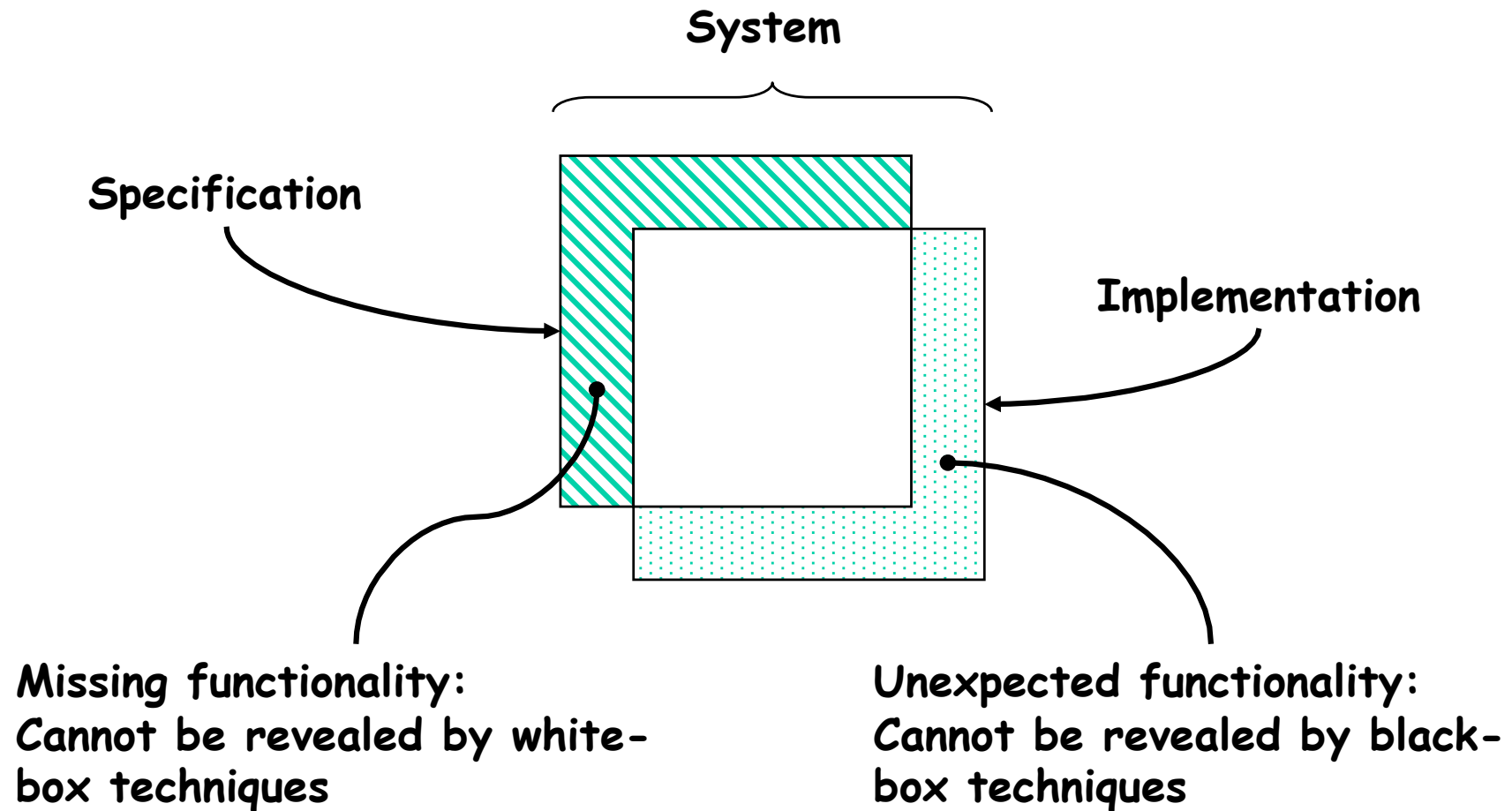


Software Verification and Validation

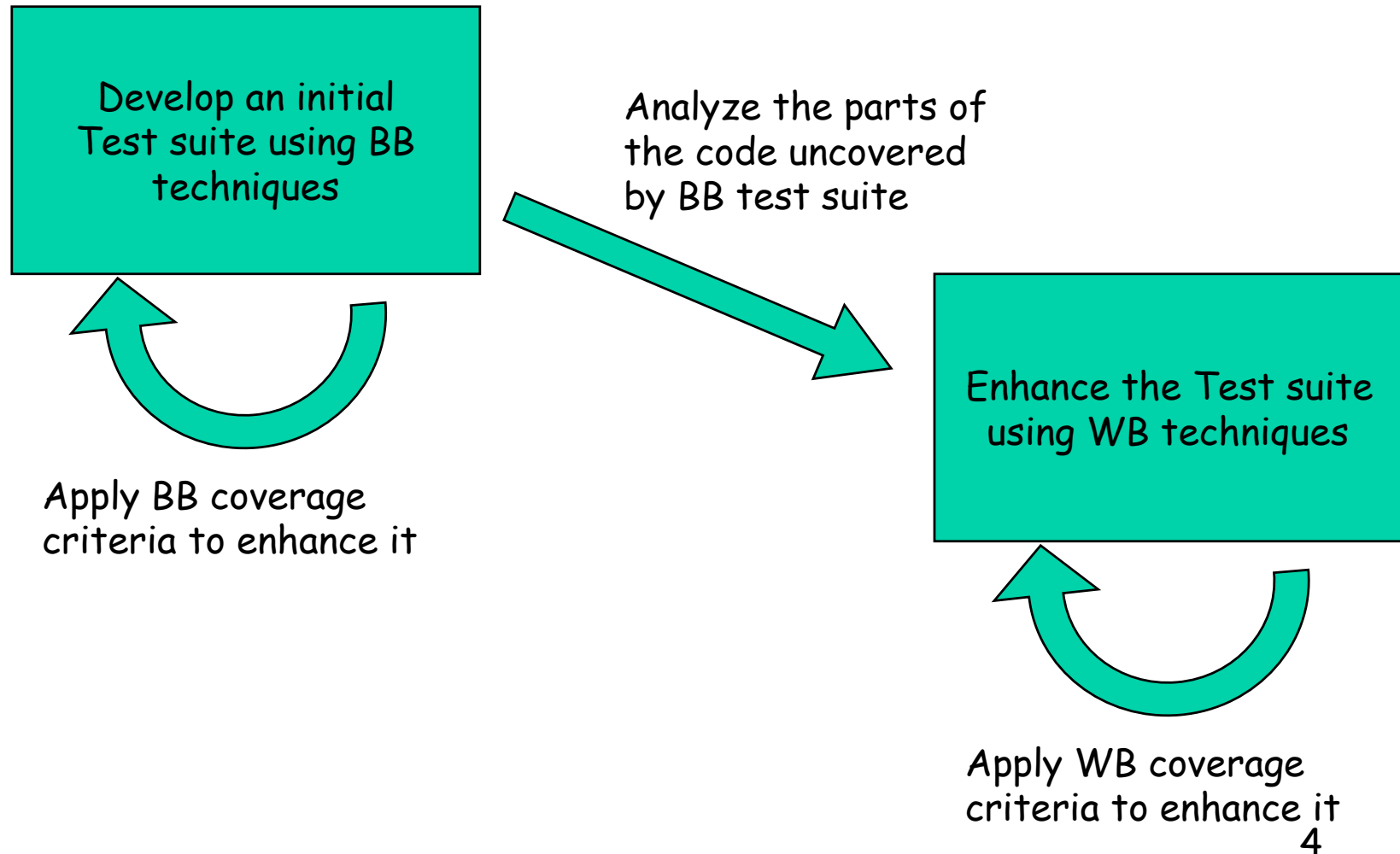
Prof. Lionel Briand
Ph.D., IEEE Fellow

White-Box Testing

Reminder: Black- vs. White (glass)-Box Testing



How do black- and white-box testing relate to one another in real-world projects?



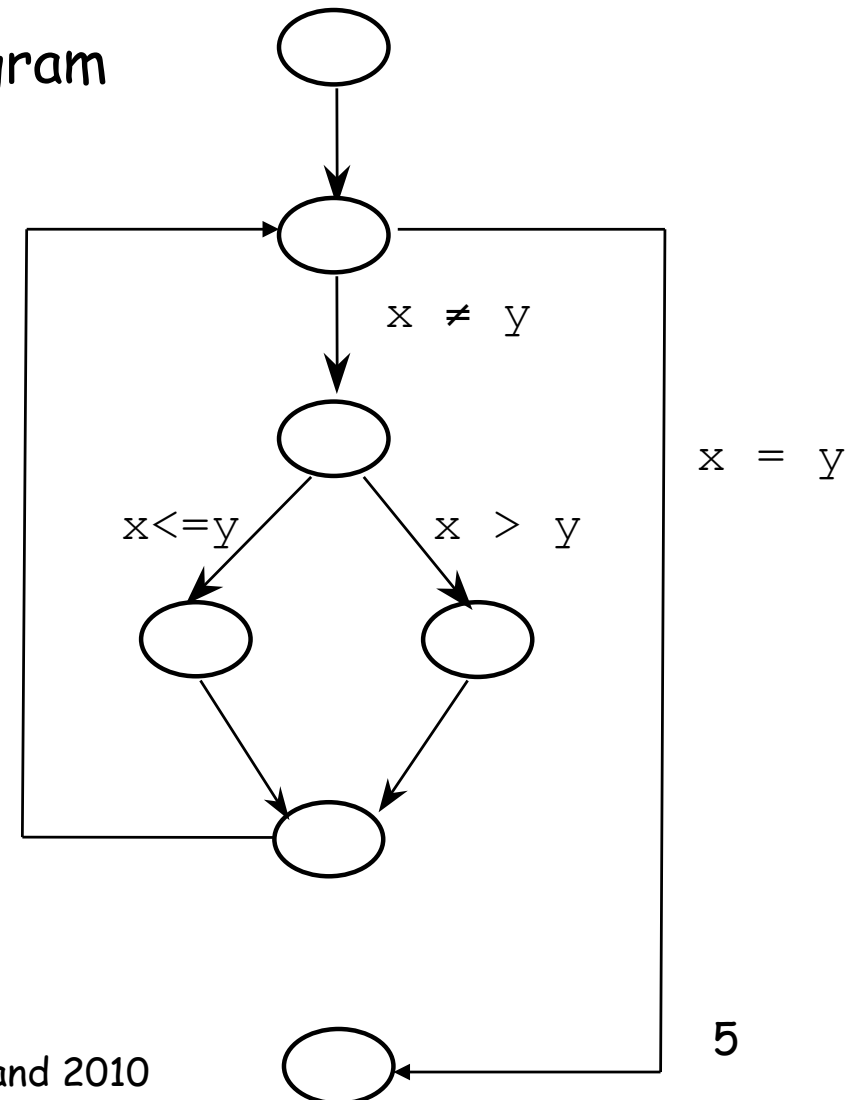
Control Flow Graph (CFG) Example

Greatest common divisor (GCD) program

```

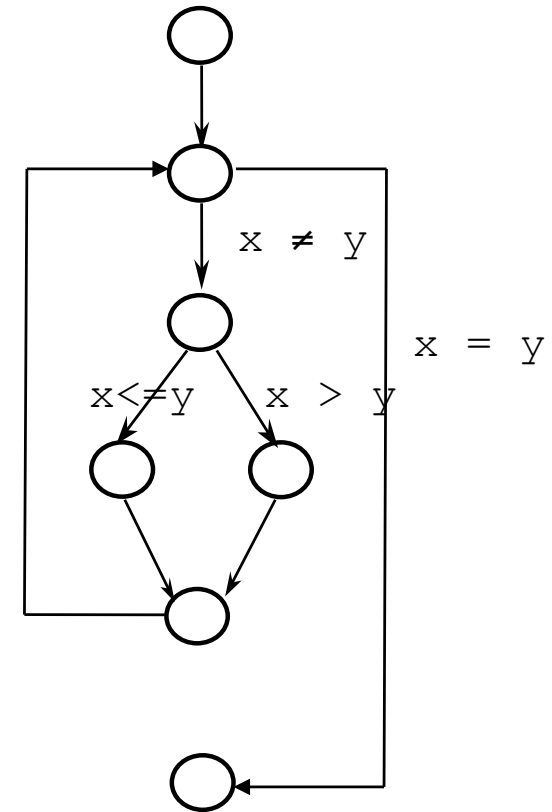
read(x);
read(y);
while x ≠ y loop
  if x > y then
    x := x - y;
  else
    y := y - x;
  end if;
end loop;
gcd := x;

```

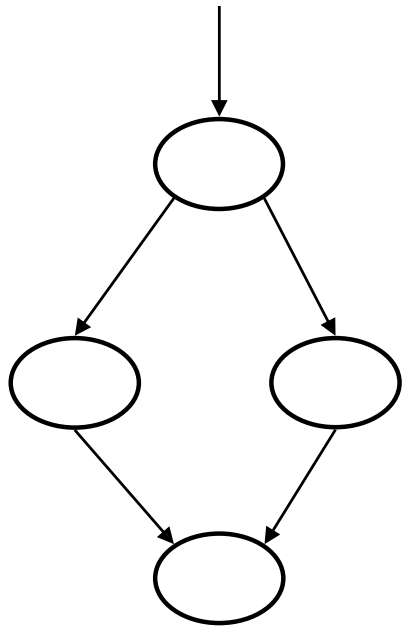


Control Flow Graphs (CFG) - Basic Definitions

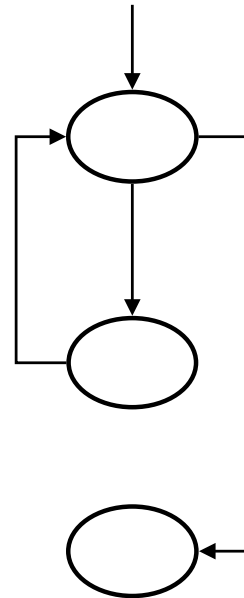
- Directed graph
- Nodes are blocks of sequential statements
- Edges are transfers of control
- Edges may be labeled with predicate representing the condition of control transfer
- There are several conventions for flow graph models with subtle differences (e.g., hierarchical CFGs, concurrent CFGs)



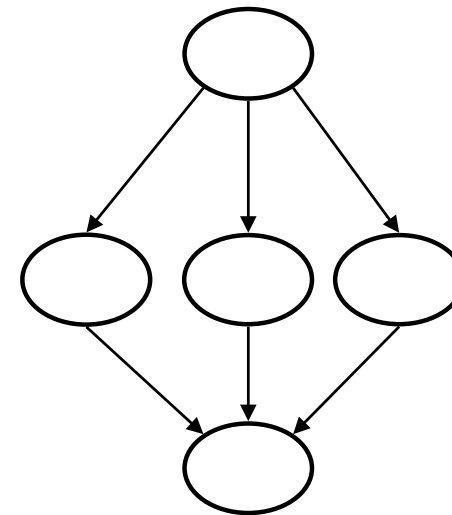
Basics of CFG: Patterns



If-Then-Else

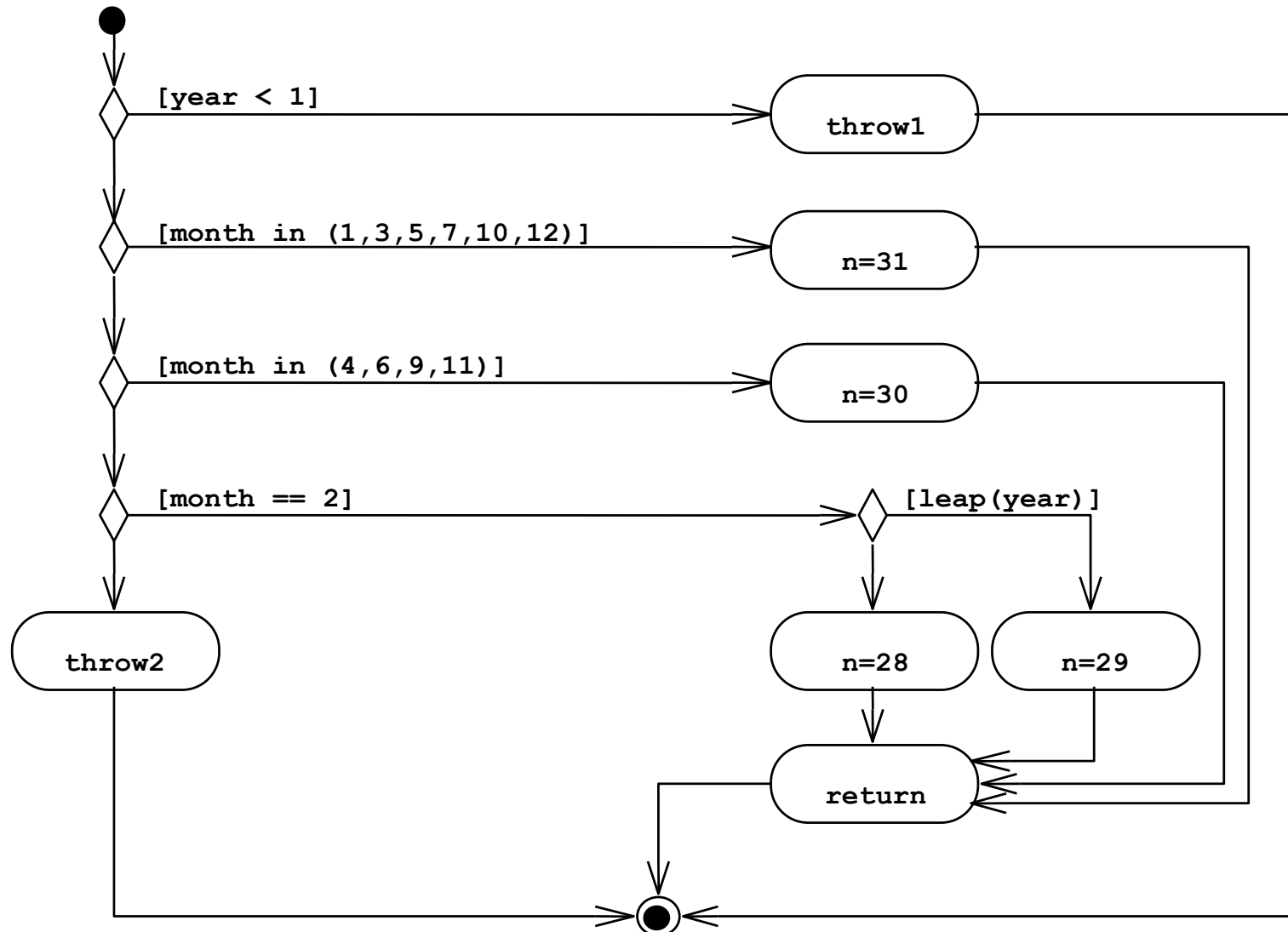


While loop



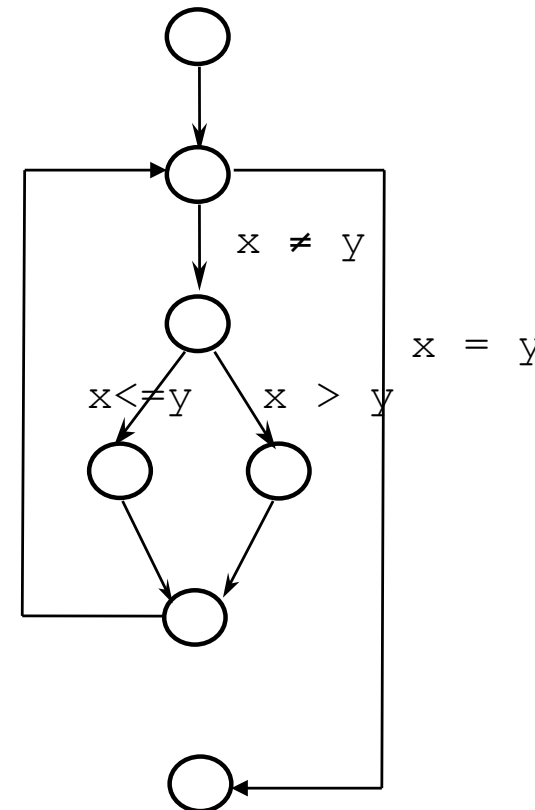
Switch

UML Activity Diagrams: Can also be used to model CFGs



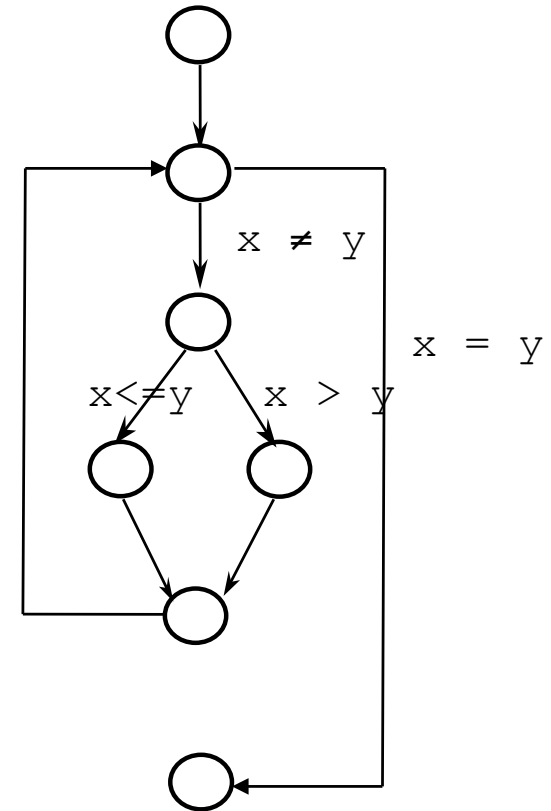
Control flow Coverage

- Depending on the (adequacy) criteria to *cover* (traverse) CFGs, we can have different types of control flow coverage:
 - Statement/Node Coverage
 - Edge Coverage
 - Condition Coverage
 - Path Coverage
- Discussed in detail next...



Statement/Node Coverage

- **Hypothesis:** Faults cannot be discovered if the statements containing them are not executed
- **Statement coverage criteria:** Equivalent to covering all nodes in CFG
- Executing a statement is a weak guarantee of correctness, but easy to achieve
- In general, several inputs execute the same statements
- An important question in practice is: how can we minimize (the number of) test cases so we can achieve a given statement coverage ratio?



Incompleteness of Statement/Node Coverage

- Though often used, statement coverage is a weak guarantee of fault detection
- An example:

```
if x < 0 then
    x := -x;
end if
z := x;
```

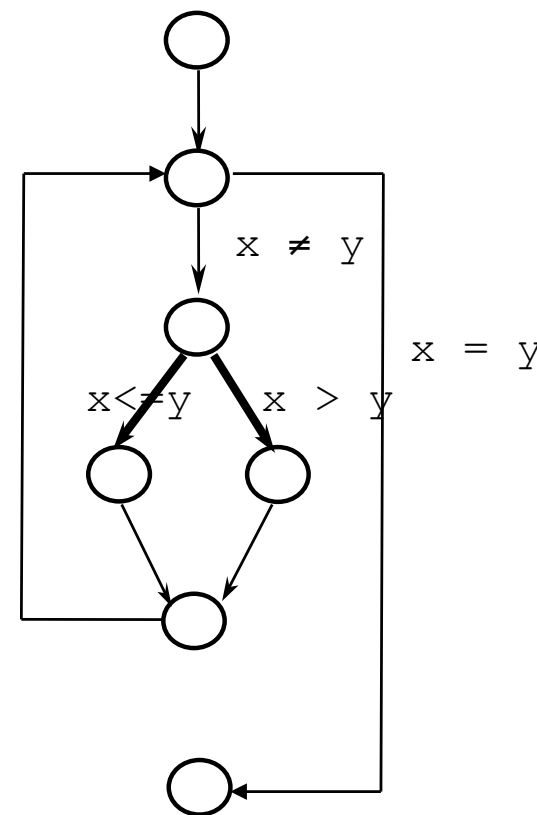
A negative x would result in the coverage of all statements.

But not exercising $x \geq 0$ is an important case.

Doing nothing for the case $x \geq 0$ may turn out to be wrong and need to be tested.

Edge (decision) Coverage

- Based on the program structure, the control flow graph (CFG)
- **Edge coverage criterion:** Select a test set T such that, by executing P for each test case t in T , each edge of P 's control flow graph is traversed at least once
- We need to exercise all conditions that traverse the control flow of the program with true and false values



Example: Searching for an element in a table

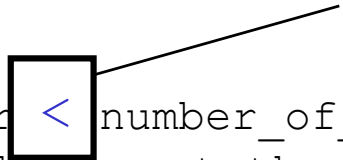
Derivation of the test set in the next slide...

```

counter:= 0;
found := false;
if number_of_items ≠ 0 then
  counter :=1;
  while (not found) and counter < number_of_items loop
    if table(counter) = desired_element then
      found := true;
    end if;
    counter := counter + 1;
  end loop;
end if;
if found then write ("the desired element exists in the table");
else write ("the desired element does not exists in the table");
end if;

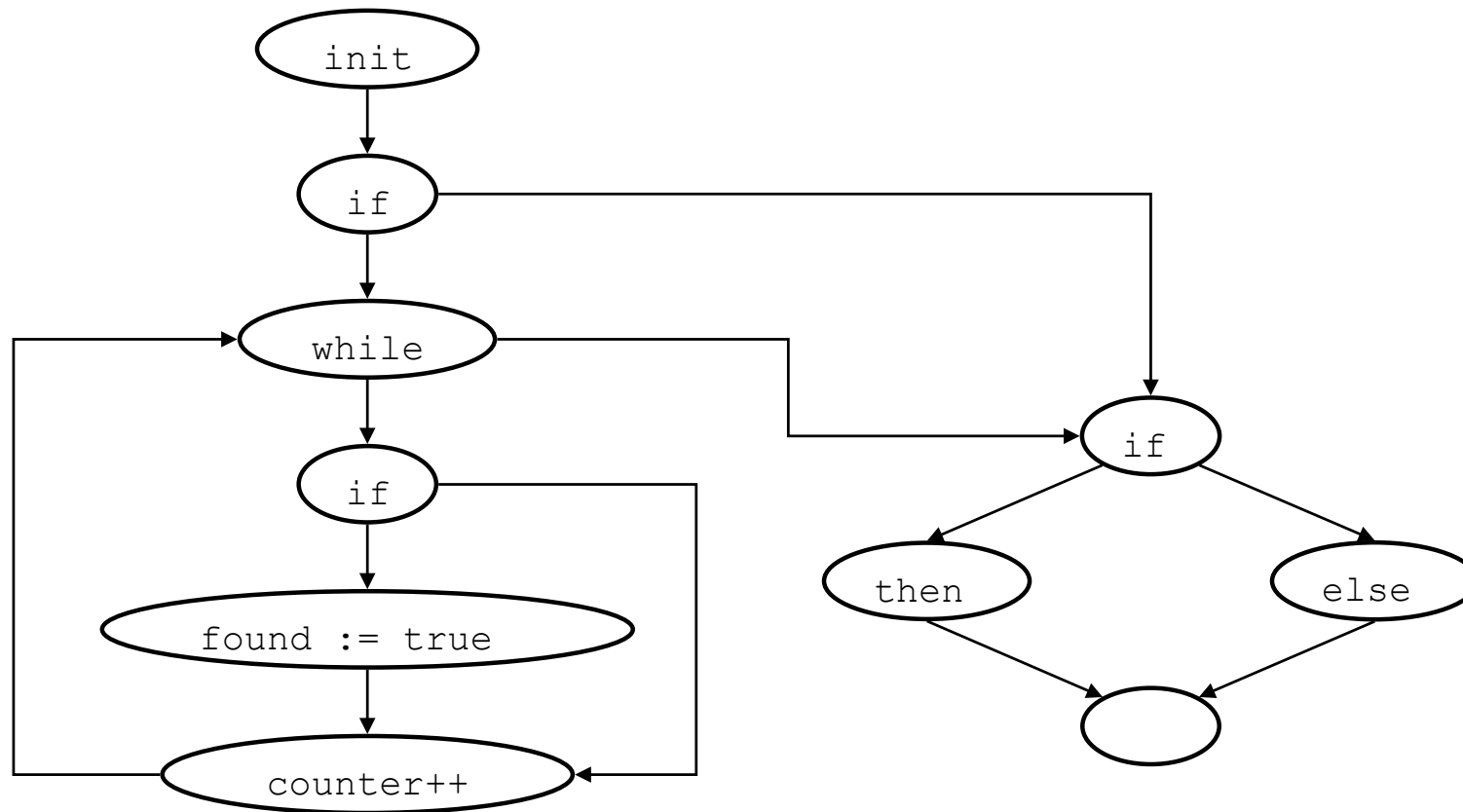
```

Should have been \leq



- Here is an example of a common mistake in a search algorithm.
- If the desired element = last element, we exit the loop before we find it.

Example's Control Flow Graph



Test Set

- We choose a test set with two test cases:
 1. One table with 0 items and,
 2. A table with 3 items, the second element being the desired one
- For the second test case, the "while" loop body is executed twice, once executing the "if-then" branch.
- The edge coverage criterion is fulfilled and the error is NOT discovered by the test set

...

```
while (not found) and counter < number_of_items loop
```

...

- The reason for the above problem?
 - Not all possible values of the *constituents* of the condition in the while loop have been exercised. `counter < number_of_items` is not evaluated to False.

Condition Coverage

- We need to further strengthen the edge coverage criterion
- **Condition Coverage (CC) Criterion:** Select a test set T such that, by executing P for each element in T , each edge of P 's control flow graph is traversed, and all possible values of the constituents of *compound conditions* (defined below) are exercised at least once
- **Compound conditions:** $C1$ and $C2$ or $C3$... where C_i 's are relational expressions or Boolean variables (atomic conditions)
- Another version: **Modified Condition-Decision Coverage (MC/DC) Criterion:** Only combinations of values such that every C_i drives the overall condition truth value twice (true and false).
- Examples next...

Condition Coverage:

Can uncover hidden edges

- Two equivalent programs: though you would write the left one

```
if c1 and c2 then
  st;
else
  sf;
end if;
```

```
if c1 then
  if c2 then
    st;
  else
    sf;
  end if;
else
  sf;
end if;
```

- Edge coverage
 - would not compulsorily cover the "hidden" edges in the right one
 - Example: **C2 = false** might not be covered
- Condition coverage would cover **C2 = false**

MC/DC Example

- **Use in industry:** The international standard DO-178B for Airborne Systems Certification (since 1992) requires testing the airborne software systems with MC/DC coverage.
- **Example :** $A \wedge (B \vee C)$, e.g., in a *while* loop, ...

	ABC	Results	Corresponding negate Case
1	TTT	T	A (5)
2	TTF	T	A (6) , B (4)
3	TFT	T	A (7) , C (4)
4	TFF	F	B (2) , C (3)
5	FTT	F	A (1)
6	FTF	F	A (2)
7	FFT	F	A (3)
8	FFF	F	-

Deriving a modified condition /decision criterion (MC/DC) Test suite:

Take a pair for each constituent:

- A: (1,5), or (2,6), or (3,7)
- B: (2,4)
- C: (3,4)

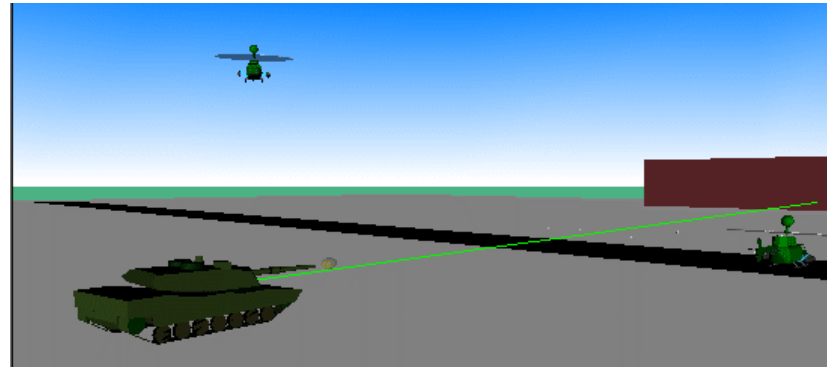
Two minimal sets to cover the MCC:

- (2,3,4,6) or (2,3,4,7)

That is 4 test cases instead of 8 for all possible combinations.

A Real Example: MC/DC coverage in Military Software

- Consider the controller SW of a modern tank with three guns



- (x, y, z) is the 3D coordinate of a hit target (e.g., from the GPS unit)
- The SW has the following requirements:
 - **R1.1:** Invoke method **Fire1** (firing with gun #1) when $(x < y)$ AND $(z * z > y)$ AND (previous direction of the gun="East").
 - **R1.2:** Invoke **Fire2** when $(x < y)$ AND $(z * z \leq y)$ OR (current direction ="South").
 - **R1.3:** Invoke **Fire3** when none of the two conditions above is true.
 - **R2:** The invocation described above must continue until an input Boolean variable becomes true (when the stop firing button is pressed).

MC/DC coverage: Example (contd.)

```

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

6    input (done);
7    current="North";
8    while ( $\neg$  done){
9      input (d);
10     prev=current; current=f(d);
11     input (x, y, z);
12     if(( $x < y$ ) and ( $z * z > y$ ) and (prev=="East"))
13       fire-1(x, y);
14     else if (( $x < y$  and ( $z * z \leq y$ ) or (current=="South"))
15       fire-2(x, y);
16     else
17       fire-3(x, y);
18     input (done);
19   }
20   output("Firing completed.");
21 end

```

← Condition C_1 .

← Condition C_2 .

← Condition C_3 .

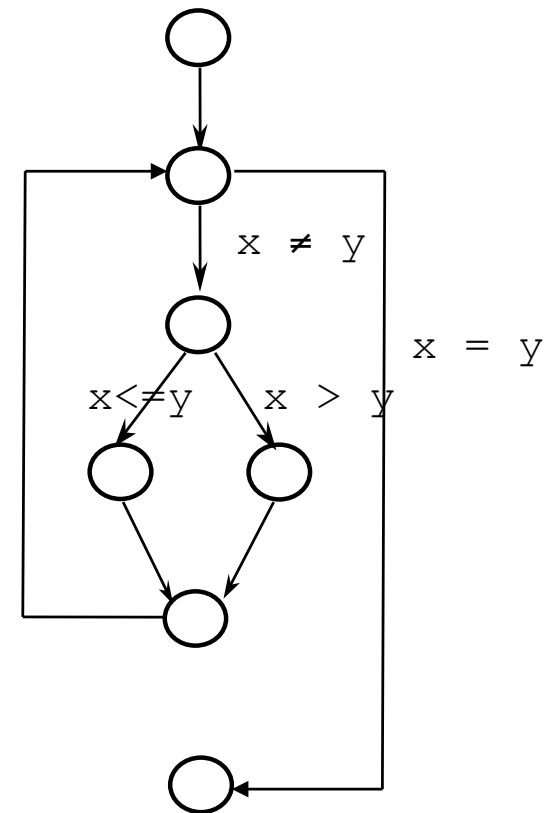
MC/DC coverage: Example (contd.)

- Verify that the following test suite of four tests, executed in the given order, is adequate with respect to statement, and decision (edge) coverage criteria but not with respect to the condition coverage criterion.
- Draw a CFG for this program and verify that all statements are covered.
- We assess the decision (edge) and condition coverage here...

Test	Requirement	done	d	x	y	z
t_1	$R_{1.2}$	false	East	10	15	3
t_2	$R_{1.1}$	false	South	10	15	4
t_3	$R_{1.3}$	false	North	10	15	5
t_4	R_2	true	-	-	-	-

Path Coverage

- **Path Coverage Criterion:** Select a test set T such that, by executing P for each test case t in T , all paths leading from the initial to the final node of P 's control flow graph are traversed
- In practice, however, the number of paths is too large, if not infinite (e.g., when we have loops)
- Some paths are infeasible (e.g., not practical given the system's business logic)
- It may be important to determine "critical paths", leading to more system load, security intrusions, etc.

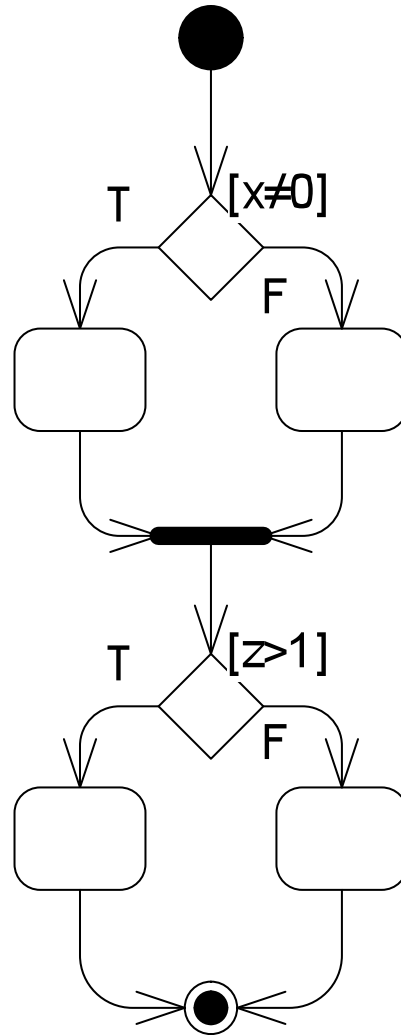


Path Coverage - Example

```

if x ≠ 0 then
  y := 5;
else
  z := z - x;
end if;
if z > 1 then
  z := z / x;
else
  z := 0;
end if;

```



Let us compare how the following two test sets cover this CFG:

T1 (test set) =

{TC11:<x=0, z =1>,}

TC12:<x =1, z=3>}

T2 =

{TC21:<x=0, z =3>,}

TC22:<x =1, z=1>}

See next...

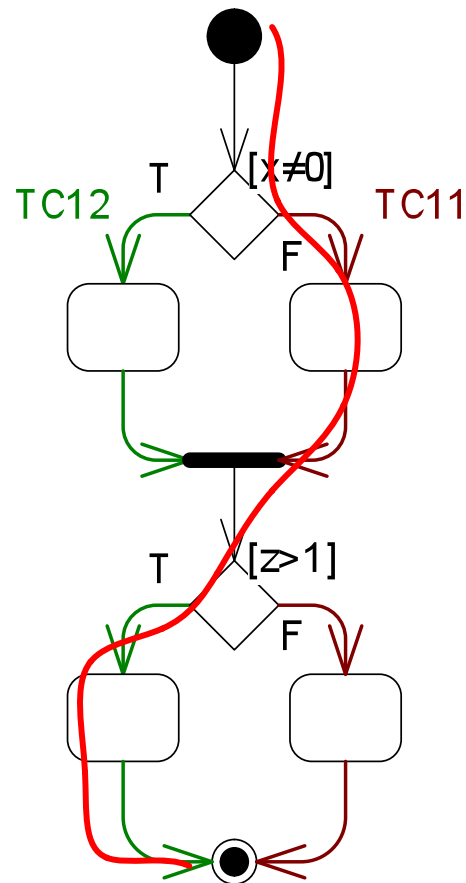
Path Coverage - Example

T1's coverage

```

if x ≠ 0 then
    y := 5;
else
    z := z - x;
end if;
if z > 1 then
    z := z / x;
else
    z := 0;
end if;

```



T1's coverage:

T1 (test set) =
 {TC11:<x=0, z =1>,
 TC12:<x =1, z=3>}

T1 executes all edges
 but...!

Do you see any testing issue
 (uncovered paths which can
 be sources of failure)?

T1 executes all edges
 and all conditions but
 does not test risk of
 division by 0. (See
 the **red** "path")

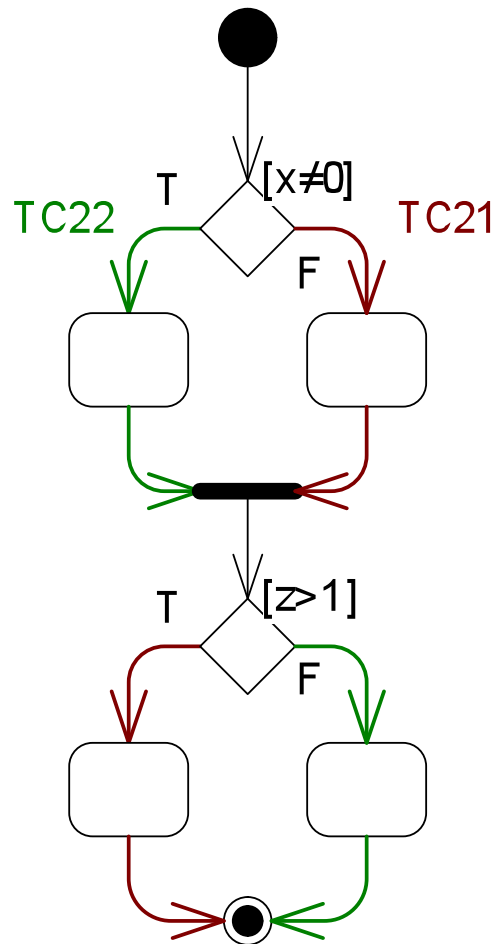
Path Coverage - Example

T2's coverage

```

if x ≠ 0 then
  y := 5;
else
  z := z - x;
end if;
if z > 1 then
  z := z / x;
else
  z := 0;
end if;

```

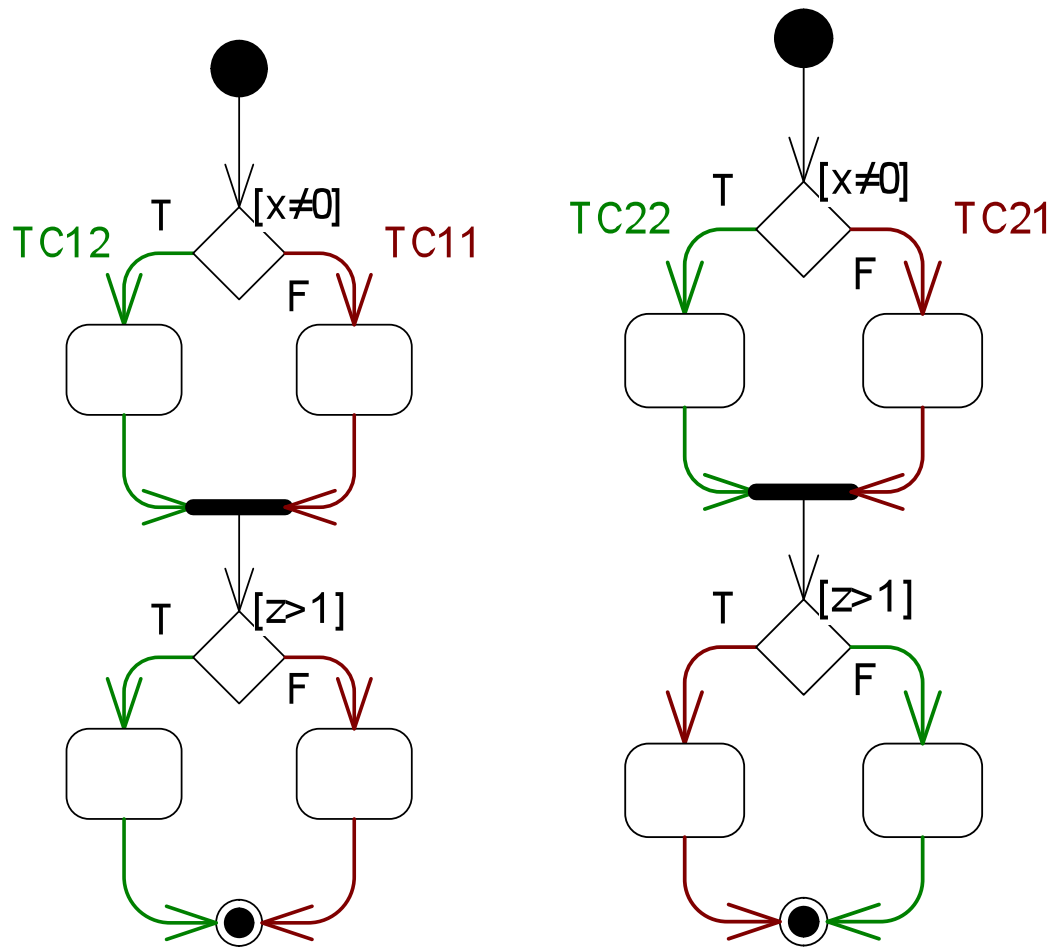


T2's coverage:

T2 =
 {TC21:<x=0, z =3>,
 TC22:<x =1, z=1>}

T2 would find the problem (triggering division by 0) by exercising the remaining possible flows of control through the program fragment.

Path Coverage - Example



T1 (test set) =
 {TC11: $\langle x=0, z=1 \rangle$,
 TC12: $\langle x=1, z=3 \rangle$ }

T1 executes all edges but do
 not show risk of division by 0

T2 = {TC21: $\langle x=0, z=3 \rangle$,
 TC22: $\langle x=1, z=1 \rangle$ }

T2 would find the problem by
 exercising the remaining
 possible flows of control
 through the program fragment

Observation:

T1 \cup T2 \rightarrow all paths
 covered

Path Coverage

Dealing with Loops

- In practice, however, the number of paths can be too large, if not **infinite** (e.g., when we have loops) → Impractical
- An pragmatic heuristic: Look for conditions that execute loops
 - Zero times
 - A maximum number of times
 - A average number of times (statistical criterion)
- For example, in the array search algorithm
 - Skipping the loop (the table is empty)
 - Executing the loop once or twice and then finding the element
 - Searching the entire table without finding the desired element

Path Coverage - Dealing with Loops

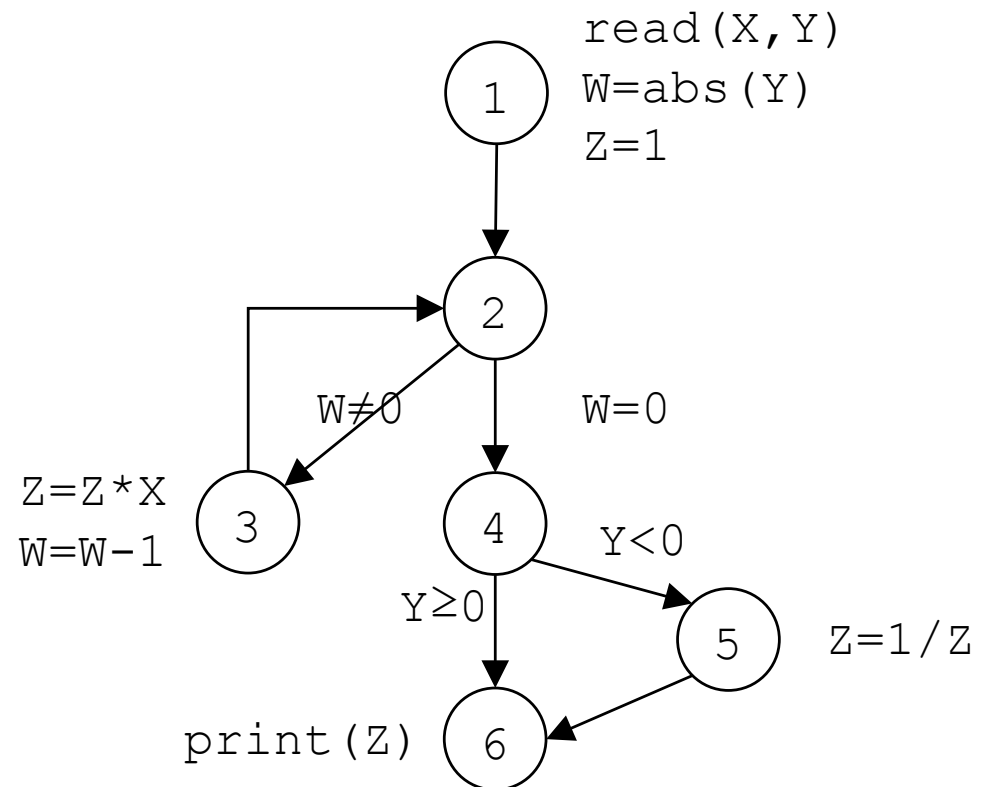
Another Example: Power Function

Program computing
 $Z=X^Y$

```

BEGIN
  read (X, Y) ;
  W = abs (Y) ;
  Z = 1 ;
  WHILE (W <> 0) DO
    Z = Z * X ;
    W = W - 1 ;
  END
  IF (Y < 0) THEN
    Z = 1 / Z ;
  END
  print (Z) ;
END

```



Path Coverage - comparison with "all branches" and "all statements"

- All paths

- Infeasible path

- $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$, Why infeasible?
 - The way Y and W relate.

- Potentially large number of paths (depends on Y)

- As many ways to iterate
 - $2 \rightarrow (3 \rightarrow 2)^* \rightarrow 4 \rightarrow 6$ as values of $Abs(Y)$

- All branches

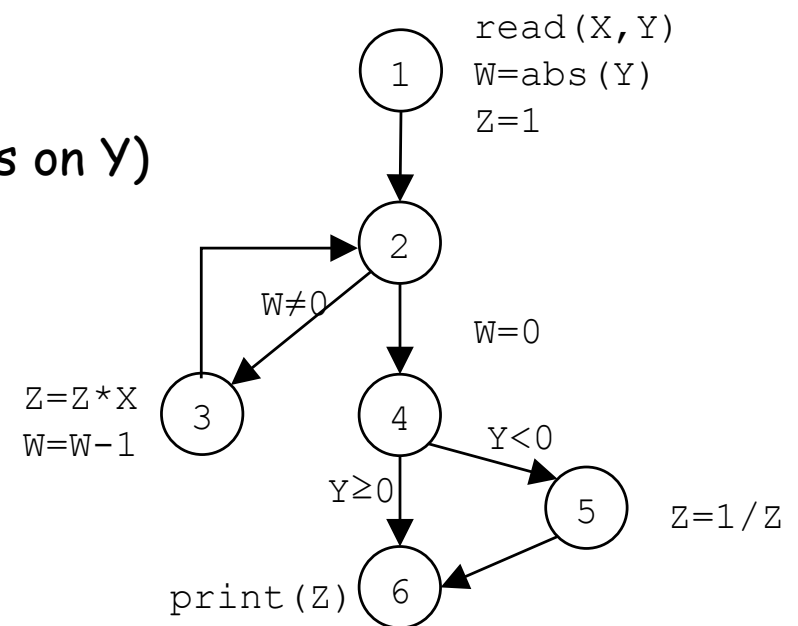
- Two test cases are enough

- $Y < 0 : 1 \rightarrow 2 \rightarrow (3 \rightarrow 2)^+ \rightarrow 4 \rightarrow 5 \rightarrow 6$
 - $Y > 0 : 1 \rightarrow 2 \rightarrow (3 \rightarrow 2)^* \rightarrow 4 \rightarrow 6$

- All statements

- One test case is enough

- $Y < 0 : 1 \rightarrow 2 \rightarrow (3 \rightarrow 2)^+ \rightarrow 4 \rightarrow 5 \rightarrow 6$



Generating Code-based (while-box) Test data

- To find test inputs that will execute an arbitrary statement Q within a program source, the tester must work backward from Q through the program's flow of control to an input statement
- For simple programs, this amounts to solving a set of simultaneous inequalities in the input variables of the program, each inequality describing the proper path through one conditional
- An example next...

Generating Code-based Test data - An Example

```
int z;
scanf("%d%d", &x, &y);
if (x > 3) {
    z = x+y;
    y+= x;
    if (2*z == y) {
        an example statement
    }
}
/* e.g., we want this statement to be covered
   by a test case */
...
```

Inequalities?

- . $2(x+y) = x+y$
 $\Leftrightarrow x = -y$
- . $x > 3$

We can have many solutions, e.g.:

$$x = 4$$

$$y = -4$$

Generating Code-based Test data - Loops

- The presence of loops and recursion in the code makes it very hard (and sometimes impossible) to write and solve the inequalities in general
- Each pass through a loop may alter the values of variables that figure in a following conditional and the number of passes cannot be determined by static analysis in general

Control Flow Coverage Reachability

- Not all statements are usually reachable in real-world programs
- It is not always possible to decide automatically if a statement is reachable and the percentage of reachable statements
- When one does not reach a 100% coverage, it is therefore difficult to determine the reason
- Tools are needed to support this activity but it cannot be fully automated
- Research focuses on search algorithms to help automate coverage
- Control flow testing is, in general, more applicable to testing in the small

Data Flow Analysis - Idea and Definitions

- In Data Flow Analysis, we focus on CFG paths that are significant for the data flow in the program
- Focus here is on the assignment of values to variables and their uses
- Analysis of occurrences of variables
- **Definition occurrence**: a value is written (bound) to a variable
- **Use occurrence**: the value of a variable is read (referred)
- **Predicate use**: a variable is used to decide whether a predicate is (evaluates to) true or false
- **Computational use**: compute a value for defining other variables or output values
- Examples next...

Definitions and uses

A program written in a procedural language, such as C and Java, contains variables. Variables are defined by assigning values to them and are used in expressions.

Statement `x = y + z` defines variable `x` and uses variables `y` and `z`

Statement `scanf ("%d %d", &x, &y)` defines variables `x` and `y`

Statement `printf ("Output: %d \n", x+y)` uses variables `x` and `y`

A parameter `x` passed as call-by-reference, can serve as a definition and use of `x`

A parameter `x` passed as call-by-value to a function, is considered as a use of (or a reference to) `x`

Definitions and uses: Pointers

Consider the following sequence of statements that use pointers.

```
z=&x;
y=z+1;
*z=25;
y=*z+1;
```

- The first of the above statements defines a pointer variable z
- the second defines y and uses z
- the third defines x through the pointer variable z , and
- the last defines y and uses x accessed through the pointer variable z

Variable z is a pointer pointing to variable x and contains the memory address of variable x .

$*z$ retrieves the value at the memory address pointed by variable z . Consequently, $*z = 25$ is to assign 25 to the memory address pointed by variable z . That is, to assign 25 to variable x .

$y=*z+1$ is to define as the sum of 1 and the value at the memory address pointed by variable z , i.e., the value of x

Definitions and uses: Arrays

Arrays are also tricky. Consider the following declaration and two statements in C:

```
int A[10];  
A[i]=x+y;
```

The first statement defines variable A .

The second statement defines A and uses i , x , and y .

Alternate: second statement defines $A[i]$ and not the entire array A .

The choice of whether to consider the entire array A as defined or the specific element depends upon how stringent is the requirement for coverage analysis.

c-use

Uses of a variable that occurs within an expression as part of an assignment statement, in an output statement, as a parameter within a function call, and in subscript expressions, are classified as c-use, where the "c" in c-use stands for computational.

How many c-uses of x can you find in the following statements?

```
z=x+1;  
A[x-1]= B[2];  
foo(x*x)  
output(x);
```

p-use

The occurrence of a variable in an expression used as a condition in a branch statement such as an *if* and a *while*, is considered as a p-use. The "p" in p-use stands for predicate.

How many p-uses of *z* and *x* can you find in the following statements?

```
if(z>0){output (x)};  
while(z>x){...};
```

p-use: possible confusion

Consider the statement:

```
if(A[x+1]>0){output(x)};
```

The use of *A* is clearly a p-use.

Is the use of *x* in the subscript, a c-use or a p-use?

c-uses within a basic block

Consider the basic block

```
p=y+z;  
x=p+1;  
p=z*z;
```

While there are two definitions of p in this block, only the second definition will propagate to the next block. The first definition of p is considered **local** to the block while the second definition is **global**.

We are concerned with global definitions, and uses.

Note that y and z are **global uses**; their definitions flow into this block from some other block.

Data Flow Analysis - FACTORIAL Example

```

1. public int factorial(int n) {
2.   int i, result = 1;
3.   for (i=2; i<=n; i++) {
4.     result = result * i;
5.   }
6.   return result;
7. }

```

Variable	Definition line	Use line
n	1	3 (Predicate)
result	2	4 (Computation)
result	2	6
result	4	4
result	4	6
i	3	3
i	3	4

Data Flow Analysis

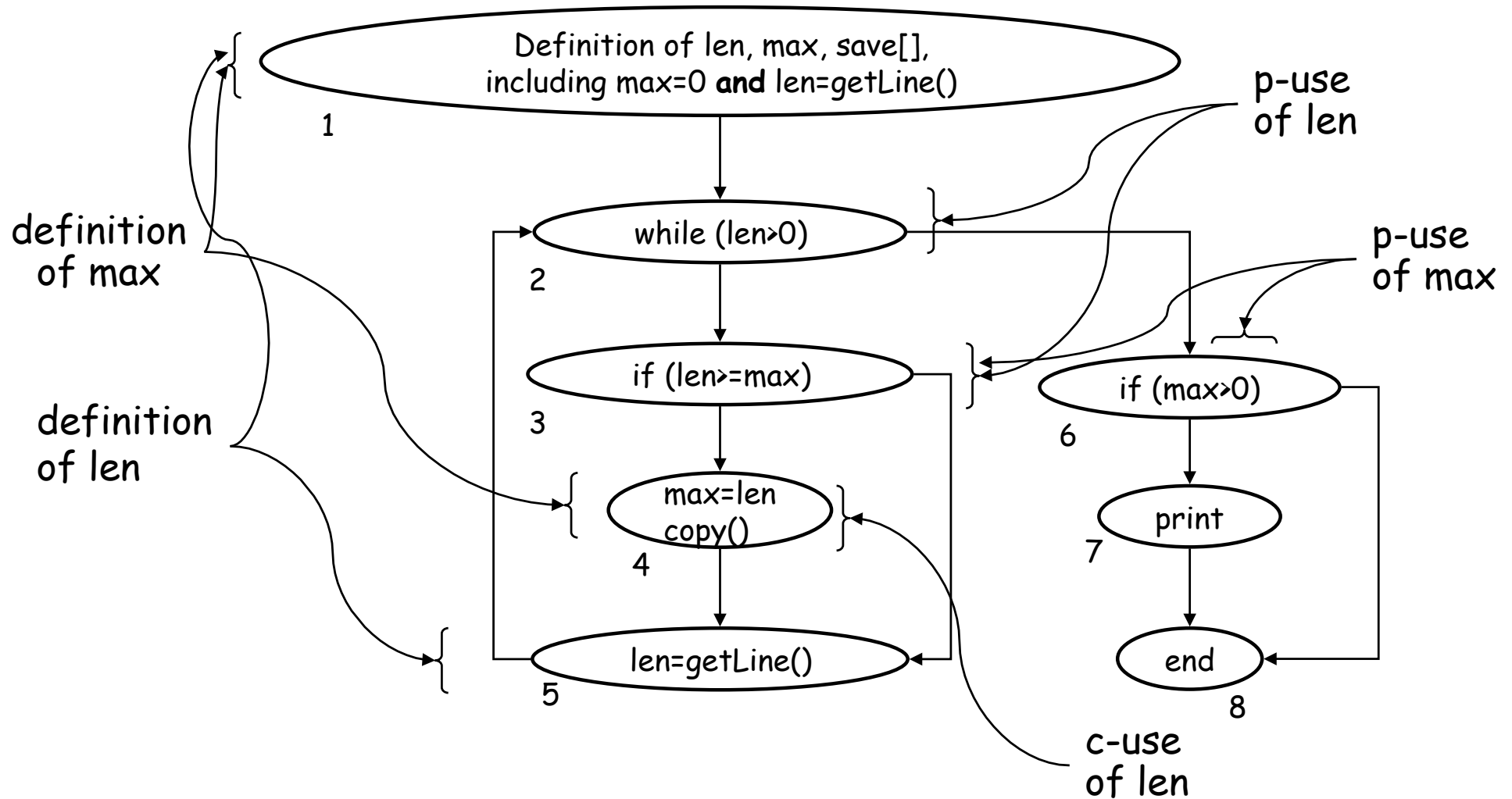
Basic Formal Definitions

- **Definition (defining) node:** Node $n \in CFG(P)$ is a defining node of the variable $v \in V$, written as $DEF(v, n)$, iff (if and only if) the value of the variable v is defined in the statement corresponding to node n
- **Usage (use) node:** Node $n \in CFG(P)$ is a usage node of the variable $v \in V$, written as $USE(v, n)$, iff the value of the variable v is used in the statement corresponding to node n
- **Predicate and Computation use:** A usage node $USE(v, n)$ is a predicate use (denoted as *P-Use*) iff the statement n is a predicate statement, otherwise $USE(v, n)$ is a computation use (denoted as *C-use*)

Data Flow Analysis - Basic Definitions II

- **PATHS(P)**: the set of all CFPs in program P
- **definition-use (du)-path**: A definition-use path with respect to a variable v (denoted **du-path**) is a path in **PATHS(P)** such that, for some $v \in V$, there are definition and usage nodes **DEF**(v, m) and **USE**(v, n) such that m and n are initial and final nodes of the path.
- **definition-clear (dc)-path**: A definition-clear path with respect to a variable v (denoted **dc-path**) is a definition-use path in **PATHS(P)** with initial and final nodes **DEF**(v, m) and **USE**(v, n) such that no other node in the path is a defining node of v .
- Examples next...

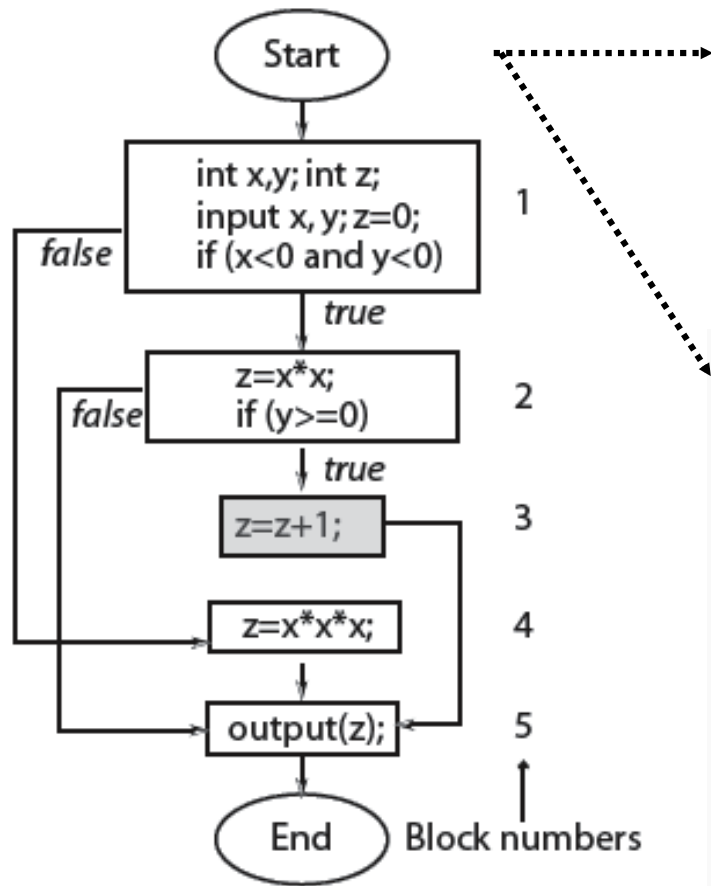
Simple Example



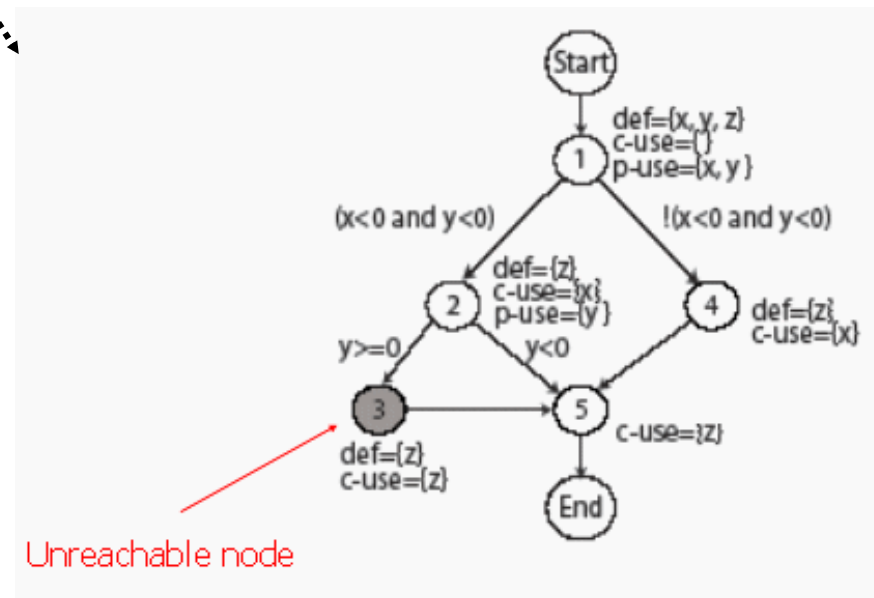
Data flow graph

- A data-flow graph of a program, denoted as def-use graph, captures the flow of definitions (denotes as defs) and uses across basic blocks in a program.
- It is similar to a control flow graph of a program in that the nodes, edges, and all paths in the control flow graph are preserved in the data flow graph.
- Attach defs, c-use and p-use to each node in the graph. Label each edge with the condition which when true causes the edge to be taken.
- We use $d_i(x)$ to refer to the definition of variable x at node i . Similarly, $u_i(x)$ refers to the use of variable x at node i .

Data flow graph: Example



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

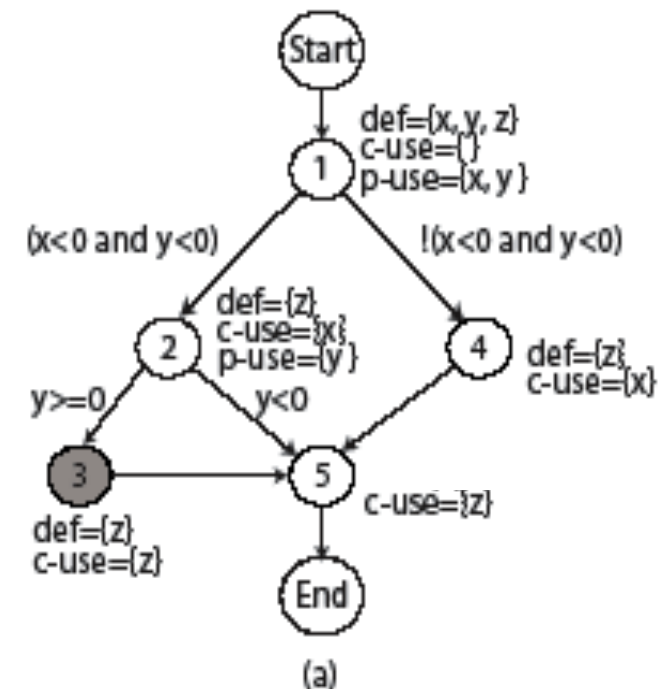


Def-clear path

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

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

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



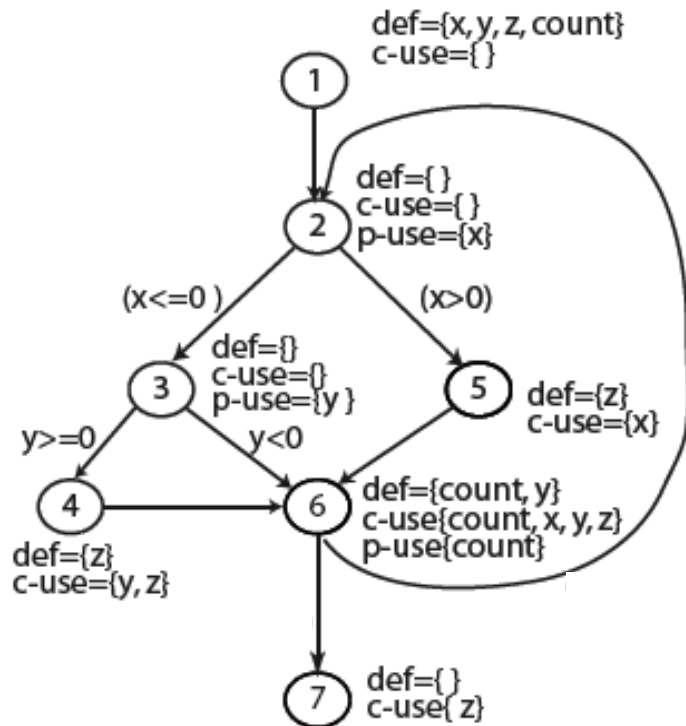
Def-use pairs

- Def of a variable at line l_1 and its use at line l_2 constitute a def-use pair. l_1 and l_2 can be the same.
- $dcu(d_i(x))$ or $dcu(x, i)$ denotes the set of all nodes where the definition of x at node i is live and used.
- $dpu(d_i(x))$ or $dpu(x, i)$ denotes the set of all edges (k, \wedge) such that there is a def-clear path from node i to edge (k, \wedge) and x is used at node k .

We say that a def-use pair $(d_i(x), u_j(x))$ is covered when a def-clear path that includes nodes i to node j is executed.

If $u_j(x)$ is a p-use then all edges of the kind (j, k) must also be taken during some executions for the def-use pair to be covered.

Def-use pairs (example)



Variable (v)	Defined in node (n)	dcu (v, n)	dpu (v, n)
x	1	{5, 6}	{(2, 3), (2, 5)}
y	1	{4, 6}	{(3, 4), (3, 6)}
y	6	{4, 6}	{(3, 4), (3, 6)}
z	1	{4, 6, 7}	{}
z	4	{4, 6, 7}	{}
z	5	{4, 6, 7}	{}
count	1	{6}	{(6, 2), (6, 7)}
count	6	{6}	{(6, 2), (6, 7)}

Data Flow Analysis - Data Flow Coverage Criteria (I)

- **all-Definitions criterion:** The test set T satisfies the **all-Definitions criterion** for the program P iff for every variable $v \in V$, T contains definition-clear paths from every defining node of v to a use of v .
- **all-Uses criterion:** The test set T satisfies the **all-Uses criterion** for the program P iff for every variable $v \in V$, T contains at least one definition-clear path from every defining node of v to every reachable use of v .
- **all-P-Uses/Some C-Uses criterion:** The test set T satisfies the **all-P-Uses/Some C-Uses criterion** for the program P iff for every variable $v \in V$, T contains at least one definition clear path from every defining node of v to every predicate use of v , and if a definition of v has no P-Uses, there is a definition-clear path to at least one computation use.

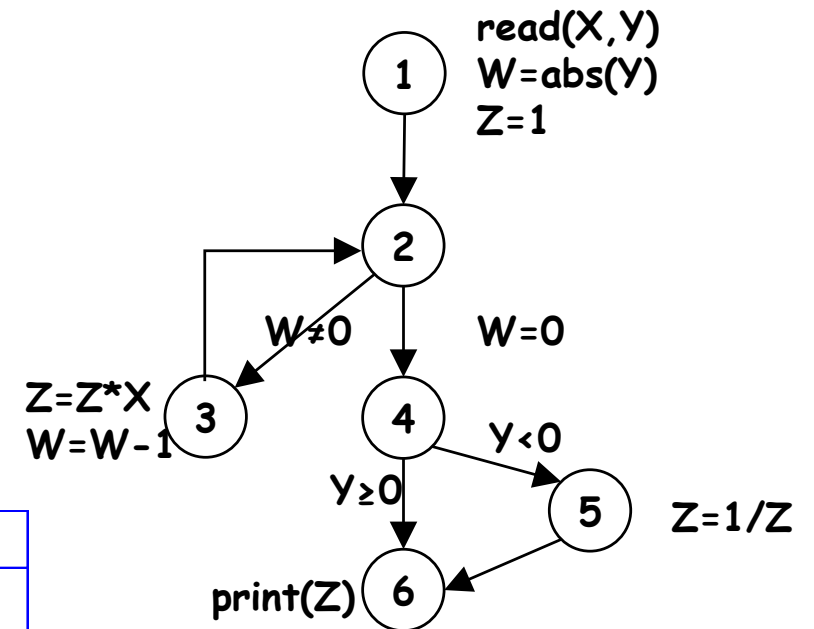
Data Flow Analysis - Data Flow Coverage Criteria (II)

- **all-C-Uses/Some P-Uses criterion:** The test set T satisfies the **all-C-Uses/Some P-Uses criterion** for the program P iff for every variable $v \in V$, T contains at least one definition-clear path from every defining node of v to every computation use of v , and if a definition of v has no C-Uses, there is a definition-clear path to at least one predicate use.
- **all-DU-Paths criterion:** The test set T satisfies the **all-DU-Paths criterion** for the program P iff for every variable $v \in V$, T contains all definition-clear paths from every defining node of v to every reachable use of v , and that these paths are either single loops traversals, or they are cycle free.

POWER Example

node i	def(i)	c-use(i)	edge(i,j)	p-use(i,j)
1	X, Y, W, Z		(1,2)	
2			(2,3) (2,4)	W W
3	W, Z	X, W, Z	(3,2)	
4			(4,5) (4,6)	Y Y
5	Z	Z	(5,6)	
6		Z		

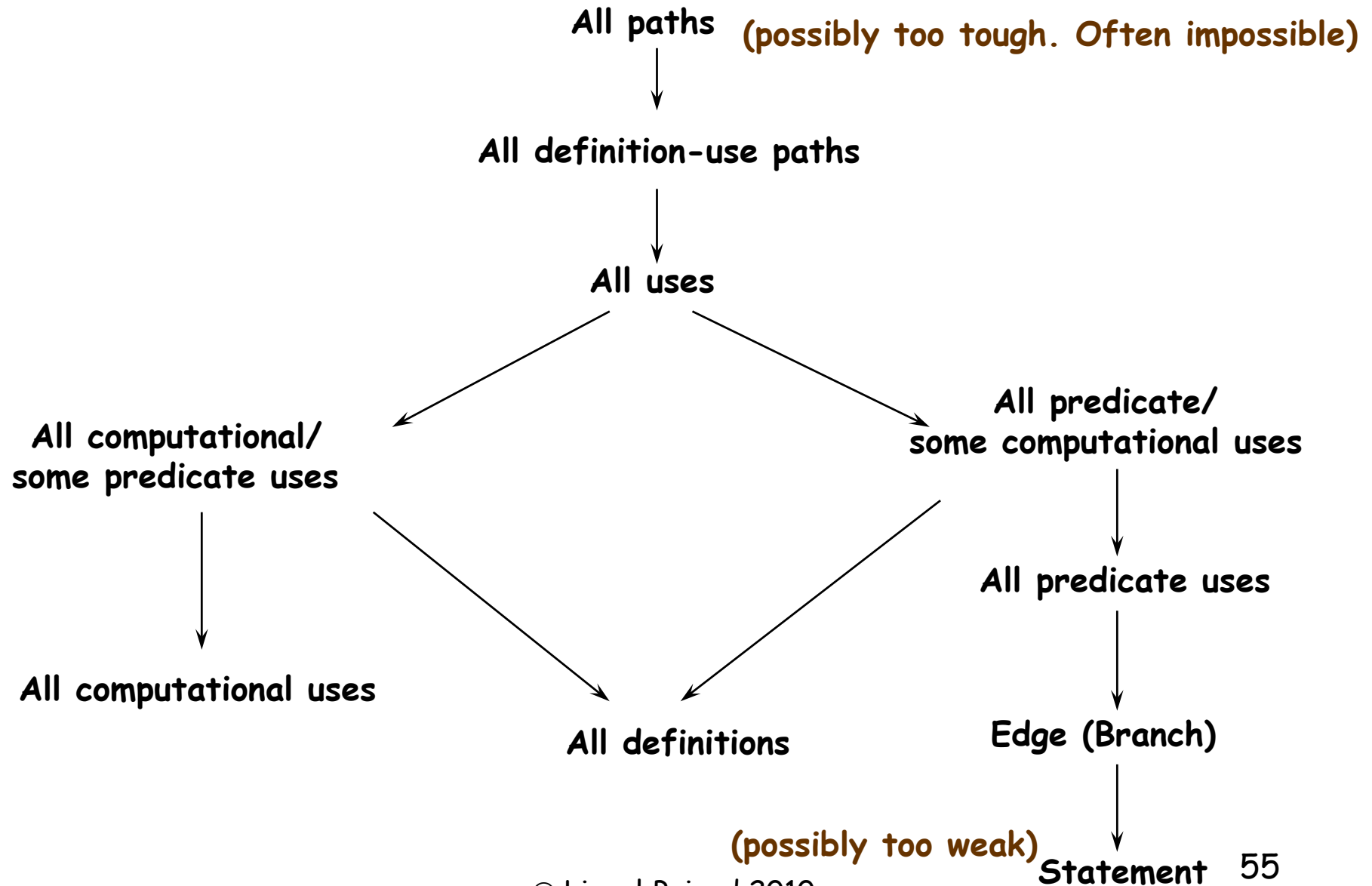
node i	dcu(v,i)	dpu(v,i)
1	dcu(X,1) = {3} dcu(Z,1) = {3,6} dcu(W,1) = {3}	dpu(Y,1) = {(4,5),(4,6)} dpu(W,1) = {(2,3),(2,4)}
3	dcu(W,3) = {3} dcu(Z,3) = {3,5,6}	dpu(W,3) = {(2,3),(2,4)}
5	dcu(Z,5) = {6}	



Data Flow Analysis (DFA) - Discussion

- In DFA, we generate test data according to the way data is manipulated in the program
- DFA can help us define intermediary CFA criteria between all-nodes testing (possibly too weak) and all-paths testing (often impossible)
- But we need effective tool support for DFA if we want to use it extensively.

The subsumption relationships



Data Flow Analysis - Measuring Code Coverage

- One advantage of structural criteria is that their coverage can be measured *automatically*
- To control testing progress
- To assess testing completeness in terms of remaining faults and reliability
- High coverage is not a guarantee of fault-free software, just an element of information to increase our confidence -> statistical models

Analysis of test coverage data

Test Productivity

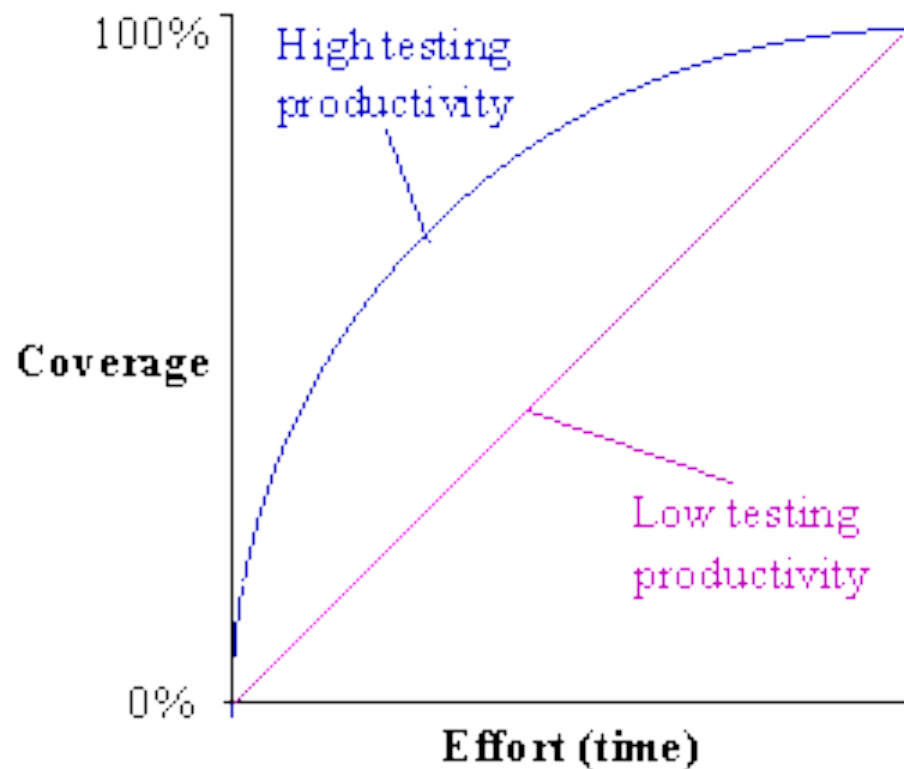


Figure 1: Coverage rate

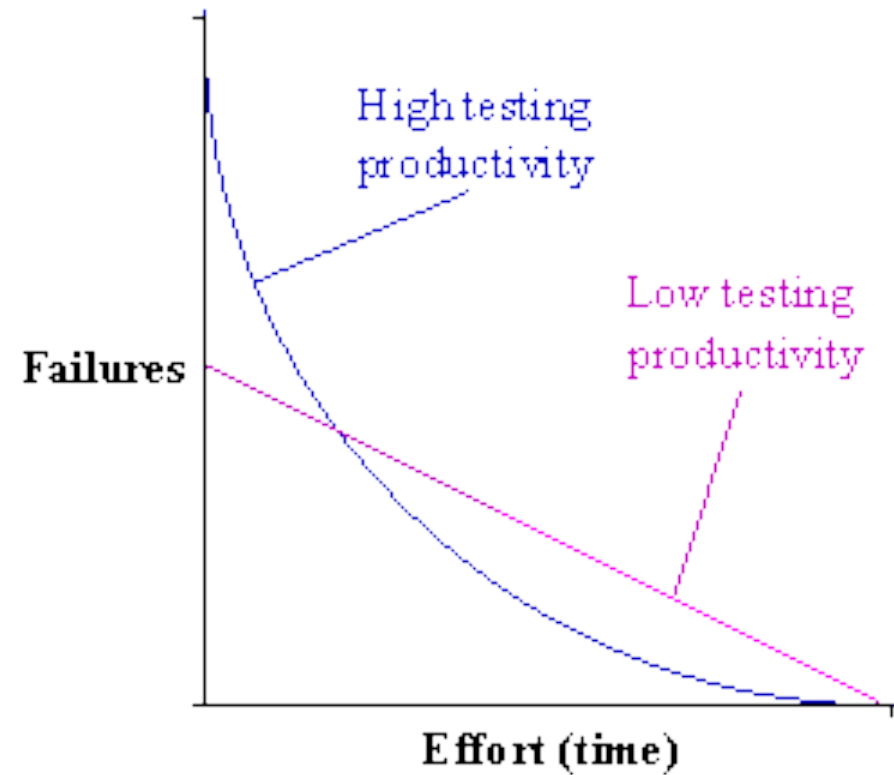
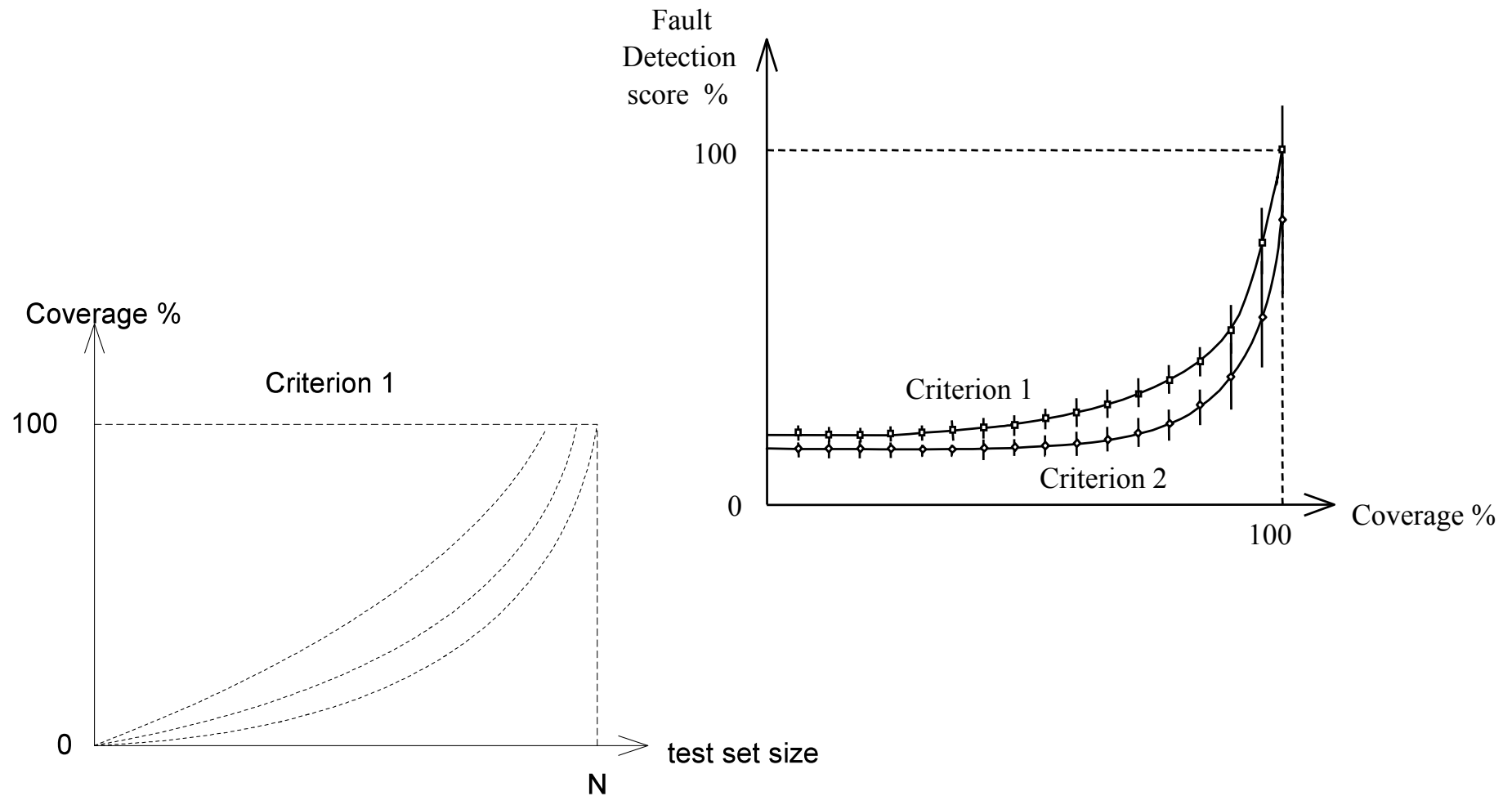


Figure 2: Failure discovery rate

Analysis of test coverage data

Typical Analyses



Analysis of test coverage data

An Experiment by Hutchins et al.

- Hutchins, Foster, Goradia, Ostrand, "Experiments on the Effectiveness of Dataflow- and Control flow-Based Test Adequacy Criteria", proceedings of the International Conference on Software Engineering, 1994
- The All-Edges and All-DU coverage criteria were applied to 130 faulty versions of 7 C programs (141-512 LOC) by seeding realistic faults
- The 130 faults were created by 10 different people, mostly without knowledge of each other's work; their goal was to be as realistic as possible.
- The test generation procedure was designed to produce a wide range both of test size and test coverage percentages, i.e., at least 30 test sets for each 2% coverage interval for each program
- They examined the relationship between fault detection and test set coverage / size

Analysis of test coverage data

One Program Example by Hutchins et al.

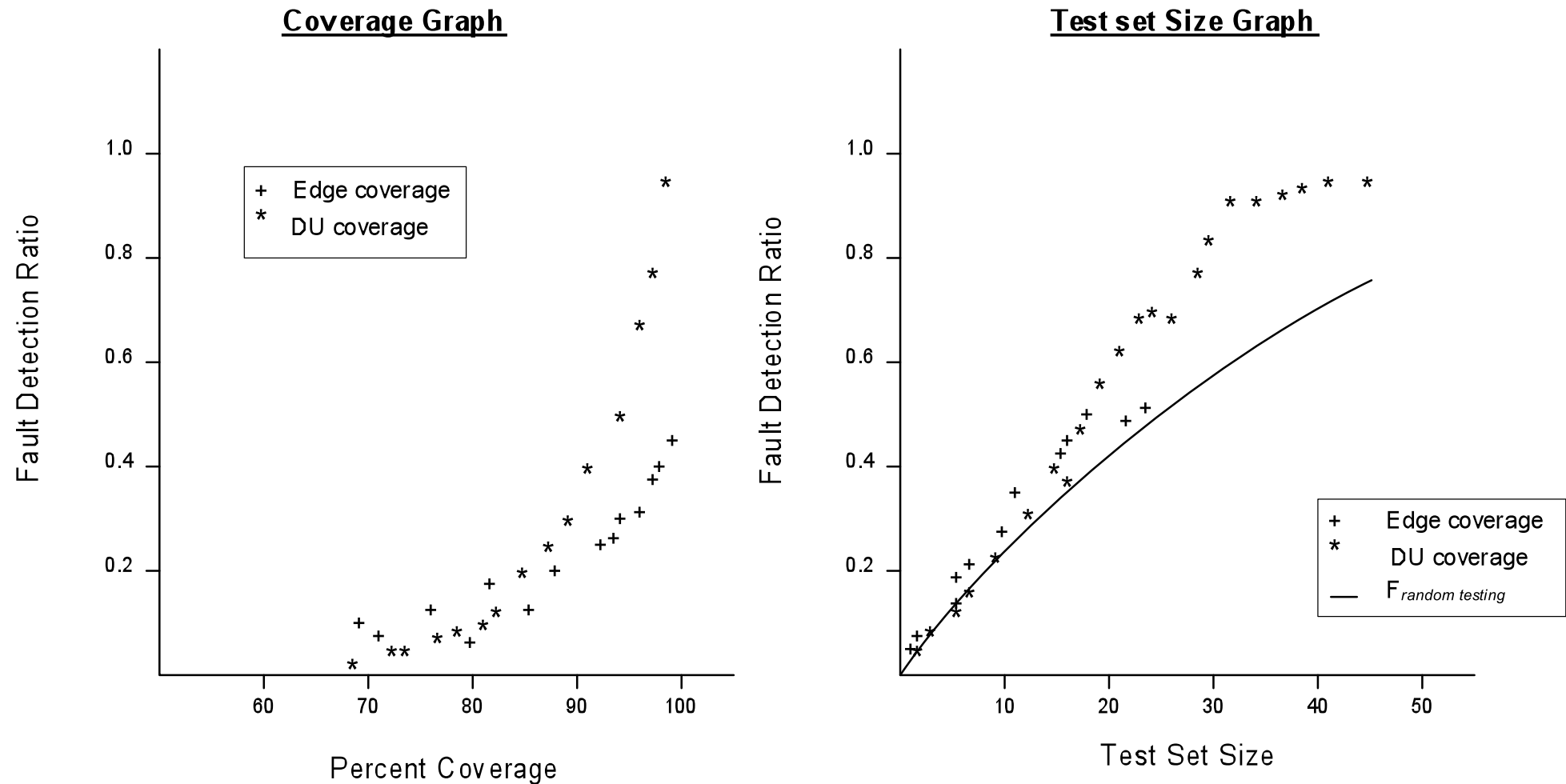


Figure : Fault Detection Ratios for One Faulty Program

Analysis of test coverage data Results of Hutchins et al.'s work

- Both coverage criteria performed better than random test selection - especially DU-coverage
- Significant improvements occurred as coverage increased from 90% to 100%
- 100% coverage alone is not a reliable indicator of the effectiveness of a test set - especially edge coverage
- Wide variation in test effectiveness for a given coverage criteria
- As expected, on average, achieving all-DU coverage required significantly larger test sets with all-Edge coverage

A short Background on Eclipse, and JUnit

- Eclipse is a very popular Integrated Development Environment (IDE) written primarily in and intended for Java.
- But it is now used for developing in many other languages too, PHPUnit, NUnit, .
- If you go to the SW industry, you are most probably going to see/use it!
- JUnit is a unit testing framework for Java.
- It is extremely popular in the SW industry.
- Check the web:
 - en.wikipedia.org/wiki/List_of_unit_testing_frameworks
 - www.junit.org
 - en.wikipedia.org/wiki/JUnit



A short Background on CodeCover

- CodeCover is an open-source code coverage tool for Java under Eclipse.
 - It can be used inside Eclipse.
 - <http://www.codecover.org>
- 
- The logo for CodeCover, featuring a blue circular icon with a white 'C' shape inside, followed by the text 'CodeCover' in a bold, sans-serif font. A horizontal blue bar is positioned below the text.
- There are many other code coverage tool out there, but this is one of the lightest and powerful ones.
 - See the list @ <http://java-source.net/open-source/code-coverage>

Demo of Eclipse and JUnit

- $T = \{t1: \langle x=2, y=3, \text{return}=5 \rangle, t2: \langle x=3, y=1, \text{return}=3 \rangle\}$
- $T = \{\text{sumTestCase}, \text{productTestCase}\}$

```

sumProductClass.java  productTestCase.java  testSuite.java  sumTestCase.java X
package sumProduct;

import junit.framework.TestCase;

public class sumTestCase extends TestCase {
    public void testSum() { //
        sumProductClass sp= new sumProductClass();
        int returnValue = sp.sumProduct(2, 3);
        assertEquals(returnValue, 5);
    }
}

```

```

sumProductClass.java  productTestCase.java X  testSuite.java
package sumProduct;

import junit.framework.TestCase;

public class productTestCase extends TestCase {
    public void testProduct() {
        sumProductClass sp= new sumProductClass();
        int returnValue = sp.sumProduct(3, 1);
        assertEquals(returnValue, 3);
    }
}

```

me	Statement	Branch	Loop	Strict Condition
SENG_607	16.8 %	17.2 %		4.2 %
Example2_CoffeeMaker	16.8 %	17.2 %		4.2 %
src	16.8 %	17.2 %		4.2 %
CoffeeMaker	47.2 %	23.3 %		12.5 %
Inventory	52.9 %	50.0 %		0.0 %
Main	0.0 %	0.0 %		0.0 %
Recipe	54.5 %	37.5 %		0.0 %