A Theory of Software Testing

2. Test Case Design Technique

A Theory of Software Testing

2. Test Case Design Technique

Test Case Design Technique

PART I

Contents

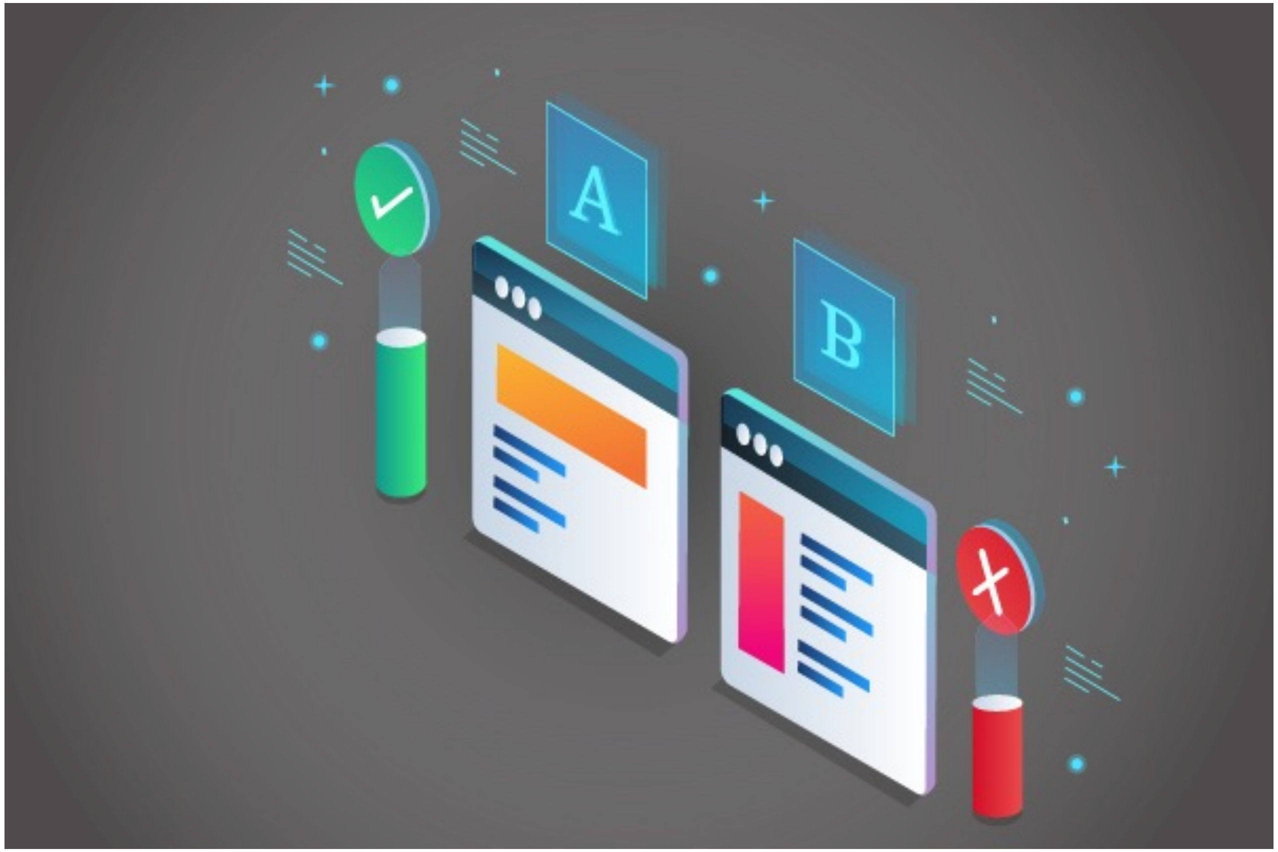
Part I

- Terminologies
- Specification-based Testing (Black-box Testing)

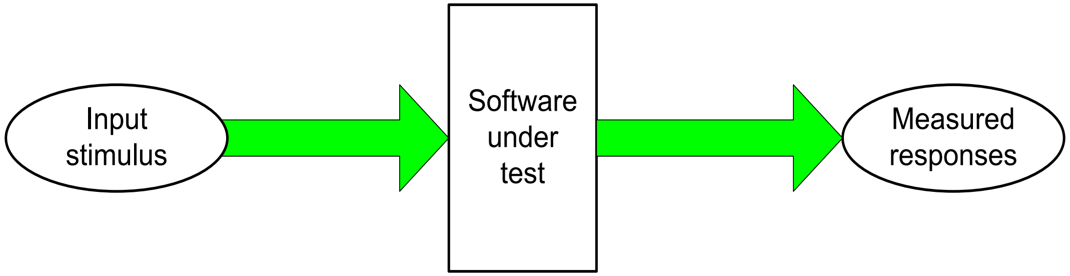
Part II (White-box Testing)

- Coverage-based Testing
- Structural Path-based Testing
- Dataflow Testing

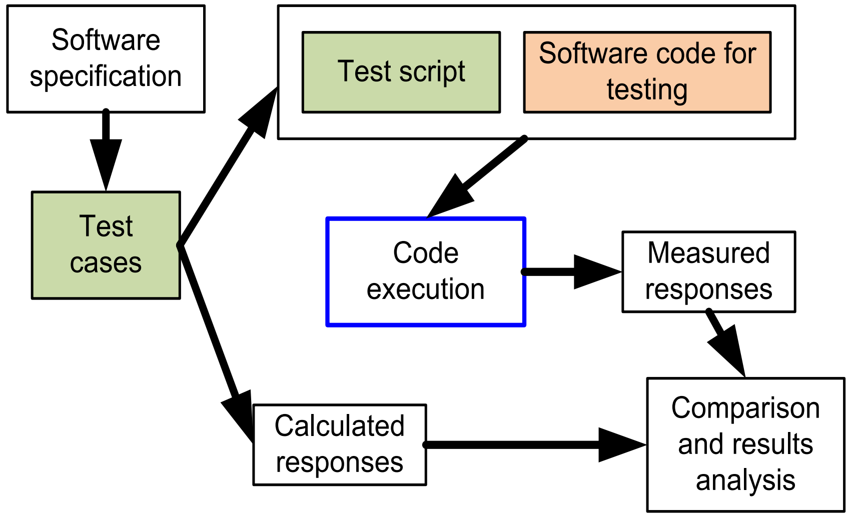
Terminologies



Dynamic Testing Procedure



(a) Fundamental concept of dynamic testing



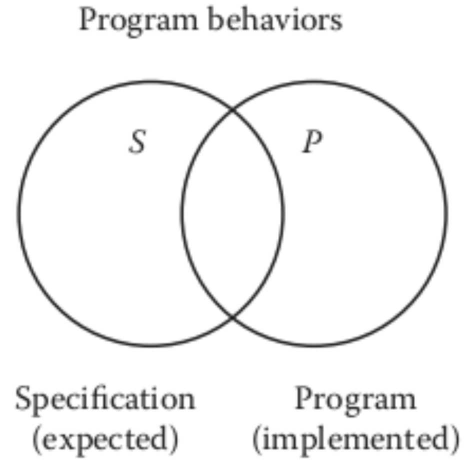
(b) Detailed aspects of dynamic testing

Test data and Test cases

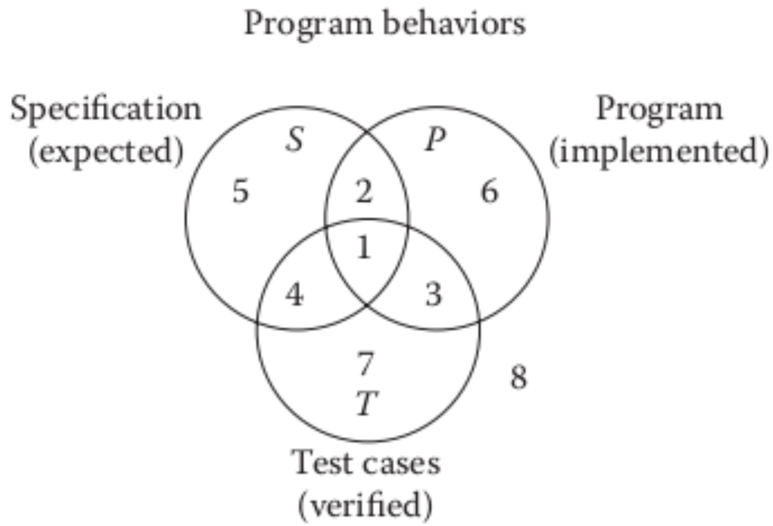
- Test data
  - Inputs which have been devised to test the system
- **Test cases**
  - **Inputs** to test the system and the **predicted outputs** from these inputs if the system operates according to its specification
- Test suite
  - A collection of test cases that are intended to be used to test a software program
- Test oracle
  - A mechanism used by software testers for determining whether a test has passed or failed
    - Specification and documentation, other products, and so on

Specification, Program, and Test Case

- Specification & Program
  - Spec. and Program are not exactly matched



- Test Cases
  - Scope 1
    - Expected, Programmed and Tested
  - Scope 2, 5
    - Expected, but not tested
  - Scope 6
    - Programmed, not expected
    and not tested



IEEE Std. Fault Types (1/2)

- IEEE Standard Classification for Software Anomalies(1993)
  - Input / Output Faults, Logic Faults

Table 1.1 Input/Output Faults

Type	*Instances*
Input	Correct input not accepted
	Incorrect input accepted
	Description wrong or missing
	Parameters wrong or missing
Output	Wrong format
	Wrong result
	Correct result at wrong time (too early, too late)
	Incomplete or missing result
	Spurious result
	Spelling/grammar
	Cosmetic

Table 1.2 Logic Faults

Missing case(s)
Duplicate case(s)
Extreme condition neglected
Misinterpretation
Missing condition
Extraneous condition(s)
Test of wrong variable
Incorrect loop iteration
Wrong operator (e.g., < instead of ≤)

IEEE Std. Fault Types (2/2)

Table 1.3 Computation Faults

Incorrect algorithm
Missing computation
Incorrect operand
Incorrect operation
Parenthesis error
Insufficient precision (round-off, truncation)
Wrong built-in function

Table 1.4 Interface Faults

Incorrect interrupt handling
I/O timing
Call to wrong procedure
Call to nonexistent procedure
Parameter mismatch (type, number)
Incompatible types
Superfluous inclusion

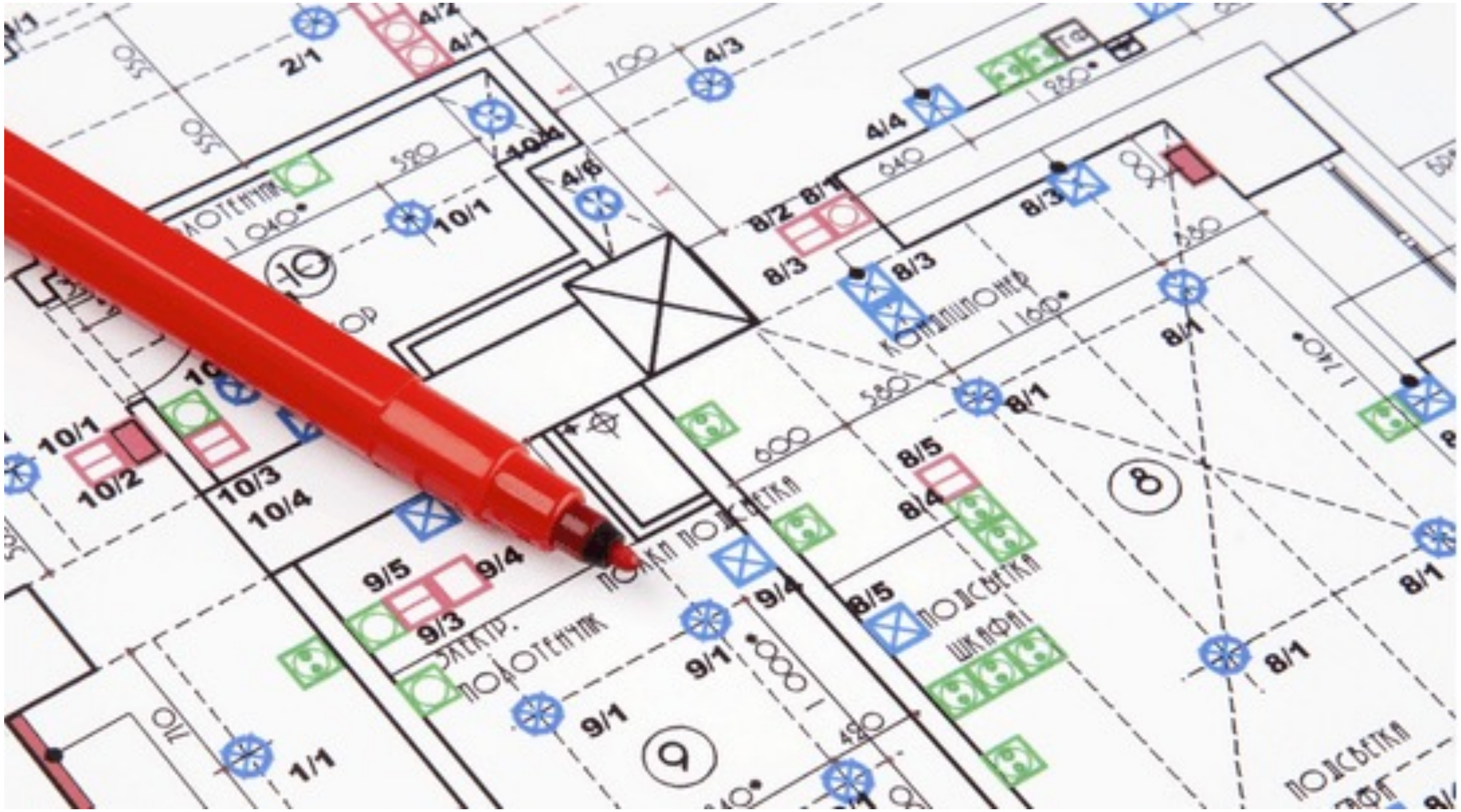
Table 1.5 Data Faults

Incorrect initialization
Incorrect storage/access
Wrong flag/index value
Incorrect packing/unpacking
Wrong variable used
Wrong data reference
Scaling or units error
Incorrect data dimension
Incorrect subscript
Incorrect type
Incorrect data scope
Sensor data out of limits
Off by one
Inconsistent data

Black-box testing vs. White-box testing

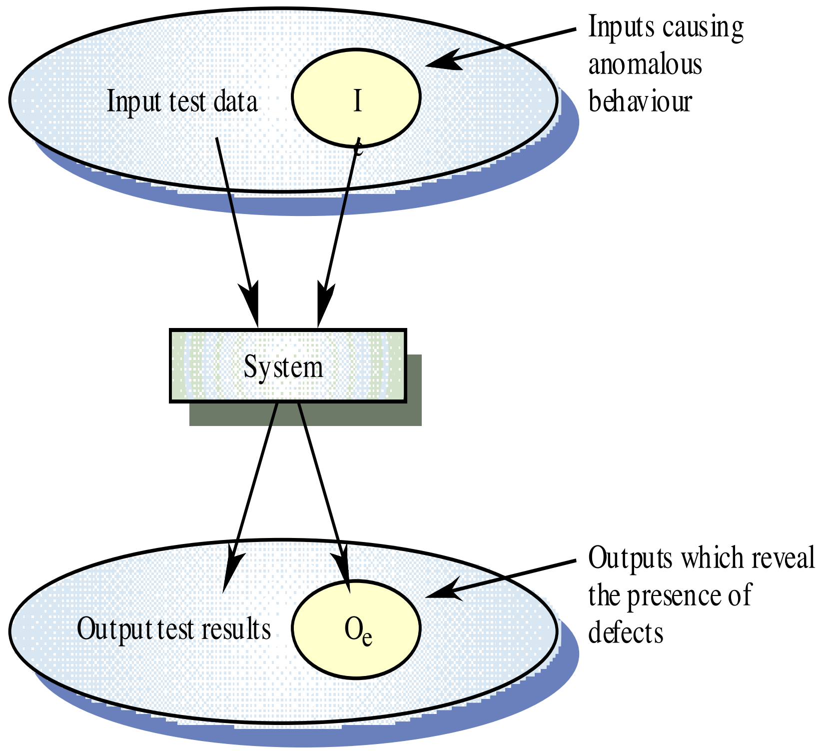
- Black-box testing
  - An approach to testing where the program is considered as a ‘black-box’
  - The program test cases are **based on the system specification**
  - Test planning can begin early in the software process
- White-box testing
  - Derivation of test cases according to program structure.
  - Knowledge of the program is used **to identify additional test cases**

Specification-based Testing



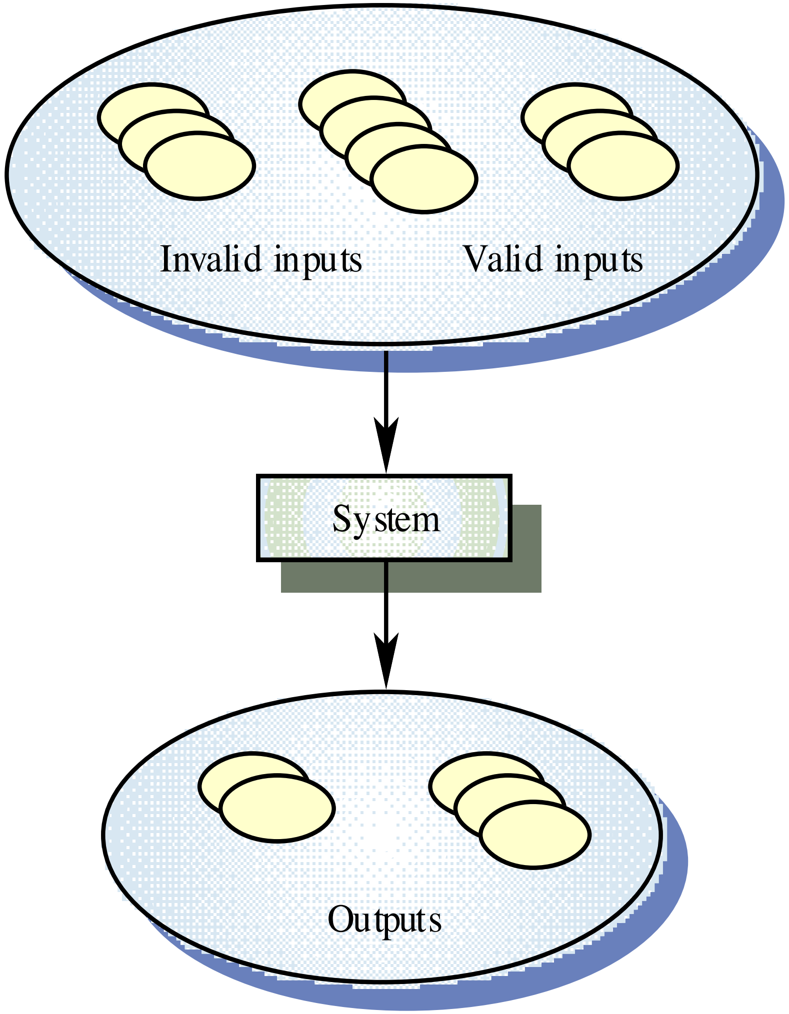
Black-box testing

- An approach to testing where the program is considered as a ‘black-box’
- The program test cases are **based on the system specification**
- Test planning can begin early in the software process



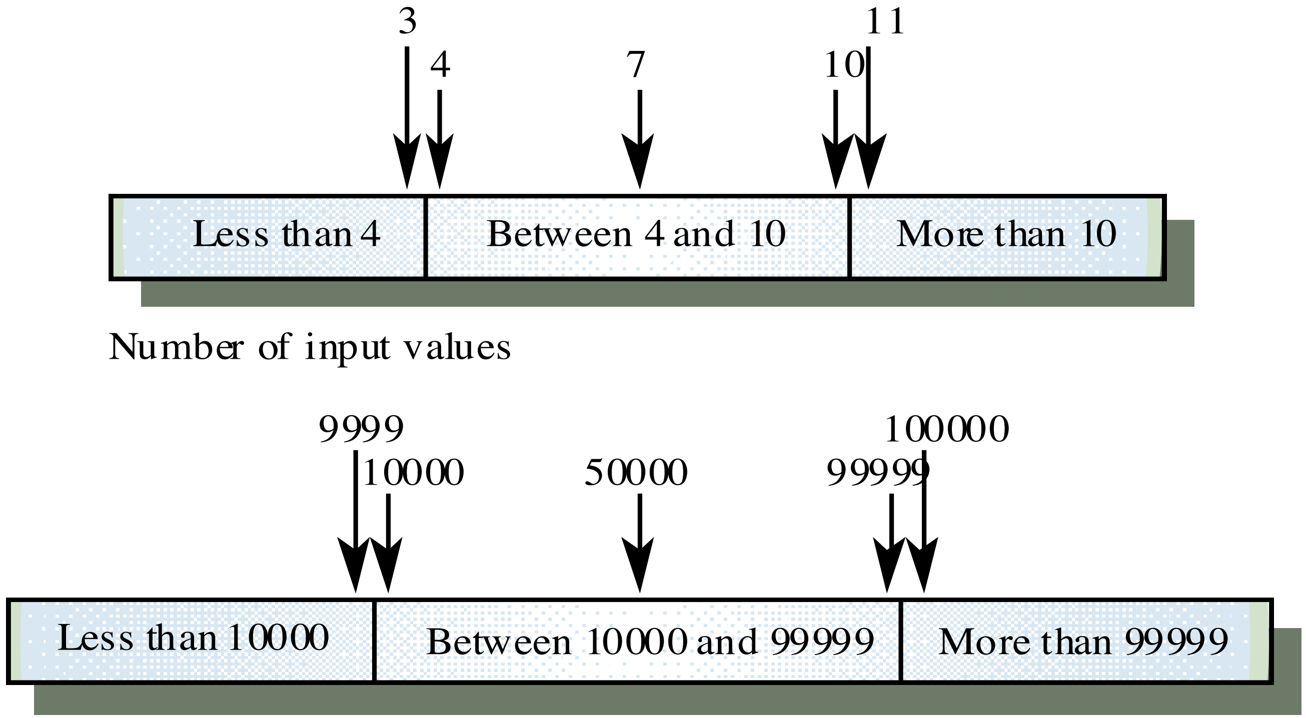
Equivalence partitioning

- Input data and output results often fall into different classes where all members of a class are related
- Each of these classes is an equivalence partition where the program **behaves in an equivalent way** for each class member
- Test cases should be chosen from each partition



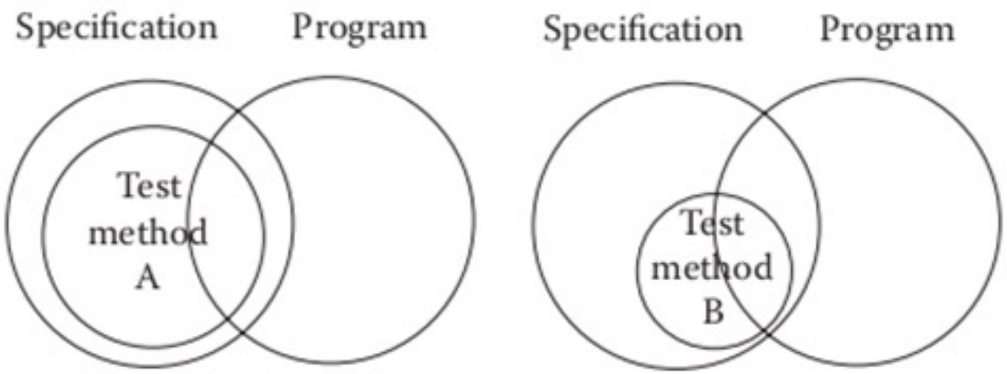
Equivalence partitioning

- Partition system inputs and outputs into ‘equivalence sets’
  - If input is a 5-digit integer between 10,000 and 99,999, equivalence partitions are <0-9999>, <10000-99999> and <100000 - >
- Choose test cases at the boundary of these sets
  - 00000, 09999, 10000, 99999, 100000



Specification-based Testing(1/2)

- Black-box Testing (called as functional testing)

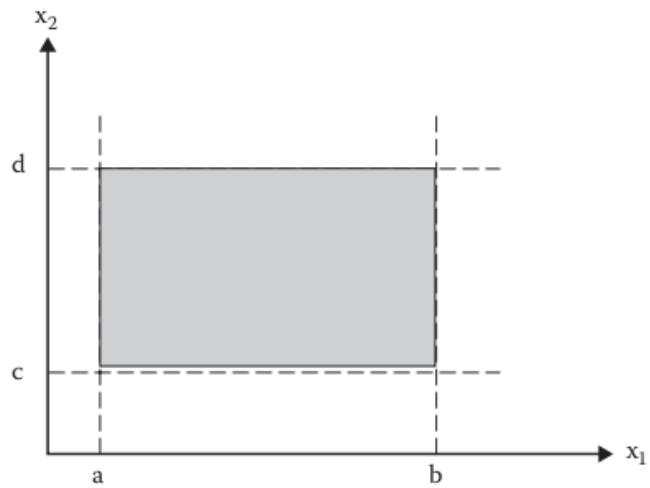


- Testing Approach
  - Boundary value Testing
  - Equivalence Class Testing
  - Decision Table-based Testing

- Strength
  - Test cases are independent of how the software is implemented
  - Test case development can occur in parallel with the implementation

Boundary Value Testing

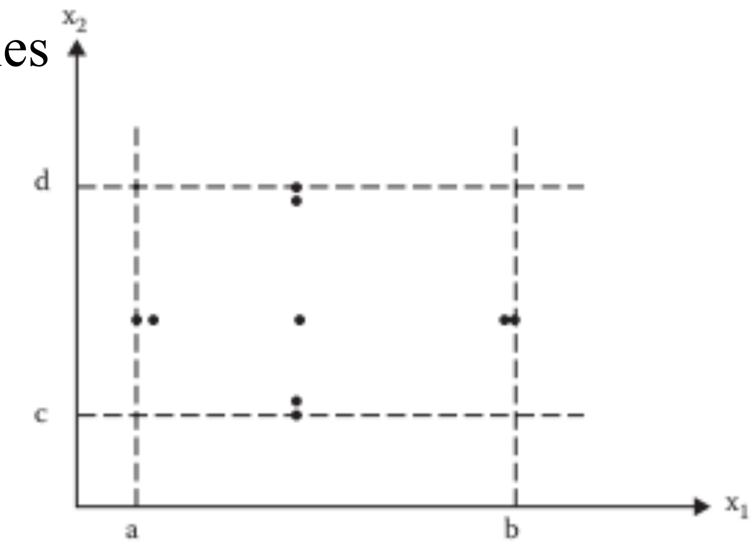
- Input Domain(or output domain) of variable is used
  - a <= x1 <= b,
    or c <= x2 <= d



- Testing Approach
  - Normal boundary value testing
  - Robust boundary value testing
  - Worst-case boundary value testing
  - Robust worst-case boundary value testing

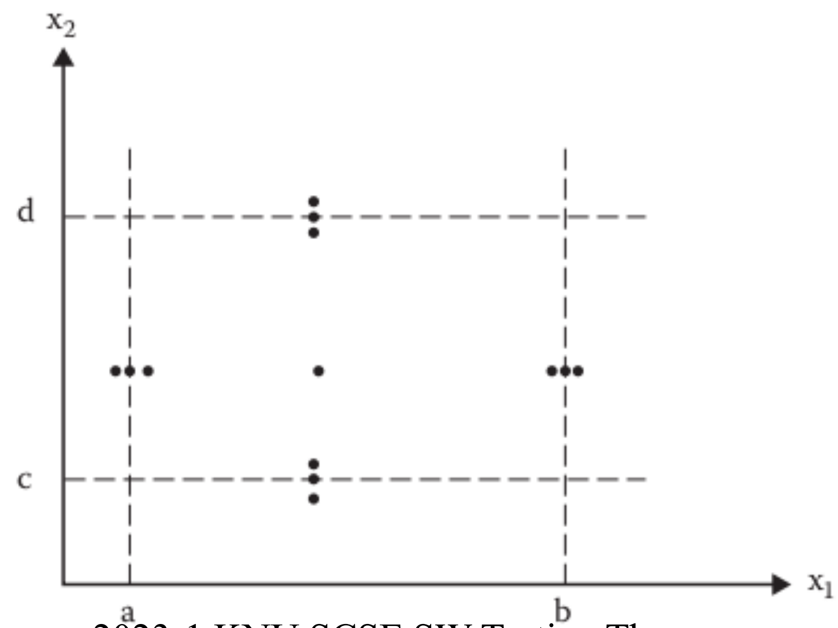
Normal Boundary Value Testing

- Testing focus on the boundary of the input space ot identify test cases
  - Usually errors occur near the boundary values
  - $< X_{min}, X_{min+1}, X_{nom}, X_{max-1}, X_{max} >$
- Input variables are considered as independent
  - Independent, bounded physical quantities
- example : x1[a, b], x2[c, d]



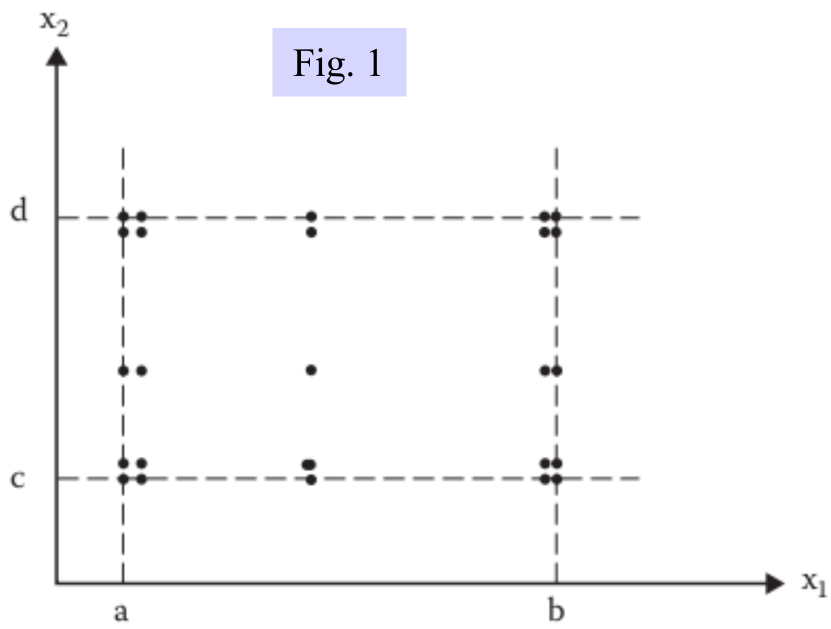
Robustness Testing

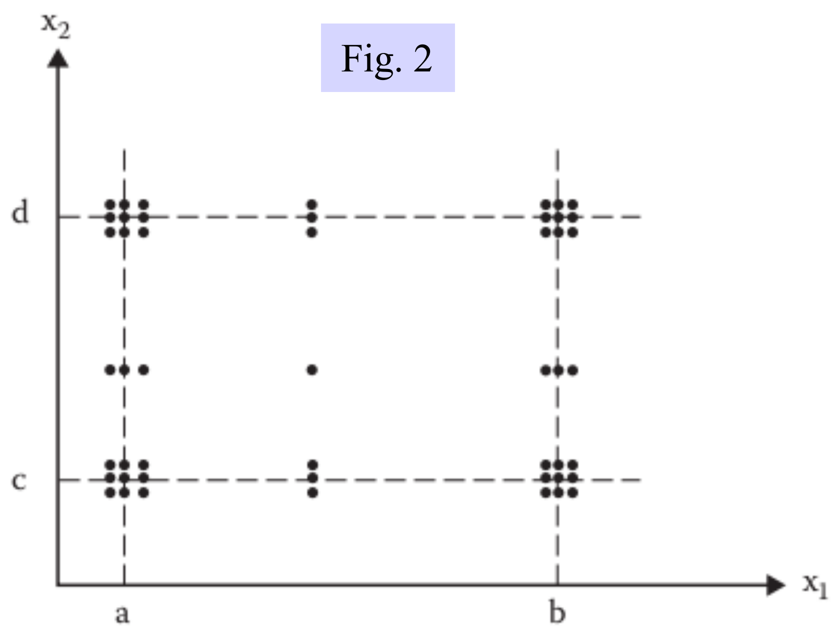
- Extension of Normal Boundary Value Testing
  - **Focus on exceptional values**
  - < X_{min-1}, X_{min}, X_{min+1}, X_{nom}, X_{max-1}, X_{max}, X_{max+1} >
- Example
  - Exceeding the maximum weighting of an elevator



Worst-Case Testing

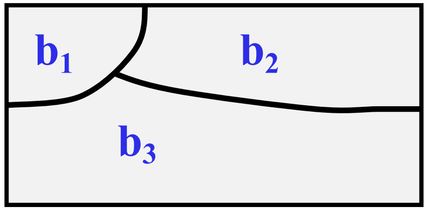
- Considering dependency of input variables
  - Worst-case test cases (Fig. 1)
  - Robust worst-case test cases (Fig. 2)





Equivalence Class Testing

- Motivation
  - A sense of complete testing
  - Hope to avoid redundancy



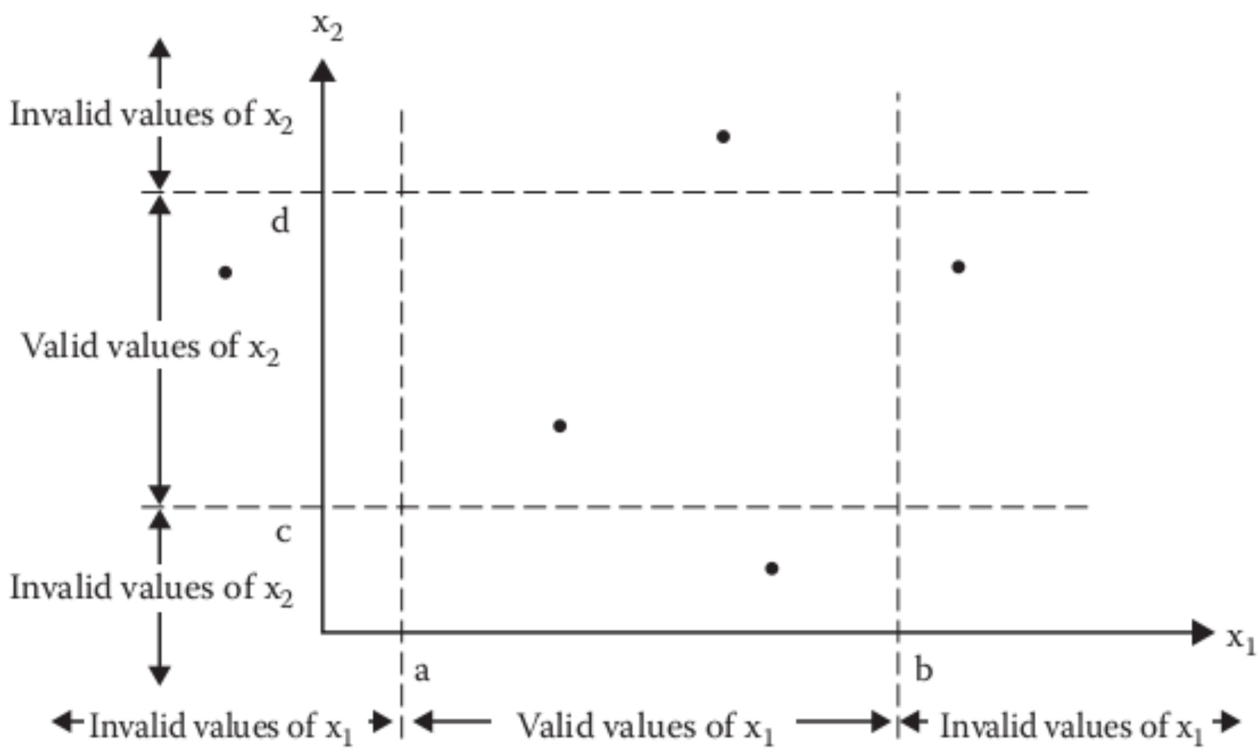
$$\bigcup_{b \in Bq} b = D$$

$$b_i \cap b_j = \Phi,\ \forall\, i \neq j,\ b_i,\ b_j \in B_q$$

- Equivalence Class
  - Modulus Operation(% 5) : Natural Number ➔ 0 ~ 4
    - [0] = {0, 5, 10, …}, [1] = {1, 6, …}, [2], [3], [4] = {4, 9, …}
  - Test data are selected one or two representative numbers in each equivalence class
- Testing Technique
  - Traditional Equivalence
  - Weak/Strong Normal Equivalence
  - Weak/Strong Robust Equivalence

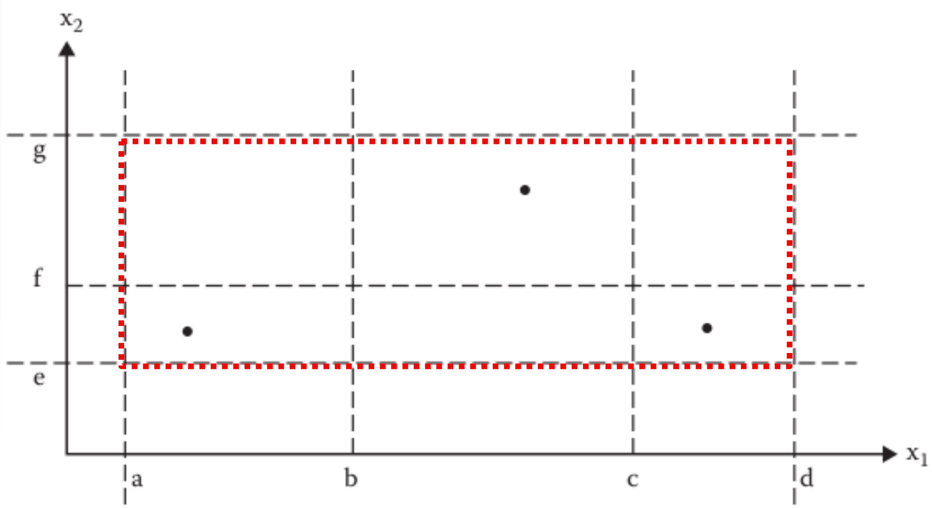
Traditional Equivalence Class Testing

- Equivalence Classes valid/invalid values
- Valid scope x1 = [a, b], x2 = [c, d]

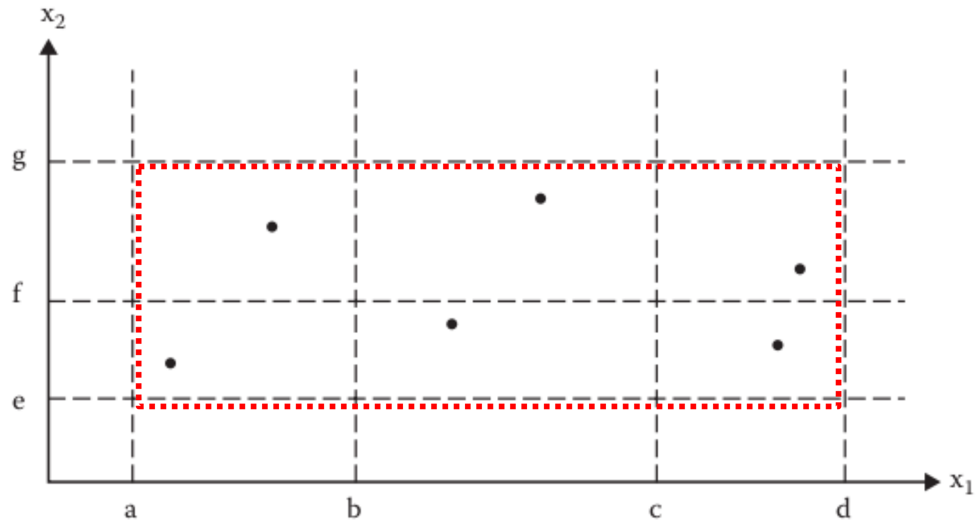


Weak/Strong Normal Equivalence Class

- Focus on normal cases
- Independent inputs(**weak**)/dependent inputs(**strong**)
- Valid scope x1 = {[a, b), [b, c), [c, d]}, x2 = {[e, f), [f, d]}



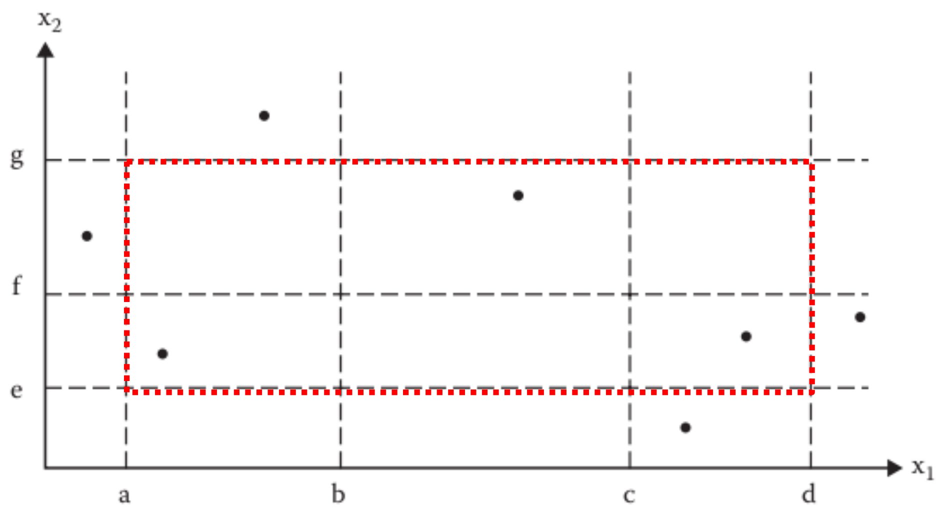
<Weak Normal Equivalence Class>



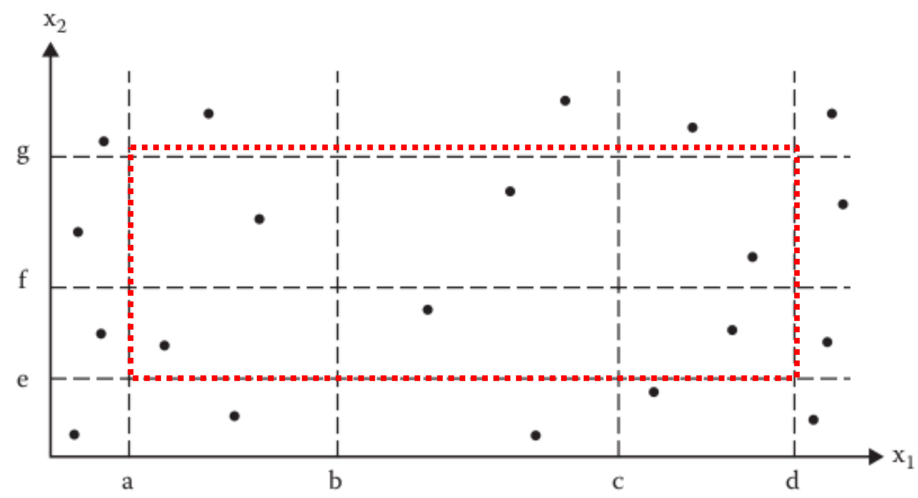
<Strong Normal Equivalence Class>

Weak/Strong Robust Equivalence Class

- Consider both valid and invalid inputs
- Independent inputs(weak)/dependent inputs(strong)
- Valid scope x1 = {[a, b), [b, c), [c, d]}, x2 = {[e, f), [f, d]}



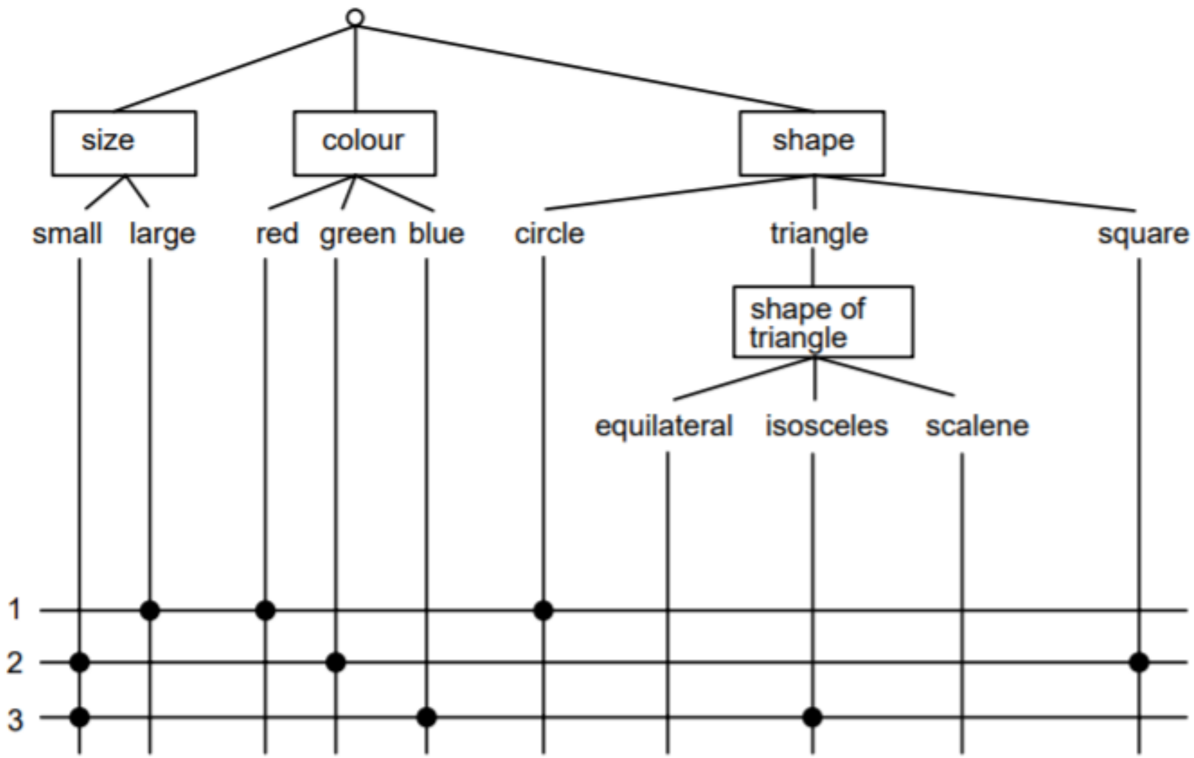
<Weak Robust Equivalence Class>



<Strong Robust Equivalence Class>

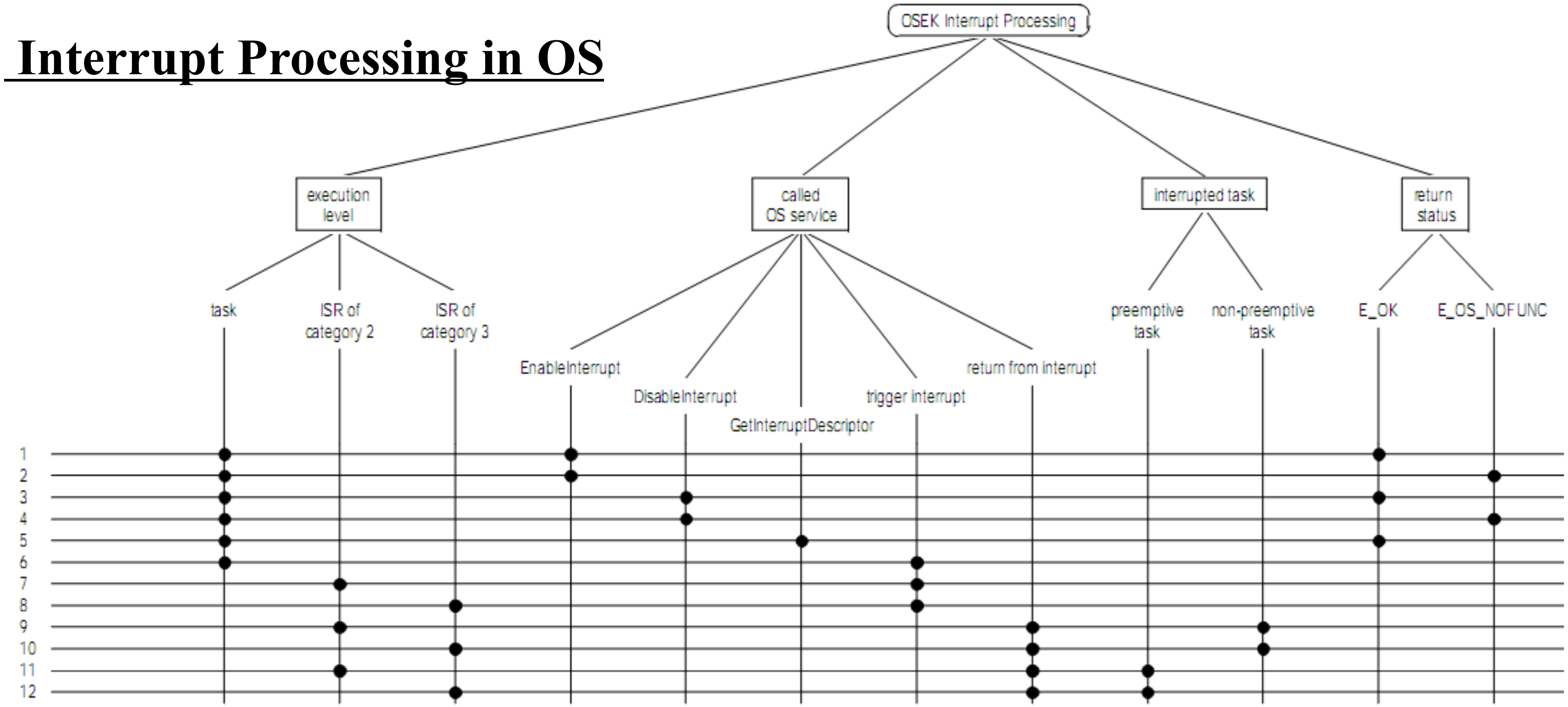
Classification Tree Method

- Selecting test objects or features (size, colour, shape, etc)
- Designing a classification tree (size : small, large)
- Combining classes to form test cases



Testing Example of CTM

Interrupt Processing in OS

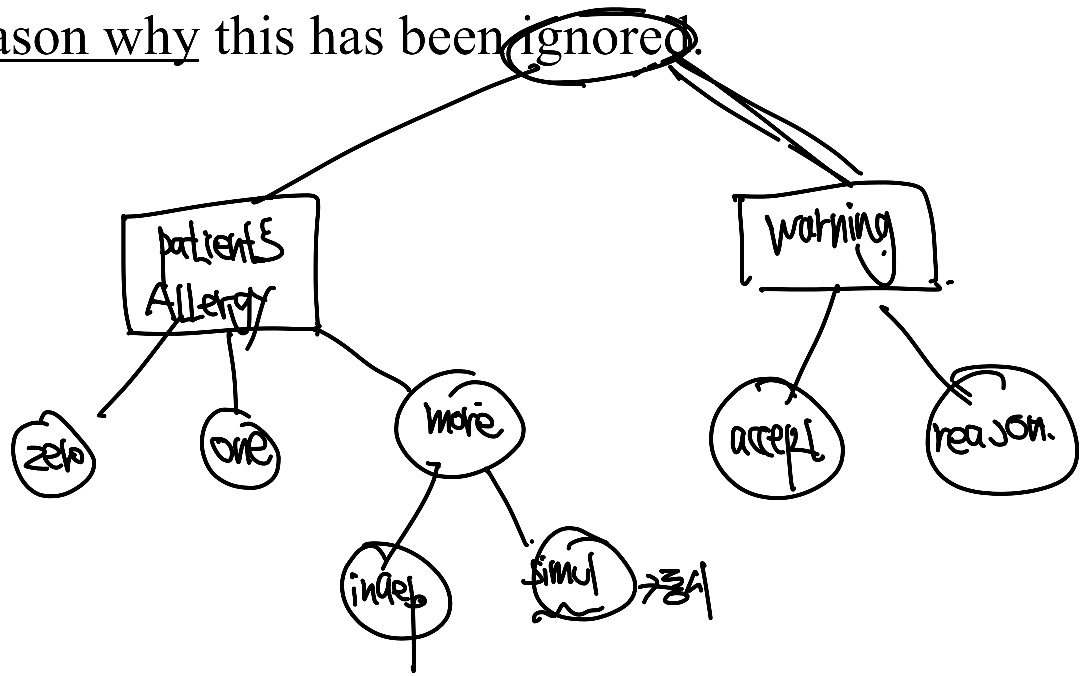


Test case No.	Sched. policy Conf. class Status	Action	Expected Result
1	n, m, f B1, B2, E1, E2 s, e	Call `EnableInterrupt()`. All requested interrupts are disabled	Enable interrupts. Service returns E_OK
2	n, m, f B1, B2, E1, E2 e	Call `EnableInterrupt()`. At least one of the requested interrupts is already enabled	Enable interrupts. Service returns E_OS_NOFUNC

Exercise : Mental Heath Care

여러 개에 대한 알러지 일 수도 있다.

- Patient Management System requirements:
  - If a patient is known to be allergic to any ~~particular medication~~, then prescription of that medication shall result in a warning message being issued to the system user.
  - If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.



Classification Tree of Mental Health Care

Testing Example : Mental Heath Care

- **Test Case #1**
  - Set up a patient record with no known allergies.
  - Prescribe medication for allergies that are known to exist.
  - Check that a warning message is not issued by the system.
- **Test Case #2**
  - Set up a patient record with a known allergy.
  - Prescribe the medication to that the patient is allergic to
  - Check that the warning is issued by the system.
- **Test Case #3**
  - Set up a patient record in which allergies to two or more drugs are recorded.
  - Prescribe both of these drugs separately
  - Check that the correct warning for each drug is issued.
- **Test Case #4**
  - Prescribe two drugs that the patient is allergic to.
  - Check that two warnings are correctly issued.
- **Test Case #5**
  - Prescribe a drug that issues a warning and overrule that warning.
  - Check that the system requires users the reason of overruling warning

Decision Table-Based Testing (1/2)

- Decision table
  - Used to test complicated logical relations from 1960s
  - Execution conditions of each action(a1, a2, …) are marked in the below
    - action a1 is related with !c1, c1 && !c2, c1 && c2 && !c3

c1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T
c2: b < a + c?	—	F	T	T	T	T	T	T	T	T	T
c3: c < a + b?	—	—	F	T	T	T	T	T	T	T	T
c4: a = b?	—	—	—	T	T	T	T	F	F	F	F
c5: a = c?	—	—	—	T	T	F	F	T	T	F	F
c6: b = c?	—	—	—	T	F	T	F	T	F	T	F
a1: Not a triangle	X	X	X								
a2: Scalene											X
a3: Isosceles							X		X	X	
a4: Equilateral				X							
a5: Impossible					X	X		X			

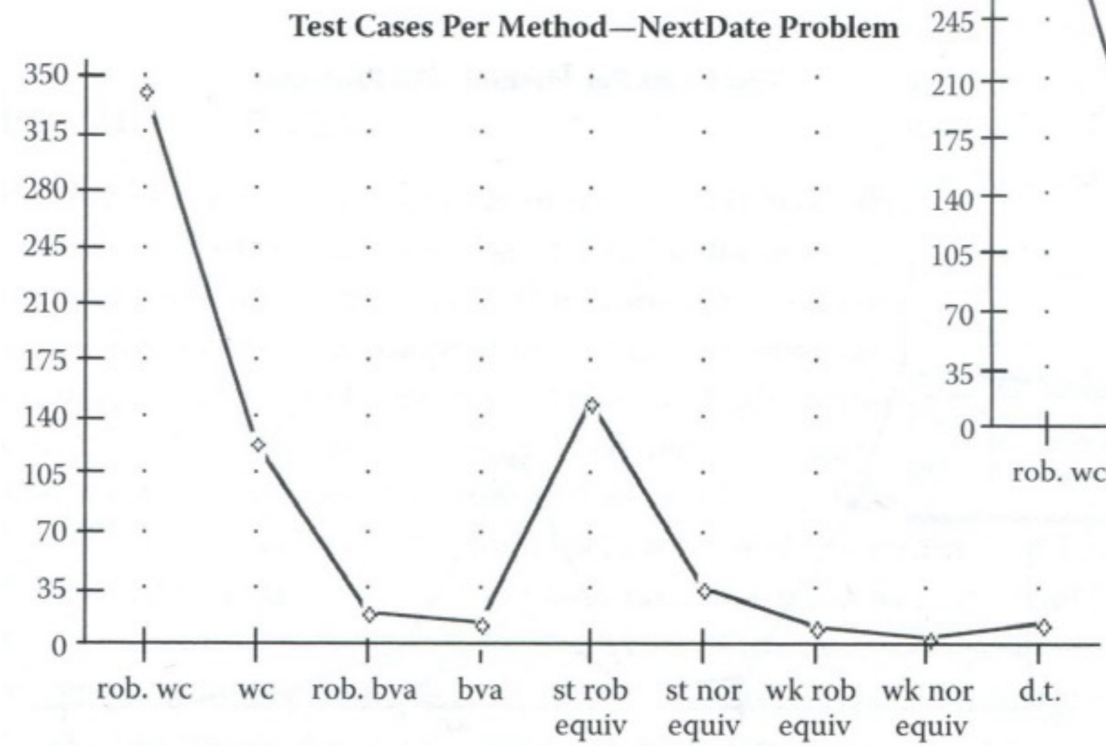
Decision Table-Based Testing (2/2)

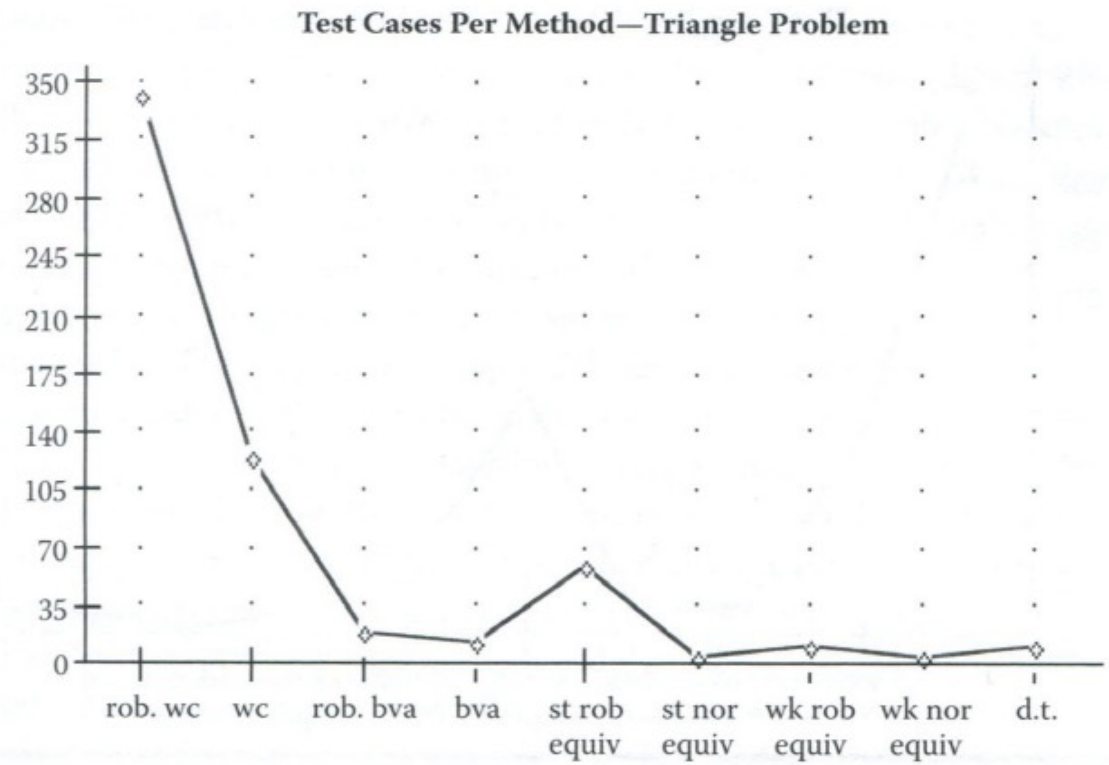
- Test cases are generated from Decision Table
  - A test case is generated from each column of decision table

c1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T
c2: b < a + c?	—	F	T	T	T	T	T	T	T	T	T
c3: c < a + b?	—	—	F	T	T	T	T	T	T	T	T
c4: a = b?	—	—	—	T	T	T	T	F	F	F	F
c5: a = c?	—	—	—	T	T	F	F	T	T	F	F
c6: b = c?	—	—	—	T	F	T	F	T	F	T	F
a1: Not a triangle	X	X	X								
a2: Scalene											X
a3: Isosceles							X		X	X	
a4: Equilateral				X							
a5: Impossible					X	X		X			
	DT01										DT11

Case ID	a	b	c	*Expected Output*
DT1	4	1	2	Not a triangle
DT2	1	4	2	Not a triangle
DT3	1	2	4	Not a triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

Testing Effort of Spec-based Testing





- wc : worst-case
- bva : boundary value
- d.t. : decision table

Test Case Design Technique

PART II

Contents

Part I

- Terminologies
- Specification-based Testing (Black-box Testing)

Part II (White-box Testing)

- Coverage-based Testing
- Structural Path-based Testing
- Dataflow Testing

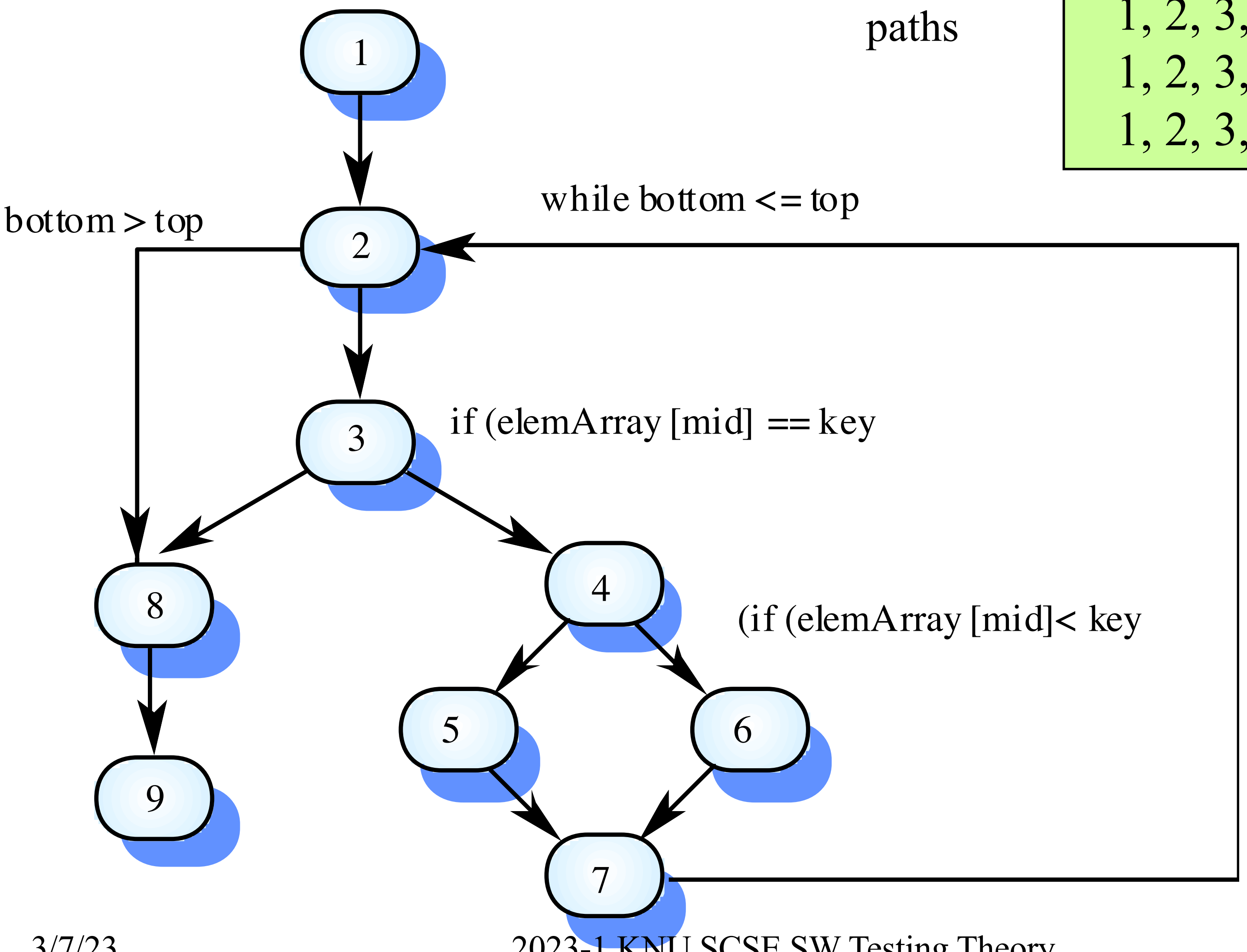
White-box testing

- Sometimes called **structural testing**
- Derivation of test cases according to program structure.
- Knowledge of the program is used **to identify additional test cases**
- Objective is to exercise all program statements, all conditions, all decisions (not all path combinations)

Program Flow Graph

Independent paths

1, 2, 8, 9
1, 2, 3, 8, 9
1, 2, 3, 4, 5, 7, 2, 8, 9
1, 2, 3, 4, 6, 7, 2, 8, 9



Test Coverage I

- Statement Coverage
  - Every statement in the program has been executed at least once
- Decision(Branch) Coverage
  - Every point of entry and exit in the program has been invoked at least once
  - **Every decision in the program has taken all possible outcomes at least once**
- Condition/Decision Coverage(C/DC)
  - Every point of entry and exit in the program has been invoked at least once
  - Every decision in the program has taken all possible outcomes at least once
  - **Every condition in a decision in the program has taken all possible outcomes at least once**

Ref : John Joseph Chilenski and Steven P. Miller, "Applicability of modified condition/decision coverage to software testing," Software Engineering Journal, Sep. 1994

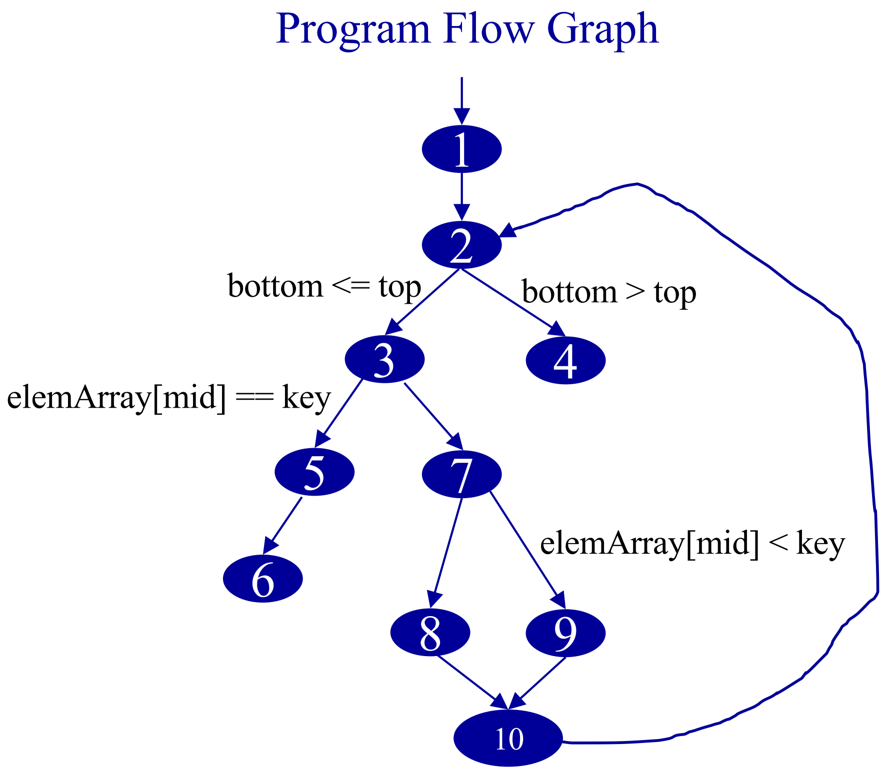
Example of Test Coverage I

- Decision Coverage of OR(||)
  - Minimal Test Set : (TT, FF), (TF, FF), (FT,FF)
- Condition Coverage of OR(||)
  - Minimal Test Set : (TT, FF), (TF, FT)
- C/DC Coverage of OR(||)
  - Minimal Test Set (TT, FF)

A	B	A||B
T	T	T
T	F	T
F	T	T
F	F	F

Exercise : Binary Search

```
public static void search ( int key, int [] elemArray, Result r )
{
    int bottom = 0 ;
    int top = elemArray.length - 1 ;
    int mid ;
    r.found = false ; r.index = -1 ;
    while ( bottom <= top )
    {
        mid = (top + bottom) / 2 ;
        if (elemArray [mid] == key)
        {
            r.index = mid ;
            r.found = true ;
            return ;
        } // if part
        else
        {
            if (elemArray [mid] < key)
                bottom = mid + 1 ;
            else
                top = mid - 1 ;
        }
    } //while loop
} // search
```



Example of Test Coverage

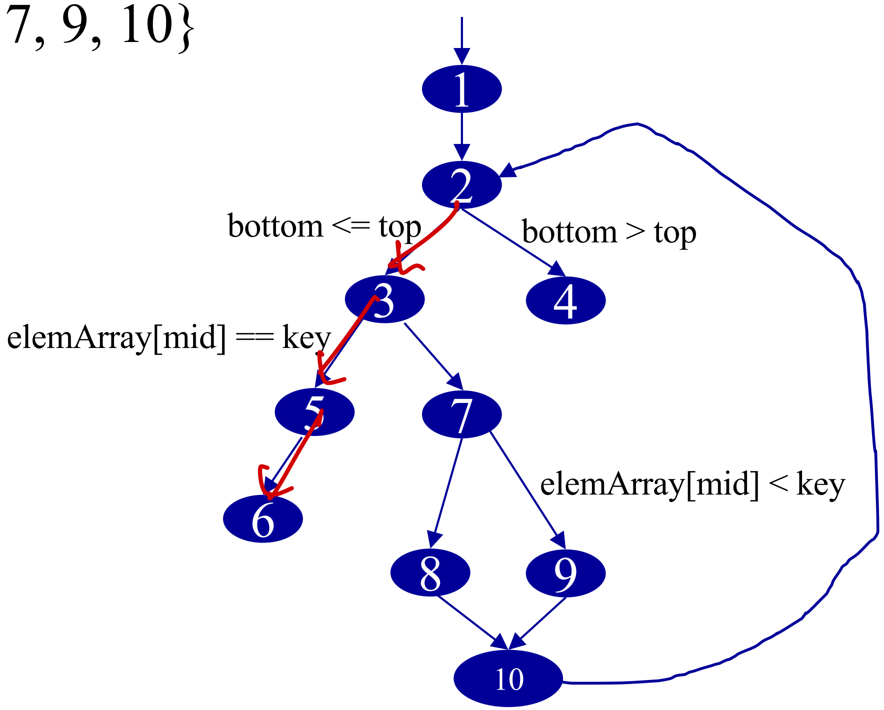
- Statement Coverage

4.

3 5-

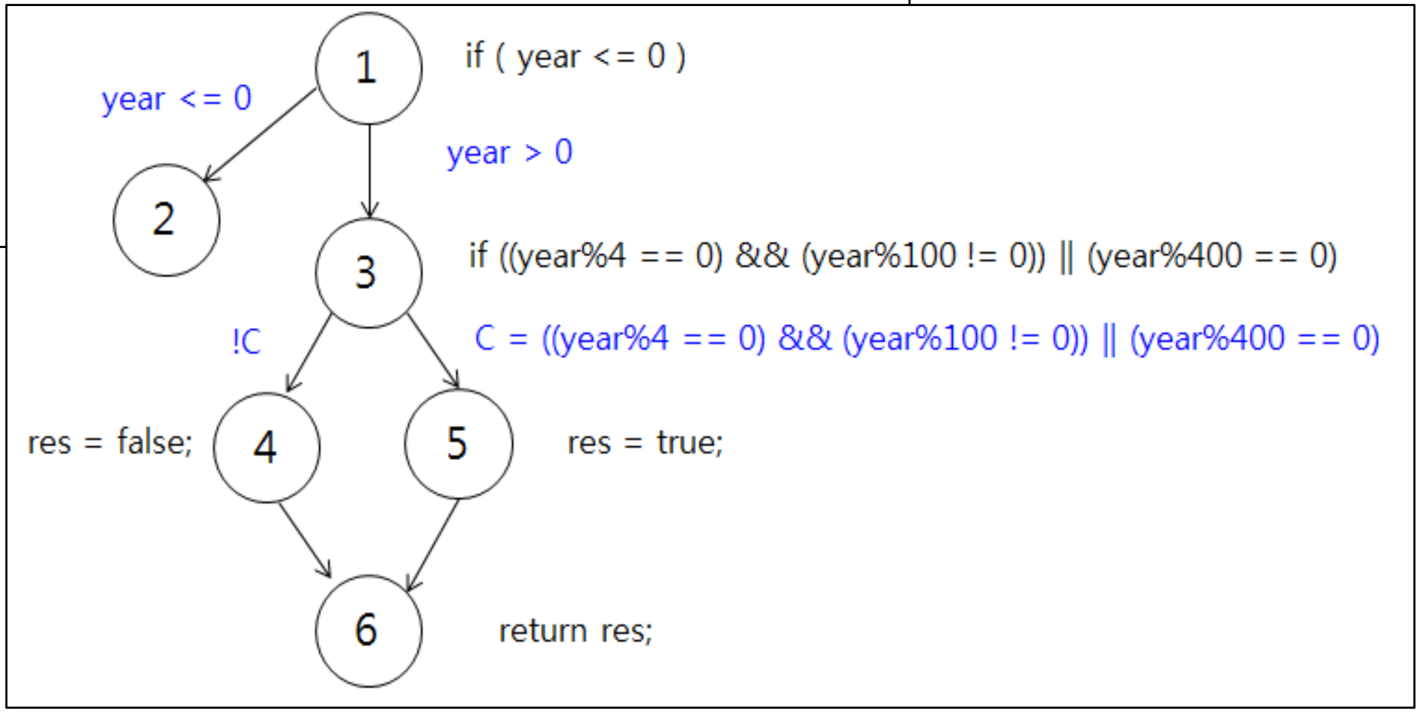
  - key : 7, elemArray[]= { 1, 3, 5, 7, 9, 10}
    - Path : (1, 2, 3, 7, 9, 10, 2, 3, 7, 8, 10, 2, 3, 5, 6)
- Decision Coverage
  - key : 7, elemArray[]= { 1, 3, 5, 7, 9, 10}
  - key : 3, elemArray[]= { }
    - Path : (1, 2, 4)

Program Flow Graph



Example2: Leap Year Function

```
bool isLeap(int year)
{
    bool res;
    if ( year <= 0 )
        return false;

    if ( ((year%4 == 0) && (year%100 != 0)) || (year%400 == 0 ) )
        res = true;
    else
        res = false;

    return res;
}
```



Test Coverage of Leap Year

- **Decision Coverage**
  - C1 : (year %4 == 0), C2 = (year%100 != 0), C3 = (year%400 == 0)

Test Input	Test Path	year<=0	(C1&&C2) || C3
year = -1	[1,2]	T	-
year = 4	[1, 3, 4, 6]	F	T
year = 5	[1, 3, 5, 6]	F	F

- **C/DC Coverage**
  - C1 : (year %4 == 0), C2 = (year%100 != 0), C3 = (year%400 == 0)

Test Input	Test Path	year<=0	C1	C2	C3	Result
year = -1	[1,2]	T	-	-	-	-
year = 4	[1, 3, 4, 6]	F	T	T	F	T
year = 5	[1, 3, 5, 6]	F	F	T	F	F
year = 400	[1, 3, 4, 6]	F	T	F	T	T

Test Coverage II

- Modified Condition/Decision Coverage (MC/DC)
  - Every point of entry and exit in the program has been invoked at least once
  - Every condition in a decision in the program has taken all possible outcomes at least once
  - **Each condition has been shown to independently affect the decision output.**
  - **A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions**

MC/DC's Example 1

- A and B
  - (T,T) can be paired with (F,T) to show the independence of A
  - (T,T) can be paired with (T,F) to show the independence of B
  - Minimal Test Set : {1,2,3} = {(T,T), (T,F), (F,T)}

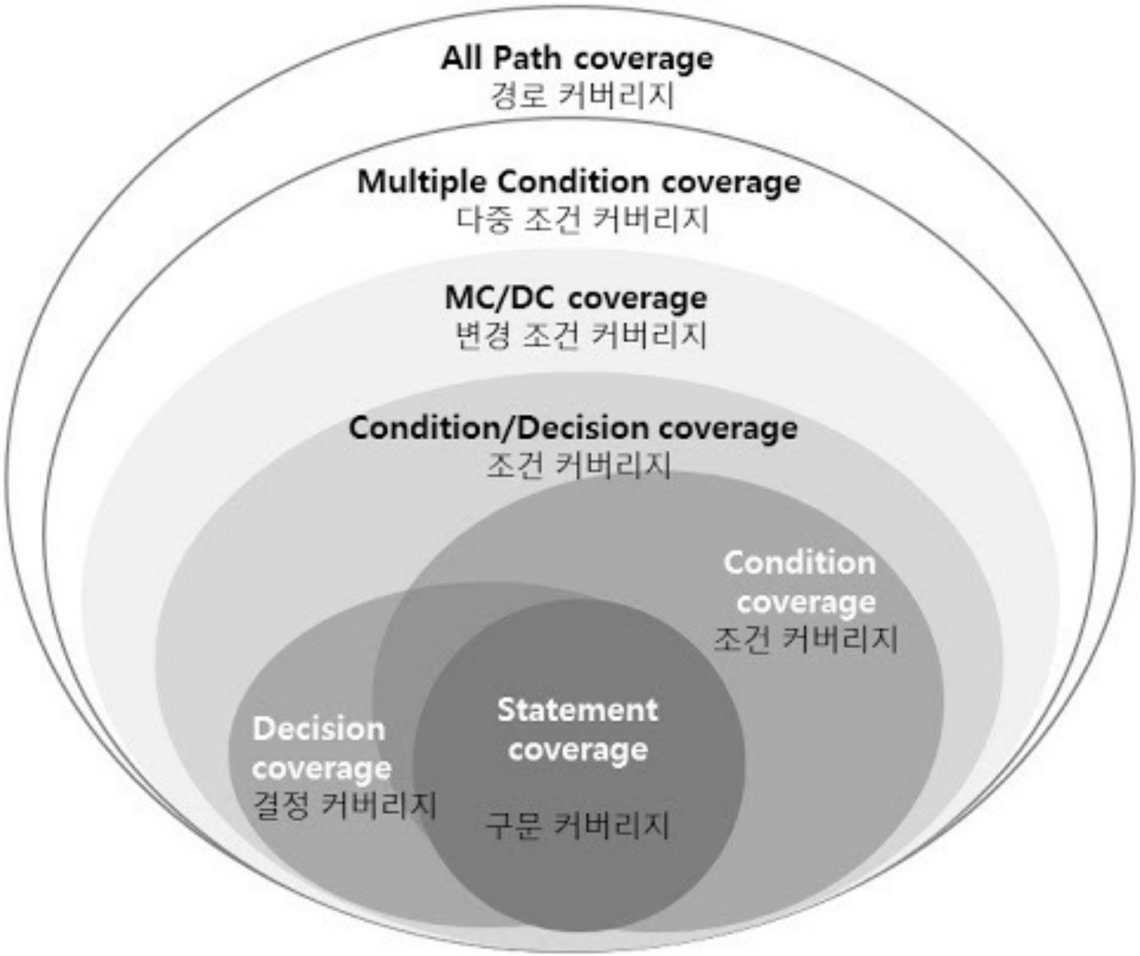
Index	A	B	A && B	A	B
1	**T**	**T**	**T**	3	2
2	T	F	F		1
3	F	T	F	1	
4	F	F	F		

MC/DC's Example 2

- A or B
  - {2, 3, 4} = {(T,F), (F,T), (F,F)}

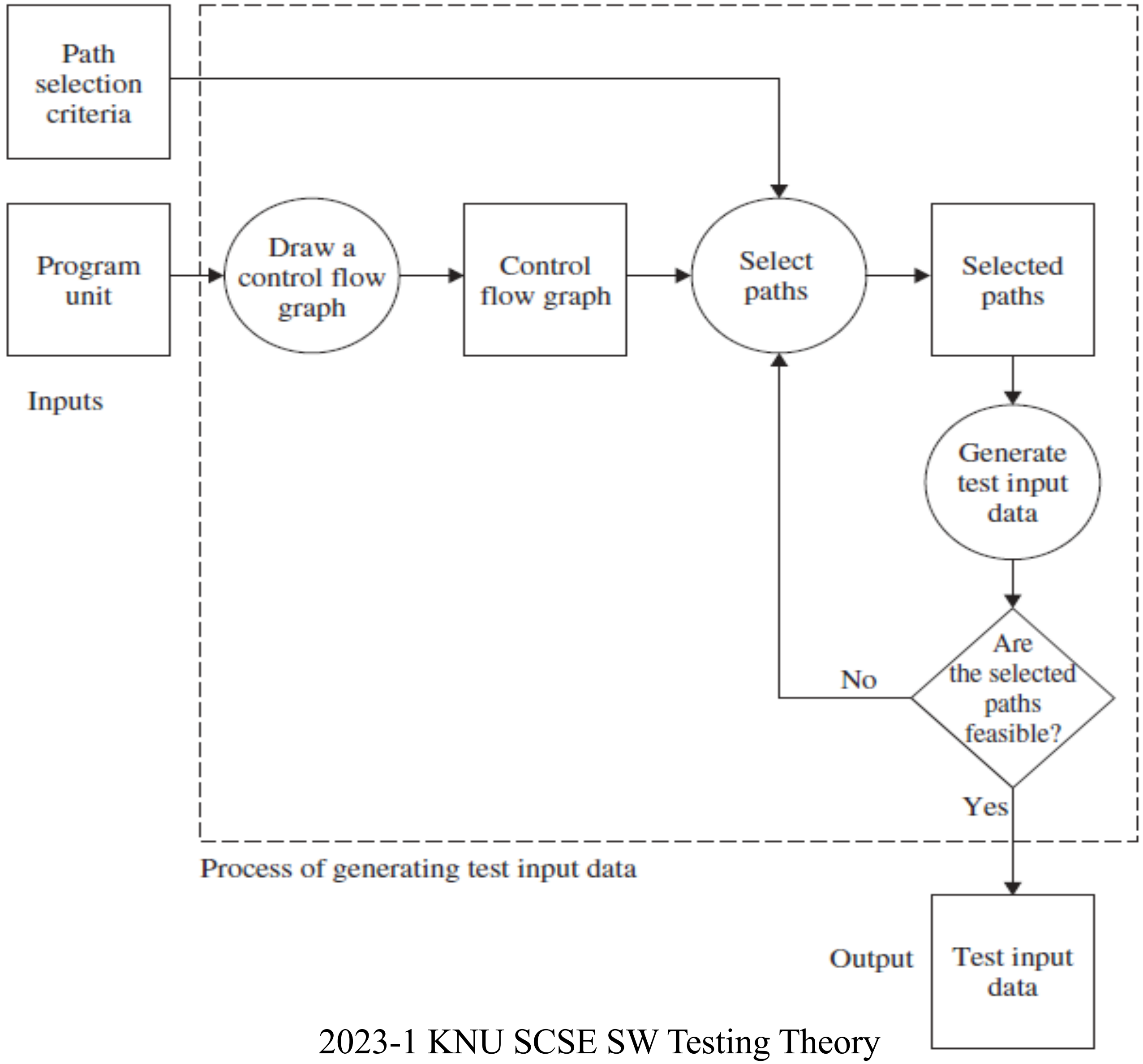
Index	A	B	A || B	A	B
1	T	T	T		
2	T	F	T	4	
3	F	T	T		4
4	**F**	**F**	**F**	2	3

Coverage-based Testing



Structural Path-based Testing

Overview of Path-based Testing

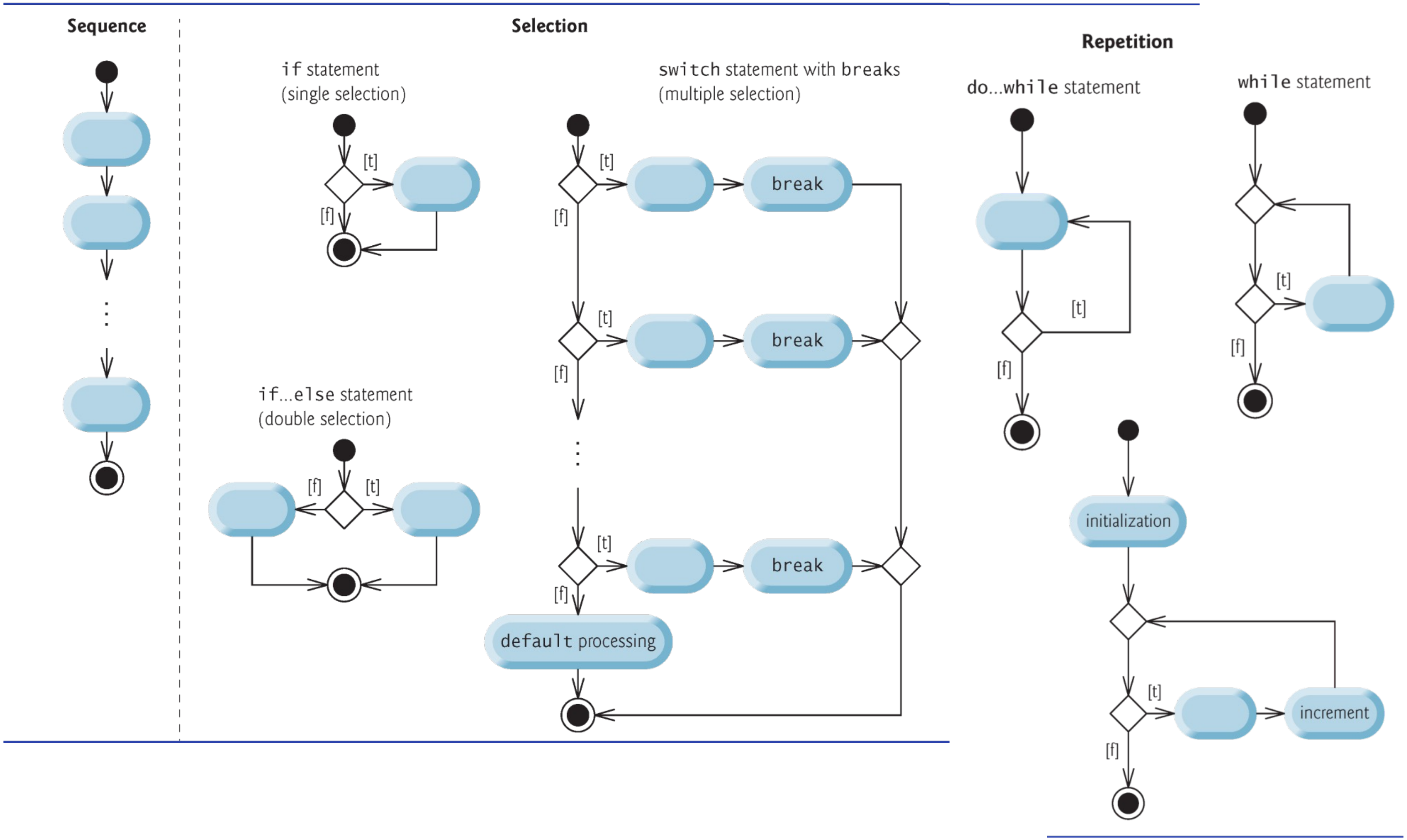


Structured Programming

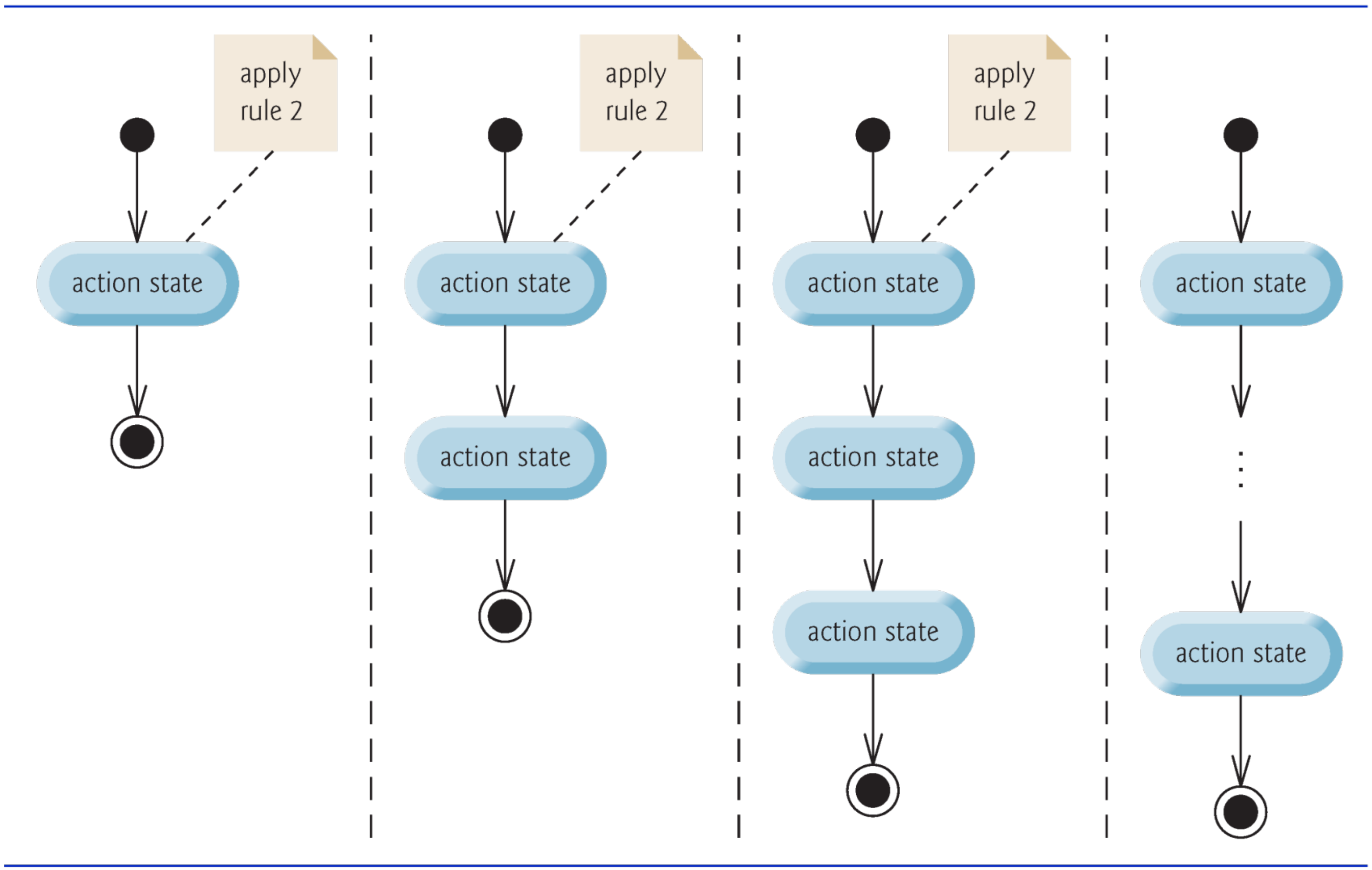
- Structured programming technique is simple
- Bohm and Jacopini provided only three basic structures for creating software program
  - Sequence
  - Selection
  - Repetition
- Rules for structured programs

Rules for forming structured programs	
1.	Begin with the simplest activity diagram (Fig. 5.22).
2.	Any action state can be replaced by two action states in sequence. (This is the stacking rule.)
3.	Any action state can be replaced by any control statement (sequence of action states, `if`, `if...else`, `switch`, `while`, `do...while` or `for`). (This is the nesting rule.)
4.	Rules 2 and 3 can be applied as often as you like and in any order.

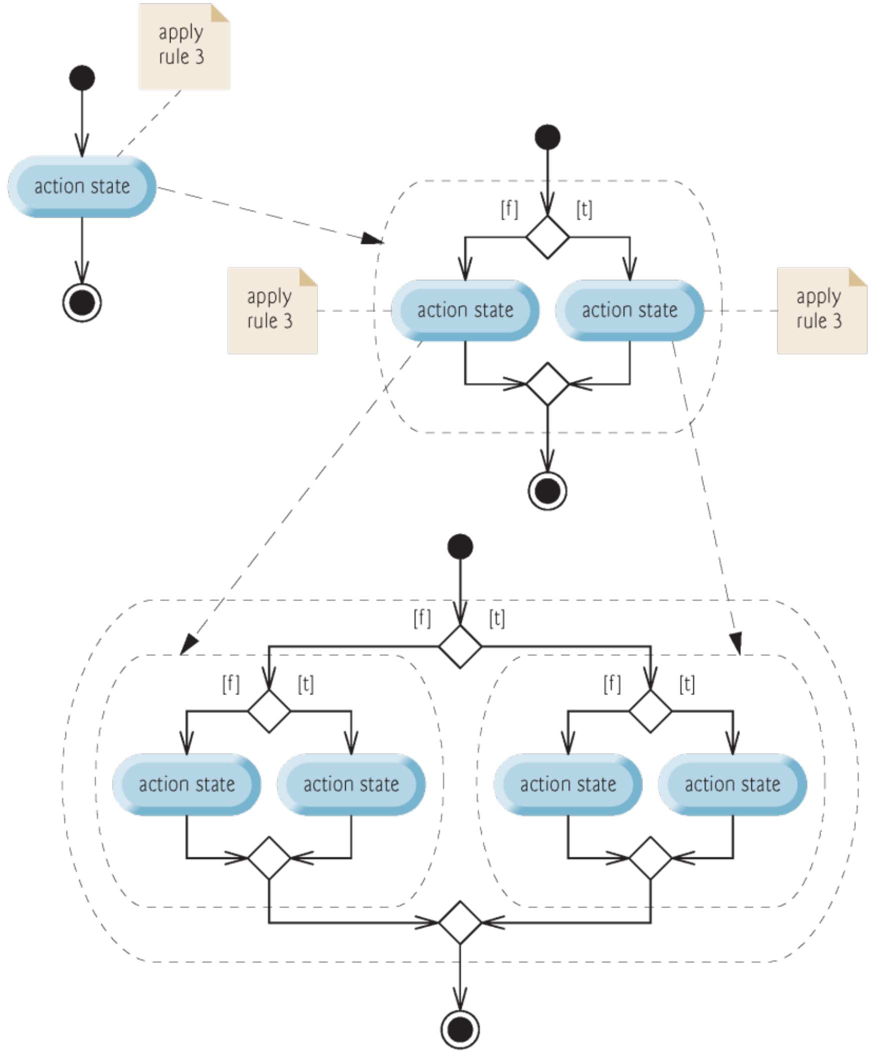
Three basic structures



Example of applying rules (1/2)



Example of applying rules (2/2)



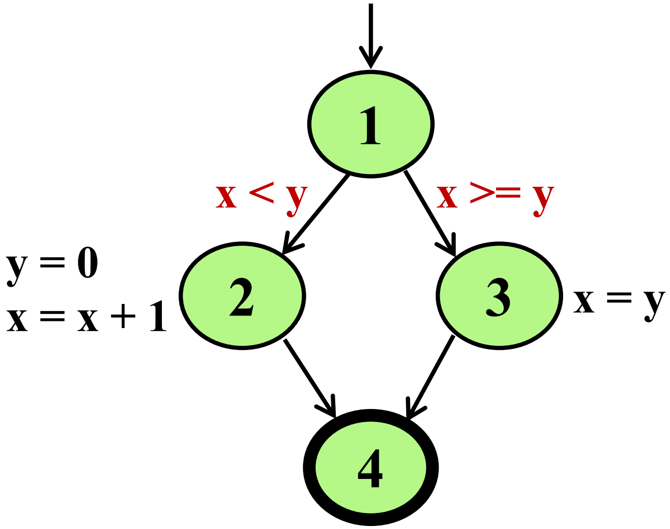
Control Flow Graphs

- A CFG models all executions of a method by describing control structures
- Nodes : Statements or sequences of statements (basic blocks)
- Edges : Transfers of control
- Basic Block : A sequence of statements such that if the first statement is executed, all statements will be (no branches)

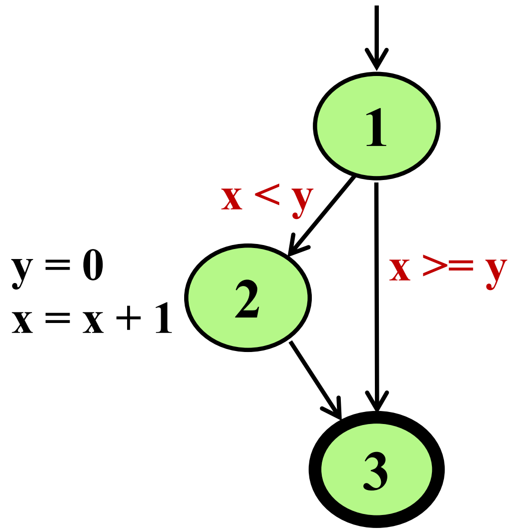
- CFGs are sometimes annotated with extra information
  - Branch predicates
  - Defs/Uses

CFG : The if Statement

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
```

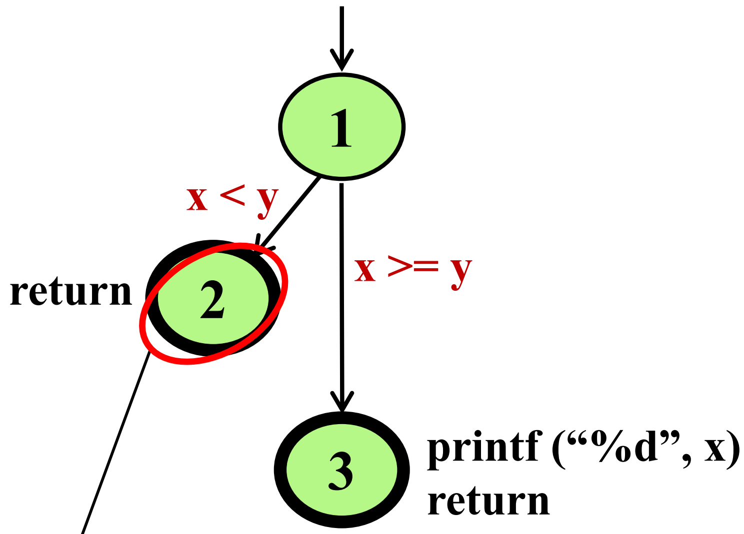



```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```



CFG : The if-return Statement

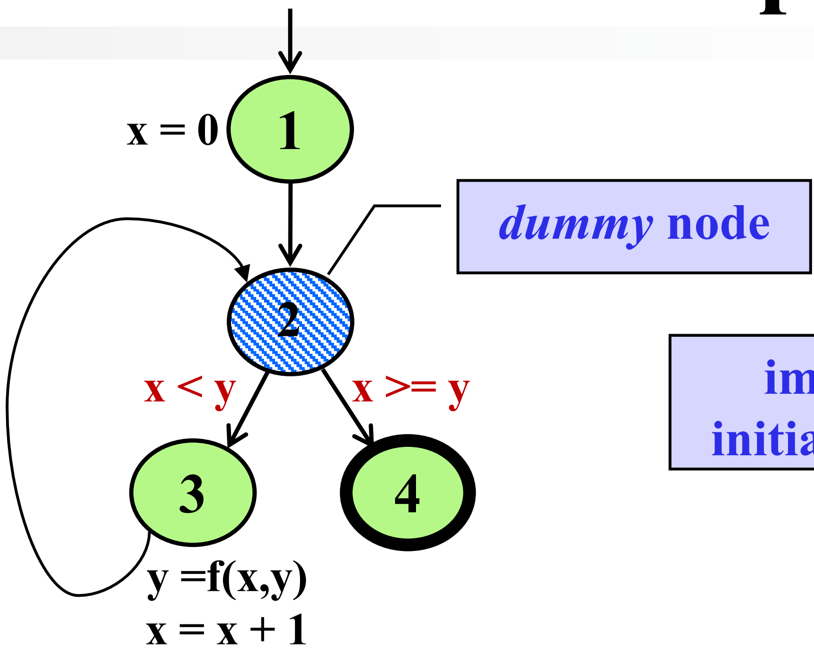
```
if (x < y)
{
   return;
}
printf("%d", x);
return;
```



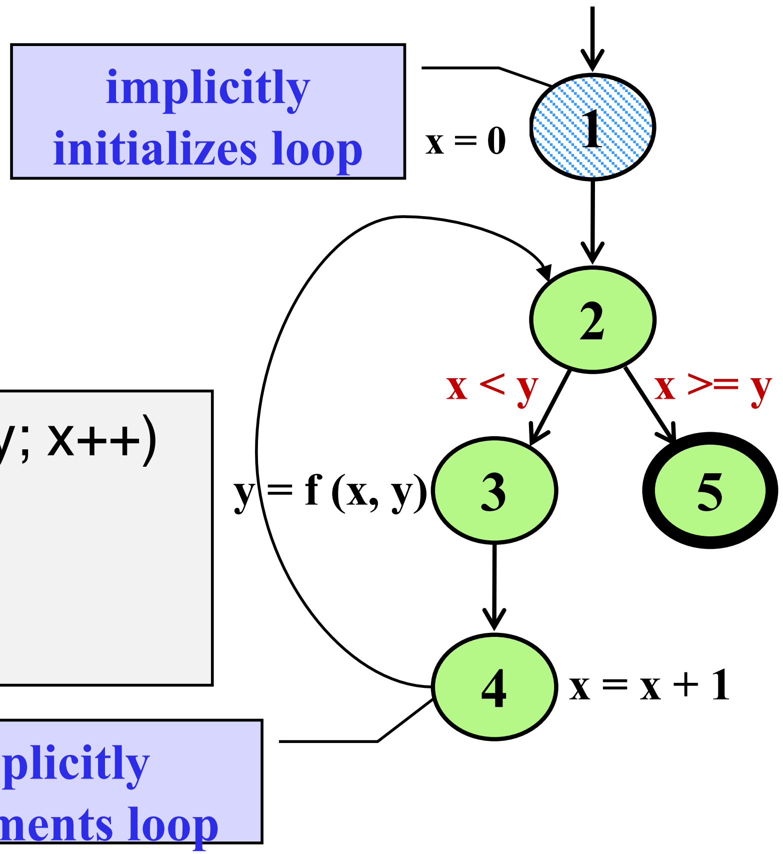
No edge from node 2 to 3.
The return nodes must be distinct.

CFG : while and for Loops

```
x = 0;
while (x < y)
{
   y = f (x, y);
   x = x + 1;
}
```

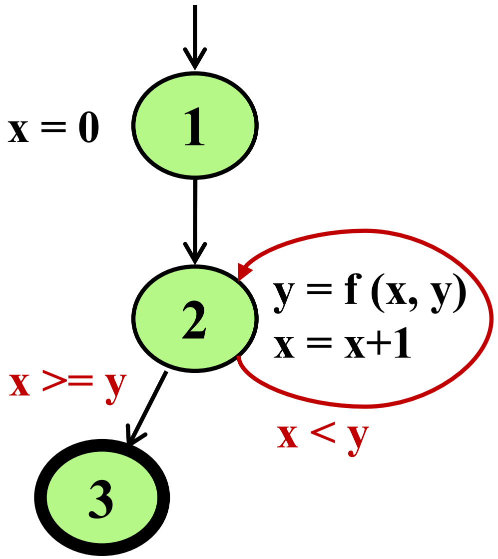



```
for (x = 0; x < y; x++)
{
   y = f (x, y);
}
```

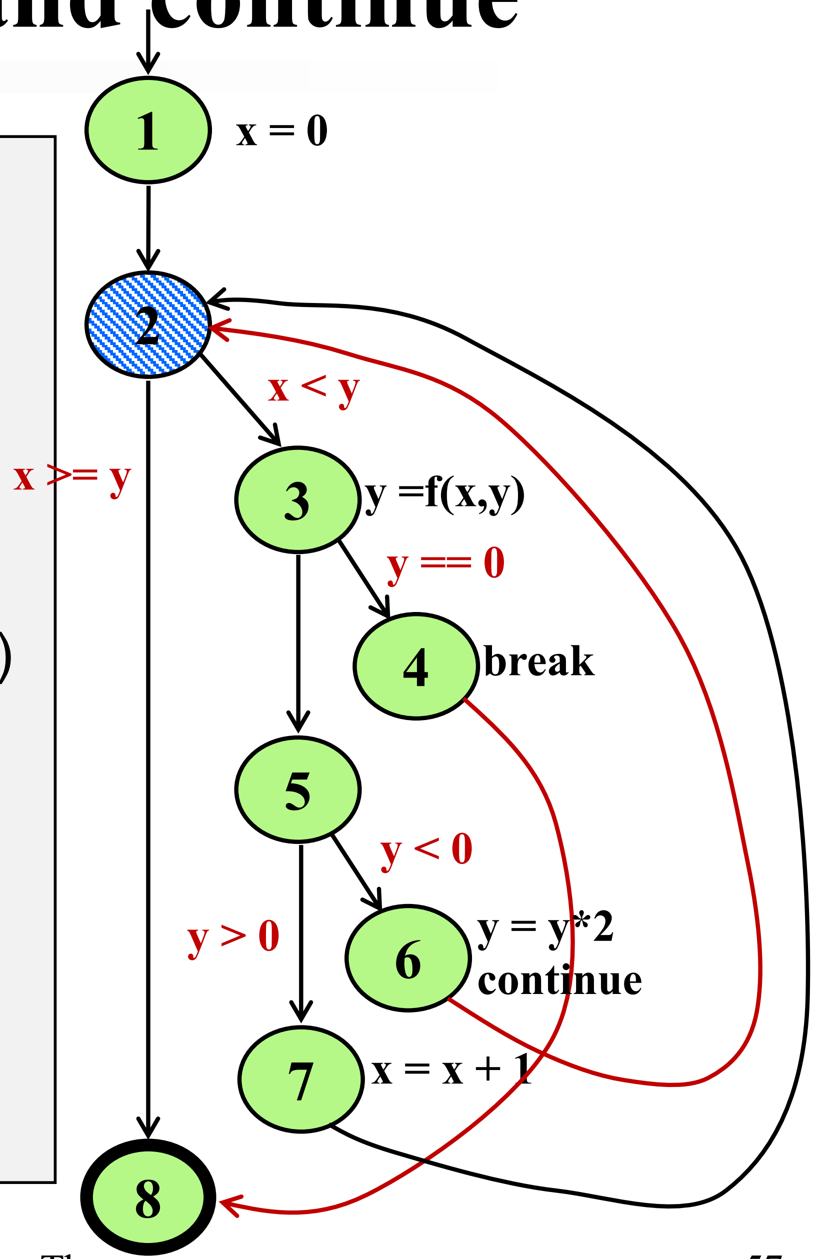


CFG : do Loop, break and continue

```
x = 0;
do {
   y = f (x, y);
   x = x + 1;
} while (x < y);
printf("%d",y);
```

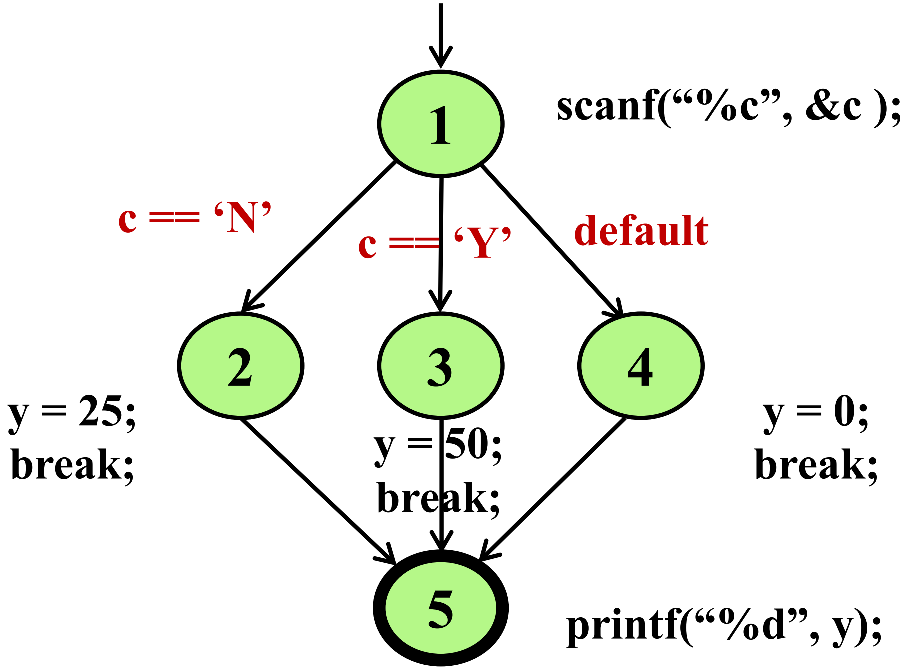



```
x = 0;
while (x < y)
{
   y = f (x, y);
   if (y == 0)
   {
      break;
   } else if (y < 0)
   {
      y = y*2;
      continue;
   }
   x = x + 1;
}
printf("%d",y);
```



CFG : The case (switch) Structure

```
scanf("%c",&c) ;
switch ( c )
{
   case 'N':
      y = 25;
      break;
   case 'Y':
      y = 50;
      break;
   default:
      y = 0;
      break;
}
printf("%d", y);
```

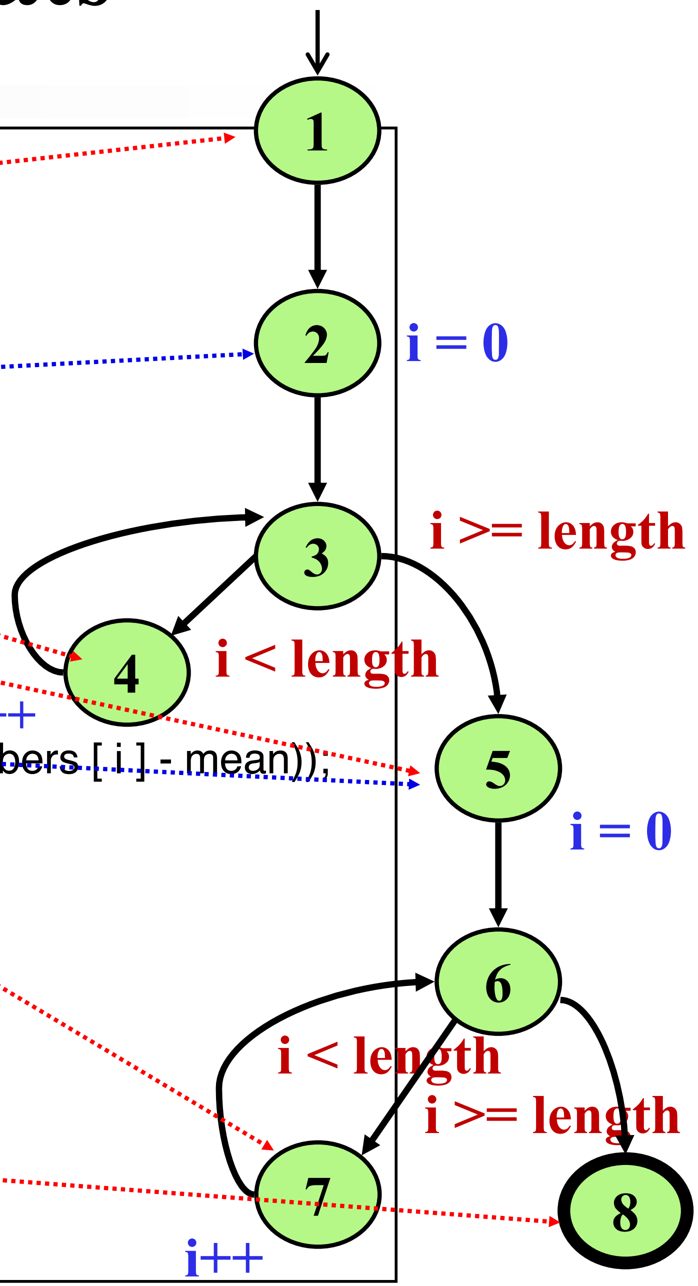


Example Control Flow – Stats

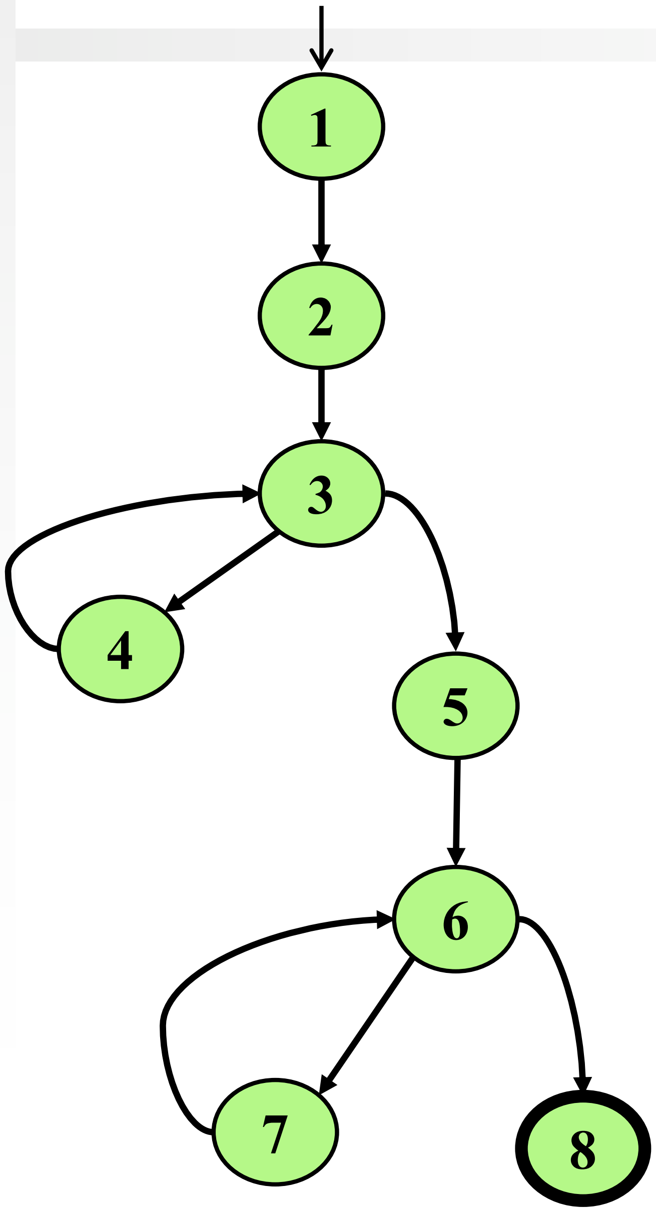
```
void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0.0;
    for (int i = 0; i < length; i++)  {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2];
    mean = sum / (double) length;

    varsum = 0.0;
    for (int i = 0; i < length; i++)  {
        varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = sqrt ( var );

    printf ("length:              %d\n", length);
    printf ("mean:                %lf\n", mean);
    printf ("median:              %lf\n", med);
    printf (“variance:            %lf\n", var);
    printf ("standard deviation:%lf \n", sd);
}
```

Control Flow Graph for Stats

```
void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0.0;
    for (int i = 0; i < length; i++)    {
        sum += numbers [ i ];
    }
    med    = numbers [ length / 2];
    mean = sum / (double) length;

    varsum = 0.0;
    for (int i = 0; i < length; i++)      {
        varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = sqrt ( var );

    printf ("length:                  %d\n", length);
    printf ("mean:                    %lf\n", mean);
    printf ("median:                  %lf\n", med);
    printf (“variance:                %lf\n", var);
    printf ("standard deviation:%lf\n", sd);
}
```



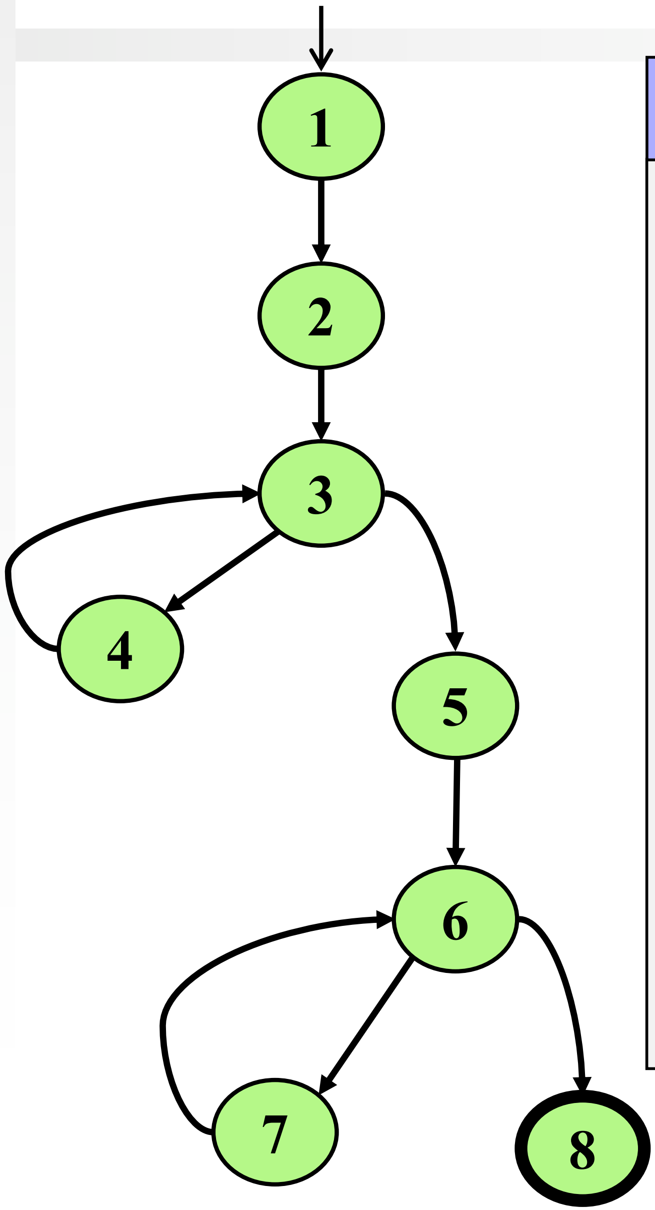
Control Flow TRs and Test Paths – EC



Edge Coverage	
TR	**Test Path**
A. [1, 2] B. [2, 3] C. [3, 4] D. [3, 5] E. [4, 3] F. [5, 6] G. [6, 7] H. [6, 8] I. [7, 6]	[1, 2, 3, 4, 3, 5, 6, 7, 6, 8]

* TR : Test Requirements

Control Flow TRs and Test Paths – EPC



Edge-Pair Coverage

TR	Test Paths
A. [1, 2, 3] B. [2, 3, 4] C. [2, 3, 5] D. [3, 4, 3] E. [3, 5, 6] F. [4, 3, 5] G. [5, 6, 7] H. [5, 6, 8] I. [6, 7, 6] J. [7, 6, 8] K. [4, 3, 4] L. [7, 6, 7]	i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8] ii. [1, 2, 3, 5, 6, 8] iii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]

TP	TRs toured	*sidetrips*
i	A, B, D, E, F, G, I, J	C, H
ii	A, C, E, H	
iii	A, B, D, E, F, G, I, J, K, L	C, H

* sidetrips : some cycles are included in paths

Exercise

```
w=x;
if (m>0){
    w++;
}
else{
    w=2*w;
}

if (y <=10){
   x=5*y;
}
else{
   x = 3*y+5;
}
z = w+x;
```

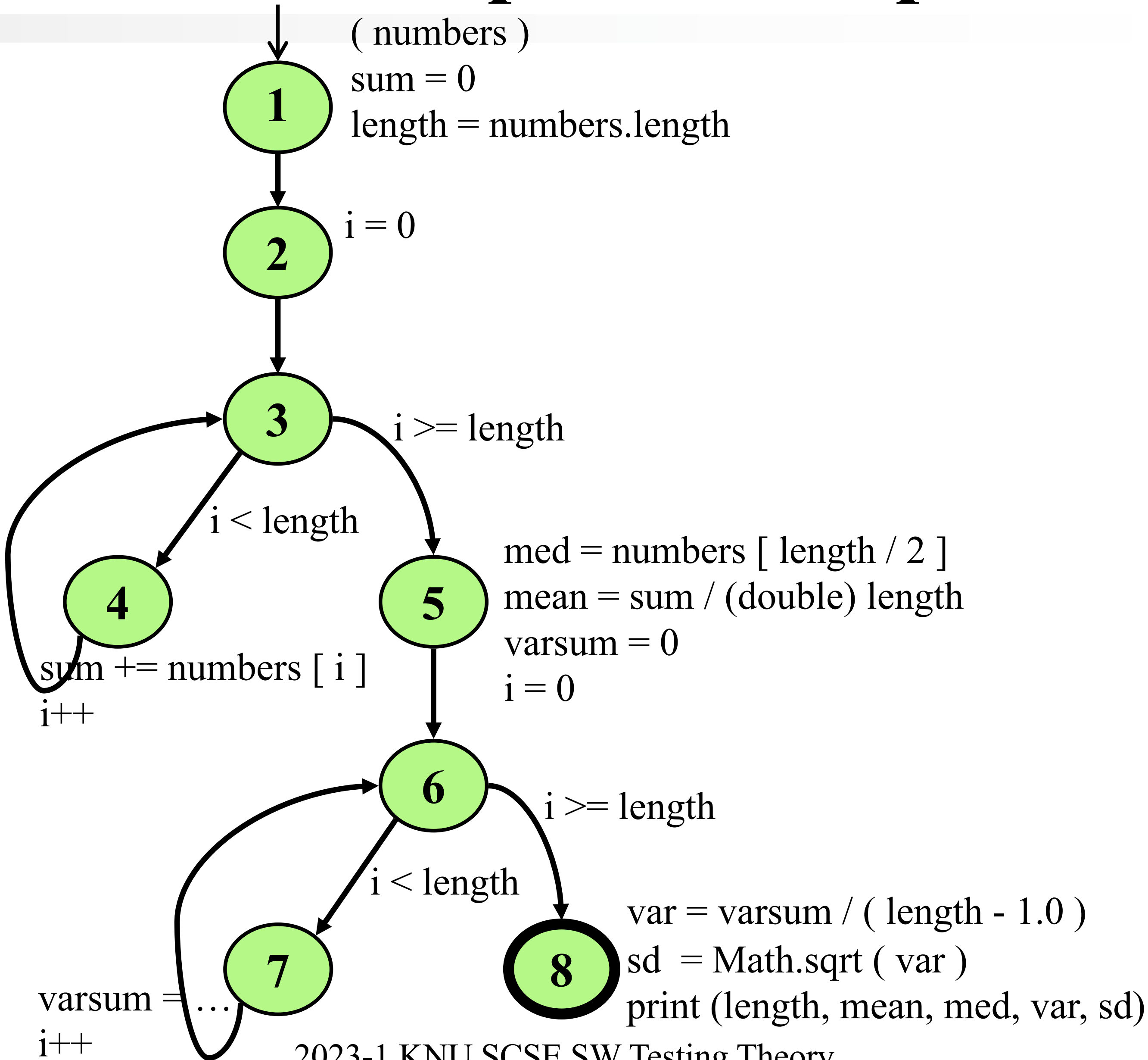
1. Draw a control flow graph
2. Find TR and Test Paths for EC
3. Find TR and Test Paths for EPC

Dataflow Testing

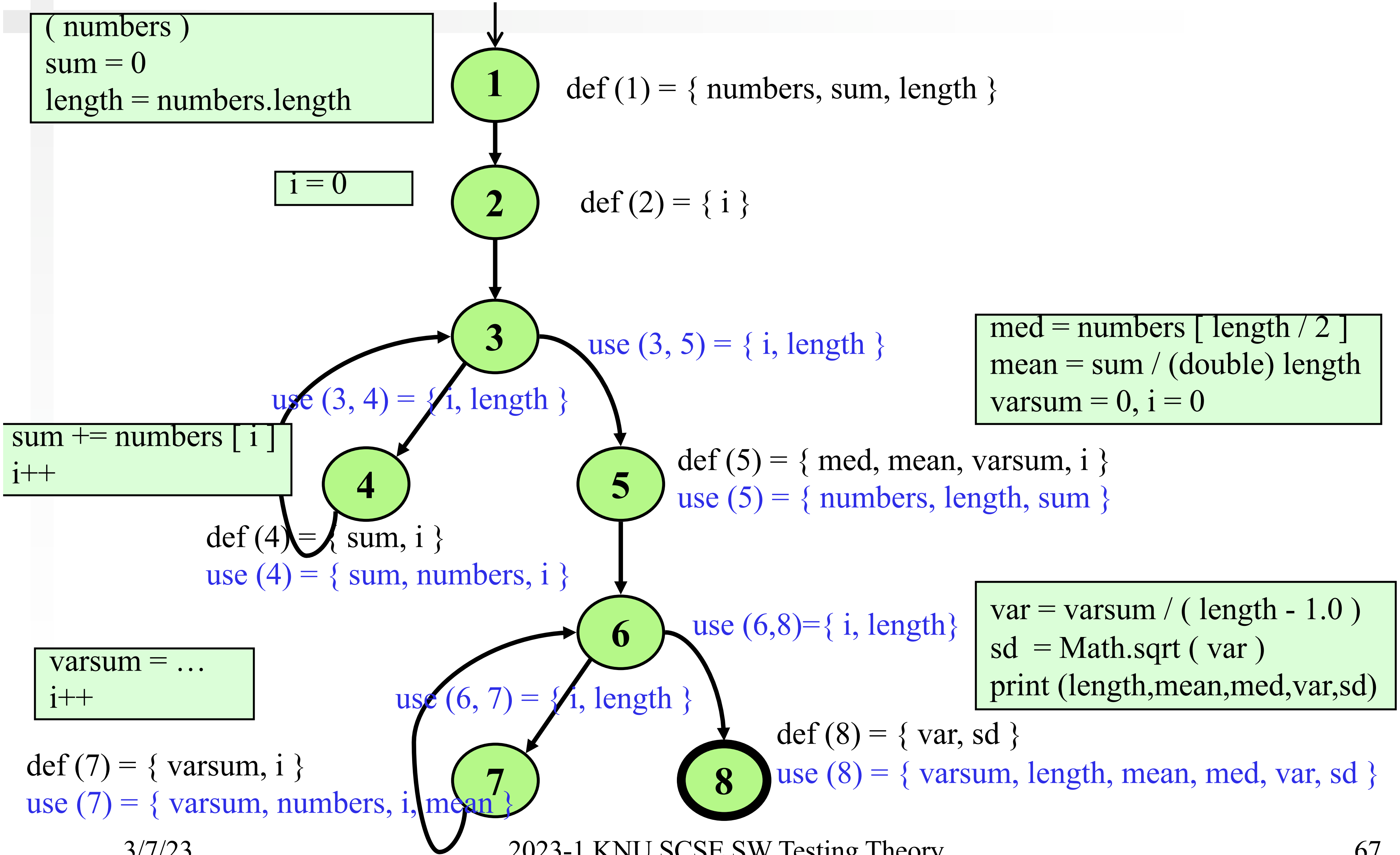
Data Flow Coverage

- def : a location where a value is stored into memory
  - x appears on the left side of an assignment (`x = 44;`)
  - x is an actual ref. parameter in a call and the method changes its value
  - x is a formal parameter (implicit DEF when method starts)
  - x is an input to a program
- use : a location where variable's value is accessed
  - x appears on the right side of an assignment or a conditional test
  - x is an actual parameter to a method
  - x is an output of the program or a method (in return statement)
- When a def and a use appear on the same node,
  - a DU-pair if the def occurs after the use in a loop
  - Not a DU-pair if the def occurs before the use

Control Flow Graph for computeStats()



CFG for Stats – With Defs & Uses



Defs and Uses Tables for computeStats()

Node	Def	Use
1	{ numbers, sum, length }	{ numbers }
2	{ i }	
3		
4	{ sum, i }	{ numbers, i, sum }
5	{ med, mean, varsum, i }	{ numbers, length, sum }
6		
7	{ varsum, i }	{ varsum, numbers, i, mean }
8	{ var, sd }	{ varsum, length, var, mean, med, var, sd }

Edge	Use
(1, 2)	
(2, 3)	
(3, 4)	{ i, length }
(4, 3)	
(3, 5)	{ i, length }
(5, 6)	
(6, 7)	{ i, length }
(7, 6)	
(6, 8)	{ i, length }

DU Pairs for computeStats()

variable	DU Pairs
numbers	(1, 4) (1, 5) (1, 7)
length	(1, 5) (1, 8) (1, (3,4)) (1, (3,5)) (1, (6,7)) (1, (6,8))
med	(5, 8)
var	(8, 8)
sd	(8, 8)
mean	(5, 7) (5, 8)
sum	(1, 4) (1, 5) (4, 4) (4, 5)
varsum	(5, 7) (5, 8) (7, 7) (7, 8)
i	(2, 4) (2, (3,4)) (2, (3,5)) ~~(2, 7) (2, (6,7)) (2, (6,8))~~ (4, 4) (4, (3,4)) (4, (3,5)) ~~(4, 7) (4, (6,7)) (4, (6,8))~~ (5, 7) (5, (6,7)) (5, (6,8)) (7, 7) (7, (6,7)) (7, (6,8))

defs come **before** uses, do not count as DU pairs

defs **after** use in loop, these are valid DU pairs

No def-clear path … different scope for i

DU Paths for computeStats()

variable	DU Pairs	DU Paths
numbers	(1, 4) (1, 5) (1, 7)	[1, 2, 3, 4] [1, 2, 3, 5] [1, 2, 3, 5, 6, 7]
length	(1, 5) (1, 8) (1, (3,4)) (1, (3,5)) (1, (6,7)) (1, (6,8))	[1, 2, 3, 5] [1, 2, 3, 5, 6, 8] [1, 2, 3, 4] [1, 2, 3, 5] [1, 2, 3, 5, 6, 7] [1, 2, 3, 5, 6, 8]
med	(5, 8)	[5, 6, 8]
var	(8, 8)	*No path needed*
sd	(8, 8)	*No path needed*
sum	(1, 4) (1, 5) (4, 4) (4, 5)	[1, 2, 3, 4] [1, 2, 3, 5] [4, 3, 4] [4, 3, 5]

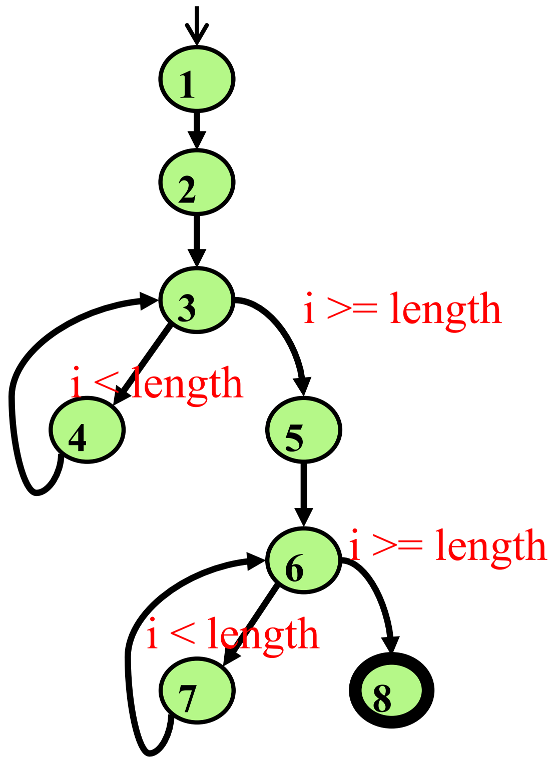
variable	DU Pairs	DU Paths
mean	(5, 7) (5, 8)	[5, 6, 7] [5, 6, 8]
varsum	(5, 7) (5, 8) (7, 7) (7, 8)	[5, 6, 7] [5, 6, 8] [7, 6, 7] [7, 6, 8]
i	(2, 4) (2, (3,4)) (2, (3,5)) (4, 4) (4, (3,4)) (4, (3,5)) (5, 7) (5, (6,7)) (5, (6,8)) (7, 7) (7, (6,7)) (7, (6,8))	[2, 3, 4] [2, 3, 4] [2, 3, 5] [4, 3, 4] [4, 3, 4] [4, 3, 5] [5, 6, 7] [5, 6, 7] [5, 6, 8] [7, 6, 7] [7, 6, 7] [7, 6, 8]

DU Paths for Stats – No Duplicates

There are 38 DU paths for Stats, but only 12 unique

Infeasible Path

[1, 2, 3, 4]	[4, 3, 4]
[1, 2, 3, 5]	[4, 3, 5]
[1, 2, 3, 5, 6, 7]	[5, 6, 7]
[1, 2, 3, 5, 6, 8]	[5, 6, 8]
[2, 3, 4]	[7, 6, 7]
[2, 3, 5]	[7, 6, 8]

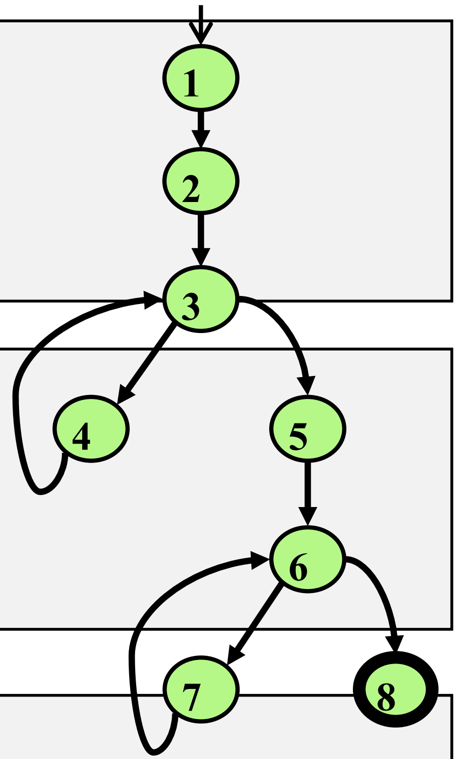


4 expect a loop not to be "entered"

6 require at least one iteration of a loop

2 require at least two iterations of a loop

Test Cases and Test Paths



Test Path : [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]

DU Paths covered

[1, 2, 3, 4] [2, 3, 4] [4, 3, 5] [5, 6, 7] [7, 6, 8]

The five stars that require at least one iteration of a loop

Test Path : [1, 2, 3, 4, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 7, 6, 8]

DU Paths covered

[4, 3, 4] [7, 6, 7]

The two stars that require at least two iterations of a loop

Test Path : [1, 2, 3, 5, 6, 8]

Additional DU Paths covered

[1, 2, 3, 5] [2, 3, 5] [5, 6, 8]

Other DU paths require arrays with length 0 to skip loops

But the method fails with index out of bounds exception…

med = numbers [length / 2];

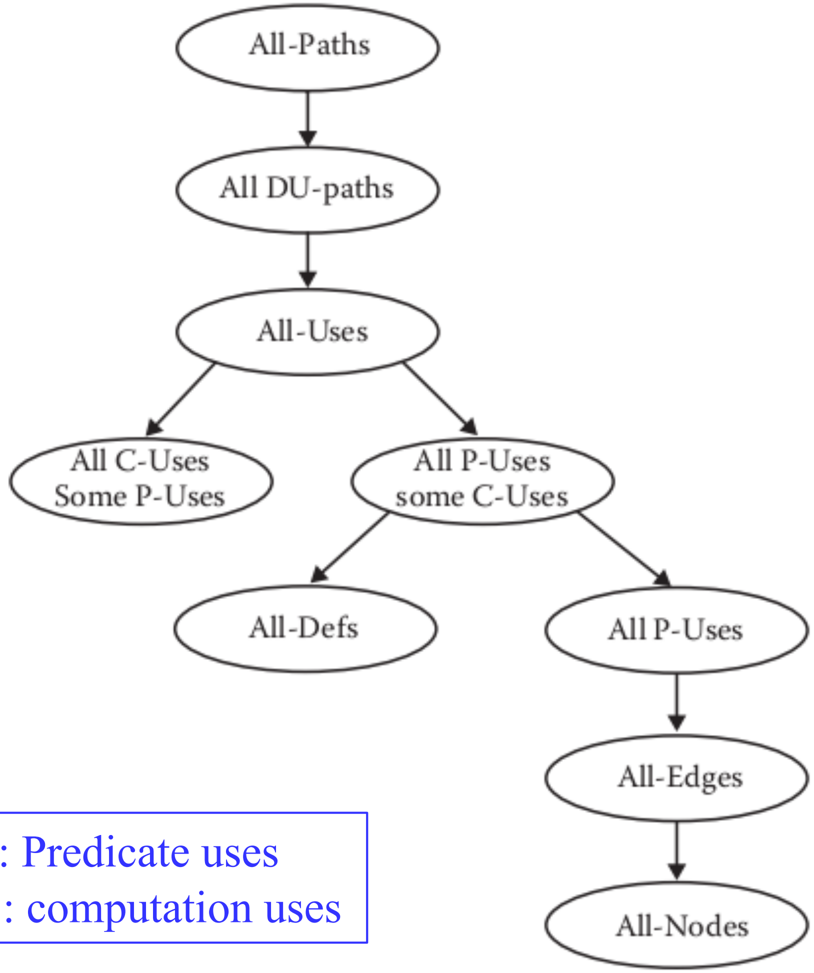
A fault was found

Exercise

```
w=x;
if (m>0){
    w++;
}
else{
    w=2*w;
}

if (y <=10){
   x=5*y;
}
else{
   x = 3*y+5;
}
z = w+x;
```

1. Find DU paths
2. Find test cases and test paths to cover all DU paths

DU-Path Coverage Metrics

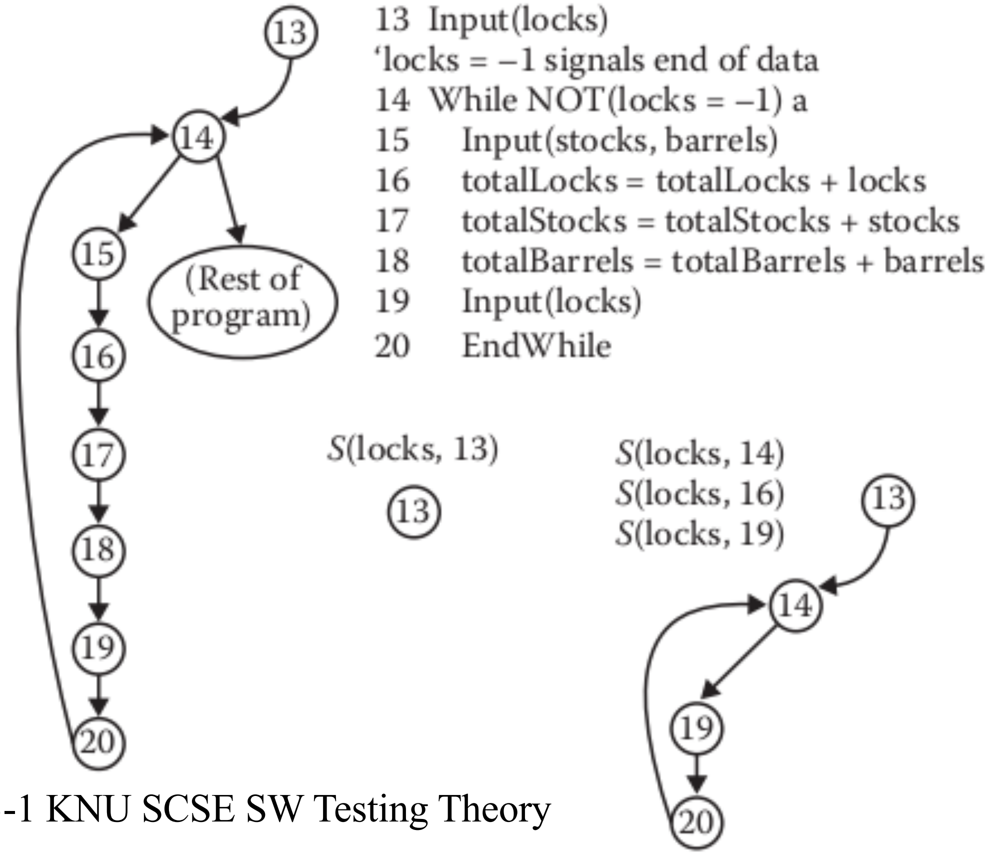


P-Uses : Predicate uses
C-Uses : computation uses

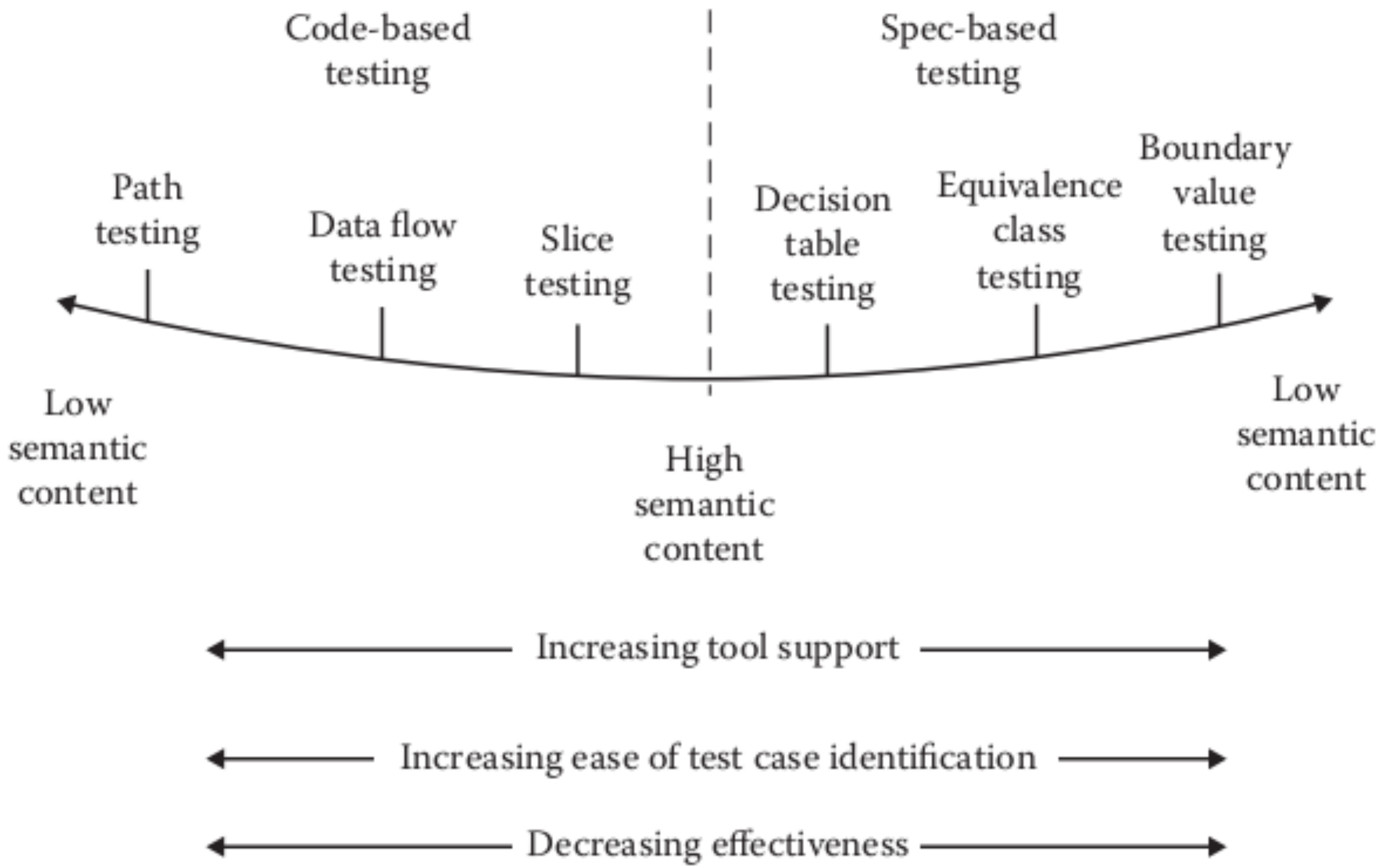
Rapps–Weyuker hierarchy of data flow coverage metrics.

Slice-Based Testing

- Slice S(V, n)
  - The set of all statement fragments in P that contribute to the value of variables in V at node n
  - Forward Slice / Backward Slice



Analysis of Structural Test Methods



Summary

- Test Cases are inputs to test the system and the predicted outputs from these inputs
- In black-box testing, test cases are constructed from the system specification
  - Boundary value Testing, Equivalence Class Testing, Decision Table-based Testing, etc.
- In white-box testing, test cases are constructed from the control flow of program code
  - Edge coverage, Edge-Pair Coverage, etc.
- Data flow testing uses DU(definition-use) pair of variables
- Test coverages include statement coverage, decision coverage, C/DC, and MC/DC coverage

참고문헌

- Paul C. Jorgensen, Software Testing : A Craftman's Approach, 4th Edition, CRC Press, 2014
- Paul Ammann, Jeff Offutt, Introduction to Software Testing, 2008
- Sommerville, Software Engineering, 9th Edition, Addison-Wesley, 2010
- Wikipedia