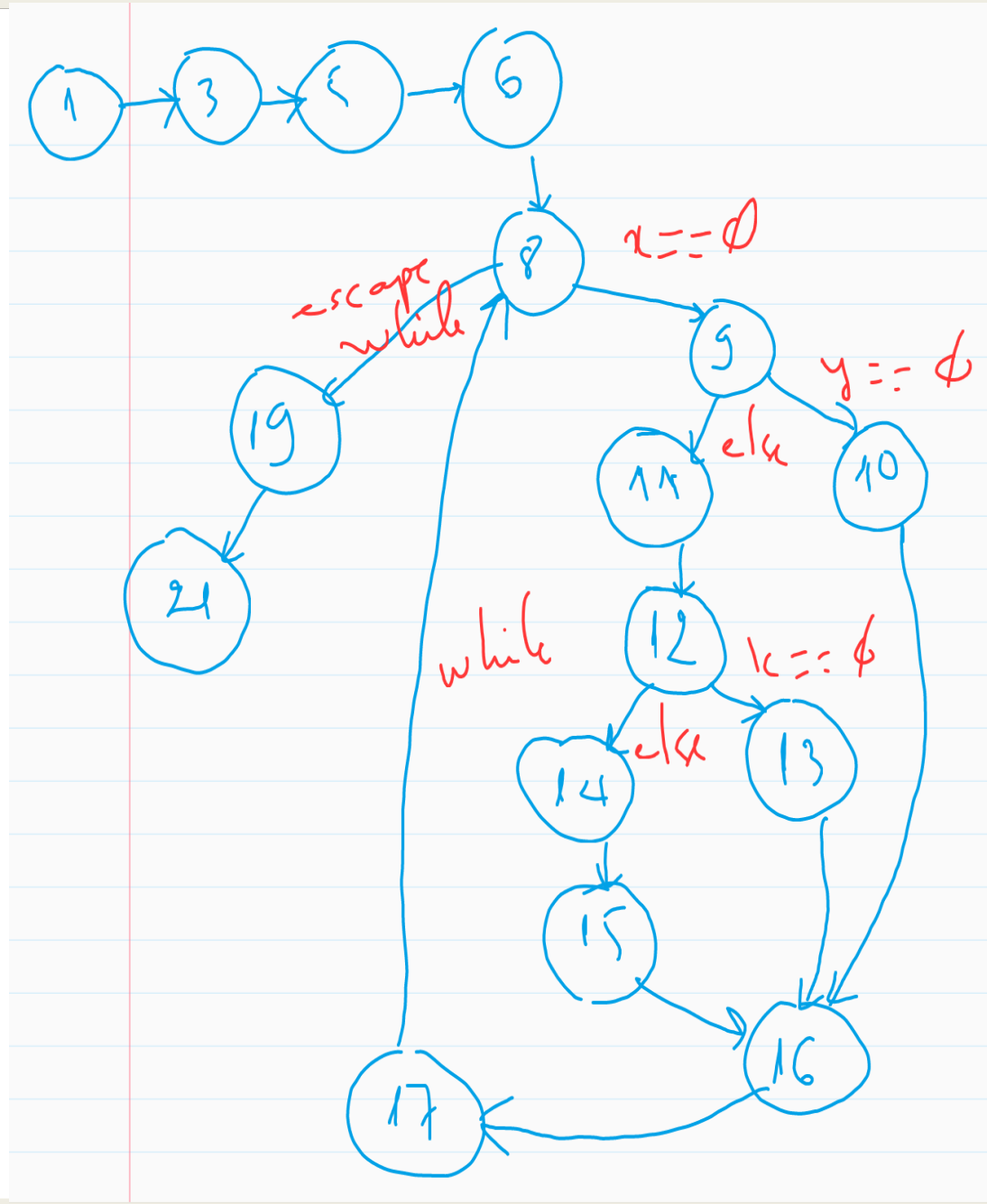
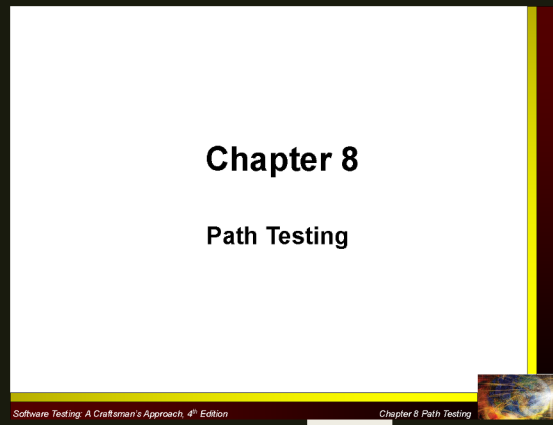


Figure 8.2 Program graph of triangle program.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int x, y, z, k;
7
8      while (x == 0) {
9          if (y == 0) {
10             z = 0;
11         } else {
12             if (k == 0)
13                 z = 1;
14             else
15                 x = 1;
16         }
17     }
18
19     cout << "End." << endl;
20
21 }
```

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int x, y, z, k;
7
8      while (x == 0) {
9          if (y == 0) {
10             z = 0;
11         } else {
12             if (k == 0)
13                 z = 1;
14             else
15                 x = 1;
16         }
17     }
18
19     cout << "End." << endl;
20
21 }
```





PATH TESTING / CODE-BASED TESTING

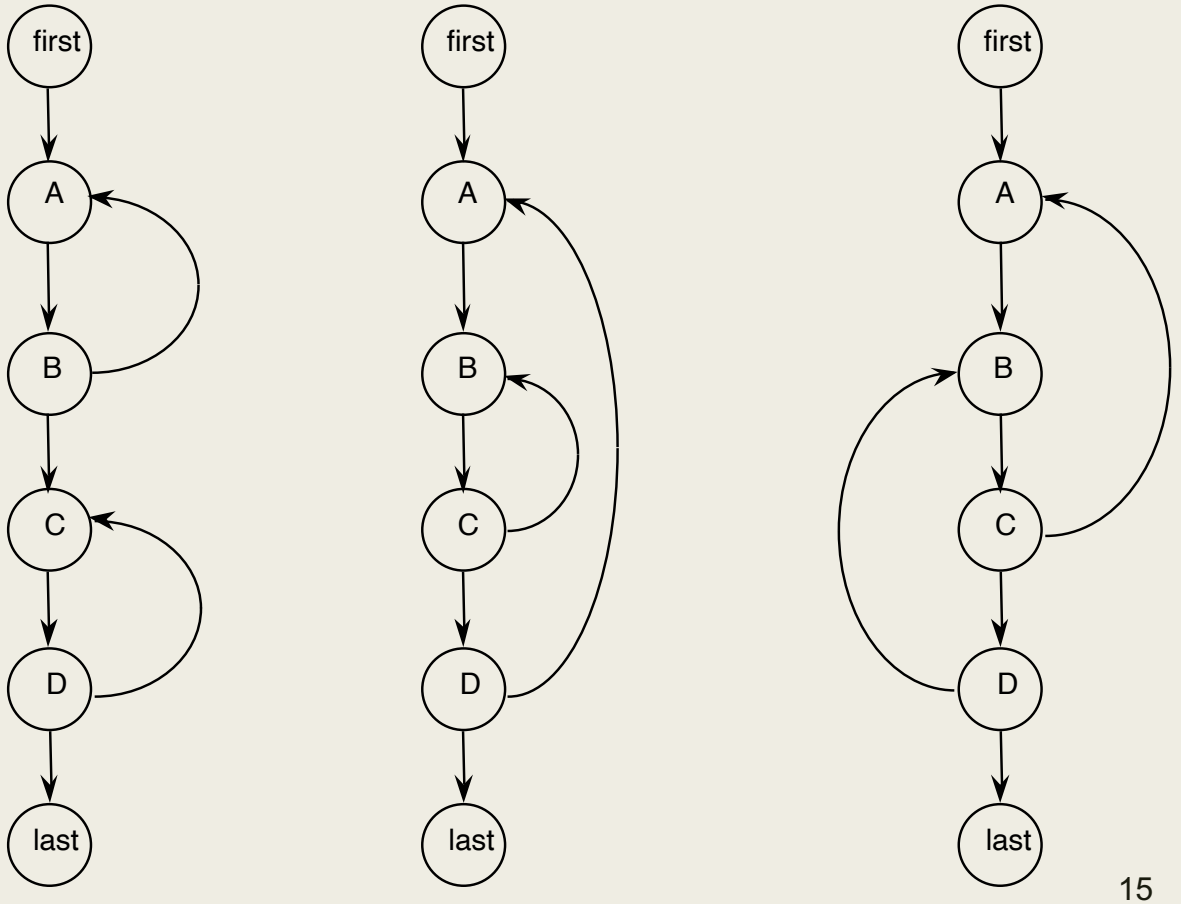
Code-Based Test Coverage Metrics (E. F. Miller, 1977 dissertation)

- 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

Huang’s theorem

- Huang's Theorem: (Paraphrased)
  - *Everything interesting will happen in two loop traversals: the normal loop traversal and the exit from the loop.*

- Types of loops:
  - *Concatenated*
  - *Nested*
  - *Knotted Loops*



Huang’s theorem

- Huang’s theorem suggests/assures 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*
  - *Work from innermost to outermost loop*
- For concatenated loops: use Huang’s Theorem
- For knotted loops: Rewrite!

Multiple Condition Testing

- Consider the multiple condition as a logical proposition, i.e., some logical expression of simple conditions.
- Make the truth table of the logical expression.
- Convert the truth table to a decision table.
- Develop test cases for each rule of the decision table (except the impossible rules, if any).

Multiple Condition Testing

Input: a, b, c are positive numbers
Output: if a, b, and c are the edges of a triangle?

```
If (a < b + c) AND (b < a + c) AND (c < a + b)
        Then IsATriangle = True
        Else  IsATriangle = False
Endif
```

- Truth Table for Triangle Inequality (a<b+c) AND (b<a+c) AND (c<a+b)

(a<b+c)	(b<a+c)	(c<a+b)	(a<b+c) AND (b<a+c) AND (c<a+b)
T	T	T	T
T	T	F	F
T	F	T	F
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F

Multiple Condition Testing

Input: a, b, c are positive numbers
Output: if a, b, and c are the edges of a triangle?

```
If (a < b + c) AND (b < a + c) AND (c < a + b)
        Then IsATriangle = True
        Else  IsATriangle = False
Endif
```

- Decision Table for (a<b+c) AND (b<a+c) AND (c<a+b)
  - *When a,b,c >0 then c1, c2 and c3 cannot be false pairwise*

c1: a<b+c	T	T	T	T	F	F	F	F
c2: b<a+c	T	T	F	F	T	T	F	F
c3: c<a+b	T	F	T	F	T	F	T	F
a1: impossible				X		X	X	X
a2:Valid test case #	1	2	3		4			

Multiple Condition Testing

Input: a, b, c are positive numbers
Output: if a, b, and c are the edges of a triangle?

```
If (a < b + c) AND (b < a + c) AND (c < a + b)
        Then IsATriangle = True
        Else  IsATriangle = False
Endif
```

- Multiple Condition Test Cases for (a<b+c) AND (b<a+c) AND (c<a+b)
  - *OR more*

Test Case		a	b	c	expected output
1	all true	3	4	5	TRUE
2	c ≥ a + b	3	4	9	FALSE
3	b ≥ a + c	3	9	4	FALSE
4	a ≥ b + c	9	3	4	FALSE

Modified Condition Decision Coverage (MCDC)

* Chilenski, John Joseph, "An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion," DOT/FAA/AR-01/18, April 2001. [http://www.faa.gov/about/office_org/headquarters_offices/ang/offices/tc/library/]

- Extension of/to Multiple Condition Testing
- Required for “Level A” software by DO-178B
- Three variations*
  - *Masking MCDC*
  - *Unique-Cause MCDC*
  - *Unique-Cause + Masking MCDC*
- Masking MCDC is
  - *the weakest of the three, AND*
  - *is recommended for DO-178B compliance*

Modified Condition Decision Coverage (MCDC) Chilenski's Definitions

- Conditions can be either simple or compound
  - *isATriangle is a simple condition*
  - *(a < b + c) AND (b < a + c) is a compound condition*
- Conditions are strongly coupled if changing one always changes the other.
  - *(x = 0) and (x ≠ 0) are strongly coupled in ((x = 0) AND a) OR ((x ≠ 0) AND b)*
- Conditions are weakly coupled if changing one may change one but not all of the others.
  - *All three conditions are weakly coupled in ((x = 1) OR (x = 2) OR (x = 3))*
- "Masking" is based on the Domination Laws
  - *(x AND false)*
  - *(x OR true)*

MCDC requires...

- 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).

MCDC Variations

- Unique-Cause MCDC [requires] a unique cause
  - *(toggle a single condition and change the expression result) for all possible (uncoupled) conditions.*
- Unique-Cause + Masking MCDC [requires] a unique cause
  - *(toggle a single condition and change the expression result) for all possible (uncoupled) conditions. In the case of strongly coupled conditions, masking [is allowed] for that condition only, i.e., all other (uncoupled) conditions will remain fixed.*
- Masking MCDC allows masking for all conditions, coupled and uncoupled
  - *(toggle a single condition and change the expression result) for all possible (uncoupled) conditions. In the case of strongly coupled conditions, masking [is allowed} for that condition only (i.e., all other (uncoupled) conditions will remain fixed.*

Decision Table for (a AND (b OR c))

```
1. If (a AND (b OR c) )
2.      Then  y = 1
3.      Else   y = 2
4. EndIf
```

Conditions	**rule 1**	**rule 2**	**rule 3**	**rule 4**	**rule 5**	**rule 6**	**rule 7**	**rule 8**
a	T	T	T	T	F	F	F	F
b	T	T	F	F	T	T	F	F
c	T	F	T	F	T	F	T	F
a AND (b OR c)	True	True	True	False	False	False	False	False
Actions								
y = 1	x	x	x	—	—	—	—	—
y = 2	—	—	—	x	x	x	x	x

MCDC Coverage of (a AND (b OR c))

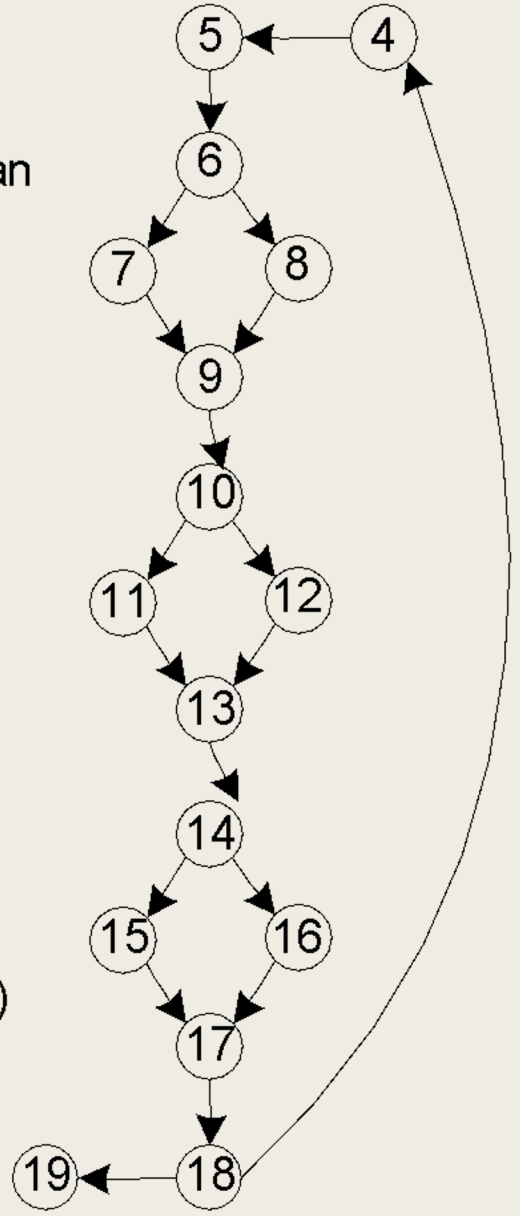
```
1. If (a AND (b OR c) )
2.     Then  y = 1
3.     Else  y = 2
4. EndIf
```

- Rules 1 and 5 toggle condition a
- Rules 2 and 4 toggle condition b
- Rules 3 and 4 toggle condition c

- If we expand (a AND (b OR c)) to ((a AND b) OR (a AND C)), we cannot do unique cause testing on variable a because it appears in both sub-expressions.

MCDC: A NextDate Example

```
 1  NextDate Fragment
 2  Dim day, month, year As Integer
 3  Dim dayOK, monthOK, yearOK As Boolean
 4  Do
 5       Input(day, month, year)
 6       If 0 < day < 32
 7          Then dayOK  = True
 8          Else  dayOK = False
 9       EndIf
10     If 0 < month < 13
11        Then monthOK  = True
12        Else  monthOK  = False
13     EndIf
14     If 1811 < year < 2013
15        Then yearOK = True
16        Else  yearOK = False
17     EndIf
18 Until (dayOK AND monthOK AND yearOK)
19  End Fragment
```



MCDC: A NextDate Example Corresponding Decision Table

Conditions	**rule 1**	**rule 2**	**rule 3**	**rule 4**	**rule 5**	**rule 6**	**rule 7**	**rule 8**
dayOK	T	T	T	T	F	F	F	F
monthOK	T	T	F	F	T	T	F	F
YearOK	T	F	T	F	T	F	T	F
The Until condition	True	False	False	False	False	False	False	False
Actions								
Leave the loop	x	—	—	—	—	—	—	—
Repeat the loop	—	x	x	x	x	x	x	x

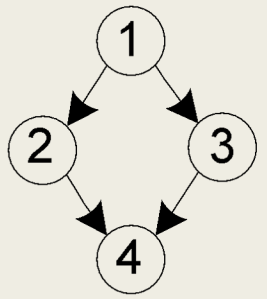
MCDC: A NextDate Example
Test Cases and Coverage

- Decision Coverage: Rule 1 and any of Rules 2 – 8
- Multiple Condition Coverage: all eight rules are needed
- Modified Condition Decision Coverage:
  - *rules 1 and 2 toggle yearOK*
  - *rules 1 and 3 toggle monthOK*
  - *rules 1 and 4 toggle dayOK*

MCDC: One more example

- In the three conditions, there are interesting dependencies that create four impossible rules.

```
1. If (a < b + c)  AND (a < b + c)  AND (a < b + c)
2.      Then  IsA Triangle = True
3.      Else   IsA Triangle = False
4. EndIf
```



MCDC: One more example Corresponding Decision Table

Conditions	**rule 1**	**rule 2**	**rule 3**	**rule 4**	**rule 5**	**rule 6**	**rule 7**	**rule 8**
(a < b + c)	T	T	T	T	F	F	F	F
(b < a + c)	T	T	F	F	T	T	F	F
(c < a + b)	T	F	T	F	T	F	T	F
IsATriangle = True	x	—	—	—	—	—	—	—
IsATriangle = False	—	x	x	—	x	—	—	—
impossible	—	—	—	x		x	x	x

Two Strategies of MCDC Testing

- Rewrite the code as a decision table
  - *algebraically simplify*
  - *watch for impossible rules*
  - *eliminate masking when possible*
- Rewrite the code as **nested IF logic**

MCC vs MCDC: Test Cases and Coverage

- Decision Coverage: Rule 1 and Rule 2 (also, rules 1 and 3, or rules 1 and 5).
- Rules 4, 6, 7, and 8 are impossible.
- Condition Coverage
  - *rules 1 and 2 toggle (c < a + b)*
  - *rules 1 and 3 toggle (b < a + c)*
  - *rules 1 and 5 toggle (a < b + c)*
- Modified Condition Decision Coverage:
  - *rules 1 and 2 toggle (c < a + b)*
  - *rules 1 and 3 toggle (b < a + c)*
  - *rules 1 and 5 toggle (a < b + c)*

Code-Based Testing Strategy

- Start with a set of test cases generated by an “appropriate” (depends on the nature of the program) specification-based test method.
- Look at code to determine appropriate test coverage metric.
  - *Loops?*
  - *Compound conditions?*
  - *Dependencies?*
- If appropriate coverage is attained, fine.
- Otherwise, add test cases to attain the intended test coverage.

Basis Path Testing

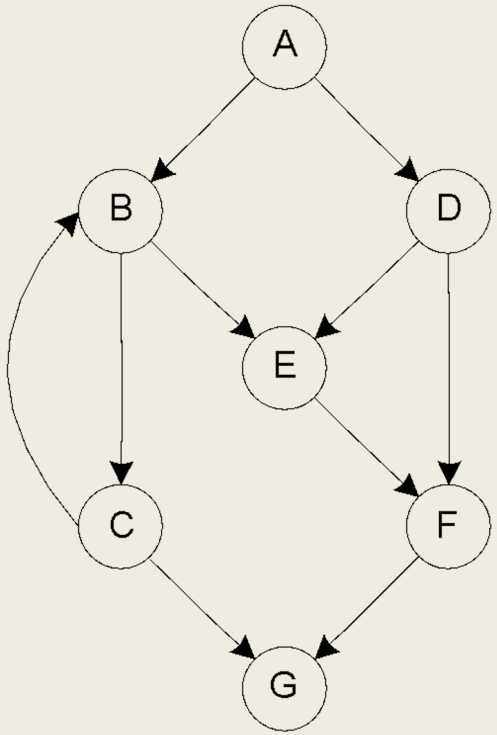
- Proposed by Thomas McCabe in 1982
- Math background
  - *a Vector Space has a set of independent vectors called basis vectors*
  - *every element in a vector space can be expressed as a linear combination of the basis vectors*
- Example: Euclidean 3-space has three basis vectors
  - *(1, 0, 0) in the x direction*
  - *(0, 1, 0) in the y direction*
  - *(0, 0, 1) in the z direction*
- The Hope: If a program graph can be thought of as a vector space, there should be a set of basis vectors. Testing them tests many other paths.

(McCabe) Basis Path Testing

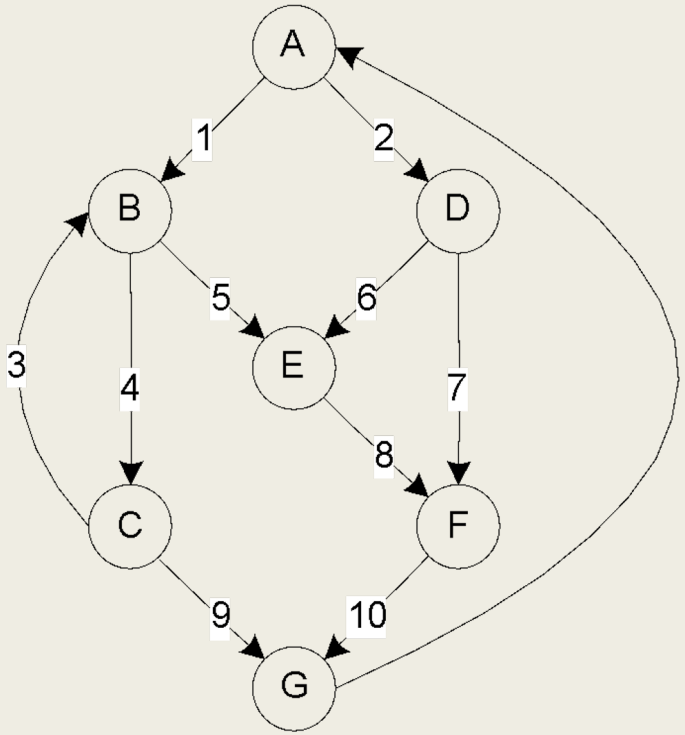
- The cyclomatic number of a strongly connected directed graph is the number of linearly independent cycles.
- Given a program graph, we can always add an edge from the sink node to the source node to create a strongly connected graph. (assuming single entry, single exit)
- Computing V(G) = e - n + p from the modified program graph yields the number of independent paths that must be tested.

McCabe's Example

McCabe's Original Graph



Derived, Strongly Connected Graph



V(G) = 10 - 7 + 2(1)
= 5

V(G) = 11 - 7 + 1
= 5

McCabe's Baseline Method

- 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.

McCabe's Example (with numbered edges)

Resulting basis paths

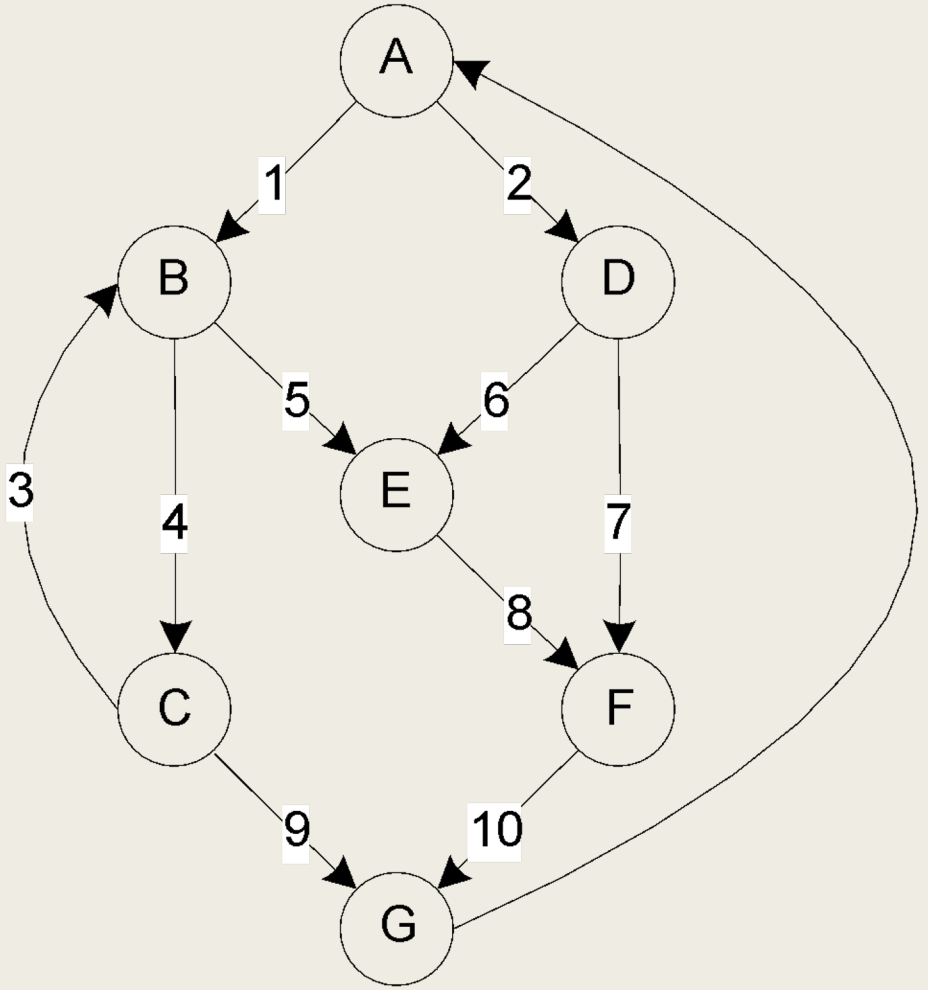
First baseline path
p1: A, B, C, G

Flip decision at C
p2: A, B, C, B, C, G

Flip decision at B
p3: A, B, E, F, G

Flip decision at A
p4: A, D, E, F, G

Flip decision at D
p5: A, D, F, G



Path/Edge Incidence

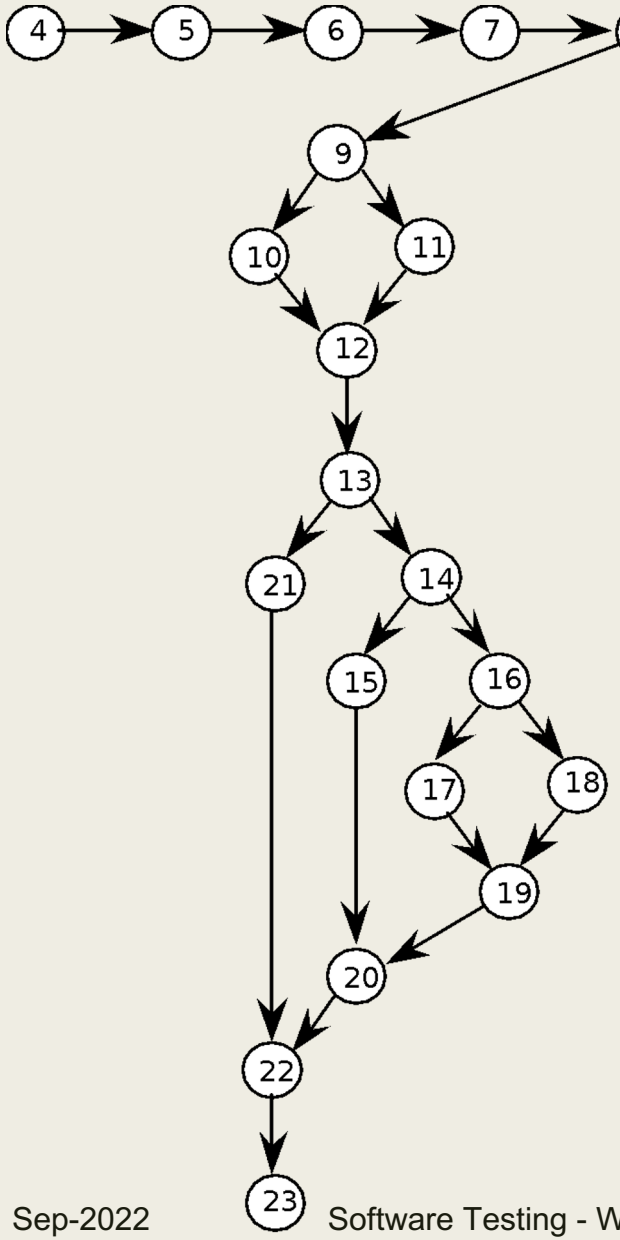
Path / Edges	*e1*	*e2*	*e3*	*e4*	*e5*	*e6*	*e7*	*e8*	*e9*	*e10*
p1: A, B, C, G	1	0	0	1	0	0	0	0	1	0
p2: A, B, C, B, C, G	1	0	**1**	2	0	0	0	0	1	0
p3: A, B, E, F, G	1	0	0	0	**1**	0	0	1	0	1
p4: A, D, E, F, G	0	1	0	0	0	**1**	0	1	0	1
p5: A, D, F, G	0	1	0	0	0	0	**1**	0	0	1
ex1: A, B, C, B, E, F, G	1	0	1	1	1	0	0	1	0	1
ex2: A, B, C, B, C, B, C, G	1	0	2	3	0	0	0	0	1	0

Sample paths as linear combinations of basis paths
ex1 = p2 + p3 - p1
ex2 = 2p2 - p1

Problems with Basis Paths

- What is the significance of a path as a linear combination of basis paths?
- What do the coefficients mean? What does a minus sign mean?
- In the path ex2 = 2p2 - p1 should a tester run path p2 twice, and then not do path p1 the next time? This is theory run amok.
- Is there any guarantee that basis paths are feasible?
- Is there any guarantee that basis paths will exercise interesting dependencies?

McCabe Basis Paths in the Triangle Program



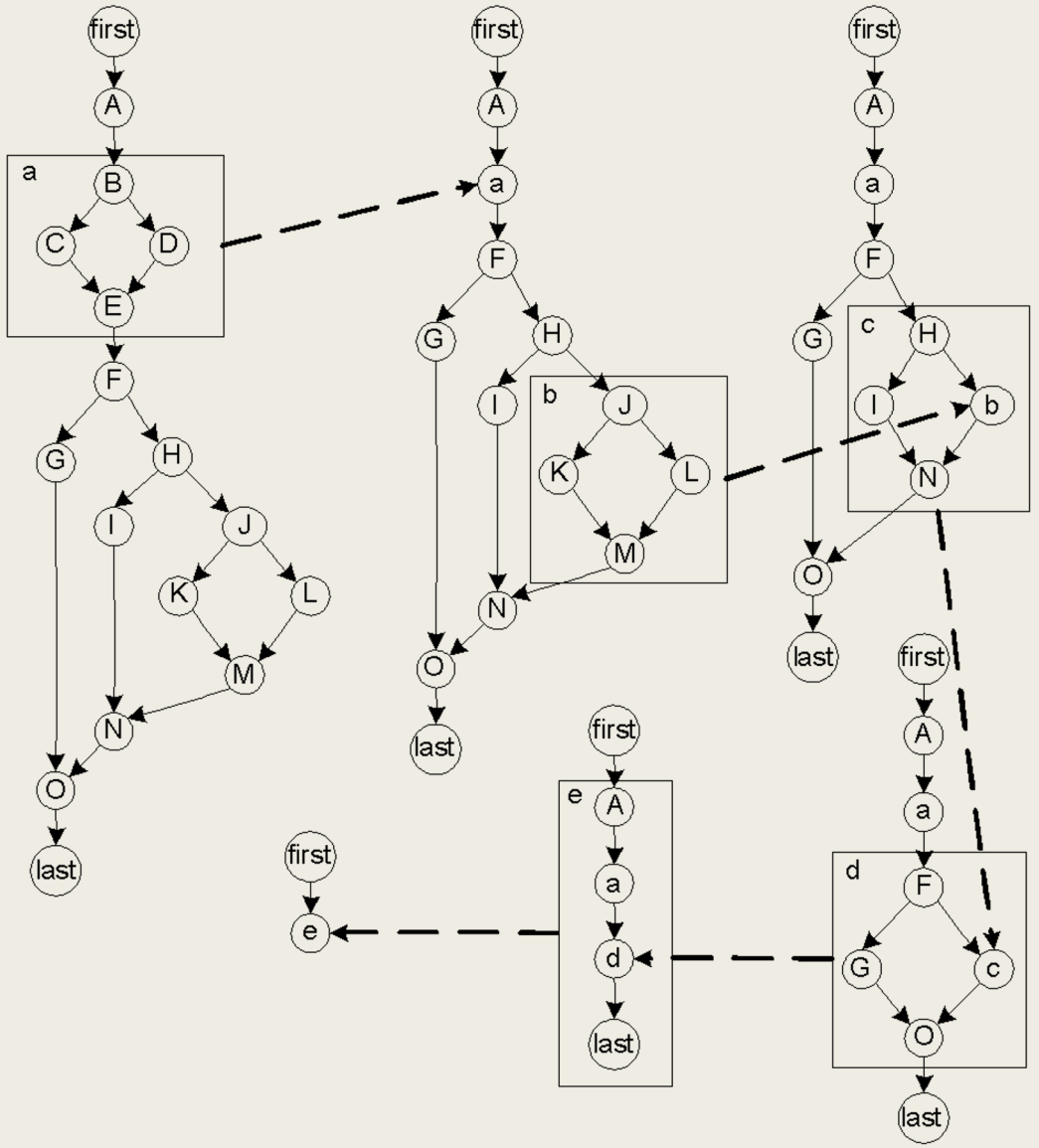
There are 8 topologically possible paths.
4 are feasible, and 4 are infeasible.

Exercise: Is every basis path feasible?

Essential Complexity

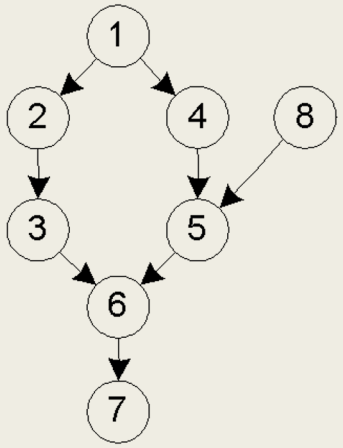
- McCabe's notion of Essential Complexity deals with the extent to which a program violates the precepts of Structured Programming.
- To find Essential Complexity of a program graph,
  - *Identify a group of source statements that corresponds to one of the basic Structured Programming constructs.*
  - *Condense that group of statements into a separate node (with a new name)*
  - *Continue until no more Structured Programming constructs can be found.*
  - *The Essential Complexityof the original program is the cyclomatic complexity of the resulting program graph.*
- The essential complexity of a Structured Program is 1.
- Violations of the precepts of Structured Programming increase the essential complexity.

Condensation with Structured Programming Constructs

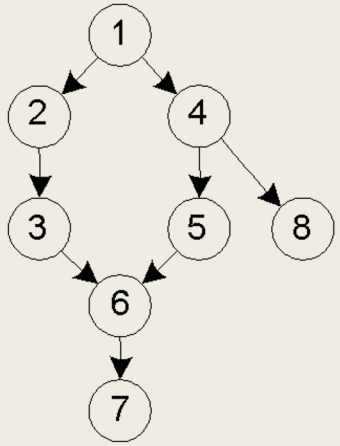


Violations of Structured Programming Precepts

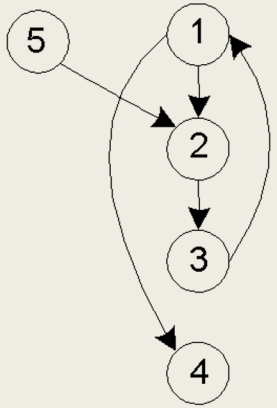
Branching into a decision



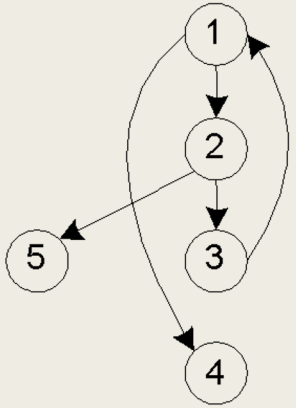
Branching out of a decision



Branching into a loop



Branching out of a loop



Cons and Pros of McCabe's Work

- Issues
  - *Linear combinations of execution paths are counter-intuitive. What does 2p2 – p1 really mean?*
  - *How does the baseline method guarantee feasible basis paths?*
  - *Given a set of feasible basis paths, is this a sufficient test?*

- Advantages
  - *McCabe's approach does address both gaps and redundancies.*
  - *Essential complexity leads to better programming practices.*
  - *McCabe proved that violations of the structured programming constructs increase cyclomatic complexity, and violations cannot occur singly.*

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

- 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

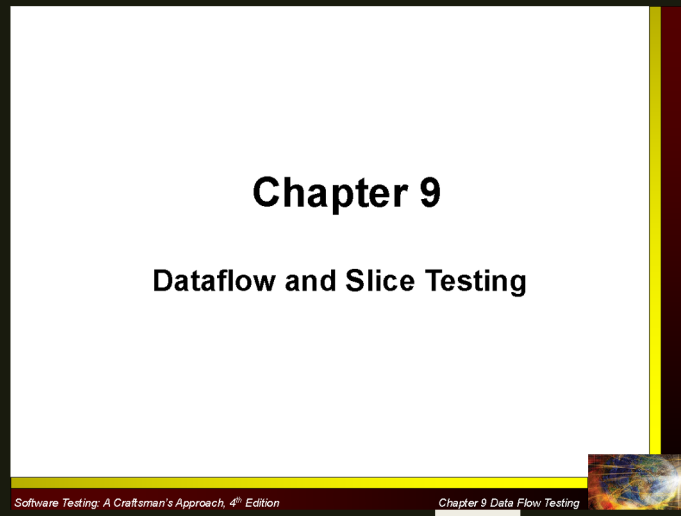
- Strategy for Loop Testing
  - *2 tests per loop is sufficient. (Judgment required, based on reality of the code.)*
  - *For nested loops:*
    - Test innermost loop first
    - Then “condense” the loop into a single node
    - Work from innermost to outermost loop
  - *For concatenated loops: use Huang’s Theorem*
  - *For knotted loops: Rewrite!*

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 the last two steps.*



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

- Main concern: places in a program where data values are defined and used.
- Static (compile time) and 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*
- Starting point is a program, P, with program graph G(P), and the set V of variables in program P.
- "Interesting" data flows are then tested as separate functions.

Data flow testing - Some definitions

- Given a program, P, with a set V of variables in P, and the program graph G(P), Node n $\in$ G(P) is
  - *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 $\in$ 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.

Data flow testing - Example

```
1.  Program Commission (INPUT,OUTPUT)
2.  Dim locks, stocks, barrels As Integer
3.  Dim lockPrice, stockPrice, barrelPrice As Real
4.  Dim totalLocks, totalStocks, totalBarrels As Integer
5.  Dim lockSales, stockSales, barrelSales As Real
6.  Dim sales, commission As Real
7.  lockPrice = 45.0
8.  stockPrice = 30.0
9.  barrelPrice = 25.0
10. totalLocks = 0
11. totalStocks = 0
12. totalBarrels = 0
13. Input(locks)
14. While NOT(locks = -1)
15.     Input(stocks, barrels)
16.     totalLocks = totalLocks + locks
17.     totalStocks = totalStocks + stocks
18.     totalBarrels = totalBarrels + barrels
19.     Input(locks)
20.  EndWhile
21.  Output("Locks sold: ", totalLocks)
22.  Output("Stocks sold: ", totalStocks)
23.  Output("Barrels sold: ", totalBarrels)
23. Output("Barrels sold: ", totalBarrels)
24. lockSales = lockPrice * totalLocks
25. stockSales = stockPrice * totalStocks
26. barrelSales = barrelPrice * totalBarrels
27. sales = lockSales + stockSales + barrelSales
28. Output("Total sales: ", sales)
29. If (sales > 1800.0)
30.   Then
31.     commission = 0.10 * 1000.0
32.     commission = commission + 0.15 * 800.0
33.     commission = commission + 0.20 *(sales-1800.0)
34.   Else If (sales > 1000.0)
35.             Then
36.               commission = 0.10 * 1000.0
37.               commission = commission + 0.15 *(sales-1000.0)
38.              Else
39.                commission = 0.10 * sales
40.           EndIf
41.  EndIf
42. Output("Commission is $", commission)
43. End Commission
```

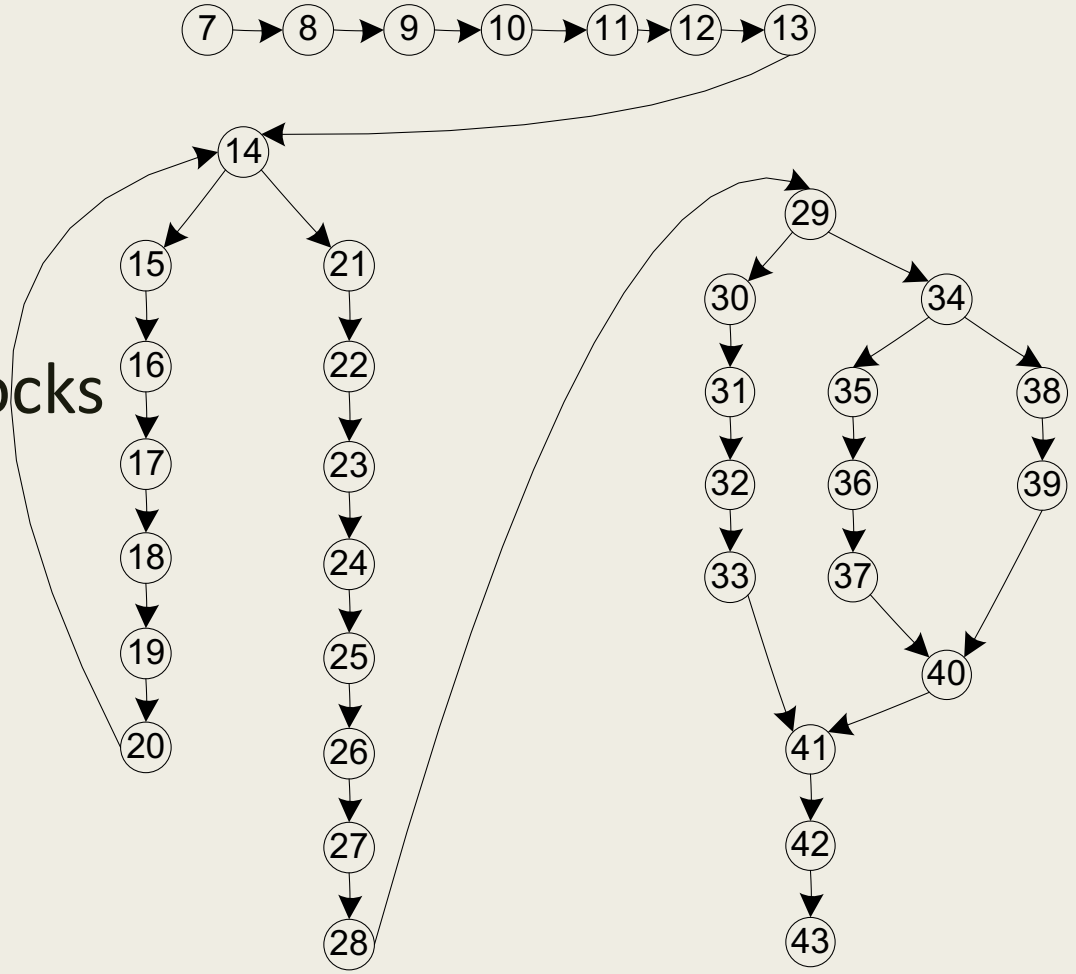
Data flow testing - Example

Variable	*Defined at Node*	*Used at Node*
lockPrice	7	24
totalLocks	10, 16	16, 21, 24
locks	13, 19	14, 16
lockSales	24	27
sales	27	28, 29, 33, 34, 37, 38
commission	31, 32, 33, 36, 37, 38	32, 33, 37, 41
stockPrice	?	?
totalStocks	?	?
locks	13, 19	14, 16
stockSales	?	?
sales	27	28, 29, 33, 34, 37, 38
commission	31, 32, 33, 36, 37, 38	32, 33, 37, 41

Data flow testing - Example

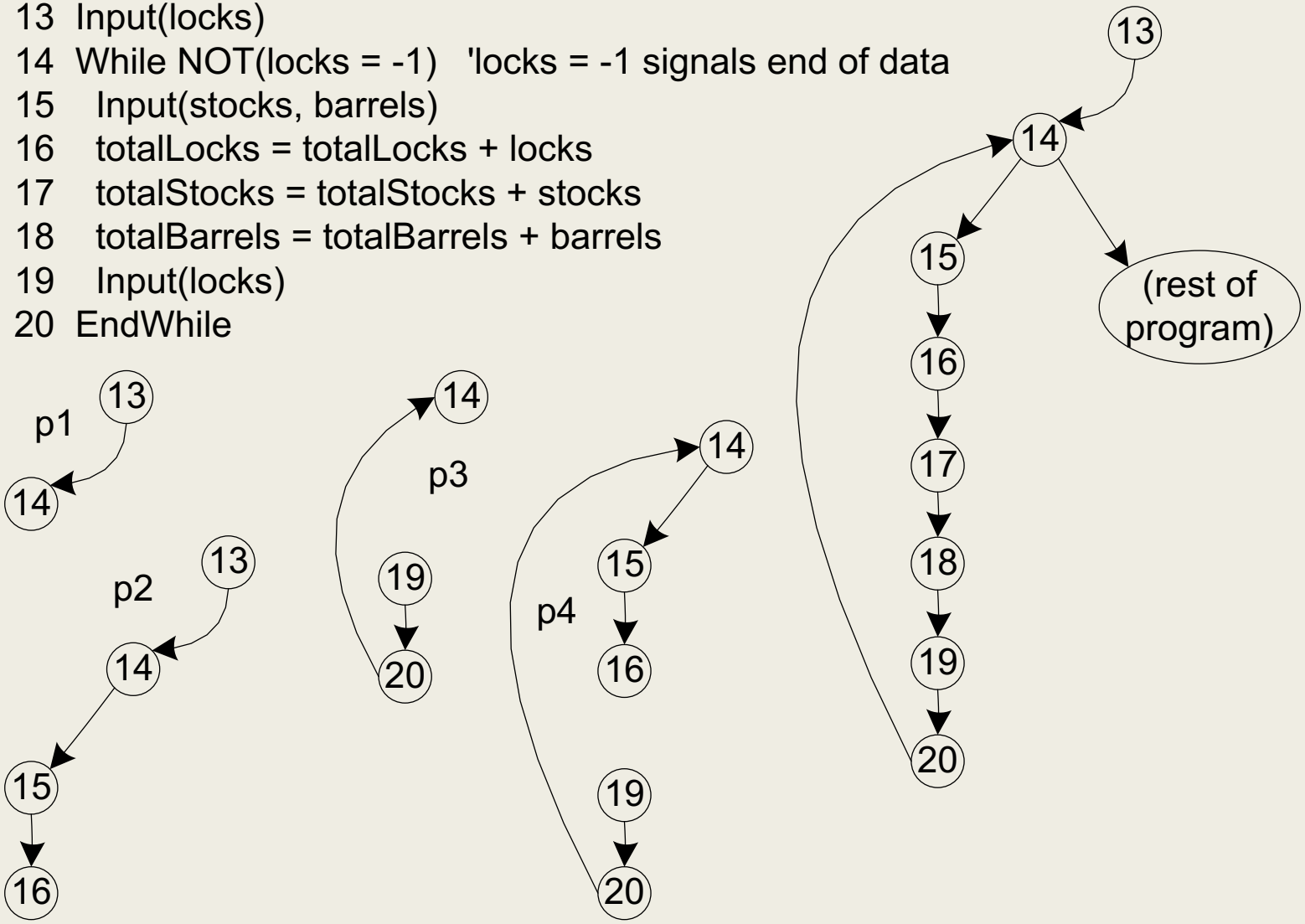
- *Variable locks:*
  - defined: 13, 19
  - used: 14, 16
- Define/Use Paths (du-paths) for locks
  - *p1 = <13, 14>*
  - *p2 = <13, 14, 15, 16>*
  - *p3 = <19, 20, 14>*
  - *p4 = <19, 20, 14, 15, 16>*

  - *(all are definition clear)*



Data flow testing - Example

```
13  Input(locks)
14  While NOT(locks = -1)   'locks = -1 signals end of data
15    Input(stocks, barrels)
16    totalLocks = totalLocks + locks
17    totalStocks = totalStocks + stocks
18    totalBarrels = totalBarrels + barrels
19    Input(locks)
20  EndWhile
```



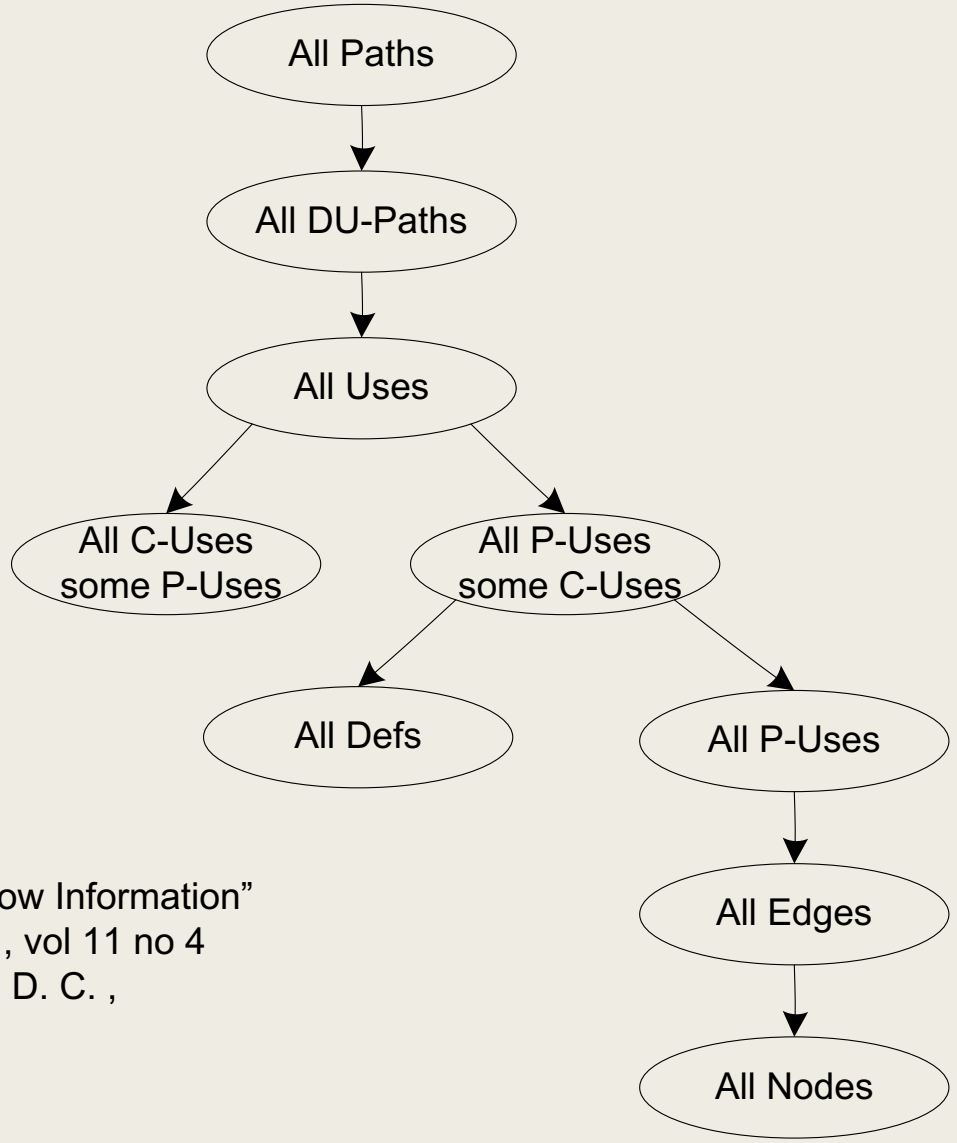
Data flow testing – Test cases

- 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.*
- Note: du-paths and dc-paths have both static and dynamic interpretations
  - *Static: just as seen in the source code*
  - *Dynamic: must consider execution-time flow (particularly for loops)*
- Definition clear paths are easier to test
  - *No need to check each definition node, as is necessary for du-paths*

Coverage Metrics Based on du-paths

- In the following definitions, T is a set of paths in the program graph G(P) of a program P, with the set V of variables.
- The set T satisfies the All-Defs criterion for the program P iff for every variable $v \in V$, 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 $v \in V$, 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 $v \in V$, 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 $v \in V$, 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 $v \in V$, 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.

Coverage Metrics Based on du-paths

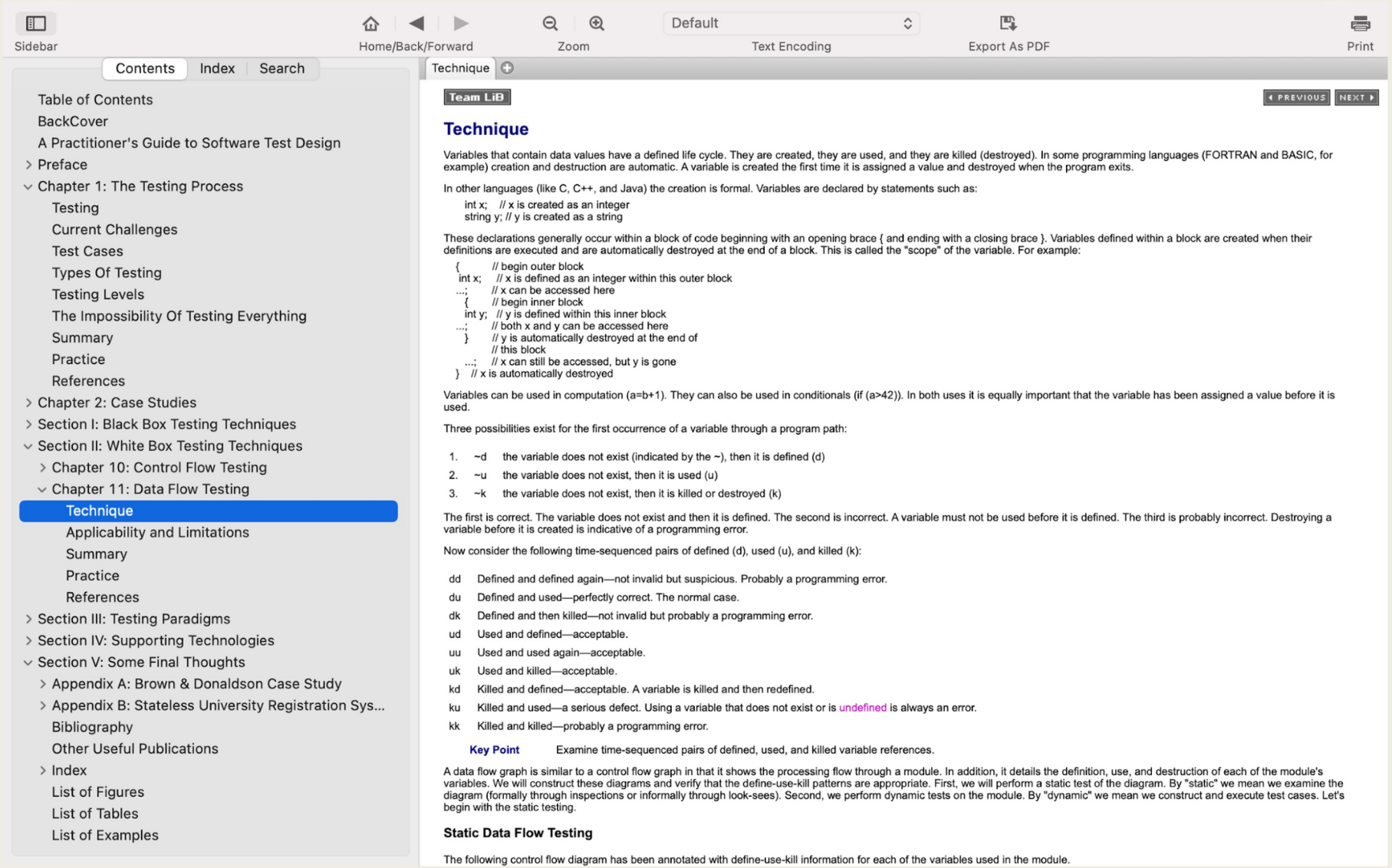


S. Rapps and E. J. Weyuker
"Selecting Software Test Data Using Data Flow Information"
*IEEE Transactions of Software Engineering*_ , vol 11 no 4
IEEE Computer Society Press, Washington, D. C. ,
April 1985, pp 367 - 375.

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.

Variable state transition (reading)



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)*