



25 YEARS ANNIVERSARY
SOICT

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY



HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Software Quality Assurance

Đảm bảo chất lượng phần mềm

Lecture 5: White Box Approach

Contents

- Control Flow Testing
- Predicate Testing
- Data Flow Testing
- Unit-level Testing
 - Mutation Testing



5.1. White Box Testing

Effective Test Case Design

- Exhaustive testing (use of all possible inputs and conditions) is impractical
 - Must use a subset of all possible test cases
 - Must have high probability of detecting faults
- Need processes that help us selecting test cases
- Effective testing - detect more faults
 - Focus attention on specific types of faults
 - Know you are testing the right thing
- Efficient testing - detect faults with less effort
 - Avoid duplication
 - Systematic techniques are measurable and repeatable

Basic Testing Strategies

- Motivation:
 - Effective Test Case Design
- Compare 2 approaches
 - Black Box
 - White Box

Test Strategy	Tester's View	Knowledge Sources	Methods
Black box		Requirements document Specifications Domain knowledge Defect analysis data	Equivalence class partitioning Boundary value analysis State transition testing Cause and effect graphing Error guessing
White box		High-level design Detailed design Control flow graphs Cyclomatic complexity	Statement testing Branch testing Path testing Data flow testing Mutation testing Loop testing

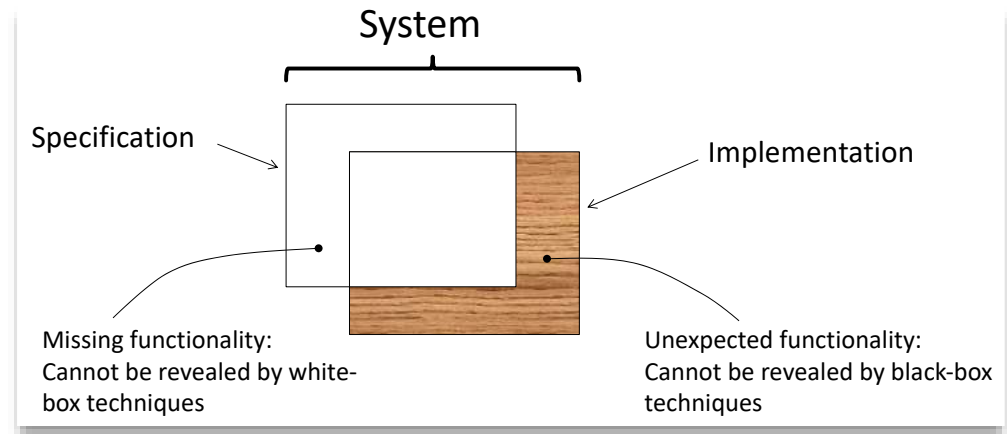
Black Box vs. White Box

- Black Box: functional testing

- Unit test
- Integration test
- System test
- Programmers & Test Engineers & Quality Assurance Engineers

- White Box: structural testing

- Unit test
- Integration test
- Programmers & Test Engineers



5.2. Control Flow Testing

Kiểm thử luồng điều khiển

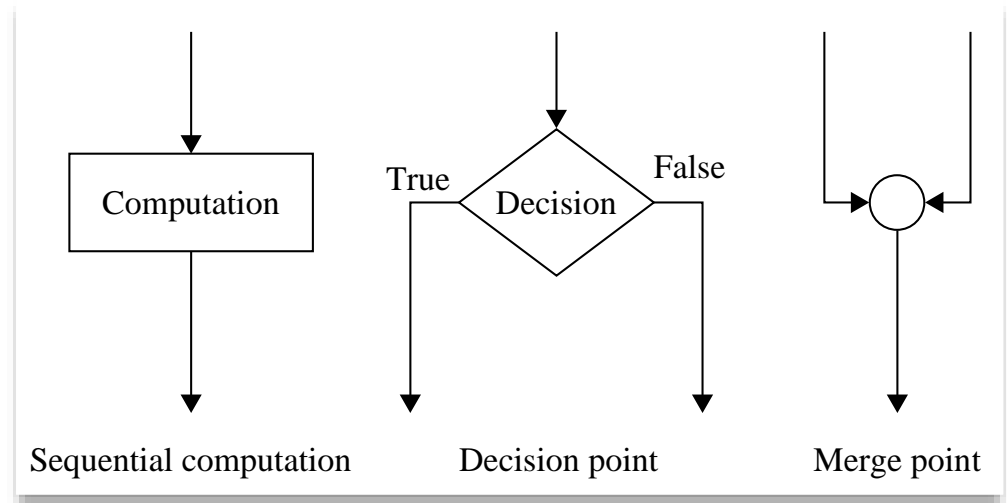
Basic terms

- Program unit: entry point to exit point
 - Procedures, Functions, Modules, Components
- 2 kinds of statement
 - Assignment statement
 - Conditional statement
- Program path
 - A sequence of statements from entry point to exit point of a program unit
 - A program unit may contains many program paths
 - An execution instance of the unit
- A given set of input data, the unit executes a different path

Control Flow Graph

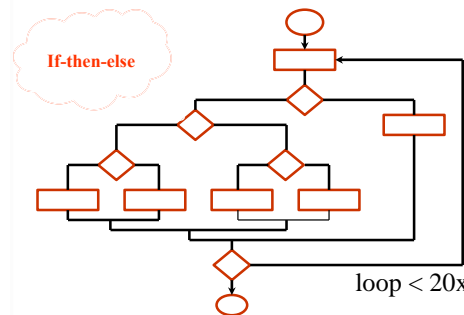
Đồ thị luồng điều khiển

- Represent the graphical structure of a program unit
- A sequence of statements from entry point to exit point of the unit depicted using graph notions



Control Flow Testing

- Main idea: select a few paths in a program unit and observe whether or not the selected paths produce the expected outcome
- Executing a few paths while trying to assess the behavior of the entire program unit



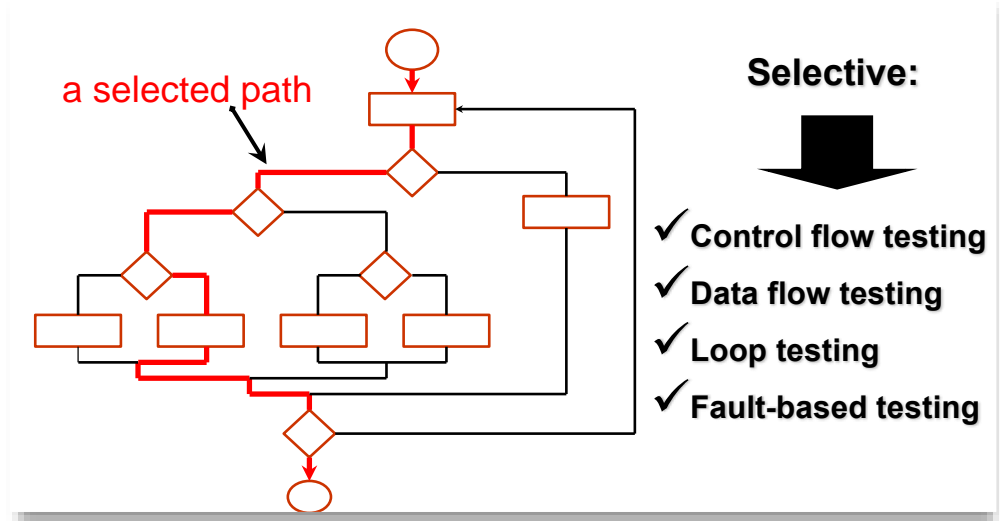
There are many possible paths!
 5^{20} ($\sim 10^{14}$) different paths



Selective Testing

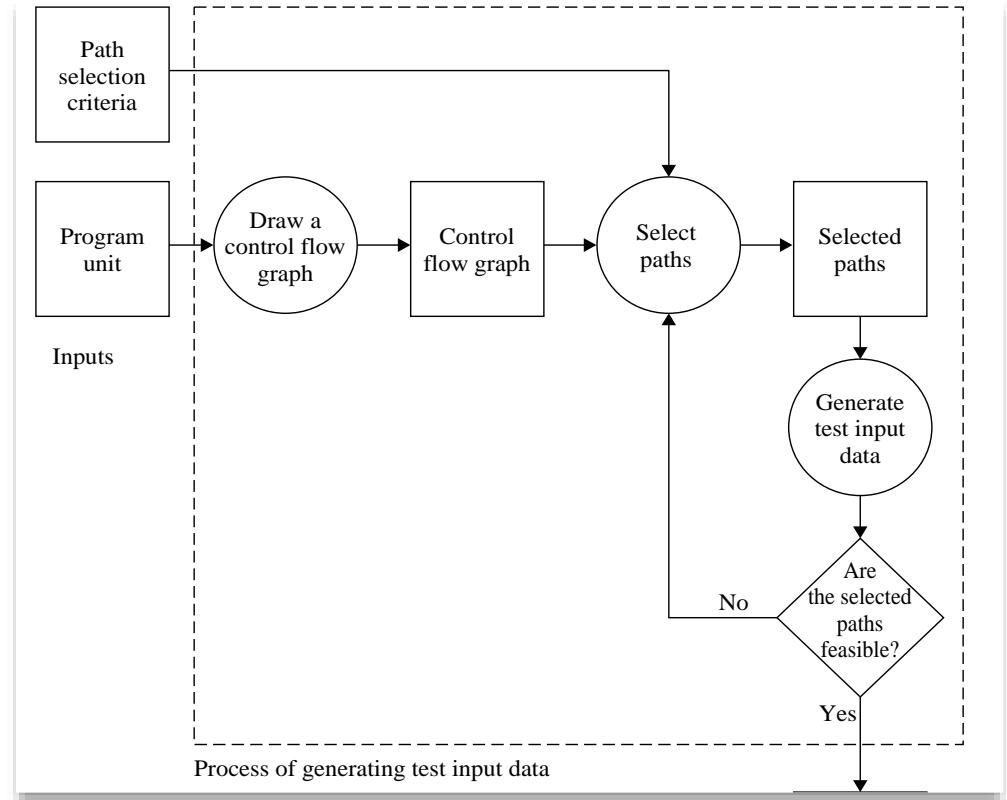
Select as few paths as possible

- Selecting relevant paths based on some criteria
 - All paths
 - Statement Coverage
 - Branch Coverage
 - Predicate Coverage
- Most applicable to new software for unit test



Outline of Control Flow Testing

- Inputs
 - Source code of unit
 - Path selection criteria
- Generate CFG: draw CFG from source code of the unit
- Selection of paths: selected paths to satisfy path selection criteria
- Generation of test input data



Path selection criteria

- Example:
 - Given the source code of the function **AccClient**
 - Draw the CFG

Life Insurance Example

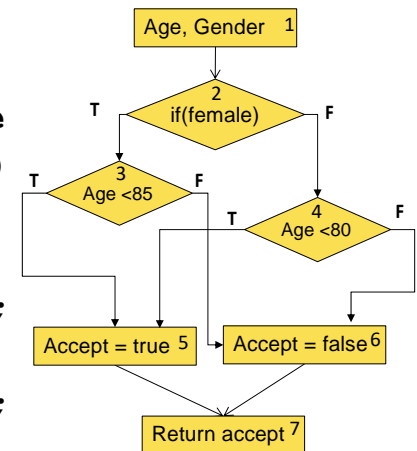
```
bool AccClient(agetype
    age; gndrtype gender)
bool accept
    if(gender=female)
        accept := age < 85;
    else
        accept := age < 80;
return accept
```

Path selection criteria

- Example:
 - Given the source code of the function **AccClient**
 - Draw the CFG

Life Insurance Example

```
bool AccClient(agetype age; gndrtype gender)
bool accept
if(gender=female)
    accept := age < 85;
else
    accept := age < 80;
return accept
```



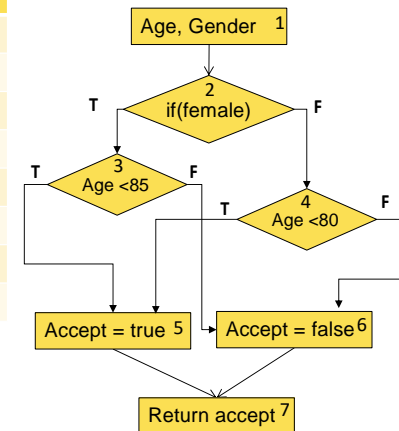
All path coverage criterion

Tiêu chí lựa chọn tất cả các đường dẫn

- Objective: Design all possible test cases so that all paths of the program are executed
- 4 test cases satisfy the all path coverage criterion

All paths

Female	Age < 85	Age < 80
Yes	Yes	Yes
Yes	Yes	No
Yes	No	Yes
Yes	No	No
No	Yes	Yes
No	Yes	No
No	No	Yes
No	No	No

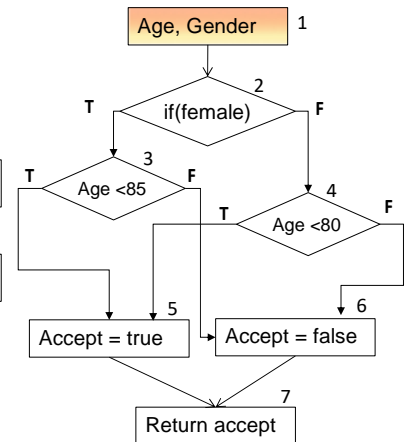


Statement coverage criterion

Tiêu chí bao phủ hết tập câu lệnh

- Main idea: Execute each statement at least once
- A possible concern may be:
 - dead code

```
bool AccClient (agetype
age; gndrtype gender)
bool accept
if (gender=female)
    accept := age < 85;
else
    accept := age < 80;
return accept
```



Disadvantages of statement coverage

- Statement coverage is the weakest, indicating the fewest number of test cases
- Bugs can easily occur in the cases that statement coverage cannot see
- The most significantly shortcoming of statement coverage is that it fails to measure whether you test simple If statements with a **false decision outcome**.

Branch coverage criterion

Tiêu chí bao phủ nhánh

- Also called Decision Coverage
- A branch is an outgoing edge from a node
 - A rectangle node has at most one out going branch
 - All diamond nodes have 2 outgoint branches
- A decision element in a program may be one of
 - If – then
 - Switch – case
 - Loop
- Main idea: selecting paths such that every branch is included in at least one path

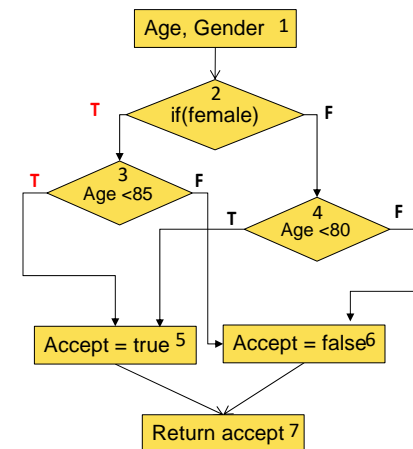
Example

- We test the path
 - 1-2(T)-3(T)-5-7

Branch Coverage /1

AccClient(83,
female)->accept

```
bool AccClient (agetype  
age; gndrtype gender)  
bool accept  
  if (gender=female)  
    accept := age < 85;  
  else  
    accept := age < 80;  
  return accept
```



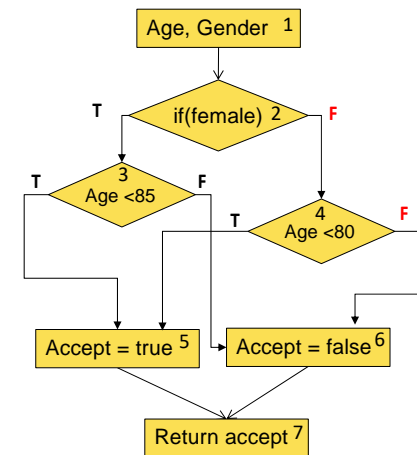
Example

- We test the path
 - 1-2(F)-4(F)-6-7

Branch Coverage /2

AccClient(83, male)
->reject

```
bool AccClient (agetype  
    age; gndrtype gender)  
bool accept  
    if (gender=female)  
        accept := age < 85;  
    else  
        accept := age < 80;  
return accept
```

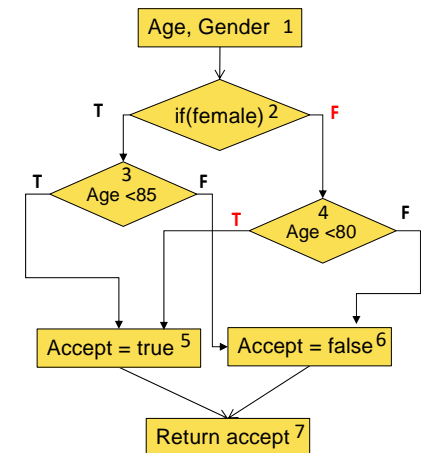


Example

- We test the path
 - 1-2(F)-4(T)-5-7

Branch Coverage /3 AccClient(78, male)->accept

```
bool AccClient (agetype
age; gndrtype gender)
bool accept
if (gender=female)
    accept := age < 85;
else
    accept := age < 80;
return accept
```

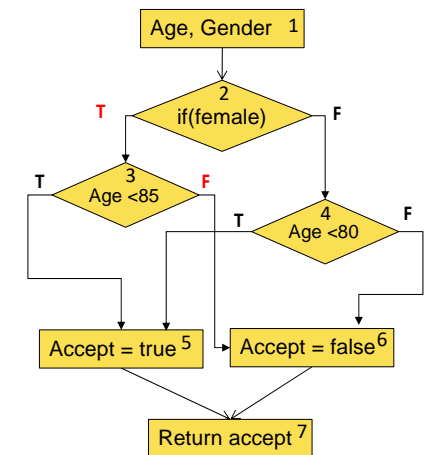


Example

- We test the path
 - 1-2(T)-3(F)-6-7

Branch Coverage /4 AccClient(88,
female) ->reject

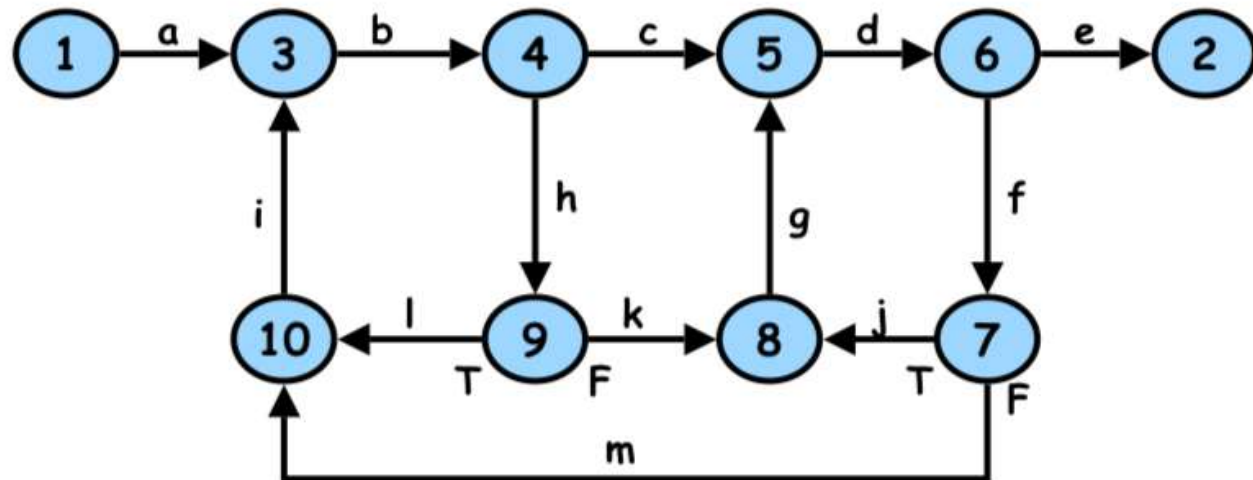
```
bool AccClient(agetype  
age; gndrtype gender)  
bool accept  
if(gender=female)  
    accept := age < 85;  
else  
    accept := age < 80;  
return accept
```



Comparing 3 criteria

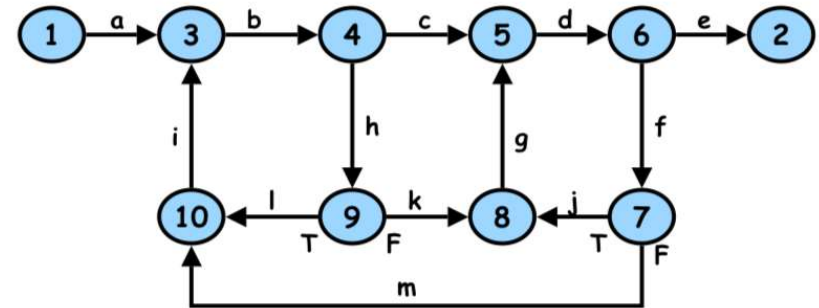
- (1) All path coverage: assure 100% paths executed
- (2) Statement coverage: pick enough paths to assure that every source statement is executed at least once
- (3) Branch coverage: assure that every branch has been exercised at least once under some test
- (1) implies (3), (3) implies (2)
- These 3 criteria are also called as **Path Testing Techniques**

Example of statement and branch coverage



Example of statement and branch coverage

- Question 1:
 - Does every decisions have a T and F values?
 - Yes → Implies branch coverage
- Question 2:
 - Is every link covered at least once?
 - Yes → Implies statement coverage



Path	Decisions				Process Link												
	4	6	7	9	a	b	c	d	e	f	g	h	i	j	k	l	m
abcde	T	T			x	x	x	x	x								
abhkgde	F	T		F	x	x		x	x		x	x			x		
abhlbcde	F	T		T	x	x	x	x	x			x	x			x	
abcdfjgde	T	F	T		x	x	x	x	x	x	x			x			
abcdfmibcde	T	F	F		x	x	x	x		x			x				x

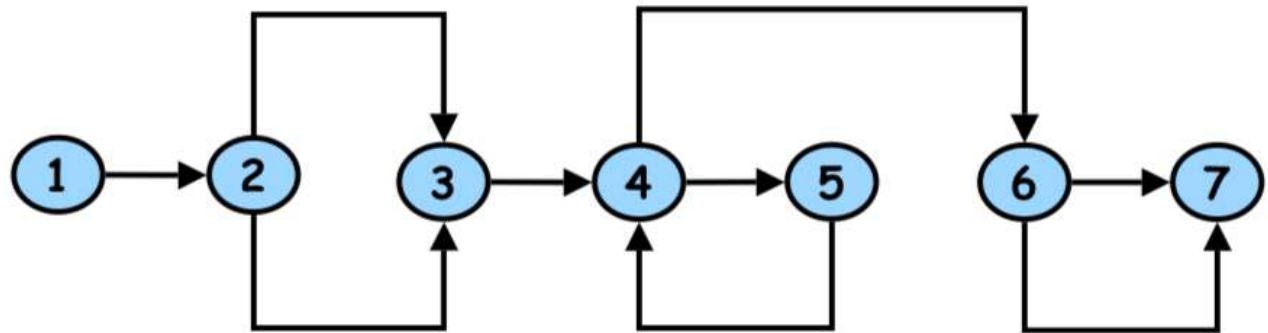
Example: Exponential Function

```
1  scanf("%d %d",&x, &y);
2  if (y < 0)
    pow = -y;
    else
        pow = y;
3  z = 1.0;
4  while (pow != 0) {
    z = z * x;
    pow = pow - 1;
5  }
6  if (y < 0)
    z = 1.0 / z;
7  printf ("%f",z);
```



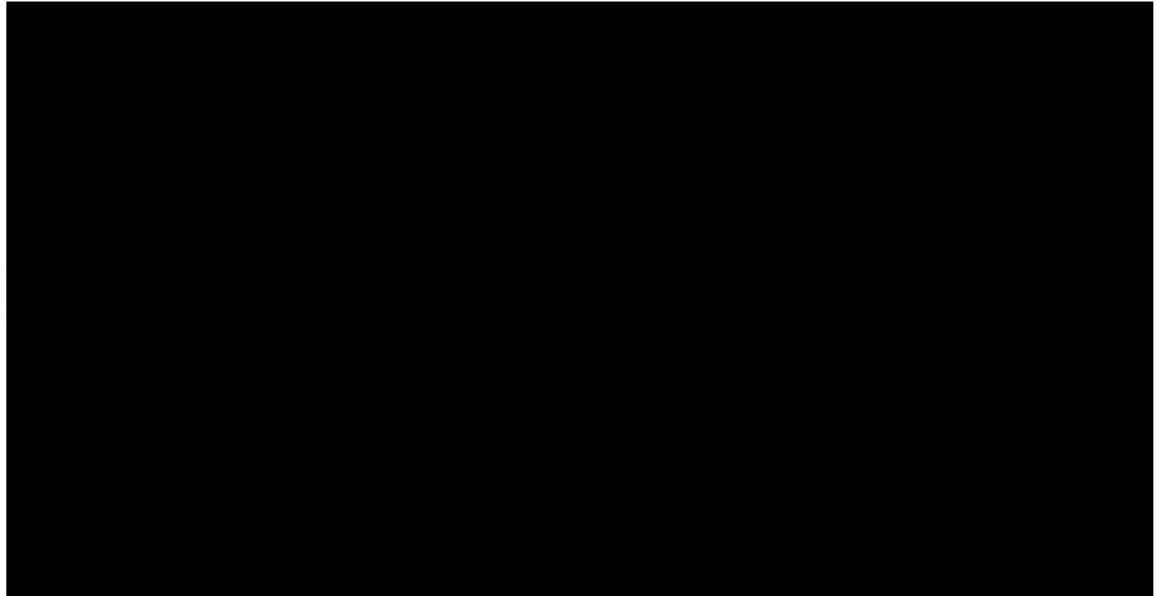
Example: Exponential Function

```
1 scanf("%d %d",&x, &y);
2 if (y < 0)
    pow = -y;
  else
    pow = y;
3 z = 1.0;
4 while (pow != 0) {
    z = z * x;
    pow = pow - 1;
5 }
6 if (y < 0)
    z = 1.0 / z;
7 printf ("%f",z);
```



Example: Bubble sort

```
1 for (j=1; j<N; j++) {  
    last = N - j + 1;  
2     for (k=1; k<last; k++) {  
3         if (list[k] > list[k+1]) {  
            temp = list[k];  
            list[k] = list[k+1];  
            list[k+1] = temp;  
4         }  
5     }  
6 }  
7 print("Done\n");
```

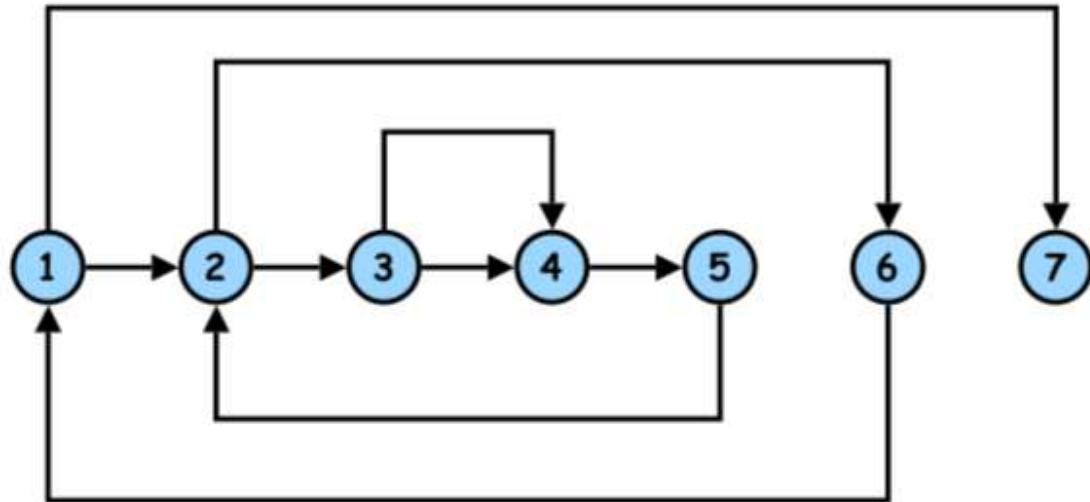


Example: Bubble sort

```

1 for (j=1; j<N; j++) {
    last = N - j + 1;
2     for (k=1; k<last; k++) {
3         if (list[k] > list[k+1]) {
            temp = list[k];
            list[k] = list[k+1];
            list[k+1] = temp;
4         }
5     }
6 }
7 print("Done\n");

```



Limitations of path testing techniques

- Path Testing is applicable to new unit
- Limitations
 - Interface mismatches and mistakes are not taken
 - Not all initialization mistakes are caught by path testing
 - Specification mistakes are not caught

5.3. Predicate Coverage

Tiêu chí bao phủ điều kiện

Basic concepts

- Predicate: are expressions that can be evaluated to a boolean value (true or false)
- Predicate may contain:
 - **Boolean** variables
 - **Non-boolean** variables that are compared with the relational operators $\{\geq, \leq, =, \neq, <, >\}$
 - **Boolean function** calls
- The internal structure is created by **logical operators**:
 - $\neg, \rightarrow, \leftrightarrow, \wedge, \vee, \oplus$
- **A clause**: is a predicate that does not contain any of the logic operator.
Example: $(a=b) \vee C \wedge p(x)$ contains 3 clauses

How to create a Path Predicate Expression?

- Write down the predicates for the decisions you meet along a path
- The result is a set of path predicate expressions
- All of these expressions must be satisfied to achieve a selected path

```
X1,X2,X3,X4,X5,X6
if (X5 > 0 || X6 < 0)          /* predicates A,B */
...
if(X1 + 3 * X2 + 17 >= 0)      /* predicate C */
...
if(X3 == 17)                  /* predicate D */
...
if(X4 - X1 >= 14 * X2)        /* predicate E */
...
P
```

Predicate Faults

Các lỗi điều kiện có thể xảy ra

- An incorrect Boolean operator is used
- An incorrect Boolean variable is used
- Missing or extra Boolean variables
- An incorrect relational operator is used
- **Parentheses** are used incorrectly

Predicate Coverage

- To ensure all predicates in the source code are implemented correctly
- Main idea:
 - For each predicate p , test cases must assure that p evaluates to true at least once and p evaluates to false at least once

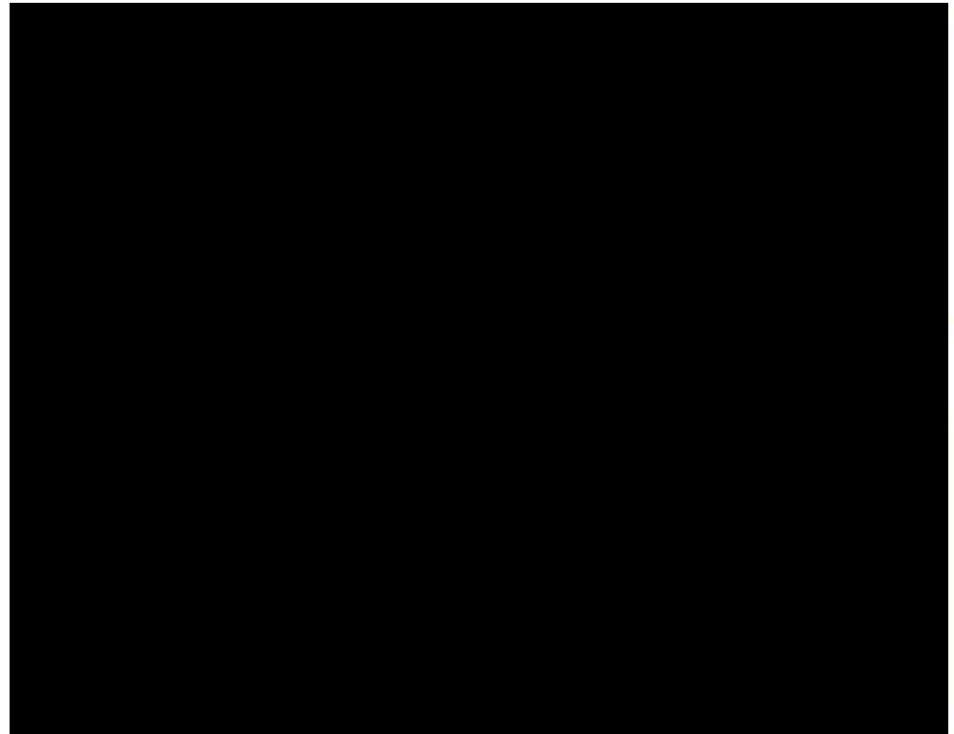
Example: $p = ((a > b) \vee C) \wedge p(x)$

	a	b	C	p(x)
1	5	4	true	true
2	5	6	false	false

“decision coverage” in literature

Example

$$p = ((a < b) \vee D) \wedge (m \geq n * o)$$



Clause Coverage

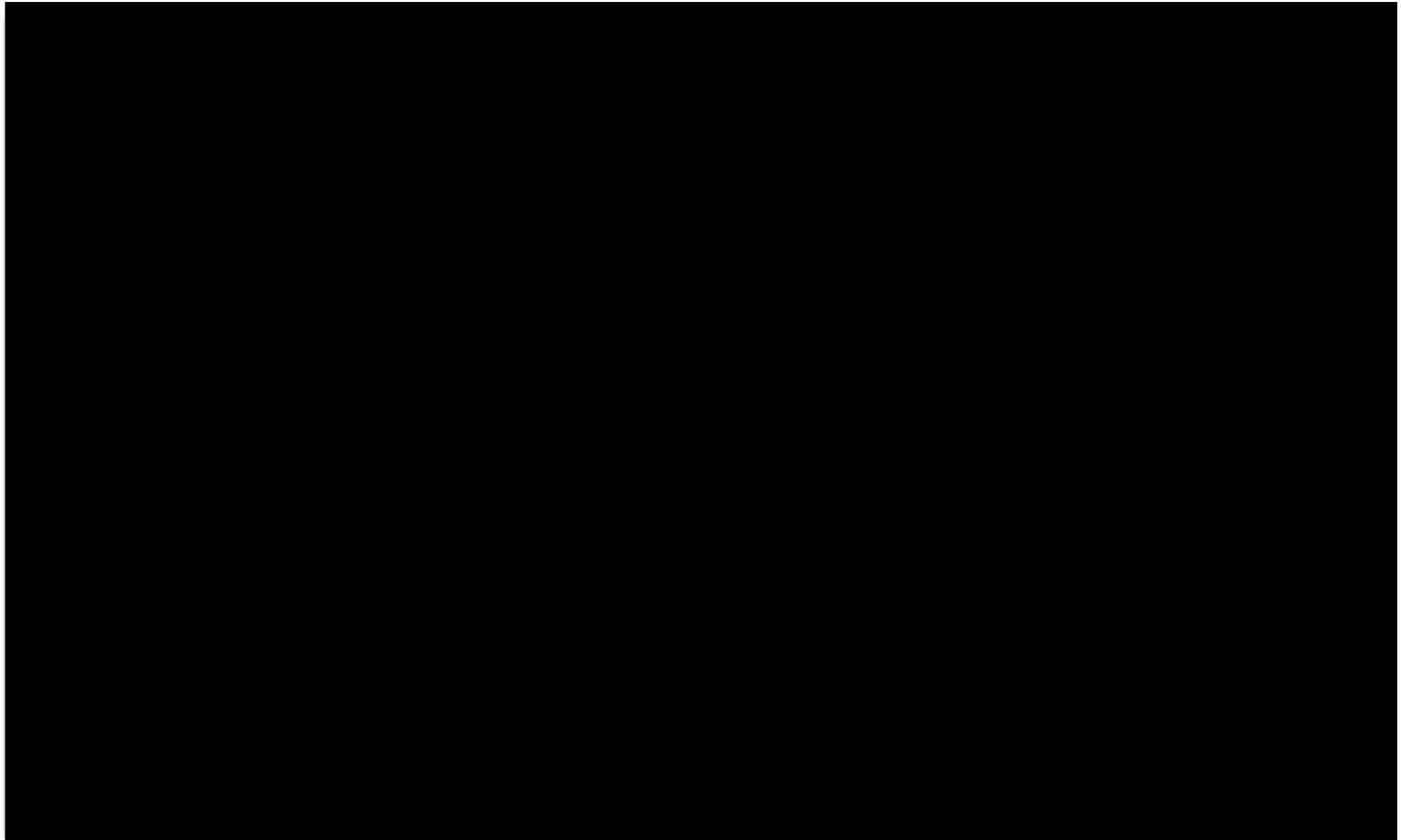
- Main idea: For each individual condition (clause) c , c should evaluate to true and c should evaluate to false at least once.

Example: $((a > b) \vee C) \wedge p(x)$

	a	b	C	p(x)
1	5	4	true	true
2	5	6	false	false

“condition coverage” in literature

Example



Predicate vs. Clause Coverage

- Does Predicate coverage subsume clause coverage?
- Does clause coverage subsume predicate coverage?
- Naturally, we want to test both the predicate and individual clauses

Example: $p = a \vee b$

	a	b	$a \vee b$
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

Predicate and Clause Coverage

- PC does not **fully** exercise all the clauses
 - For example: $p = a \vee b$. In most programming languages, at run time, when a evaluates to True, p evaluates to True and **b does not exercise**.
- CC does not always ensure PC
- This is, we can satisfy CC without causing the predicate to be both true and false
- This is definitively not what we want
 - We need to come up with another approach
- Inversely, with PC, it is not always that all individual clauses evaluate to both true and false

Combinatorial Coverage

Bao phủ kết hợp

- Main idea: For each predicate p , test cases should ensure that the clauses in p should evaluate to each possible combination of truth values

Example: $(a \vee b) \wedge c$

CoC requires every possible combination

	a	b	c	$(a \vee b) \wedge c$
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Example 1

- Write all the clauses and the Combination of Clauses (CoC) of the given predicate $P = ((a > b) \vee C) \wedge p(x)$

Example 1 - Answer

$$P = ((a > b) \vee C) \wedge p(x)$$

$$P = (X \vee Y) \wedge Z$$

	X	Y	Z	Predicate
1	F	F	F	F
2	F	F	T	F
3	F	T	F	F
4	F	T	T	T
5	T	F	F	F
6	T	F	T	T
7	T	T	F	F
8	T	T	T	T

Z is more important clause in this predicate than the others

What is the problem with combinatorial coverage?

- Combinatorial coverage is very expensive if we have multiple clauses in the predicate
- For each decision point with N conditions, we need 2^N test cases

Motivation

- Predicate should evaluate to both true and false in test cases
- Each individual condition (clause) is tested independently of each other
- Each individual clause should affect the predicate
 - i.e., changing the value of an individual clause, the value of predicate is also changed

Active clause

- Major clause: the clause which is being focused upon
- Minor clause: all other clauses in the predicate
- Determination: clause c in p determines p

A clause c_i in predicate p , called the major clause, determines p if and only if the values of the remaining minor clauses c_j are such that changing c_i changes the value of p

$$P = A \vee B$$

if $B = \text{true}$, P is always true.
so if $B = \text{false}$, A determines P .
if $A = \text{false}$, B determines P .

$$P = A \wedge B$$

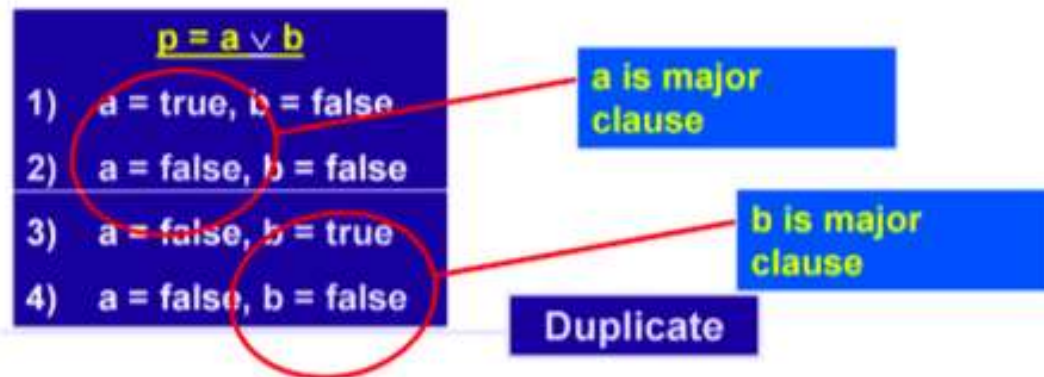
if $B = \text{false}$, P is always false.
so if $B = \text{true}$, A determines P .
if $A = \text{true}$, B determines P .

Example: $P = a \wedge (b \vee c)$

- Identify, when considering a as active clause, which values should be assigned to b and c?
- Answer:
 - a is active clause, the values of b and c are those such that changing the value of a changes the value of P.
 - So, $(b \vee c) = 1$, the value of P is determined by the value of a. We can assign $b = 1, c = 1$
- Test case $T = \{(a=T, b=T), (a=F, b=F)\}$ satisfy GACC, but not PC, all both cases, $P = T$

Active Clause Coverage (ACC)

- For each predicate **p** and each major clause **c** of **p**, choose minor clauses so that **c** determines **p**. 2 test requirements for assuring ACC: for each **c**: **c** evaluates to **true** and **c** evaluates to **false**
- With a decision point consists of **N** conditions, only **N+1** test cases are required. Why?



Example

```
public static void printHonorRollStatus(double cumulativeGPA,  
    double termGPA, int creditsCompleted, boolean fullTimeStatus) {  
    // Determine if the student is on the deans list.  
    if ((creditsCompleted > 30) && (cumulativeGPA > 3.20)  
        && (fullTimeStatus == true) && (termGPA > 2.0)) {  
        System.out.println("You are on the dean's list.");  
    } else if ((creditsCompleted > 30) && (cumulativeGPA > 3.70)  
        && (fullTimeStatus == true) && (termGPA > 2.0)) {  
        System.out.println("You are on the high honors dean's list.");  
    } else if ((creditsCompleted > 30) && (cumulativeGPA > 2.0)  
        && (fullTimeStatus == true) && (termGPA > 3.2)) {  
        System.out.println("You are on the honor list.");  
    } else {
```

Example - Answer

	CC	CGPA	FT	TGPA	Predicate
	29	3.3	True	2.4	False
➔	31	3.3	True	2.4	True
	31	3.0	True	2.4	False
➔	31	3.3	True	2.4	True
	31	3.3	False	2.4	False
➔	31	3.3	True	2.4	True
	31	3.3	True	1.9	False
➔	31	3.3	True	2.4	True

- The Green cells indicate active clauses,
- The Orange color cells indicate minor clauses
- We can test this with only **5 test cases**

Resolving the Ambiguity problem

Giải quyết vấn đề nhập nhằng

- With the example $p = a \vee (b \wedge c)$, **a is active clause**
- How we can choose the value of b and c? c should be different from b or not?

The diagram shows a logical expression $p = a \vee (b \wedge c)$ inside a box. Below it, the text "Major clause : a" is present. Two sets of variable assignments are listed: "a = true, b = false, c = true" and "a = false, b = false, c = false". In the second set, the value "c = false" is circled in red. A blue speech bubble points to this circled text and contains the question "Is this allowed?".

General Active Clause Coverage (GACC)

- For each clause c , choose minor clauses' values such that c determines p
- Clause c has to evaluate to both true and false
- Minor clauses don't need to be the same when clause c is true as well as when c is false.
- $c_j(c_i = \text{true}) = c_j(c_i = \text{false})$ for all c_j
OR $c_j(c_i = \text{true}) \neq c_j(c_i = \text{false})$ for all c_j

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Does GACC subsume predicate coverage? NOT always, ex: $P = A \leftrightarrow B$

General Active Clause Coverage (GACC)

a and (b or c)	1	0
a is active	1 . .	0 . .
b is active	. 1 .	. 0 .
c is active	. . 1	. . 0

All active values are in diagonal

General Active Clause Coverage (GACC)

Values of minor clauses are different when active clause evaluates to true and false

a and (b or c)	1	0
a is active	1 1 0	0 0 1
b is active	1 1 0	1 0 0
c is active	1 0 1	1 0 0

Test case selected: 1 1 0, 1 0 1, 0 0 1, 1 0 0

General Active Clause Coverage (GACC)


Values of minor clauses are the same when active clause evaluates to true and false

a and (b or c)	1	0
a is active	1 1 0	0 1 0
b is active	1 1 0	1 0 0
c is active	1 0 1	1 0 0

Test case selected: 1 1 0, 1 0 1, 0 1 0, 1 0 0

Example

$a > b$ C $p(x)$ $((a > b) \parallel C) \&\& p$



GACC and subsumption of CC/PC

- By definition: GACC subsumes CC, that is, satisfying GACC leads to satisfy CC, but not PC
- Example: $P = a \leftrightarrow b$.
 - Test case $T = \{(a=T, b=T), (a=F, b=F)\}$ satisfy GACC, but not PC, all both cases, $P = T$
 - Test case $T = \{(a=T, b=F), (a=F, b=T)\}$ satisfy GACC too, but not PC either, all both cases, $P = F$

Correlated Active Clause Coverage (CACC)

- For each clause c_i in P , choose minor clauses c_j such that c_i determines the predicate P
- Clause c_i has to evaluate to true or false in test cases
- The values chosen for the minor clauses c_j must cause P to be true for one value of the major clause c_i and false for the other value of c_i
- $P(c_i = \text{true}) \neq P(c_i = \text{false})$
- Minor clauses don't need to be the same. That is:
 - $c_j(c_i = \text{true}) = c_j(c_i = \text{false})$ for all c_j **OR** $c_j(c_i = \text{true}) \neq c_j(c_i = \text{false})$ for all c_j

Example

- When a is active clause, which rows can be chosen to satisfy CACC?
 - When a is active, values of b and c are chosen such that $P(a=\text{true}) \neq P(a=\text{false})$
 - Any of rows 1, 2, 3 and any of row 5,6,7 - one of total nine pairs satisfies CACC

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Test cases for CACC

a and (b or c)	1	0
a is active	1 1 0	0 1 0
b is active	1 1 0	1 0 0
c is active	1 0 1	1 0 0

Test case selected: 1 1 0, 1 0 1, 0 1 0, 1 0 0

CACC and subsumption of GACC/CC/PC

- By definition, CACC subsumes GACC thus CC and also PC
- Example: $P = a \leftrightarrow b$
 - Which test cases satisfy PC?
 - Which test cases satisfy CACC?

Example: $P = a \wedge (b \leftrightarrow c)$

- Which test cases satisfy GACC?
- Which test cases satisfy CACC?

t	a	b	c	$p = a \wedge (b \leftrightarrow c)$
1	T	T	T	T
2	T	T	F	F
3	T	F	T	F
4	T	F	F	T
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Restricted Active Clause Coverage (RACC)

- For each predicate P and each active clause c_i in P , choose minor clause c_j so that c_i determines P . c_i evaluates to both true and false in test cases
- Predicate P evaluates to true in one case of major clause and false in the other (that is CACC)
- The values chosen for minor clauses c_j must be the same when c_i evaluates to true as when c_i evaluates to false.
- That is, $c_j(c_i=\text{true})=c_j(c_i=\text{false})$ for all c_j
- This is the stricter version of CACC

RACC Example

- When a is active clause, which rows can be chosen to satisfy RACC?
 - Rows 1 and 8 are not chosen because a is not active in these two cases
 - b and c should have the same values as when a is true as well as when a is false
 - One of three pairs: {1,5}, {2, 6} and {3, 7} satisfy RACC

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Test cases for RACC

a and (b or c)	1	0
a is active	1 1 0	0 1 0
b is active	1 1 0	1 0 0
c is active	1 0 1	1 0 0

Test case selected: 1 1 0, 1 0 1, 0 1 0, 1 0 0

RACC and subsumptions of CACC/GACC/PC/CC

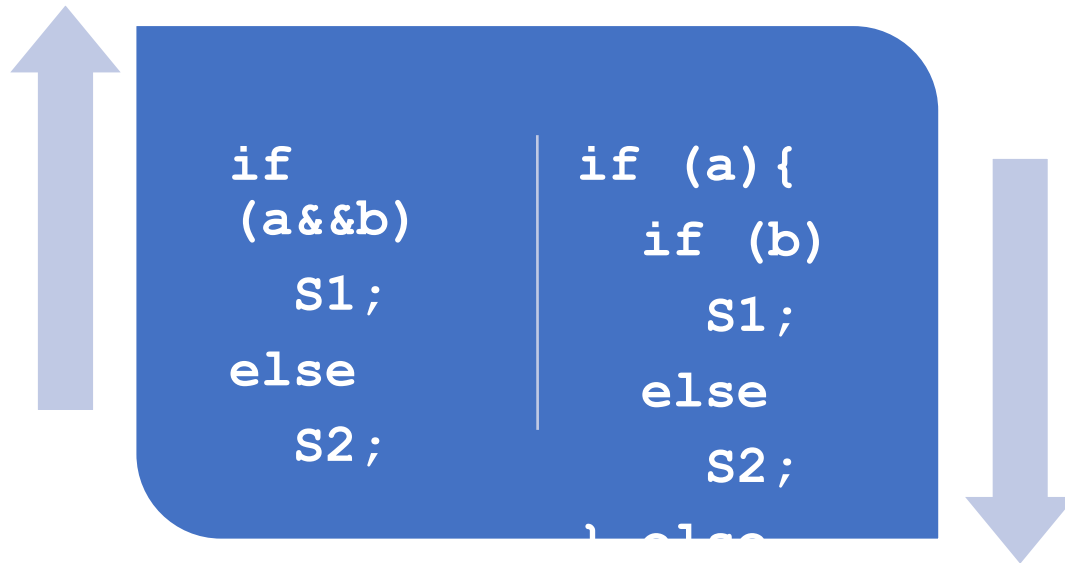
- By definition, RACC subsumes CACC, thus subsumes GACC, CC and PC
- RACC often leads to infeasible test requirements
- Example: Consider the predicate $P = ((a > b) \ \&\& \ (b < c)) \ || \ (c > a) = (X \ \&\& \ Y) \ || \ Z$
 - When Z is active clause, we choose $X = 1, Y = 0$. Infeasible test case: $Z = 0, X = 1, Y = 0$

Logical Operators in Source Code

- $\&$ | \sim \wedge correspond also to logical operators.
- What is the difference between $(a\&\&b)$ and $(a\&b)$? $(a||b)$ and $(a|b)$?

logical expression	Java expression
$a \wedge b$	<code>a && b</code>
$a \vee b$	<code>a b</code>
$\neg a$	<code>! a</code>
$a \rightarrow b$	<code>a ? b : true;</code>
$a \leftrightarrow b$	<code>a == b</code>
$a \oplus b$	<code>a != b</code>

Code transformation



Exercise 1

- Identify test cases for
 - PC
 - CC
 - CoC
 - GACC
 - CACC
 - RACC

```
public static boolean isLeapYear(int y) {  
    return y % 400 == 0 || (y % 4 == 0 && y % 100 != 0);  
}
```

```
a: y % 400 == 0    b: y % 4 == 0    c: y % 100 != 0  
p: a ∨ (b ∧ c)
```

Exercise 2

- Determine predicates and clauses in the source code
- Identify reachable predicates of the source code
- Identify test cases that satisfy PC, CC and CoC
- Are there infeasible requirements ?

```
public static int daysInMonth(int m, int y) {  
    if (m <= 0 || m > 12)  
        throw new IllegalArgumentException("Invalid month: " + m);  
    if (m == 2) {  
        if (y % 400 == 0 || (y % 4 == 0 && y % 100 != 0))  
            return 29;  
        else  
            return 28;  
    }  
    if (m <= 7) {  
        if (m % 2 == 1)  
            return 31;  
        return 30;  
    }  
    if (m % 2 == 0)  
        return 31;  
    return 30;  
}
```

○ Predicates and clauses

- p1: c1 || c2 ; c1: m <= 0, c2: m > 12
- p2: c3; c3: m == 2;
- p3: c4 || (c5 && c6) ; c4: y % 400 == 0, c5: y % 4 == 0, c6: y % 100 != 0;
- p4: c7; c7: m <= 7;
- p5: c8; c8: m % 2 == 1;
- p6: c9; c9: m % 2 == 0

Exercise 2

- Determine predicates and clauses in the source code
- Identify reachable predicates of the source code
- Identify test cases that satisfy PC, CC and CoC
- Are there infeasible requirements?

```
public static int daysInMonth(int m, int y) {  
    if (m <= 0 || m > 12) // p1  
        throw new IllegalArgumentException("Invalid month: " + m);  
    if (m == 2) { // p2  
        if (y % 400 == 0 || (y % 4 == 0 && y % 100 != 0)) // p3  
            return 29;  
        else  
            return 28;  
    }  
    if (m <= 7) { // p4  
        if (m % 2 == 1) // p5  
            return 31;  
        return 30;  
    }  
    if (m % 2 == 0) // p6  
        return 31;  
    return 30;  
}
```

○ Reachability predicates - $r(p)$

- $r(p1) = \text{true}$ (always reached)
- $r(p2) = \neg p1 = m \geq 0 \wedge m < 12$
- $r(p3) = r(p2) \wedge p2 = (m > 0 \wedge m < 12 \wedge m == 2) = (m == 2)$
- $r(p4) = r(p2) \wedge \neg p2$
- $r(p5) = r(p4) \wedge p4 = (m > 0 \wedge m < 12 \wedge m != 2) \wedge m \leq 7$
- $r(p6) = r(p4) \wedge \neg p4 = (m > 0 \wedge m < 12 \wedge m != 2) \wedge m > 7$

PC Test Cases

○ Predicates and clauses

- **p1:** `c1 || c2 ; c1: m <= 0, c2: m > 12`
- **p2:** `c3; c3: m == 2;`
- **p3:** `c4 || (c5 && c6) ; c4: y % 400 == 0, c5: y % 4 == 0, c6: y % 100 != 0;`
- **p4:** `c7; c7: m <= 7;`
- **p5:** `c8; c8: m % 2 == 1;`
- **p6:** `c9; c9: m % 2 == 0`

#	PC	Test Input	
		m	y
1	p1	0	-
2	\neg p1	1	-
3	p2	2	-
4	\neg p2	1	-
5	p3	2	2000
6	\neg p3	2	2017
7	p4	3	-
8	\neg p4	8	-
9	p5	3	-
10	\neg p5	4	-
11	p6	8	-
12	\neg p6	9	-

CC Test Cases

O Predicates and clauses

- **p1:** $c1 \vee c2$; **c1:** $m \leq 0$, **c2:** $m > 12$
- **p2:** $c3$; **c3:** $m == 2$;
- **p3:** $c4 \vee (c5 \wedge c6)$; **c4:** $y \% 400 == 0$,
c5: $y \% 4 == 0$, **c6:** $y \% 100 != 0$;
- **p4:** $c7$; **c7:** $m \leq 7$;
- **p5:** $c8$; **c8:** $m \% 2 == 1$;
- **p6:** $c9$; **c9:** $m \% 2 == 0$

#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
CC	c1	$\neg c1$	c2	$\neg c2$	c3	$\neg c3$	c4	$\neg c4$	c5	$\neg c5$	c6	$\neg c6$	c7	$\neg c7$	c8	$\neg c8$	c9	$\neg c9$

m	0	1	13	1	2	1	2	2	2	2	2	2	3	8	3	4	8	9
y	-	-	-	-	-	-	2000	2017	2000	2017	2017	2000	-	-	-	-	-	-

CoC Test Case

#	c1	c2	c3	c4	c5	c6	c7	c8	c9
1	T	T	infeasible						
2	T	F	m=0						
3	F	T	m=13						
4	F	F	m=1						
5			T	m=2					
6			F	m=1					
7					m=3		T		
8					m=8		F		
9						m=3		T	
10						m=4		F	
11							m=8		T
12							m=9		F

CoC Test Case

#	c1	c2	c3	c4	c5	c6	c7	c8	c9
13	infeasible			T	T	T			
14	m=2, y=2000			T	T	F			
15	infeasible			T	F	T			
16	infeasible			T	F	F			
17	m=2, y=2016			F	T	T			
18	m=2, y=2100			F	T	F			
19	m=2, y=2017			F	F	T			
20	infeasible			F	F	F			

Exercise 3

```
public static TClass triangleType(int a, int b, int c) {  
    if (a <= 0 || b <= 0 || c <= 0) /* p1 */  
        return INVALID;  
    if (a >= b + c || b >= a + c || c >= a + b) /* p2 */  
        return INVALID;  
    int count = 0;  
    if (a == b) /* p3 */  
        count++;  
    if (a == c) /* p4 */  
        count++;  
    if (b == c) /* p5 */  
        count++;  
    if (count == 0) /* p6 */  
        return SCALENE;  
    if (count == 1) /* p7 */  
        return ISOSCELES;  
    return EQUILATERAL;  
}
```

○ Identify:

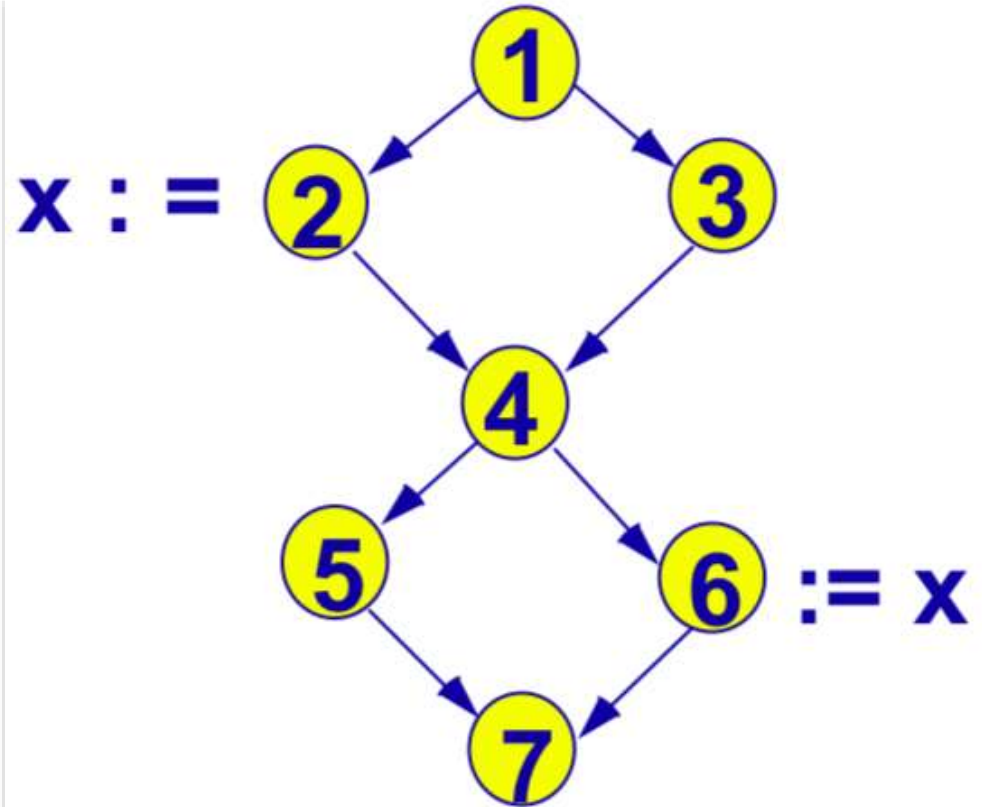
- ☼ 1) the reachability predicates;
- ☼ 2) TR(CC) and TR(PC)
- ☼ 3) test cases that satisfy a) PC b) CC
- ☼ 4) determination predicates for the clauses of p1 and p2
- ☼ 5) TR(CACC)
- ☼ 6) test cases that satisfy a) CACC b) RACC [are there infeasible requirements?]

5.4. Data Flow Testing

Kiểm thử luồng dữ liệu

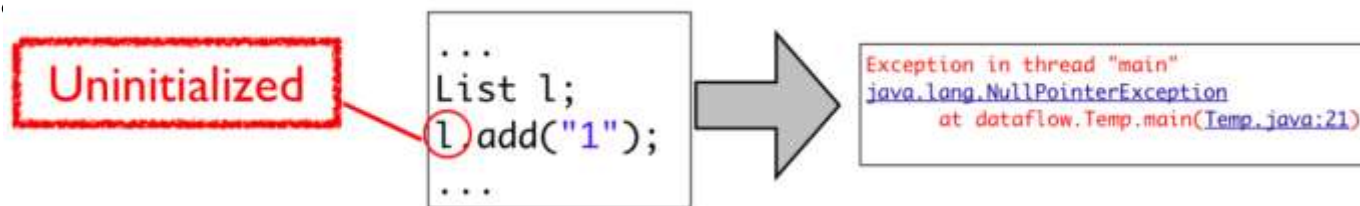
Motivation

- S2: define x, assign to x a value
- S6: use x to assign to other variable
- If Branch Coverage
 - 1-2-4-5-7
 - 1-3-4-6-7
- We cannot explore simultaneously the relationship between definition of x in statement 2 and the use of x in statement 6



Motivation

- GOAL: Try to ensure that values are computed and used correctly
- Consider: how data gets accessed and modified in the system and how it can get corrupted
- Common access-related bugs:
 - Using an undefined or uninitialized variable
 - Deallocating or reinitializing a variable before it is constructed, initialized or used



Variable Definition

Định nghĩa biến

- A program variable is **DEFINED** whenever its value is modified
 - on the left hand side of an assignment statement, e.g., **y = 20**
 - in an input statement, e.g., **read(y)**
 - as an call-by-reference parameter in a subroutine call, e.g., **update(x, &y)**

Variable Use

Sử dụng biến

- A program variable is **USED** whenever its value is read:
 - On the right hand side of an assignment statement, e.g., $y = x + 17$, x is used, y is defined
 - as an call-by-value parameter in a subroutine or function call, e.g., $y = \text{sqrt}(x)$
 - in the predicate of a branch statement, e.g., $\text{if } (x > 0)\{\dots\}$

Variable use: p-use and c-use

- Use in the predicate of a branch statement is a **predicate-use** or **p-use**
- Any other use is a **computation-use** or **c-use**
- For example, in the code below, there is a p-use of x and a c-use of y

```
if (x > 0) {  
    print(y);  
}
```

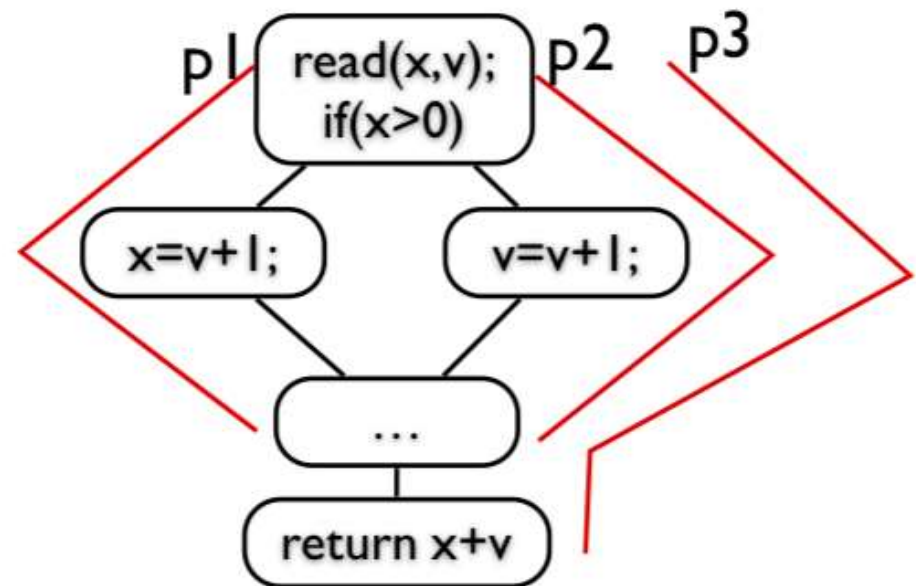
Variable Use

- A variable can also be used and then re-defined in a single statement when it appears
 - on both sides of an assignment statement, e.g., $y = y + x$
 - as an call-by-reference parameter in a subroutine call, e.g., **increment(&y)**

Terminology

Các thuật ngữ khác

- *Definition-Clear-Path*
 - A path is definition-clear ("**def-clear**") with respect to a variable **v** if it has no variable re-definition of **v** on this path
 - p1 is def-clear path of v while p2 is not

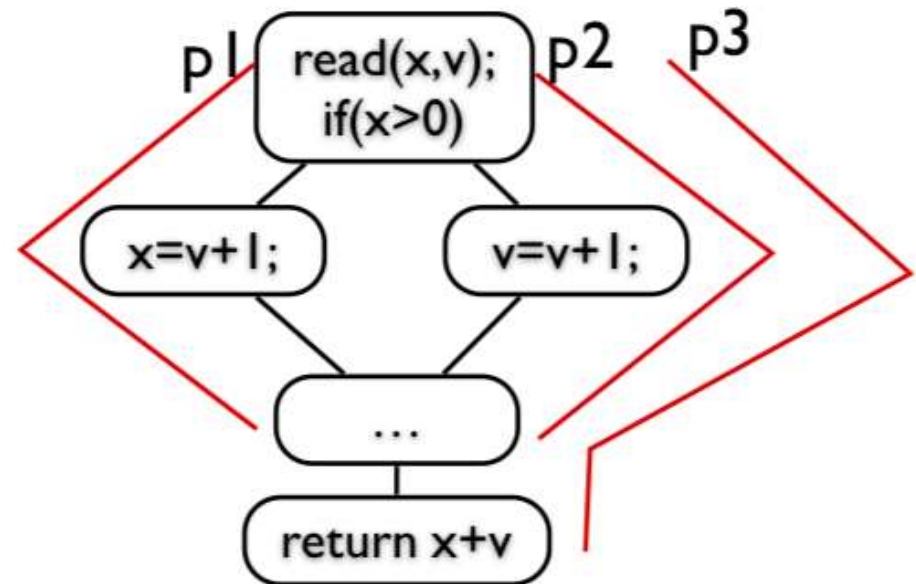


Terminology

Các thuật ngữ khác

- *Complete-Path*

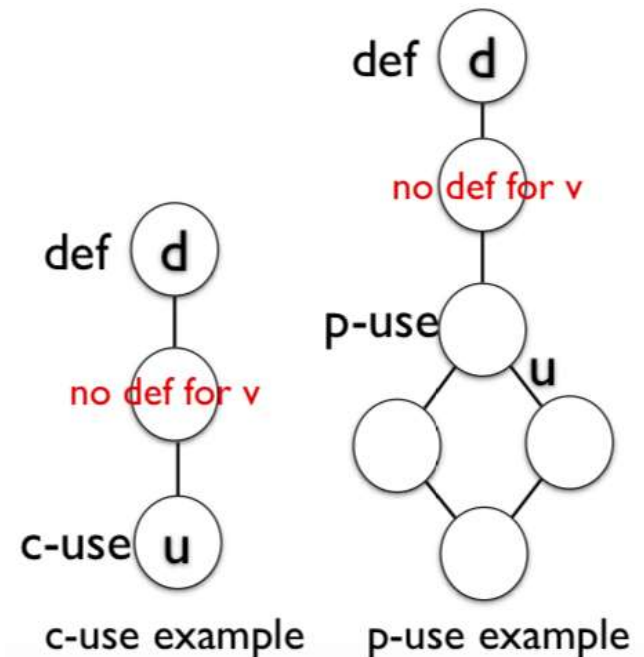
- A path is complete if the initial node of this path is a entry node and the final node of this path is an exit node
- p1, p2 are not complete while p3 is



Definition-Use Pair (DU-pair)

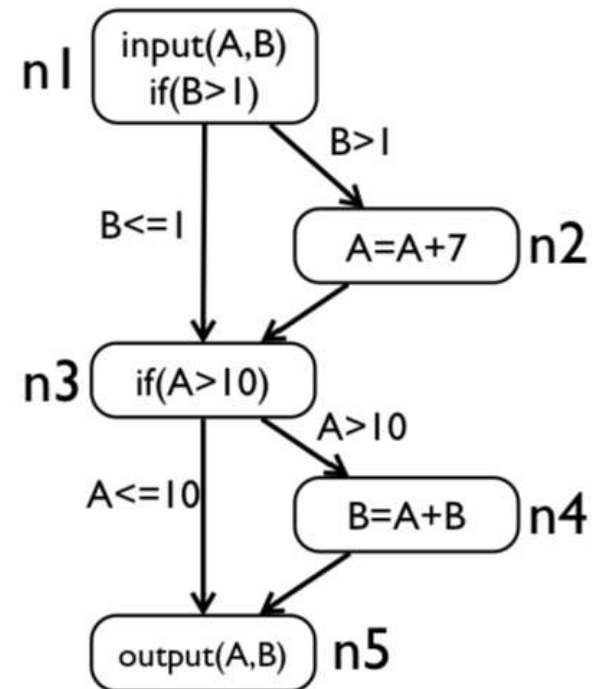
Cặp định nghĩa – sử dụng

- A *definition-use pair* (**du-pair**) with respect to a variable v is a pair (d, u) such that
 - d is a node defining v
 - u is a node or edge using v
 - When it is a ***p-use*** of v , u is an outgoing edge of the predicate statement
 - There is a ***def-clear*** path with respect to v from d to u



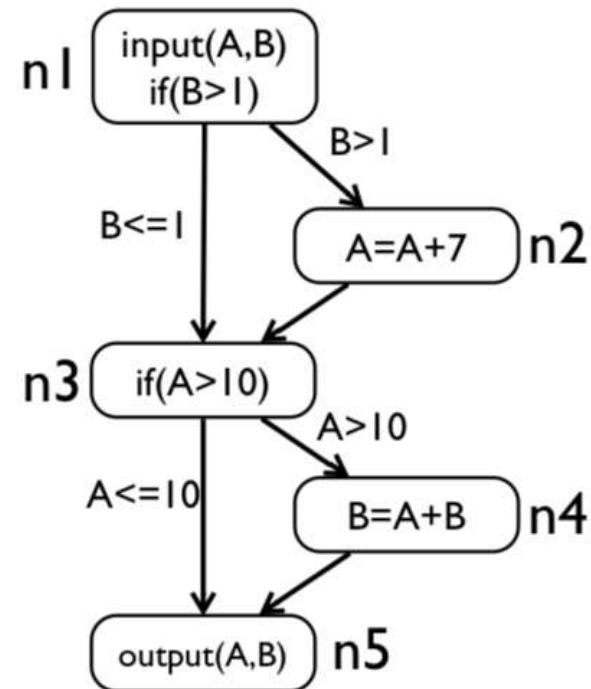
Example

```
1. input(A,B)
   if (B>1){
2.   A = A + 7
   }
3.   if (A > 10){
4.     B = A + B
   }
5. output(A,B)
```



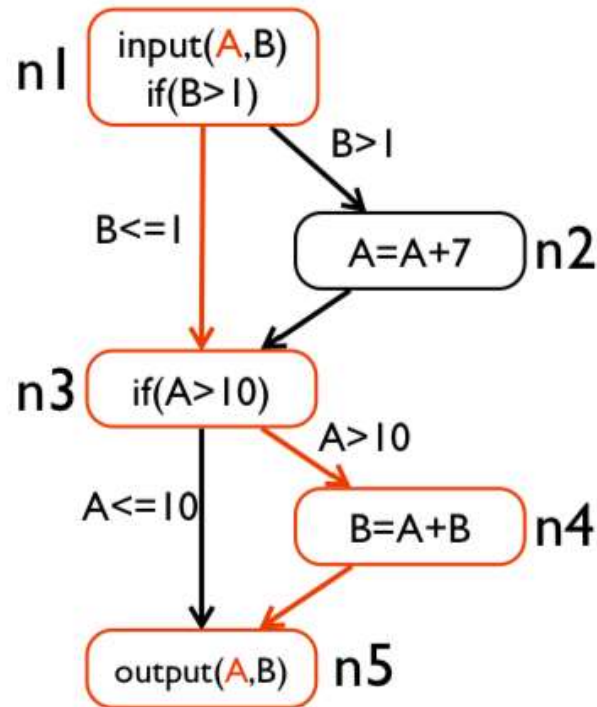
Example: Identifying DU-Pairs of Variable A

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



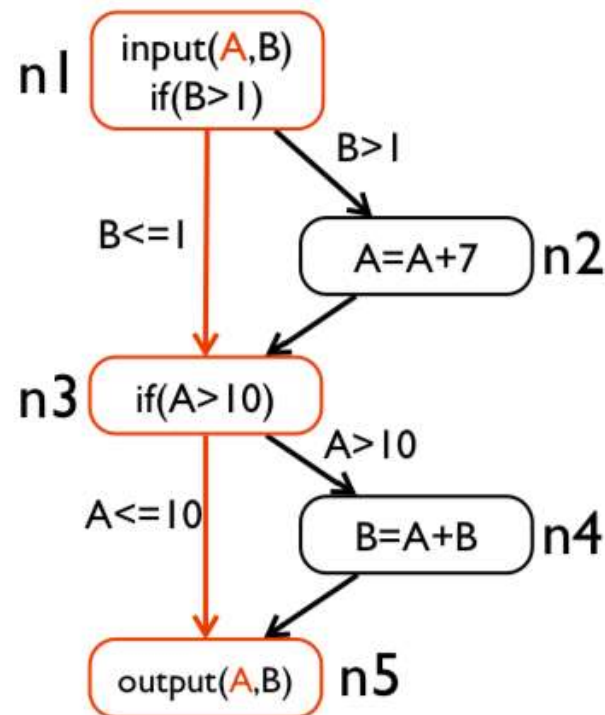
Example: Identifying DU-Pairs of Variable A

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



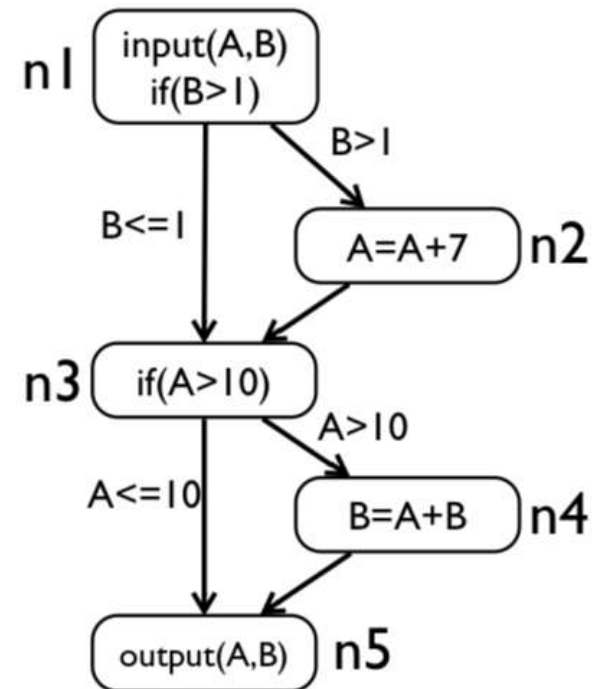
Example: Identifying DU-Pairs of Variable A

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



Example: Identifying DU-Pairs of Variable B

<u>du-pair</u>	<u>path(s)</u>
(1,4)	<1,2,3,4>
	<1,3,4>
(1,5)	<1,2,3,5>
	<1,3,5>
(1,<1,2>)	<1,2>
(1,<1,3>)	<1,3>
(4,5)	<4,5>



Data Flow Test Coverage Criteria

Các tiêu chí kiểm thử bao phủ luồng dữ liệu

- ***All-defs coverage*** – Bao phủ tất cả các điểm định nghĩa dữ liệu trên đồ thị
- ***All-uses coverage*** – Bao phủ tất cả các điểm sử dụng dữ liệu trên đồ thị
- ***All-DU-Paths coverage*** – Bao phủ tất cả các đường dẫn DU trên đồ thị
- ***All-p-uses/Some-c-uses coverage*** – Bao phủ tất cả các điểm sử dụng dữ liệu trong các câu lệnh rẽ nhánh và một vài điểm sử dụng dữ liệu
- ***All-c-uses/Some-p-uses coverage*** – Bao phủ tất cả các điểm sử dụng dữ liệu và một vài điểm sử dụng dữ liệu trong các câu lệnh rẽ nhánh
- ***All-p-uses coverage*** – Bao phủ tất cả các loại sử dụng dữ liệu trong điều kiện
- ***All-c-uses coverage*** – Bao phủ tất cả các loại sử dụng thông thường của dữ liệu

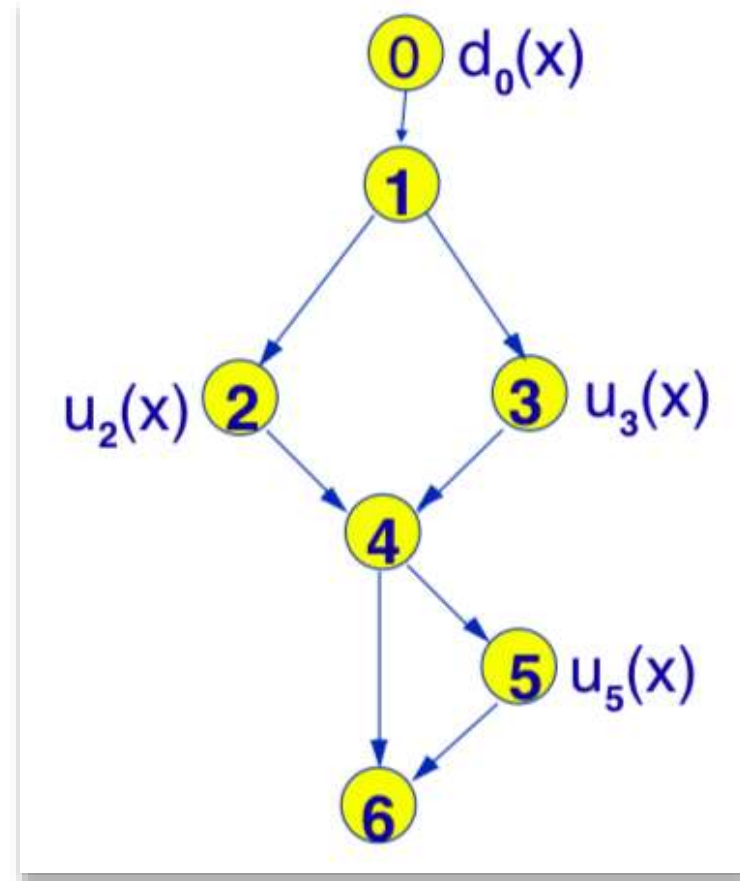
All-Defs Coverage

Bao phủ tất cả các điểm định nghĩa

- For every program variable v , at least one *def-clear path* from every definition of v to at least one *c-use* or one *p-use* of v must be covered

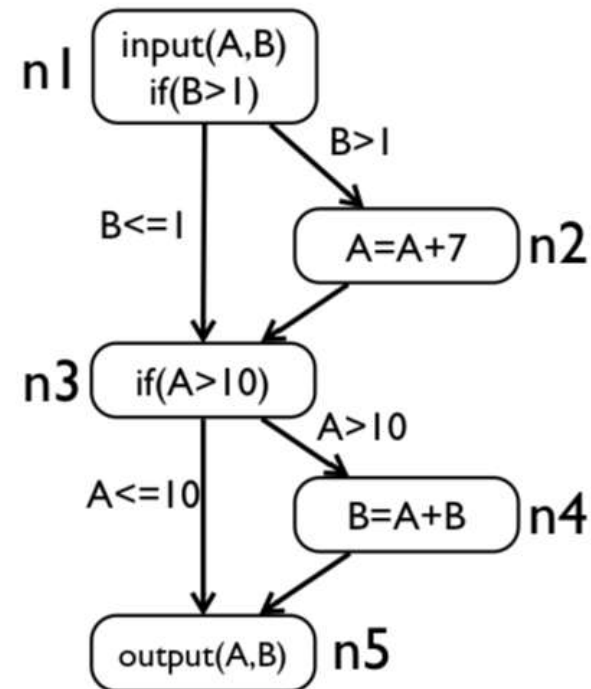
Example 1

- All Defs requires
 - $d_0(x)$ to a use
- Satisfactory Path:
 - $\langle 0-1-2-4-6 \rangle$



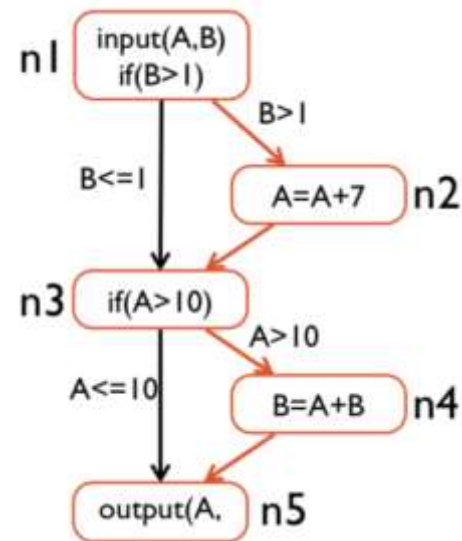
Example 2

- Identify all test cases to satisfy All-defs
- Two variables: A and B
- Consider a test case executing path
 - t1: <1,2,3,4,5>
- t1 satisfy All-defs or not?



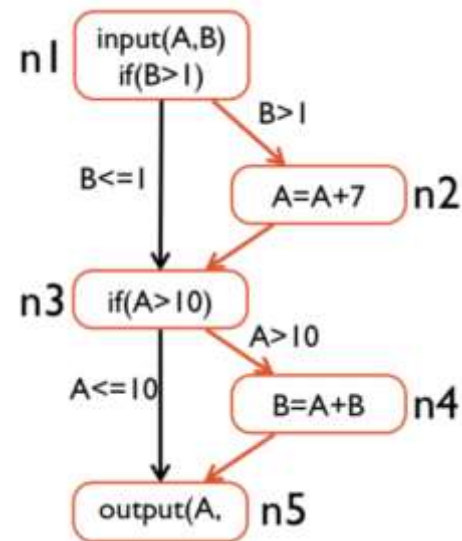
Example – Considering Variable A

du-pair	path(s)
(1,2)	<1,2> a
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4> a
(2,5)	<2,3,4,5> a
	<2,3,5>
(2,<3,4>)	<2,3,4>a
(2,<3,5>)	<2,3,5>



Example – Considering Variable B

<u>du-pair</u>	<u>path(s)</u>
(1,4)	<1,2,3,4> _a
	<1,3,4>
(1,5)	<1,2,3,5>
	<1,3,5>
(1,<1,2>)	<1,2> _a
(1,<1,3>)	<1,3>
(4,5)	<4,5> _a



Answer

- Since $\langle 1, 2, 3, 4, 5 \rangle$ covers at least one def-clear path from **every definition** of A or B to at least one c-use or p-use of A or B \rightarrow At least one du-pair from every definition of A (n_1 and n_2) and of B (n_1 , n_4) are covered!
- **All-Defs coverage is achieved**

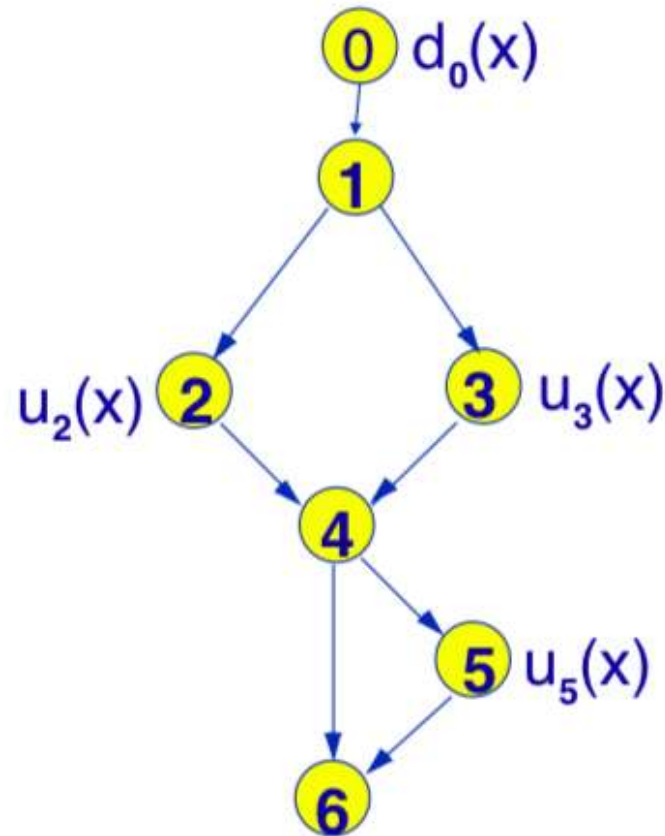
All-Uses Coverage

Bao phủ tất cả các điểm sử dụng

- For every program variable v , at least one def-clear path from every definition of v to every **c-use** and every **p-use** (including all outgoing edges of the predicate statement) of v must be covered
- Requires that all du-pairs covered

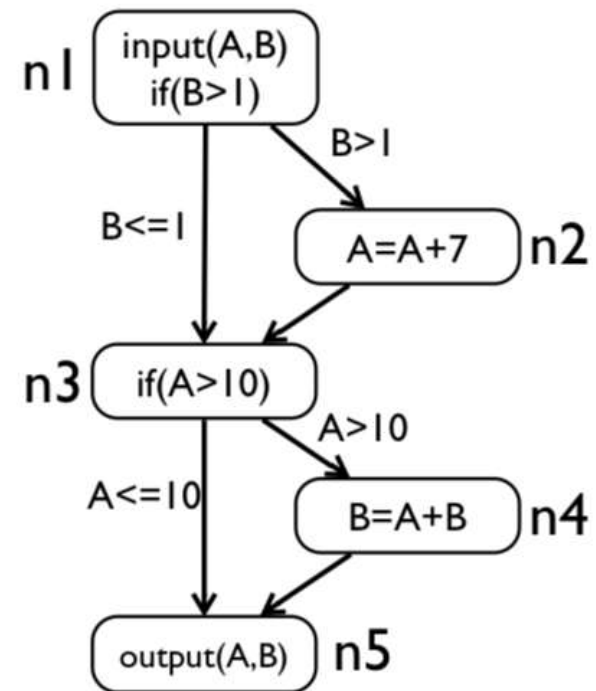
Example 1

- All-Uses requires
 - a: $d_0(x)$ to $u_2(x)$
 - b: $d_0(x)$ to $u_3(x)$
 - c: $d_0(x)$ to $u_5(x)$
- Satisfactory Path:
 - $\langle 0-1-2-4-5-6 \rangle$
satisfies a, c
 - $\langle 0-1-3-4-6 \rangle$
satisfies b



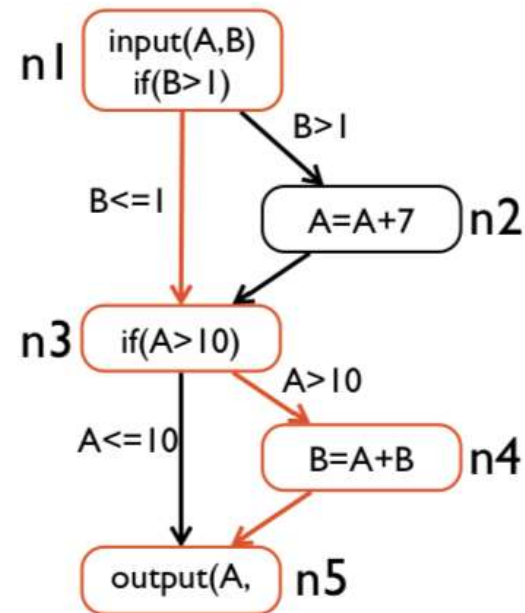
Example 2

- Consider two executing paths:
 - t_2 : $\langle 1, 3, 4, 5 \rangle$
 - t_3 : $\langle 1, 2, 3, 5 \rangle$
- Do all three test cases t_1 , t_2 , t_3 provide All-Uses coverage?



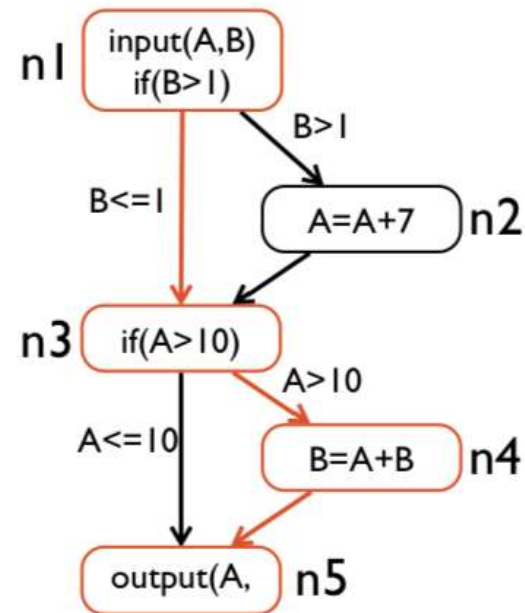
Example: Def-clear paths by t2 for Variable A

du-pair	path(s)
(1,2)	<1,2> a
(1,4)	<1,3,4> a
(1,5)	<1,3,4,5> a
	<1,3,5>
(1,<3,4>)	<1,3,4> a
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4> a
(2,5)	<2,3,4,5> a
	<2,3,5>
(2,<3,4>)	<2,3,4> a
(2,<3,5>)	<2,3,5>



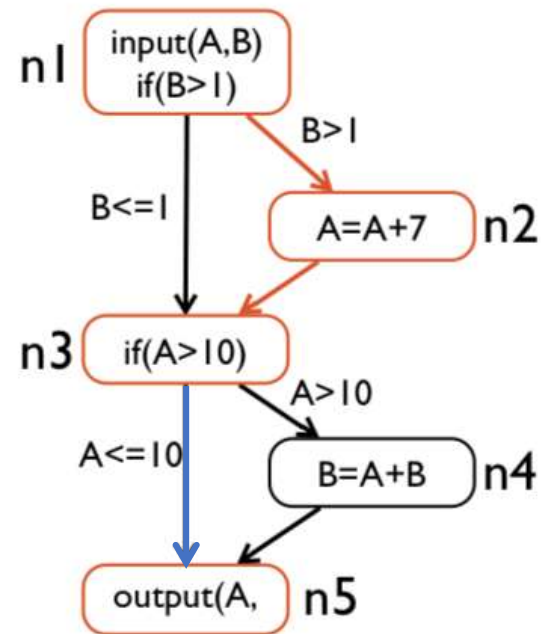
Example: Def-clear paths by t2 for Variable B

<u>du-pair</u>	<u>path(s)</u>
(1,4)	<1,2,3,4> a
	<1,3,4> a
(1,5)	<1,2,3,5>
	<1,3,5>
(1,<1,2>)	<1,2> a
(1,<1,3>)	<1,3> a
(4,5)	<4,5> a a



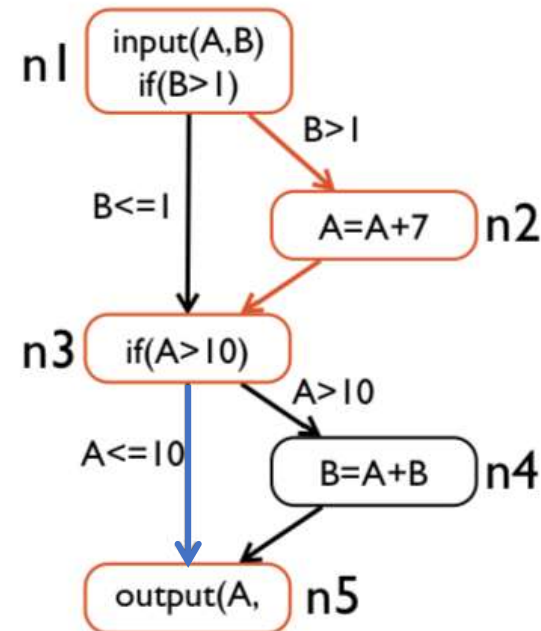
Example: Def-clear paths by t3 for Variable A

du-pair	path(s)
(1,2)	<1,2> a a
(1,4)	<1,3,4> a
(1,5)	<1,3,4,5> a
	<1,3,5>
(1,<3,4>)	<1,3,4> a
(1,<3,5>)	<1,3,5> a
(2,4)	<2,3,4> a
(2,5)	<2,3,4,5> a
	<2,3,5> a
(2,<3,4>)	<2,3,4> a
(2,<3,5>)	<2,3,5> a



Example: Def-clear paths by t3 for Variable B

<u>du-pair</u>	<u>path(s)</u>
(1,4)	<1,2,3,4>a
	<1,3,4>a
(1,5)	<1,2,3,5>a
	<1,3,5>a
(1,<1,2>)	<1,2>a a
(1,<1,3>)	<1,3>a
(4,5)	<4,5>a a

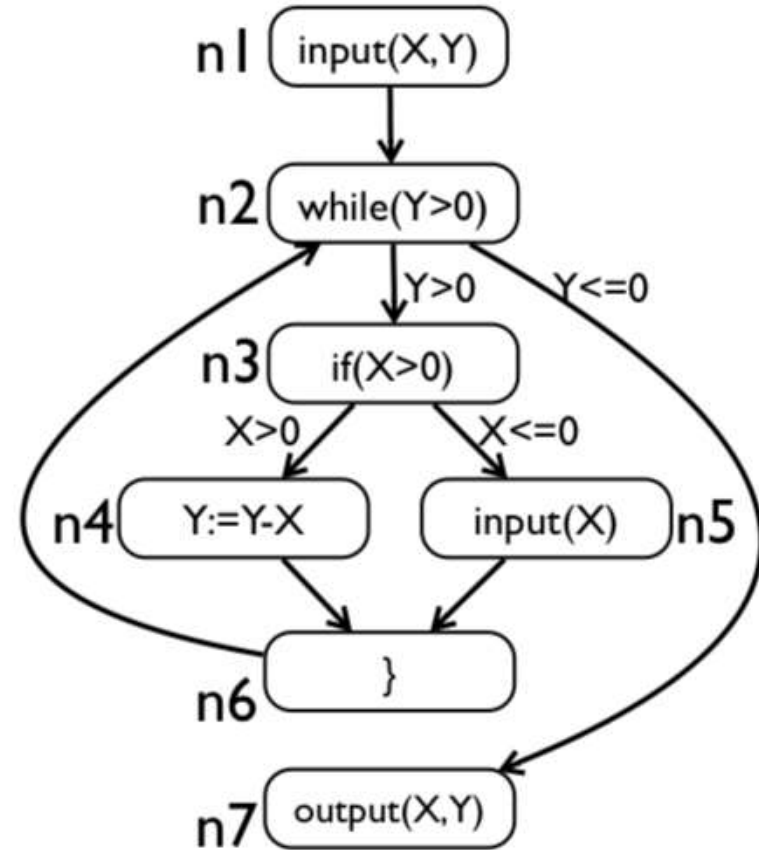


Answer

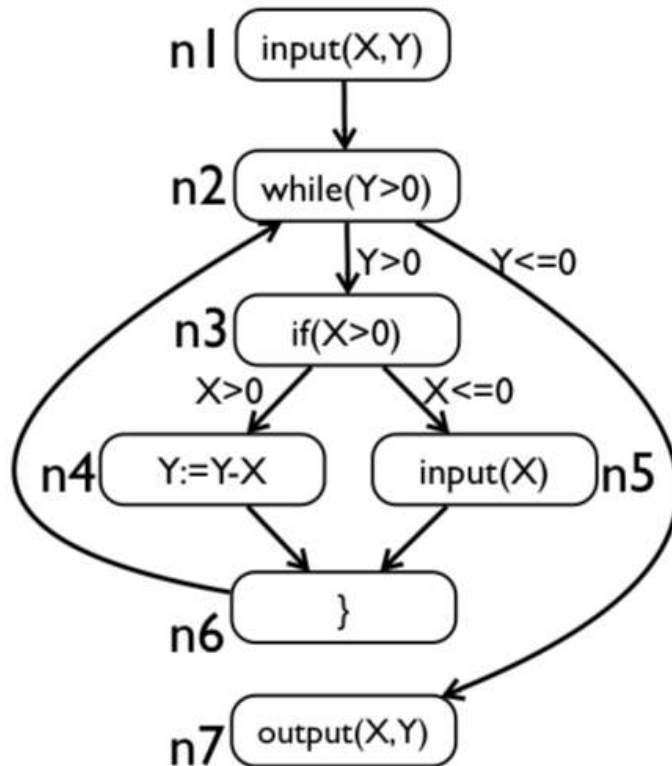
- Since all the du-pairs for variables A and B are covered by 3 test cases → All-Uses coverage is achieved
- **Note:** all du-pairs means: at least one def-clear path from every definition to every c-use/p-use of the variable

Example 3

```
1. input(X,Y)
2. while (Y>0) {
3.   if (X>0)
4.     Y := Y-X
5.   else
6.     input(X)
7. }
8. output(X,Y)
```

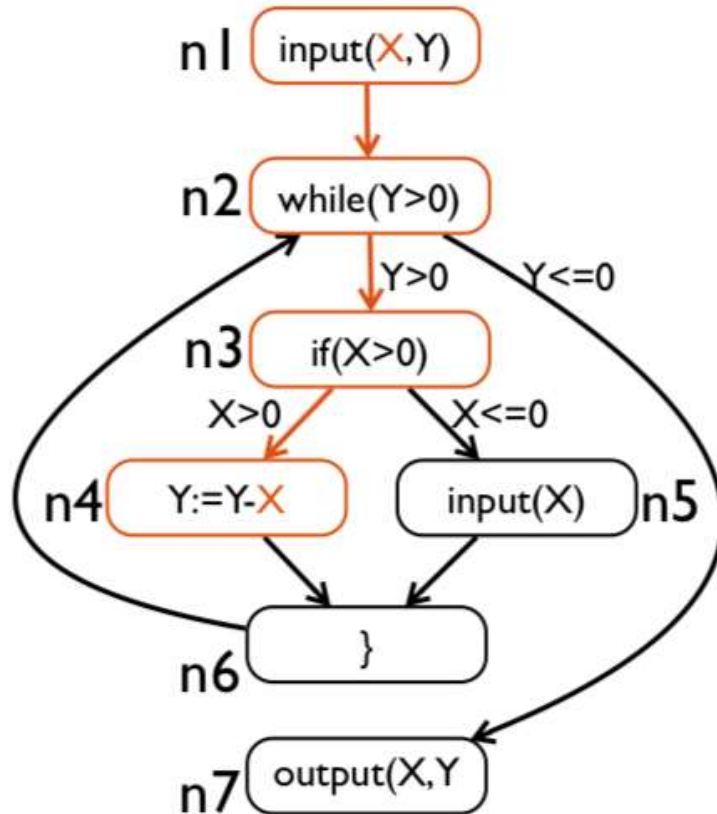


Du-Pairs with respect to Variable X



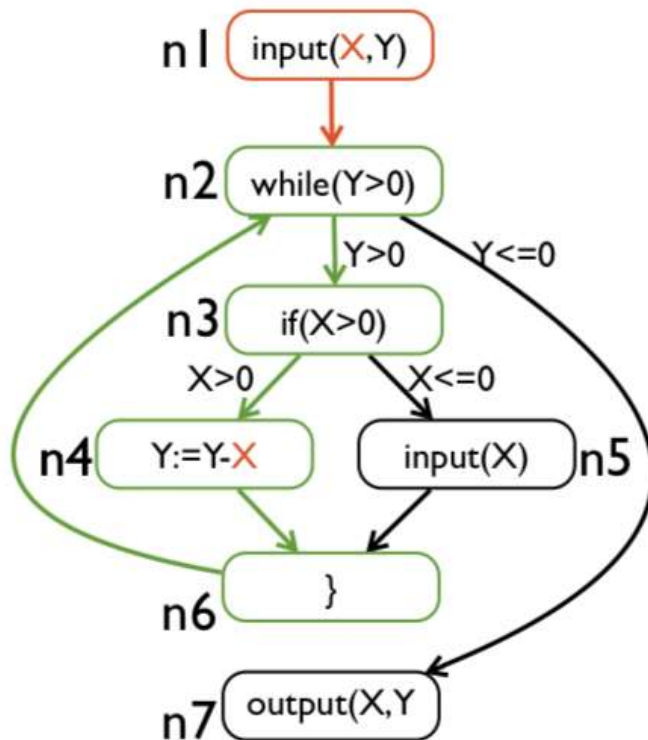
du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,7)	<1,2,7>
	<1,2,(3,4,6,2)*,7>
(1,<3,4>)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,<3,5>)	<1,2,3,5>
	<1,2,3,5,(6,2,3,5)*>
(5,4)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,7)	<5,6,2,7>
	<5,6,2,(3,4,6,2)*,7>
(5,<3,4>)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,<3,5>)	<5,6,2,3,5>
	<5,6,2,(3,4,6,2)*,3,5>

Du-Pairs with respect to Variable X



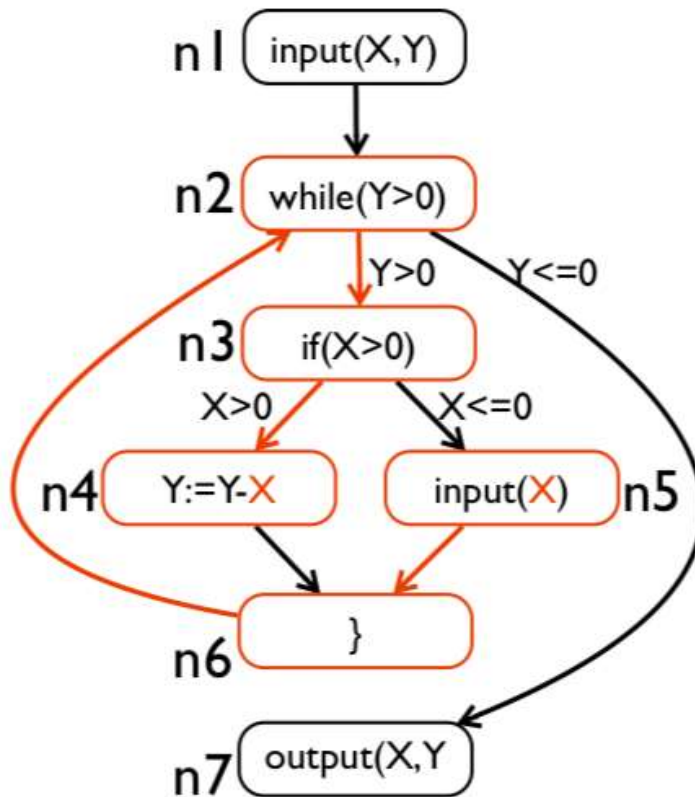
du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,7)	<1,2,7>
	<1,2,(3,4,6,2)*,7>
(1,<3,4>)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,<3,5>)	<1,2,3,5>
	<1,2,3,5,(6,2,3,5)*>
(5,4)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,7)	<5,6,2,7>
	<5,6,2,(3,4,6,2)*,7>
(5,<3,4>)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,<3,5>)	<5,6,2,3,5>
	<5,6,2,(3,4,6,2)*,3,5>

Du-Pairs with respect to Variable X



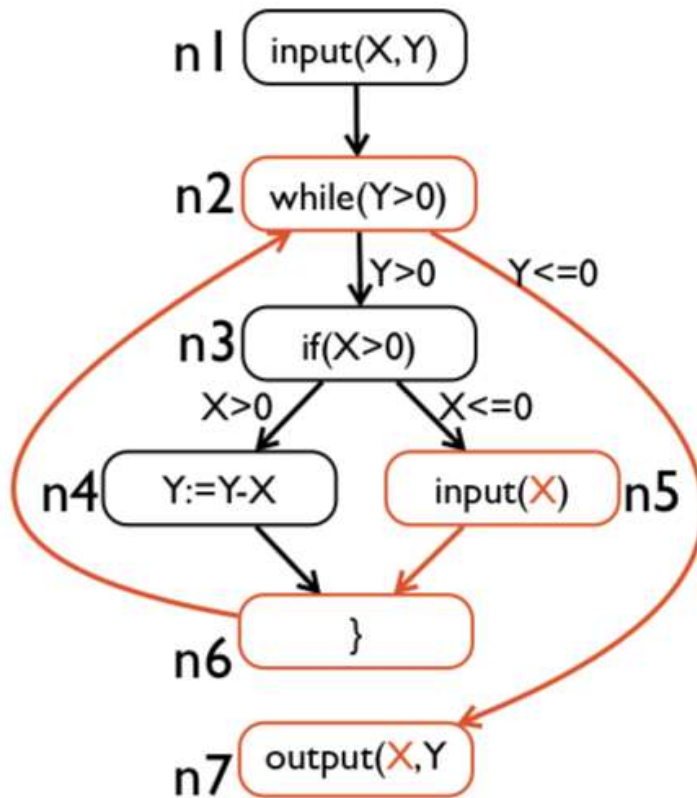
du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,7)	<1,2,7>
	<1,2,(3,4,6,2)*,7>
(1,<3,4>)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,<3,5>)	<1,2,3,5>
	<1,2,3,5,(6,2,3,5)*>
(5,4)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,7)	<5,6,2,7>
	<5,6,2,(3,4,6,2)*,7>
(5,<3,4>)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,<3,5>)	<5,6,2,3,5>
	<5,6,2,(3,4,6,2)*,3,5>

Du-Pairs with respect to Variable X



du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,7)	<1,2,7>
	<1,2,(3,4,6,2)*,7>
(1,<3,4>)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,<3,5>)	<1,2,3,5>
	<1,2,3,5,(6,2,3,5)*>
(5,4)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,7)	<5,6,2,7>
	<5,6,2,(3,4,6,2)*,7>
(5,<3,4>)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,<3,5>)	<5,6,2,3,5>
	<5,6,2,(3,4,6,2)*,3,5>

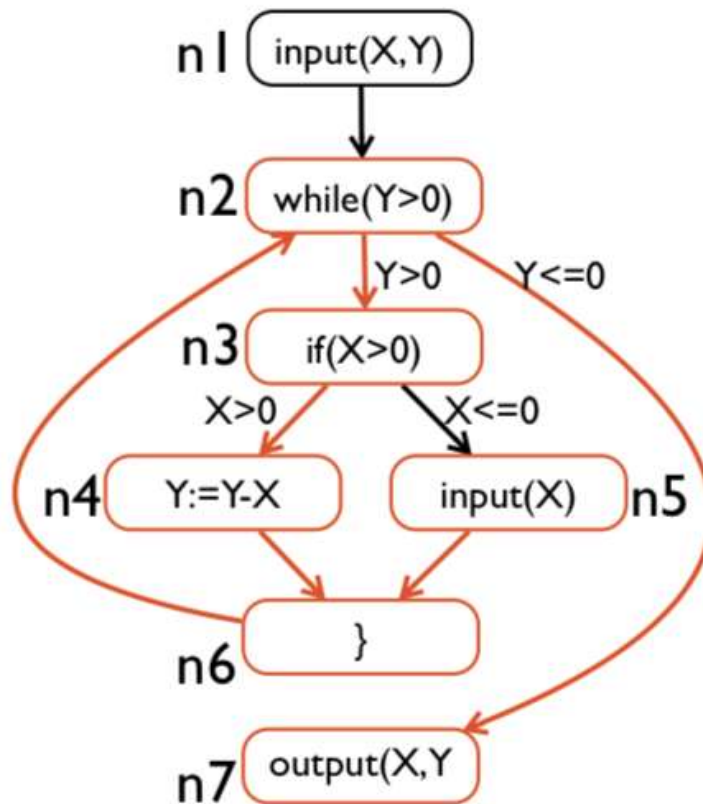
Du-Pairs with respect to Variable X



du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,7)	<1,2,7>
	<1,2,(3,4,6,2)*,7>
(1,<3,4>)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,<3,5>)	<1,2,3,5>
	<1,2,3,5,(6,2,3,5)*>
(5,4)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,7)	<5,6,2,7>
	<5,6,2,(3,4,6,2)*,7>
(5,<3,4>)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,<3,5>)	<5,6,2,3,5>
	<5,6,2,(3,4,6,2)*,3,5>

Infeasible!

Du-Pairs with respect to Variable X



du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,7)	<1,2,7>
	<1,2,(3,4,6,2)*,7>
(1,<3,4>)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,<3,5>)	<1,2,3,5>
	<1,2,3,5,(6,2,3,5)*>
(5,4)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,7)	<5,6,2,7>
	<5,6,2,(3,4,6,2)*,7>
(5,<3,4>)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,<3,5>)	<5,6,2,3,5>
	<5,6,2,(3,4,6,2)*,3,5>

More Dataflow Terms and Definitions

- A path (either partial or complete) is **simple** if all edges within the path are distinct, i.e., different
- A path is **loop-free** if all nodes within the path are distinct, i.e., different

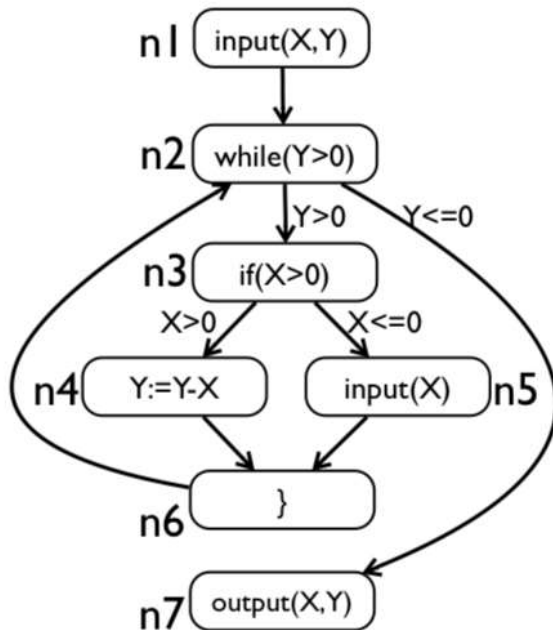
Example: Simple and Loop-Free Paths

path	Simple?	Loop-free?
<1,3,4,2>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<1,2,3,2>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<1,2,3,1,2>	<input type="checkbox"/>	<input type="checkbox"/>
<1,2,3,2,4>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

DU-Path

- A path $\langle n_1, n_2, \dots, n_j, n_k \rangle$ is a du-path with respect to a variable v , if v is defined at node n_1 and either:
 - there is a c-use of v at node n_k and $\langle n_1, n_2, \dots, n_j, n_k \rangle$ is a def-clear **simple** path, or
 - there is a p-use of v at edge $\langle n_j, n_k \rangle$ and $\langle n_1, n_2, \dots, n_j \rangle$ is a def-clear **loop-free** path

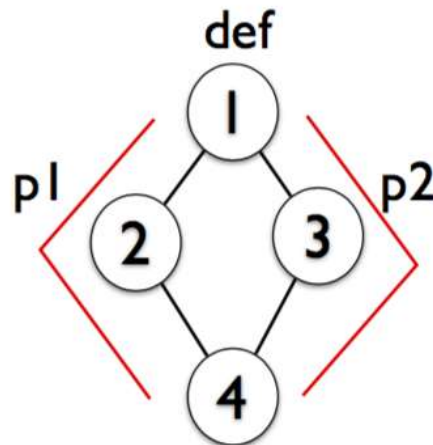
Example 3: Identify du-path



du-pair	path(s)	du-path?
(5,4)	<5,6,2,3,4>	<input type="checkbox"/>
	<5,6,2,3,4,(6,2,3,4)*>	<input type="checkbox"/>
(5,7)	<5,6,2,7>	<input type="checkbox"/>
	<5,6,2,(3,4,6,2)*,7>	<input type="checkbox"/>
(5,<3,4>)	<5,6,2,3,4>	<input type="checkbox"/>
	<5,6,2,3,4,(6,2,3,4)*>	<input type="checkbox"/>
(5,<3,5>)	<5,6,2,3,5>	<input type="checkbox"/>
	<5,6,2,(3,4,6,2)*,3,5>	<input type="checkbox"/>

All DU-Paths Coverage

For every program variable v , every **du-path** from every definition of v to every **c-use** and every **p-use** of v must be covered



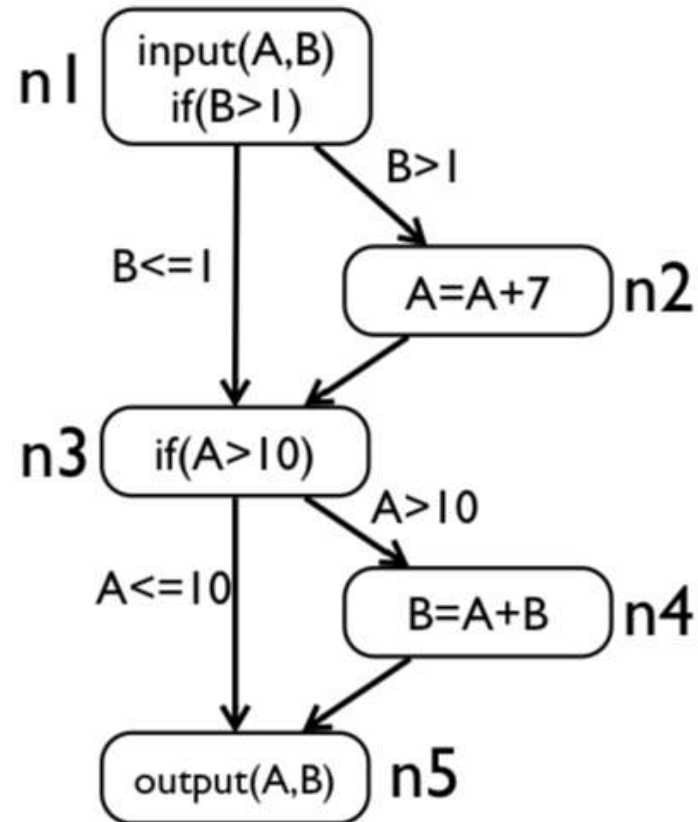
node 1 is the only def node, and 4 is the only use node for v

p1 satisfies all-defs and all-uses, but not all-du-paths

p1 and p2 together satisfy all-du-paths

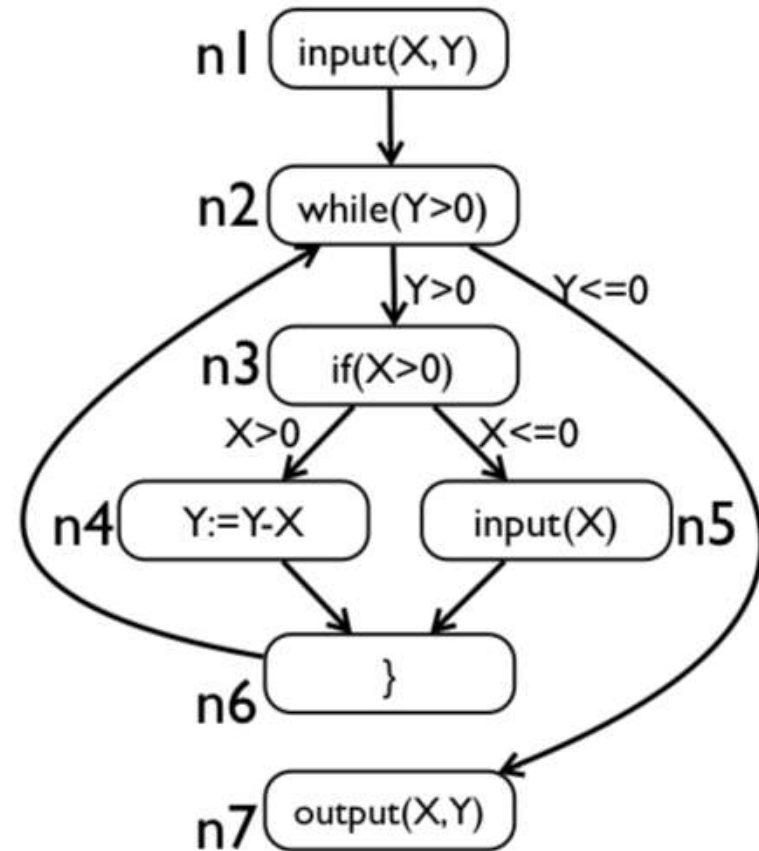
Example 2: All DU-paths coverage

- Identify all DU-Paths for variable A and B



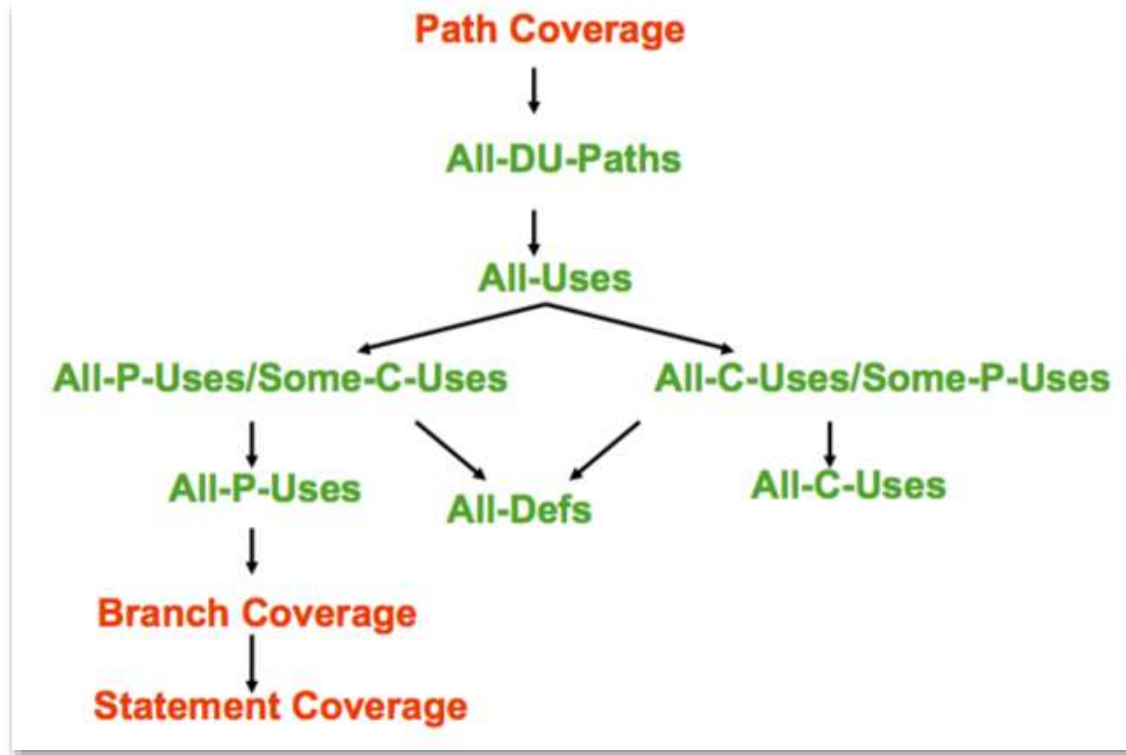
Example 3: All DU-paths coverage

- Identify all DU-Paths for variable X and Y



Data Flow Testing Summary

- Relationship between Data Flow Testing and Path Testing
- Path testing with **Branch Coverage** and Data-flow testing with **All-uses** is a very good combination in practice



Exercise 1

- Draw a data flow graph for the *binsearch* function
- Find a set of complete paths satisfying all-defs criterion with respect to variable *mid*
- Do the same for variable *high*

```
1 public class BinarySearch {
2     public int binsearch(int x, int[] V, int n) {
3         int low, high, mid;
4         low = 0;
5         high = n - 1;
6         while (low <= high) {
7             mid = (low + high) / 2;
8             if (x < V[mid])
9                 high = mid - 1;
10            else if (x > V[mid])
11                low = mid + 1;
12            else
13                return mid;
14        }
15        return -1;
16    }
17 }
```

Exercise 2

- Using **All-DU paths** to test the **pow** function
- Using **All-Uses** combining with Branch Coverage to test this function

```
1 public class Pow {  
2     public void pow(int x, int y) {  
3         float z;  
4         int p;  
5         if (y < 0)  
6             p = 0 - y;  
7         else p = y;  
8         z = 1.0f;  
9         while (p!=0)  
10        {  
11            z = z * x;  
12            p = p - 1;  
13        }  
14        if (y < 0)  
15            z = 1.0f/z;  
16        System.out.println(z);  
17    }  
18 }
```

5.5. Mutation Testing

Kiểm thử đột biến

Mutation Testing

- Mutation testing (Kiểm thử đột biến) is a structural testing method
 - i.e., We have to use the structure of code to guide the test process
- We cover the following aspects of Mutation Testing
 - What is a mutation?
 - What is mutation testing?
 - When should we use mutation testing?
 - Mutations
 - Mutation Testing tools

What is a mutation?

- A mutation is a small change in the program
- Such small changes are intended to model low level defects that arise in the process of coding systems
- Ideally, mutations should model low-level defect creation

When should we use mutation testing?

- Structural test suites are directed at identifying defects in the code
- One goal of mutation testing: **To assess or improve the efficacy of test suites in discovering defects**
- Defects remaining in the code is measured through Residual Defect Density (RDD)
- RDD is measured in defects per thousand lines of code
- Given the program under test P
 - P satisfies all the tests in a test suite T
 - Mutation testing on P provide an estimate of RDD on P

Using Mutation Testing to estimate RDD

- Given program P , test suite T
- Suppose we have r , an estimate of RDD on P – could be based on the number of faults our tests have already detected
- Generate n mutants of P
- Test each mutant with the test suite T
- Find the number, k , of mutants that are killed by T
- $RDD = r \times (n-k)/k$
- k/n is a measure of the adequacy of T in finding defects in P

An approach to Mutation

- Consider mutation operators in the form of rules that match a context and create some systematic mutation of the context to create a mutant
- Simple approach: to consider all possible mutants but this may create a very large number of mutants

Kinds of mutation

- **Value Mutation**

- Changing the values of constants or parameters (by adding or subtracting a value)
- e.g., changing loop bounds

- **Decision Mutation**

- Modifying conditions to reflect potential slips and errors in the coding of conditions in program
- e.g., replacing < by >

- **Statement Mutation**

- Deleting certain lines to reflect omissions in coding
- Swapping the order of lines of code
- Changing operations in arithmetic expressions

- ...

Mutations for Inter-class Testing

Language Feature	Operator	Description
Access Control	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	overriding method calling position change
	IOR	Overriding method rename
	ISK	<i>super</i> keyword deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism	PNC	<i>new</i> method call with child class type
	PMD	Instance variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PRV	Reference assignment with other comparable type
Overloading	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAD	Argument order change
	OAN	Argument number change
Java-Specific Features	JTD	<i>this</i> keyword deletion
	JSC	<i>static</i> modifier change
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common Programming Mistakes	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Value Mutation (đột biến giá trị)

- Typical examples:
 - Changing values to one larger or smaller
 - Swapping values in initializations
- The commonest approach is to change constants by one in an attempt to generate a one-off error
- Coverage criterion: Perturb all constants in the program or unit at least once or twice

Value Mutation - Example

```
public int Segment(int t[], int l, int u){  
    // Assumes t is in ascending order, and l<u,  
    // counts the length of the segment  
    // of t with each element l<t[i]<u  
    int k = 0;  
    for(int i=0; i<t.length && t[i]<u; i++){  
        if(t[i] > l){  
            k++;  
        }  
    }  
    return(k);  
}
```

Mutating to k=1 causes miscounting

Here we might mutate the code to read i=1, a test that would kill this would have t length 1 and have $l < t[0] < u$, then the program would fail to count t[0] and return 0 rather than 1 as a result

Decision Mutation

(Đột biến điều kiện)

- Typical examples:
 - Modelling “one-off” error by changing $<$ to \leq or vice versa (common in checking loop bounds)
 - Modelling confusion about larger and smaller by changing $<$ to $>$ or vice versa
 - Getting parenthesisation wrong in logical expressions: mistaking precedence between $\&\&$ and $\|\|$
- Coverage Criterion: consider one mutation for each condition in the program

Decision Mutation (Đột biến điều kiện)

```
public int Segment(int t[], int l, int u){  
    // Assumes t is in ascending order, and l<u,  
    // counts the length of the segment  
    // of t with each element l<t[i]<u  
    int k = 0;  
    for(int i=0; i<t.length && t[i]<u; i++){  
        if(t[i]>l){  
            k++;  
        }  
    }  
    return(k);  
}
```

Mutating to $t[i]>u$ will cause miscounting

We can model “one-off” errors in the loop bound by changing this condition to $i \leq t.length$ - provided array bounds are checked exactly this will provoke an error on every execution.

Statement Mutation (Đột biến câu lệnh)

- Typical examples:
 - Deleting a line of code
 - Duplicating a line of code
 - Permuting the order of statements
- Coverage Criterion:
 - Applying this procedure to each statement in the program

Statement Mutation (Đột biến câu lệnh)

```
public int Segment(int t[], int l, int u){  
    // Assumes t is in ascending order, and l<u,  
    // counts the length of the segment  
    // of t with each element l<t[i]<u  
    int k = 0;  
  
    for(int i=0; i<t.length && t[i]<u; i++){  
        if(t[i]>l){  
            k++;  
        }  
    }  
    return(k);  
}
```

Here we might consider deleting this statement (then count would be zero for all inputs, we might also duplicate this line in which case all counts would be doubled.

Mutation Testing Tools

- Stryker JS: mutation testing for javascript
- Mutant: automated code reviews via mutation testing
- Infection: PHP mutation testing library
- Pitest: State of the art mutation testing for the JVM
- Cosmic Ray: mutation testing for Python
- Muter: automated mutation testing for Swift
- ...



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you
for your
attention!!!

