TVVS – Test, Verification and Validation of Software

Test case design

Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Index

- Introduction
- Black box testing techniques
- White box testing techniques

What is adequacy?

- Consider a program *P* written to meet a set *R* of functional requirements.
- Let *R* contain *n* requirements labeled R_1, R_2,…, R_n .
- A set *T* containing *k* tests has been constructed to test *P*.
- *P* has been executed against each test in *T* and has produced correct behavior.
- **We now ask:**
  - Is *T* good enough?
  - Has *P* been tested thoroughly?
  - Is *T* adequate?

In the context of software testing, the terms "thorough", "good enough", and "adequate", used in the questions above, have the same meaning.

Measurement of adequacy

- Measurement of adequacy is done against a given criterion **C**.
  - A test set is considered adequate with respect to criterion C when it satisfies C. The determination of whether or not a test set T for program P satisfies criterion C depends on the criterion itself.
- Examples
  - C: A test T for program (P, R) is considered adequate if for each requirement R_i in R there is at least one test case in T that tests the correctness of P with respect to R_i.
- A criterion C is a **white-box test adequacy criterion** if the corresponding coverage domain Ce depends solely on program P under test (is a **black-box test adequacy criterion** when Ce depends solely on requirements for the program P under test.

Example

- Program sumProduct must meet the following requirements:
  - R1 - Input two integers, say x and y, from the standard input device.
  - R2.1 - Find and print to the standard output device the sum of x and y, if x<y .
  - R2.2 - Find and print to the standard output device the product of x and y, if x≥ y.
- Obviously, T={t: <x=2, y=3>} is inadequate with respect to C (at least one test for each R_i) for program sumProduct. The lone test case t in T tests R1 and R2.1, but not R2.2.
- In this case the finite set of elements Ce={R1, R2.1, R2.2}. T covers R1 and R2.1 but not R2.2. Hence T is not adequate with respect to C. The coverage of T with respect to C, P, and R is 2/3 = 0.66.

Another Example

- Consider the following criterion: **"A test T for program P with a set of requirements R is considered adequate if each path in P is traversed at least once."**
- Assume that P has exactly two paths, one corresponding to condition x<y and the other to x≥ y. We refer to these as p1 and p2, respectively. For the given adequacy criterion C we obtain the coverage domain Ce to be the set {p1, p2}.
- We execute P against each test case in T={t: <x=2, y=3>}.
- As T contains only one test for which **x<y**, **only the path p1** is executed. Thus the coverage of T with respect to C, P, and R is 0.5 and hence T is not adequate with respect to C. We can also say that p1 is tested and p2 is not tested.

Code-based coverage domain

- In the previous example we assumed that P contains exactly two paths. This assumption is based on a knowledge of the requirements. However, when the coverage domain must contain elements from the code, these elements must be derived by analyzing the code and not only by an examination of its requirements.
- Errors in the program and incomplete or incorrect requirements might cause the program, and hence the coverage domain, to be different from the expected.

Example

R1 - Input two integers, say x and y, from the standard input device.
R2.1 - Find and print to the standard output device the sum of x and y, if x<y .
R2.2 - Find and print to the standard output device the product of x and y, if x≥ y.

sumProduct1

```
1    begin
2      int x, y;
3      input (x, y);
4      sum=x+y;
5      output (sum);
6    end
```

This program is obviously incorrect as per the requirements of sumProduct.

There is only one path denoted as p1. This path traverses all the statements. Using the path-based coverage criterion C, we get coverage domain Ce={p1}. T={t: <x=2, y=3>} is adequate w.r.t. C but does not reveal the error.

Example (contd.)

R1 - Input two integers, say x and y, from the standard input device.
R2.1 - Find and print to the standard output device the sum of x and y, if x<y .
R2.2 - Find and print to the standard output device the product of x and y, if x≥ y.

sumProduct2

```
1    begin
2      int x, y;
3      input (x, y);
4      if(x<y)
5      then
6        output(x+y);
7      else
8        output(x*y);
9    end
```

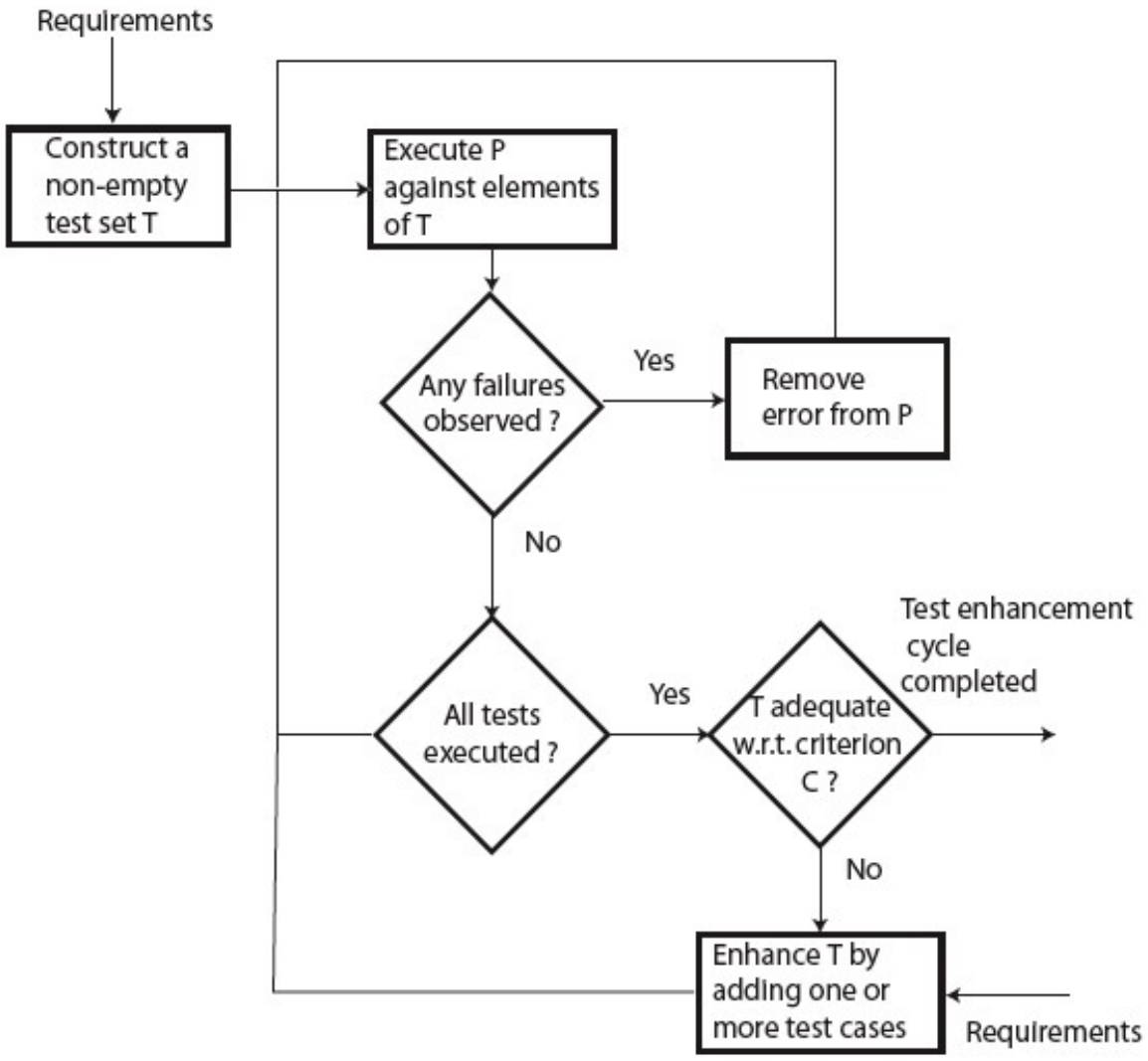
This program is correct as per the requirements of *sumProduct*. It has two paths denoted by p1 and p2.

Ce={ p1, p2}. T={t: <x=2, y=3>} is inadequate w.r.t. the path-based coverage criterion C.

Lesson

- An adequate test set might not reveal even the most obvious error in a program. This does not diminish in any way the need for the measurement of test adequacy as increasing coverage might reveal an error!

Test Enhancement: Procedure

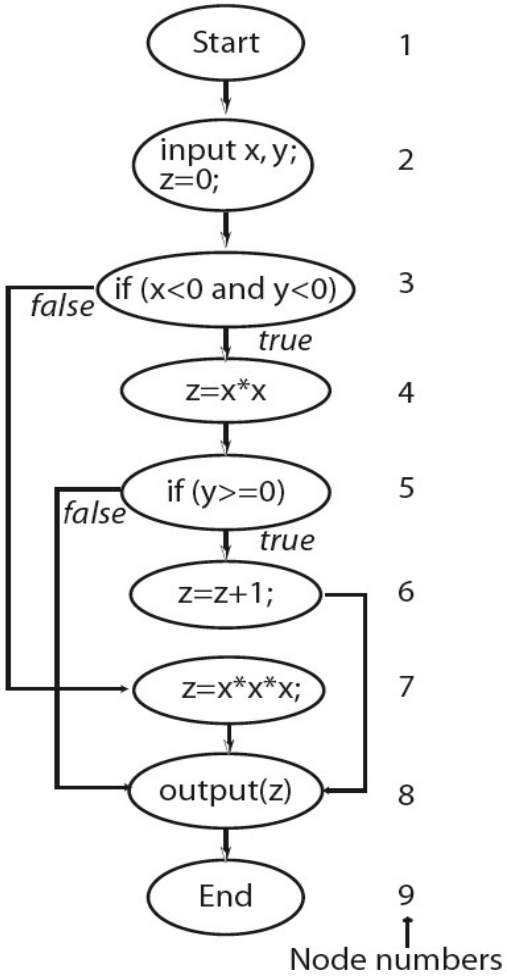


Infeasibility and test adequacy

- An element of the coverage domain is infeasible if it cannot be covered by any test in the input domain of the program under test.

- There does not exist an algorithm that would analyze a given program and determine if a given element in the coverage domain is infeasible or not. Thus it is usually the tester who determines whether or not an element of the coverage domain is infeasible.

Example: Flow graph and paths



p_1: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9]$

p_2: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 9]$

p_3: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 9]$

p1 is **infeasible** and cannot be traversed by any test case. This is because when control reaches node 5, condition $y \geq 0$ is false and hence control can never reach node 6.

Thus any test adequate with respect to the path coverage criterion for the exponentiation program will only cover p2 and p3

Index

- Introduction
- Black box testing techniques
- White box testing techniques

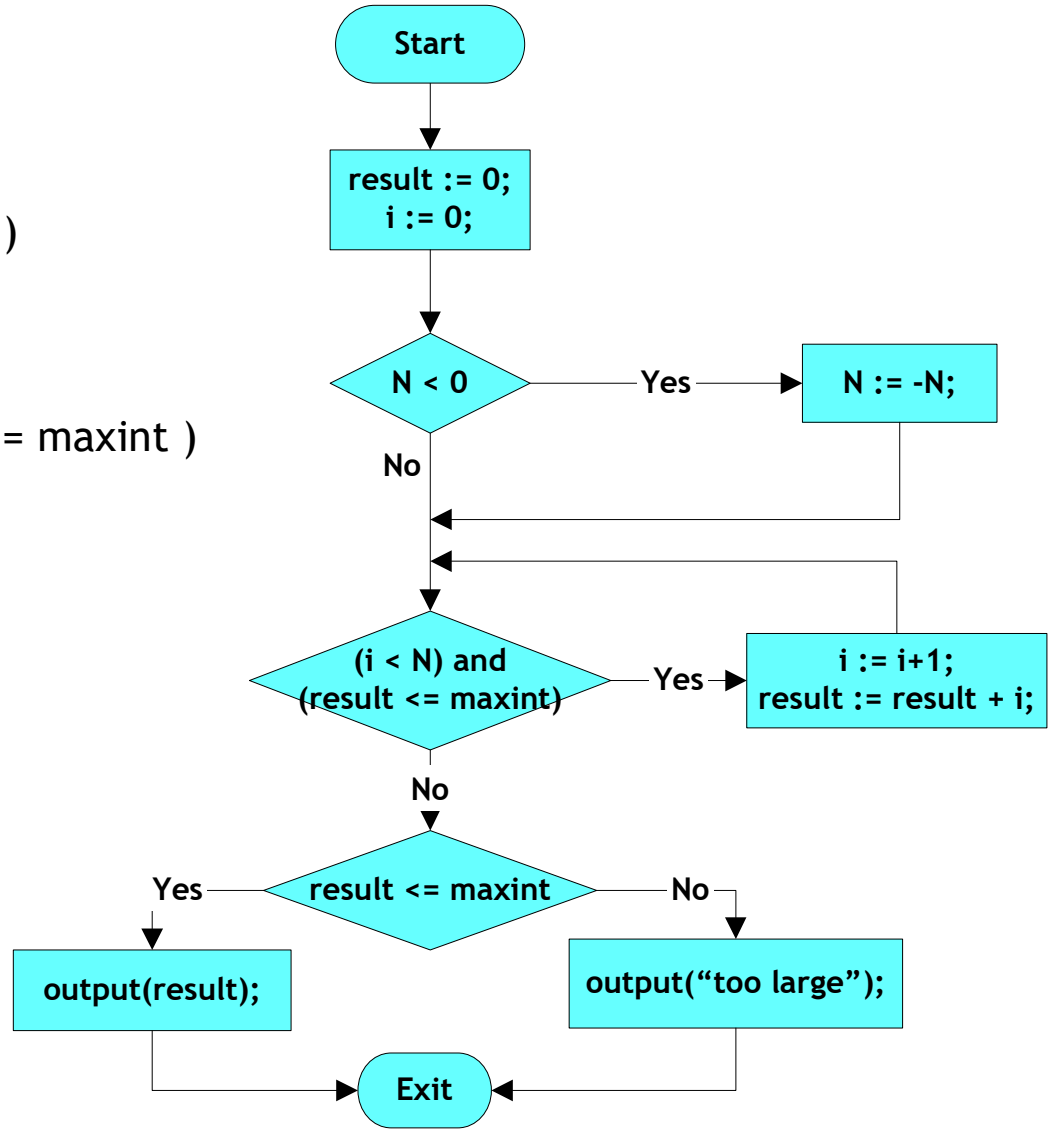
White-box testing

- Derivation of test cases according to program structure. Knowledge of the program is used to identify additional test cases
- Used mainly for unit testing
- Programming language dependent
- Extent to which (source) code is executed, i.e., covered
- Different kind of coverage :
  - based on control flow analysis - statement, decision, condition, decision and condition, MC/DC, path, ...
  - based on data flow analysis – p-use, c-use, ...

Control flow analysis
Example

```
1   PROGRAM  sum ( maxint, N : INT )
2     INT    result := 0 ;   i := 0 ;
3     IF   N < 0
4     THEN   N  :=  - N ;
5     WHILE   ( i < N )  AND   ( result <= maxint )
6     DO   i  :=  i + 1 ;
7            result  :=  result + i ;
8     OD;
9     IF   result <= maxint
10    THEN   OUTPUT ( result )
11    ELSE   OUTPUT ( “too large” )
12  END.
```

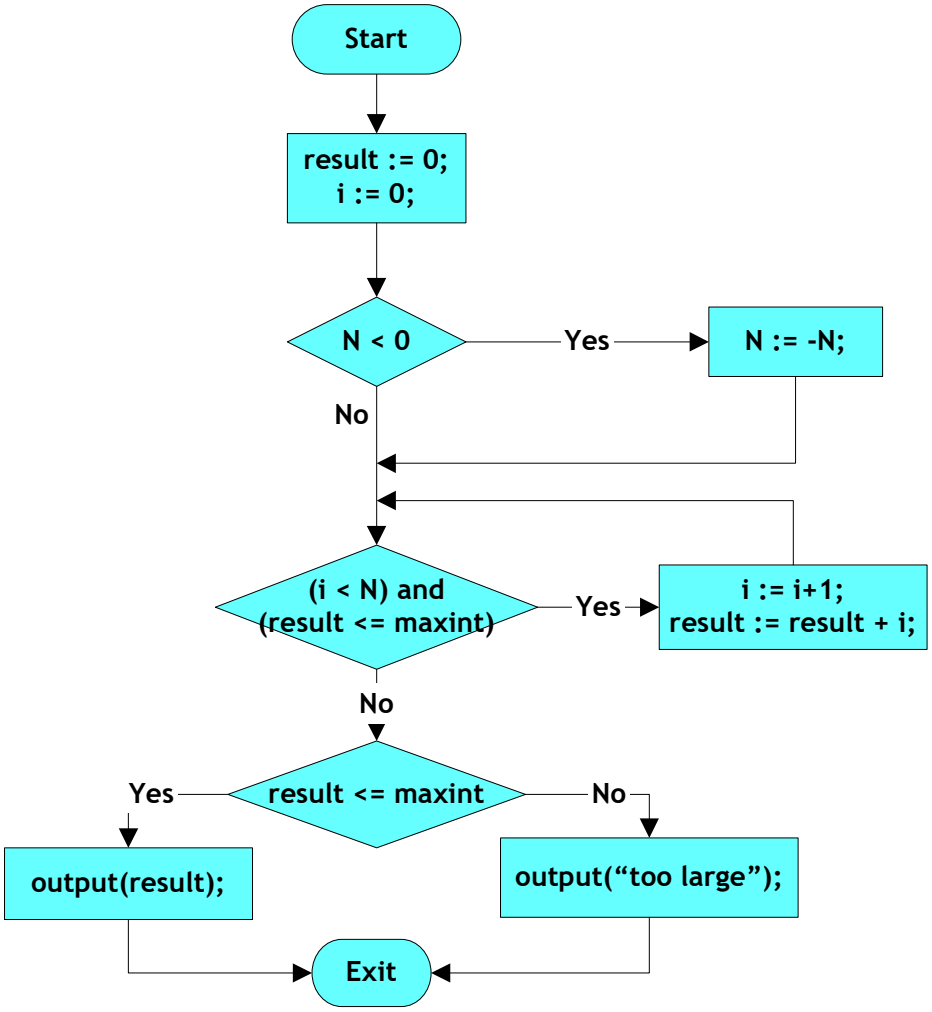
$$result = \begin{cases} \sum_{K=0}^{|N|} k, & \text{if this <= maxint} \\ error, & \text{otherwise} \end{cases}$$



Statement coverage

- Execute (exercise) every statement of a program
  - Generate a set of test cases such that each statement of the program is executed at least once
- Weakest white-box criterion
- Analysis supported by many commercial and freeware tools (test coverage or code coverage tools)
  - Standard Unix tool: *tcov*
  - A listing indicates how often each statement was executed and the percentage of statements executed
- Note: in case of unreachable statements, full statement coverage is not possible

Example : statement coverage



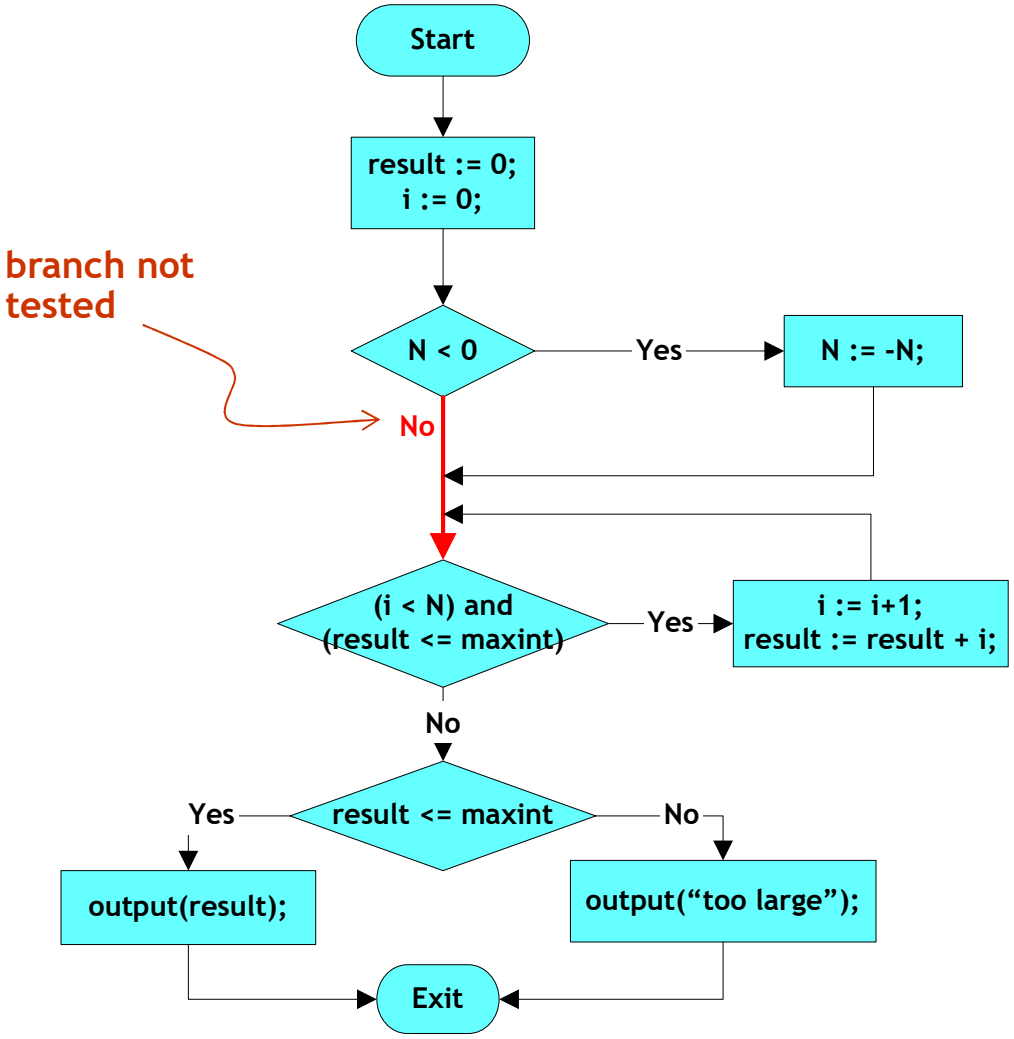
Tests for complete statement (node) coverage:

inputs		outputs
maxint	N	result
10	-1	1
0	-1	too large

Decision (or branch) coverage

- **Execute every branch of a program :**
  each possible outcome of each decision occurs at least once
- Example:
  - simple decision: IF b THEN s1 ELSE s2
    - b should be tested for true and false
  - multiple decision:
    CASE x OF
    1 : ….
    2 : ….
    3 : ….
- Stronger than statement coverage
  - IF THEN without ELSE – if the condition is always true all the statements are executed, but branch coverage is not achieved (infeasibility)

Example : decision (or branch) coverage



Tests for complete statement (node) coverage:

inputs		outputs
maxint	N	result
10	-1	1
0	-1	too large

are not sufficient for decision (branch) coverage!

Take:

inputs		outputs
maxint	N	result
10	3	6
0	-1	too large

for complete decision (branch) coverage

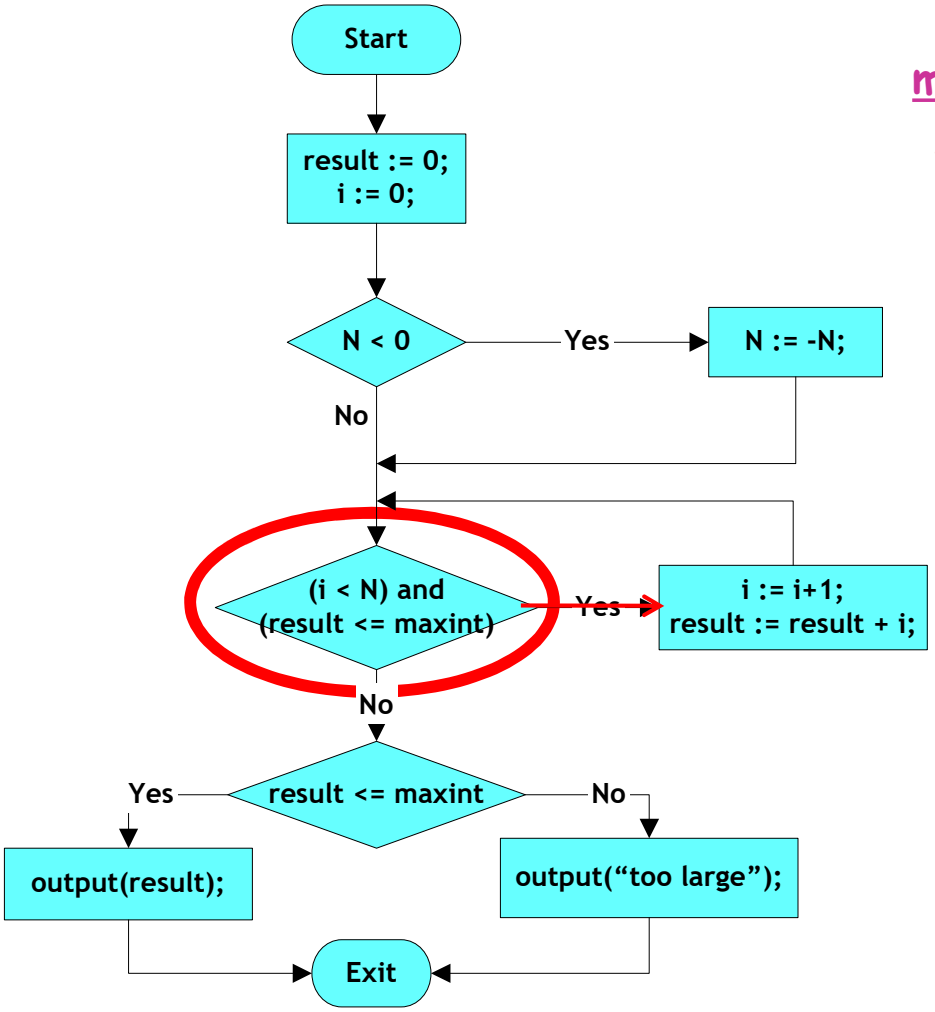
Condition coverage

- Design test cases such that **each possible outcome** of each condition in a decision (composite condition) **occurs at least once**

- Example:
  - **decision** (i < N) AND (result <= maxint)
    consists of **two** conditions : (i < N) , (result <= maxint)
  - test cases should be designed such that each condition gets value **true** and **false** at least once

- Last test cases of previous slides already guarantee condition (and branch) coverage

Condition and decision (or condition / decision) coverage



Test cases:

maxint	N	i	result	i<N	result<=maxint
-1	1	0	0	true	false
1	0	0	0	false	true

give condition coverage
for all conditions

But don't preserve
branch coverage

⇓

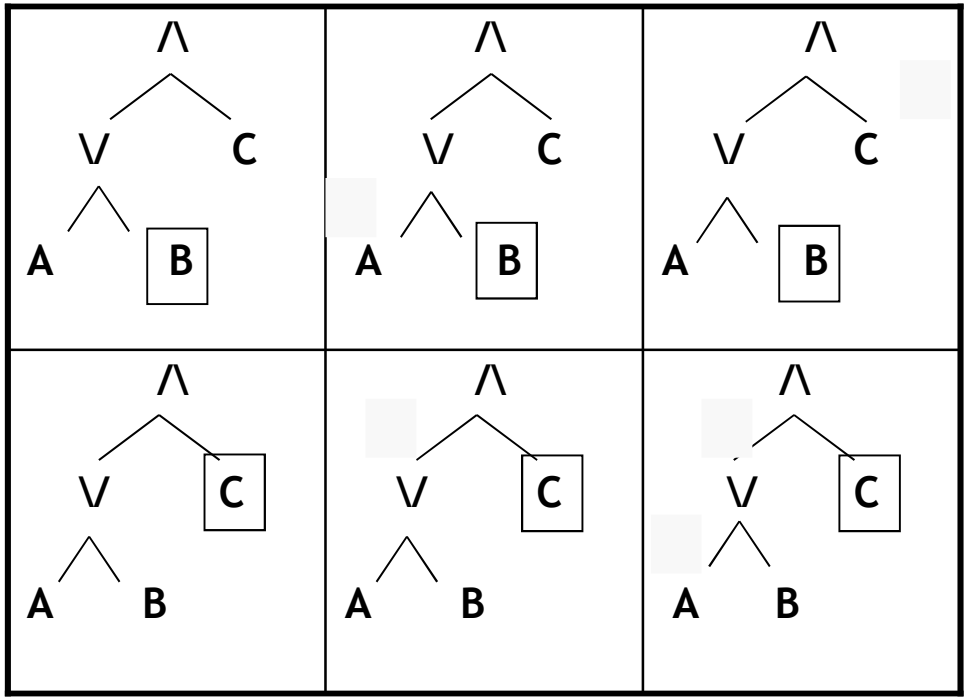
always take care that
condition coverage
preserves branch coverage :
condition and decision coverage

Modified Condition/Decision Coverage

- Also known as MC/DC or MCDC
- Design test cases such that
  - every decision in the program has taken **all possible outcomes at least once** (decision coverage)
  - every condition in a decision in the program has taken **all possible outcomes** at least once (condition coverage)
  - every condition in a decision has been shown to **independently affect that decision's outcome**; a condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions
    - **condition** – a Boolean expression containing no Boolean operators
    - **decision** – a Boolean expression composed of conditions and zero or more Boolean operators
- Created at Boeing, required for level A (critical) software for the Federal Aviation Administration (FAA) in the USA by RCTA/DO-178B

Modified Condition/Decision Coverage

(A \/ B) /\ C



Modified Condition/Decision Coverage

- Test cases required to meet the MC/DC criteria

Test case	Conditions		Decision
	A	B	A **and** B
1	True	True	True
2	False	True	False
3	True	False	False

Test case	Conditions		Decision
	A	B	A **or** B
1	False	False	False
2	True	False	True
3	False	True	True

Test case	Condition	Decision
	A	**not** A
1	True	False
2	False	True

Test case	Conditions		Decision
	A	B	A **xor** B
1	True	True	False
2	False	True	True
3	True	False	True

(or another combination)

Modified Condition/Decision Coverage

- Consider the following fragment of code:

```
if
    A or (B and C)
then
    do_something;
else
    do_something_else;
end if;
```

- MC/DC may be achieved with the following set of test inputs (note that there are alternative sets of test inputs, which will also achieve MC/DC):

Case	A	B	C	Outcome
1	FALSE	FALSE	TRUE	FALSE
2	TRUE	FALSE	TRUE	TRUE
3	FALSE	TRUE	TRUE	TRUE
4	FALSE	TRUE	FALSE	FALSE

are not evaluated if logical operators short-circuit

cases 2 a 4 are sufficient for branch and condition coverage, but only if logical operators do not short-circuit

- Because:
  - A is shown to independently affect the outcome of the decision condition by case 1 and case 2
  - B is shown to independently affect the outcome of the decision condition by case 1 and case 3
  - C is shown to independently affect the outcome of the decision condition by case 3 and case 4

Multiple condition coverage

- Design test cases for each combination of conditions
- Example:
  - (i < N) (result <= maxint)

    false false

    false true

    true false

    true true
- Implies decision, condition, decision and condition, modified branch/condition coverage
- But : exponential blow-up ($2^{\text{number of conditions}}$)
- Again : some combinations may be infeasible

Path coverage

- Execute every possible path of a program, i.e., every possible sequence of statements
- Strongest white-box criterion (based on control flow analysis)
- Usually impossible: infinitely many paths (in case of loops)
- So: not a realistic option
- But note : enormous reduction w.r.t. all possible test cases (each sequence of statements executed for only one value) (doesn't mean exhaustive testing)

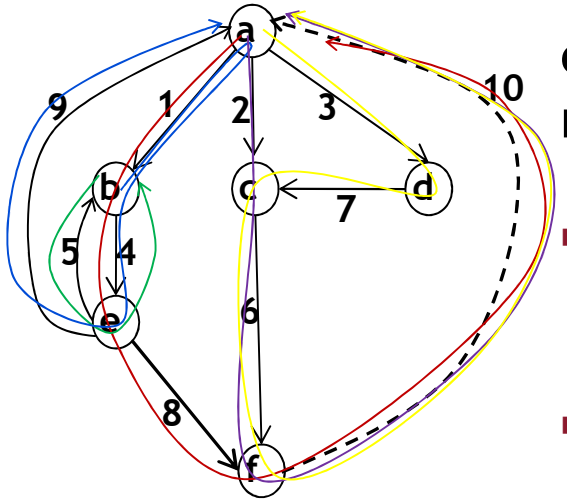
Independent path (or basis path) coverage

- Obtain a maximal set of linearly independent paths (also called a basis of independent paths)
  - If each path is represented as a vector with the number of times that each edge of the control flow graph is traversed, the paths are linearly independent if it is not possible to express one of them as a linear combination of the others
- Generate a test case for each independent path
- The number of linearly independent paths is given by the McCabe's *cyclomatic complexity* of the program
  - Number of edges - Number of nodes + 2 in the control flow graph
  - Measures the structural complexity of the program

Independent path (or basis path) coverage

- Problem: some paths may be impossible to execute
- Also called structured testing (see McCabe for details)
- McCabe's argument: this approach produces a number of test cases that is proportional to the complexity of the program (as measured by the cyclomatic complexity), which, in turn, is related to the number of defects expected
- More information:
  - http://www.mccabe.com/iq_research_metrics.htm
  - "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric", Arthur H. Watson, Thomas J. McCabe, NIST Special Publication 500-235

Example: Independent path coverage

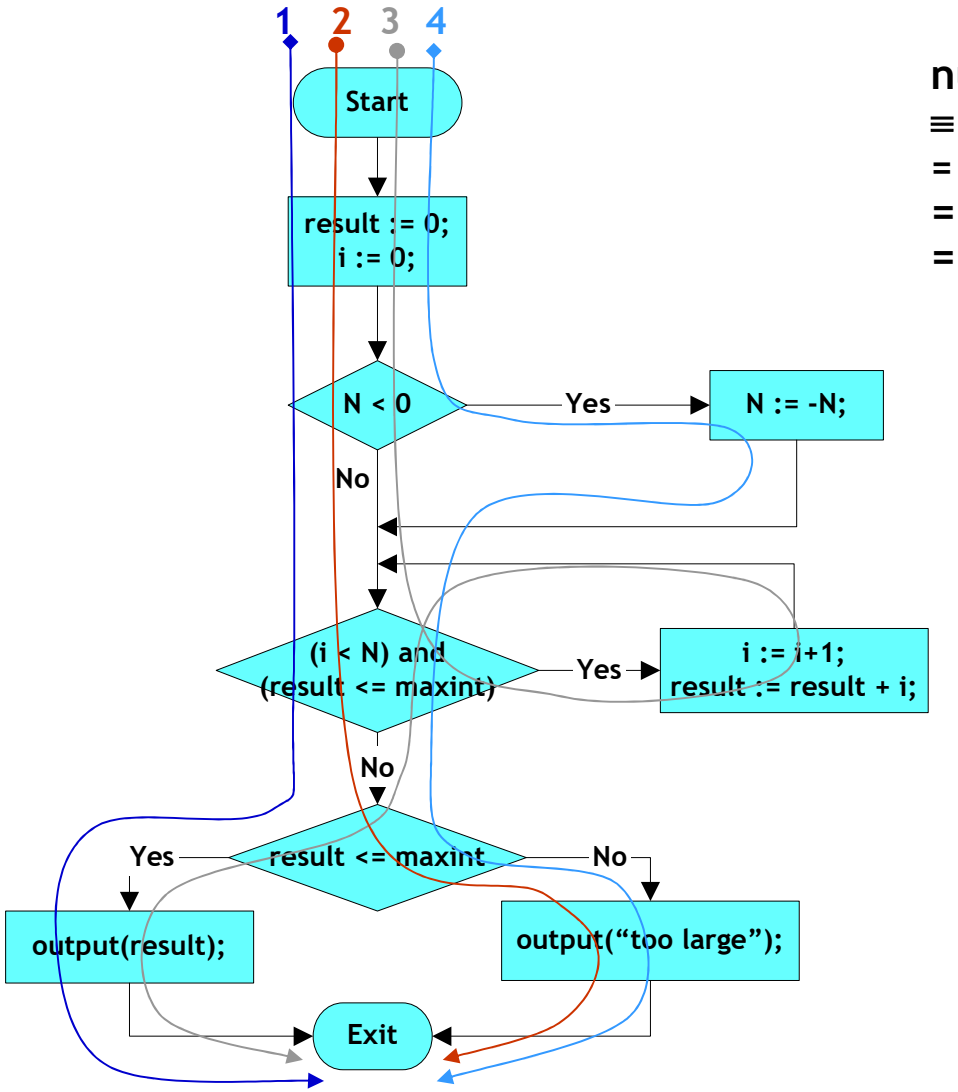


	1	2	3	4	5	6	7	8	9	10
abefa	1	0	0	1	0	0	0	1	0	1
beb	0	0	0	1	1	0	0	0	0	0
abea	1	0	0	1	0	0	0	0	1	0
acfa	0	1	0	0	0	1	0	0	0	1
adcfa	0	0	1	0	0	1	1	0	0	1

Theorem: In a strongly connected graph, G, the cyclomatic number is equal to the maximum number of linear independent circuits.

- 5 independent circuits
  - (abefa),(beb),(abea),(acfa),(adcfa)
- 5 independent paths
  - (abef), (abebef),(abeabef),(acf),(adcf)
  - This set of independent paths forms the basis for the set of all circuits in the graph.
- For instance
  - The path $(abea(be)^3f) = 2(abebef)-(abef)$

Example: Independent path coverage

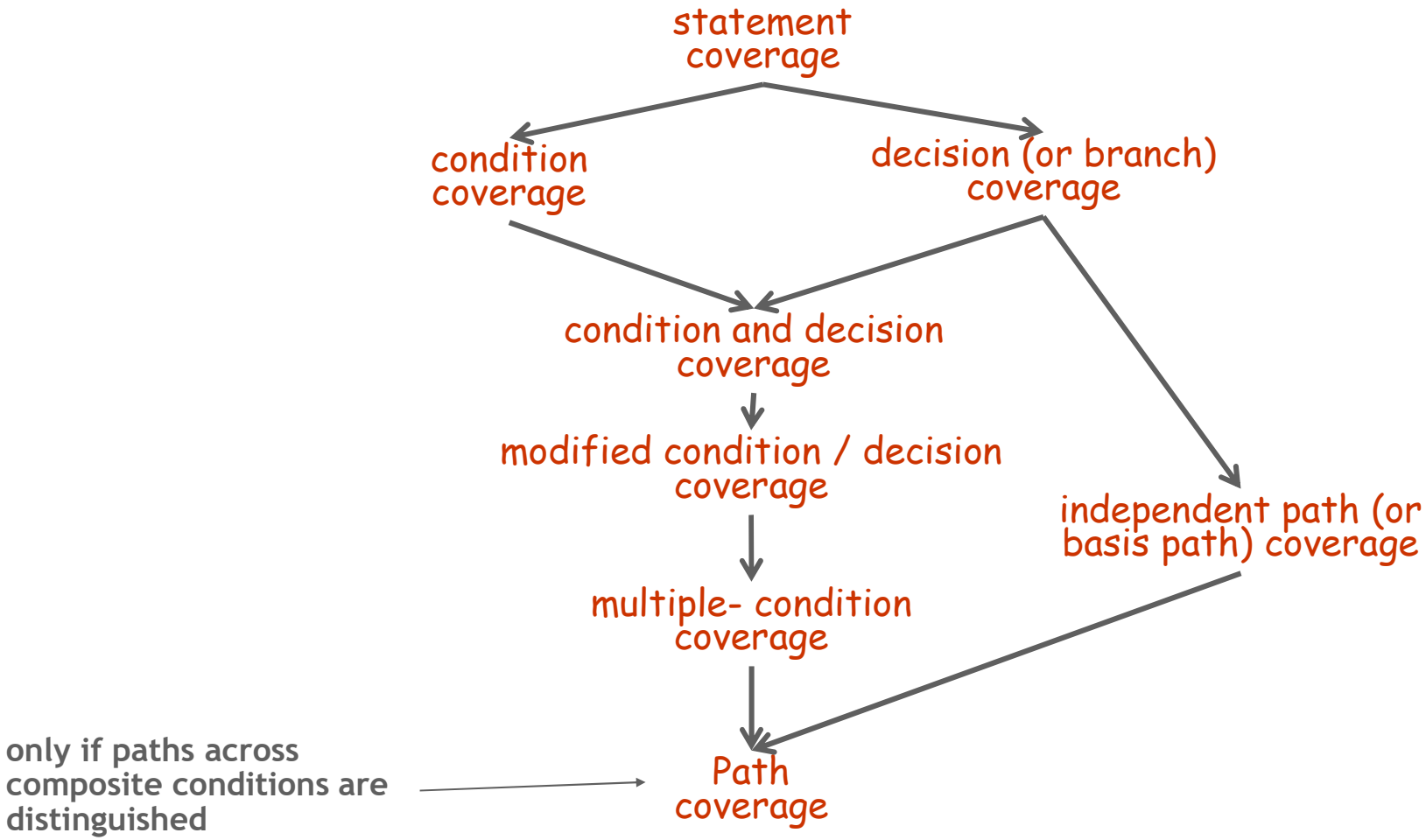


number of independent paths
≡ cyclomatic complexity
= number of edges - number of nodes + 2
= 12 - 10 + 2
= 4

Test cases

Path	inputs		outputs
	maxint	N	result
1	1	0	0
2	-1	0	too large
3	-1	-1	too large
4	10	1	1

White-Box Testing : Overview



Data flow testing

- We say a variable is **defined** in a statement when its value is assigned or changed

```
Y = 26 * X
```

  - This is indicated as a *def* for the variable Y

- We say a variable is **used** in a statement when its value is utilized in a statement. The value of the variable is not changed. Data flow roles:
  - *c-use* for variable X:

```
Y = 26 * X
```

  - *p-use* for variable X:

```
if (X  > 98)
      Y  = max
```

  - others: *undefined* or *dead*

Data flow testing

- Coverage criteria
  - All def
  - All p-uses
  - All c-uses/some p-uses
  - All p-uses/some c-uses
  - All uses
  - All def-use paths
  - … and several variants of this technique …

- The strongest of these criteria is *all def-use paths*. This includes all *p-* and *c-uses*.

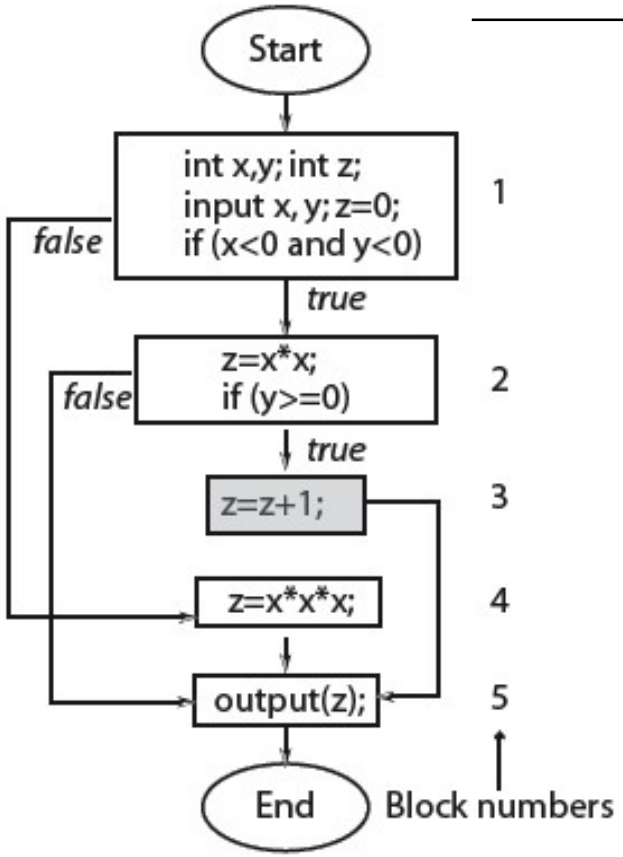
Data flow graph

- A data-flow graph of a program, also known as def-use graph, captures the flow of definitions (also known as defs) across basic blocks in a program.
- It is similar to a control flow graph of a program in that the nodes, edges, and all paths thorough the control flow graph are preserved in the data flow graph. An example follows.

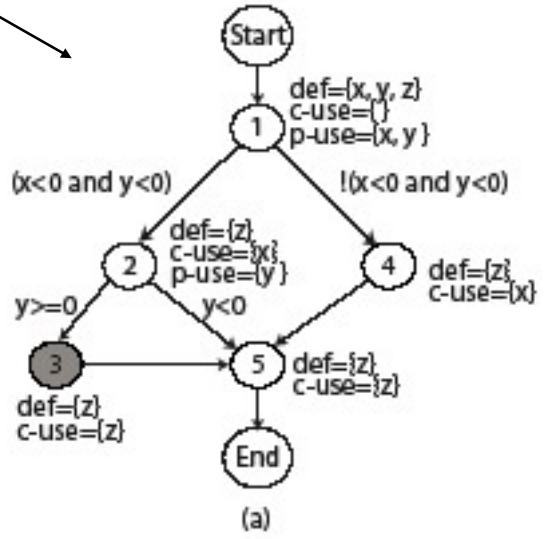
Data flow graph: Example

- Given a program, find its basic blocks, compute defs, c-uses and p-uses in each block. Each block becomes a node in the def-use graph (this is similar to the control flow graph).
- Attach defs, c-use and p-use to each node in the graph. Label each edge with the condition which when true causes the edge to be taken.
- We use $d_i(x)$ to refer to the definition of variable x at node i. Similarly, $u_i(x)$ refers to the use of variable x at node i.

Data flow graph: Example (contd.)



Node (or Block)	def	c-use	p-use
1	{x, y, z}	{ }	{x, y}
2	{z}	{x}	{y}
3	{z}	{z}	{ }
4	{z}	{x}	{ }
5	{ }	{z}	{ }



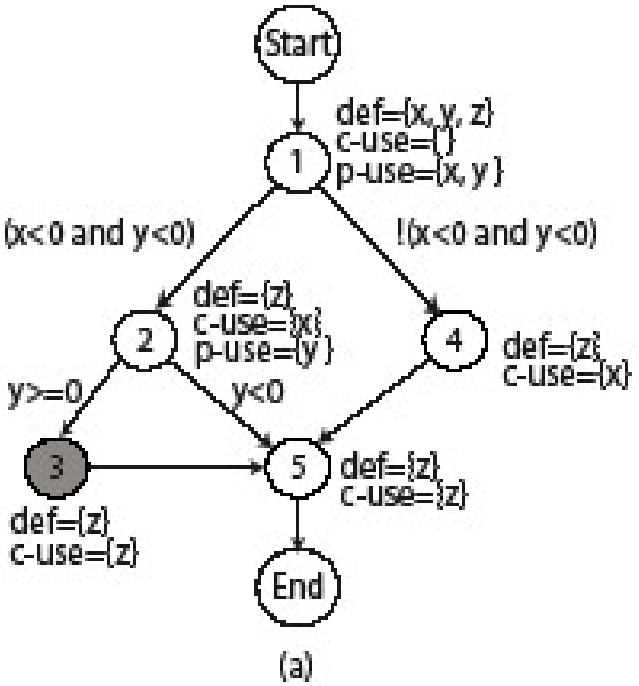
Unreachable node

Def-clear path

Any path starting from a node at which variable x is defined and ending at a node at which x is used, without redefining x anywhere else along the path, is a *def-clear* path for x.

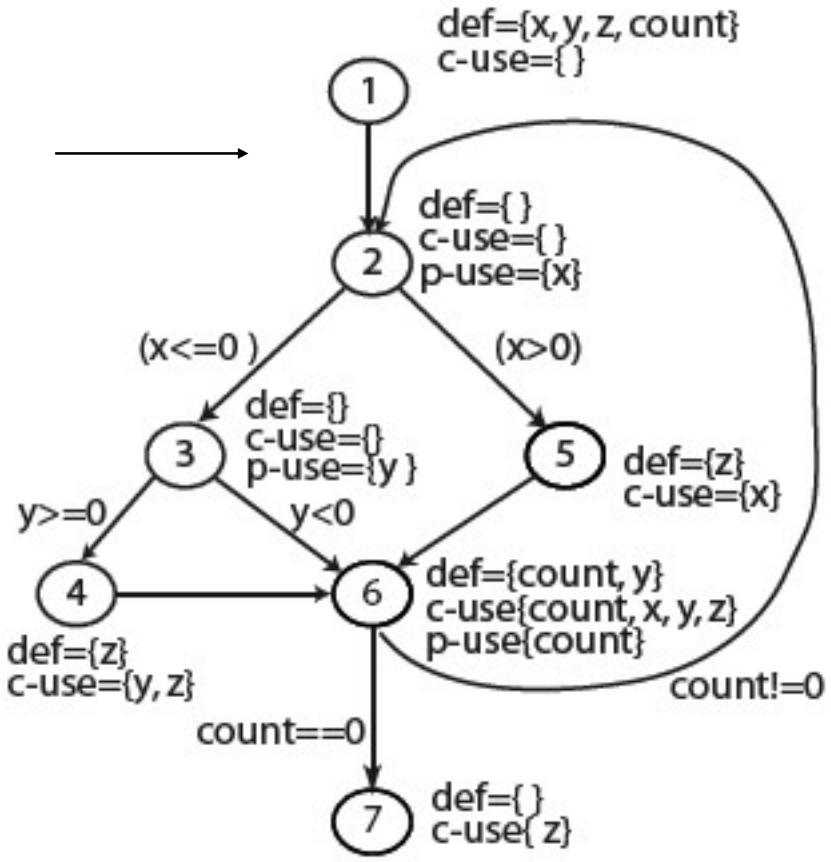
Path 2-5 is def-clear for variable z defined at node 2 and used at node 5. Path 1-2-5 is NOT def-clear for variable z defined at node 1 and used at node 5.

Thus definition of z at node 2 is live at node 5 while that at node 1 is not live at node 5.



Def-clear path (another example)

```
1    begin
2      float x, y, z=0.0;
3      int count;
4      input (x, y, count);
5      do {
6        if (x≤0) {
7          if (y≥0) {
8            z=y*z+1;
9          }
10       }
11       else{
12         z=1/x;
13       }
14       y=x*y+z
15       count=count-1
16     while (count>0)
17     output (z);
18   end
```



Node	Lines
1	1, 2, 3, 4
2	5, 6
3	7
4	8, 9, 10
5	11, 12, 13
6	14, 15, 16
7	17, 18

Find def-clear paths for defs and uses of x and z.
Which definitions are live at node 4?

Mutation testing (fault injection)

- Starts with a code component and its associated test cases (in a state such that the code passes all test cases)
- The original code component is modified in a simple way (replace operators, constants, etc.) to provide a set of similar components that are called mutants, based on typical errors
- The original test cases are run with each mutant
- Live mutants cannot be distinguished from the original program (parent)
- Distinguishing a mutant from its parent is referred to as killing such mutant
- If a mutant remains live (passes all the test cases), then either the mutant is equivalent to the parent (and is ignored), or it is not equivalent, in which case **additional test cases** should be developed in order to kill such mutant
- The rate of mutants "killed" (after removing mutants that are equivalent to the original code) gives an indication of the rate of undetected defects that may exist in the original code

Class testing / coverage

- In object oriented systems, the units to be tested are typically classes, but can also be methods or clusters of related classes
- Complete test coverage of a class involves
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - *Exercising the object in all possible states*
- Each test case typically exercises a possible object lifecycle, with alternating operation calls to change and query the object's state
- Encapsulation, inheritance and polymorphism complicate the design of test cases
  - Encapsulation – how to check the object's state?
  - Inheritance – how to (unit) test abstract classes, abstract methods and interfaces?
  - Polymorphism – how to test methods with *callbacks* (from super-class to sub-class)? (same problem with event handlers)

White-Box testing : How to Apply ?

- Don’t start with designing white-box test cases!
- Start with black-box test cases
  (equivalence partitioning, boundary value analysis,
  cause effect graphing, derivation with formal methods,…)
- Check white-box coverage
  (statement, branch, condition,… , coverage)
- Use a testing coverage tool
- Design additional white-box test cases for not covered code

References and further reading

- Practical Software Testing, Ilene Burnstein, Springer-Verlag, 2003
- Software Testing, Ron Patton, SAMS, 2001
- The Art of Software Testing, Glenford J. Myers, Wiley & Sons, 1979 (Chapter 4 - Test Case Design)
- Software testing techniques, Boris Beizer, Van Nostrand Reinhold, 2nd Ed, 1990
- Foundations of Software Testing, by Aditya P. Mathur, Pearson Education, 2008
- Testing Computer Software, 2nd Edition, Cem Kaner, Jack Falk, Hung Nguyen, John Wiley & Sons, 1999
- Software Engineering, Ian Sommerville, 6th Edition, Addison-Wesley, 2000
- http://www.swebok.org/ - Guide to the Software Engineering Body of Knowledge (SWEBOK), IEEE Computer Society
- http://www.mccabe.com/iq_research_metrics.htm
- "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric", Arthur H. Watson, Thomas J. McCabe, NIST Special Publication 500-235
- RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment Certification", Radio Technical Commission for Aeronautics, USA, December 1992
- Chilenski1994 John Joseph Chilenski and Steven P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing", Software Engineering Journal, September 1994, Vol. 9, No. 5, pp.193-200.
- A Practical Tutorial on Modified Condition / Decision Coverage, NASA / TM-2001-210876 (http://www.faa.gov/certification/aircraft/av-info/software/Research/MCDC%20Tutorial.pdf)
- A Complexity Measure, Thomas J. McCabe, IEEE Trans. Software Eng., volume 2, number 4, 1976, pages 308-320.

Exercise 1

If the pseudocode below were a programming language ,how many tests are required to achieve 100% statement coverage?

```
If x=3 then
        Display_messageX;
        If y=2 then
                Display_messageY;
        Else
                Display_messageZ;
Else
        Display_messageZ;
```

- 1. Choose the correct answer

  a) 1; b) 2; c) 3; d) 4

Using the same code example as question 17,how many tests are required to achieve 100% branch/decision coverage?

- 2. Choose the correct answer

  a) 1; b) 2; c) 3; d) 4

Exercise 2

- **Given the following code, which is true about the minimum number of test cases required for full statement and branch coverage:**

```
Read P
Read Q
IF P+Q > 100 THEN
   Print "Large"
ENDIF
If P > 50 THEN
   Print "P Large"
ENDIF
```

a) 1 test for statement coverage, 3 for branch coverage

b) 1 test for statement coverage, 2 for branch coverage

c) 1 test for statement coverage, 1 for branch coverage

d) 2 tests for statement coverage, 3 for branch coverage

e) 2 tests for statement coverage, 2 for branch coverage

Exercise 3 (1)

- A Program is written to meet the following requirements:
  - R1: Given coordinate positions x, y and z, and a direction valued d, the program must invoke one of the three functions fire-1, fire-2, fire-3 as per conditions bellow:
    - R1.1: Invoke fire-1 when (x<y) AND (z * z > y) AND (prev="East").
    - R1.2: Invoke fire-2 when (x<y) AND (z * z $\leq$ y) OR (current="South").
    - R1.3: Invoke fire-3 when none of the two conditions above is true.
  - R2: The invocation described above must continue until an input Boolean variable becomes true.

Exercise 3 (2)

```
1  begin
2    float x, y, z;
3    direction d;
4    string prev, current;
5    bool done;

6    input(done);
7    current="North";
8    while (~done)  {                                          ← condition C1
8              input (d);
10             prev=current; current=f(d);
11             input(x,y,z);
12             if ((x<y) and (z*z > y) and (prev=="East"))      ← Condition C2
13                             fire-1(x,y);
14             else if ((x<y) and (z*z<=y) or (current=="South"))   ← Condition C3
15                             fire-2(x,y);
16             else
17                             fire-3(x,y);
17             input(done);
18   }
19   output("Firing completed.");
20 end
```

Exercise 3 (3)

- Verify that the following set T1 of four tests, executed in the given order, is adequate with respect to statement, block, and decision coverage criteria but not with respect to the condition coverage criterion.

Test set T1						
Test	Requirement	done	d	x	y	z
t1	R1.2	False	East	10	15	3
t2	R1.1	False	South	10	15	4
t3	R1.3	False	North	10	15	5
t4	R2	True	-	-	-	-

Exercise 3 (4)

Test set T2						
Test	Requirement	done	d	x	y	z
t1	R1.2	False	East	10	15	3
t2	R1.1	False	South	10	15	4
t3	R1.3	False	North	10	15	5
t5	R1.1 and R1.2	False	South	10	5	5
t4	R2	True	-	-	-	-

Test set T2 is adequate according to MC/DC?

Exercise 3 (5)

- Verify that the following set T3, obtained by adding t6, t7, t8, and t9 to T2 is adequate with respect to MC/DC coverage criterion. Note again that sequencing of tests is important in this case (especially for t1 and t7)!

Test set T3						
Test	Requirement	done	d	x	y	z
t1	R1.2	False	East	10	15	3
t6	R1	False	East	10	5	2
t7	R1	False	East	10	15	3
t2	R1.1	False	South	10	15	4
t3	R1.3	False	North	10	15	5
t5	R1.1 and R1.2	False	South	10	5	5
t8	R1	False	South	10	5	2
t9	R1	False	North	10	5	2
t4	R2	True	-	-	-	-

Exercise 4

- Suppose that condition C=C1 AND C2 AND C3 has been coded as C'=C1 AND C3. Four tests that form an MC/DC adequate set for C' are in the following table. Verify that the following set of four tests is MC/DC adequate but does not reveal the error.

	Test	C	C'	Error detected?
	C1, C2, C3	C1 and C2 and C3	C1 and C3	
t1 t2	true, true, true false, false, false			
t3 t4	true, true, false false, false, true			