

Control flow and data flow testing

Jingyue Li (Bill)

Outline

- Control flow testing and coverage
 - Data flow testing and coverage
 - Mutation testing and coverage

Industry standards define test needed

- IEEE 1012 – general software verification and validation
- IEC 61508 – general safety critical software
- ISO 26262 – automotive software
- Other domain specific standards, e.g.,
 - Healthcare
 - Oil & Gas
 - Railway

ISO 26262 – automotive software

The ASIL (Automotive System Integrity Level) – A, B, C, or D – is the outcome of the combination of three factors









- S – Severity. How dangerous is the event
- E – Probability. How likely is the event
- C – Controllability. How easy is it to control the event if it occurs

Finding the ASIL level

Severity	Probability	C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

ASIL chart*

ASIL chart

Functional category	Hazard	ASIL-A	ASIL-B	ASIL-C	ASIL-D
Driving	Sudden start				
	Abrupt acceleration				
	Loss of driving power				
Braking	Maximum 4 wheel braking				
	Loss of braking function				
Steering	Self steering				
	Steering lock				
	Loss of assistance				

* <https://www.jnovel.co.jp/en/service/compiler/iso26262.html>

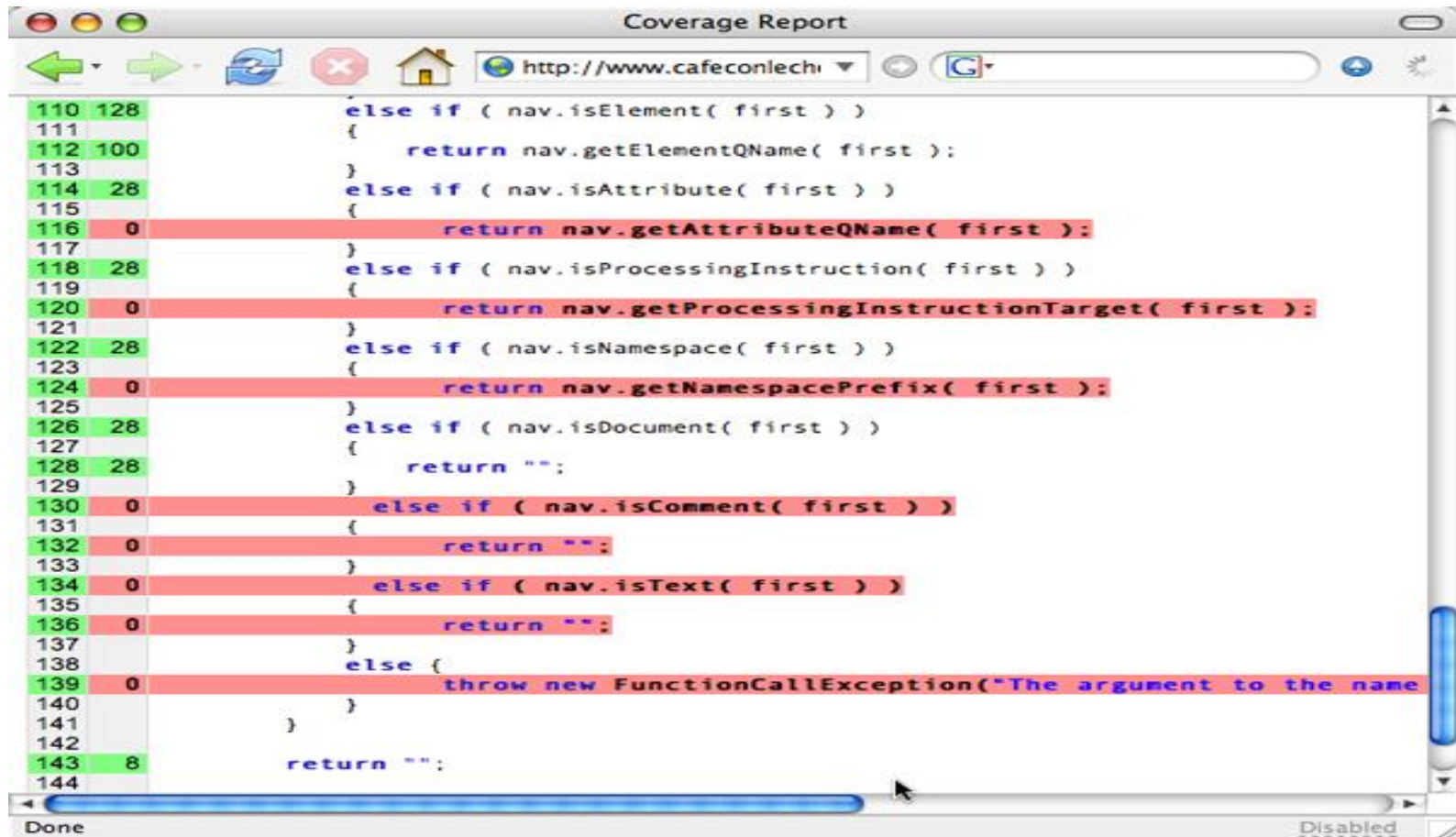
Methods and measurements for software unit testing

Methods and measures	Sub-methods/Measures	ASIL			
		A	B	C	D
Functional tests			
Structural coverage	Statement coverage	++	++	+	+
	Decision coverage	+	+	++	++
	...				
Resource usage measurement	...	+	+	+	++
Back-to-back test between simulation	...	+	+	+	++

Control flow coverage

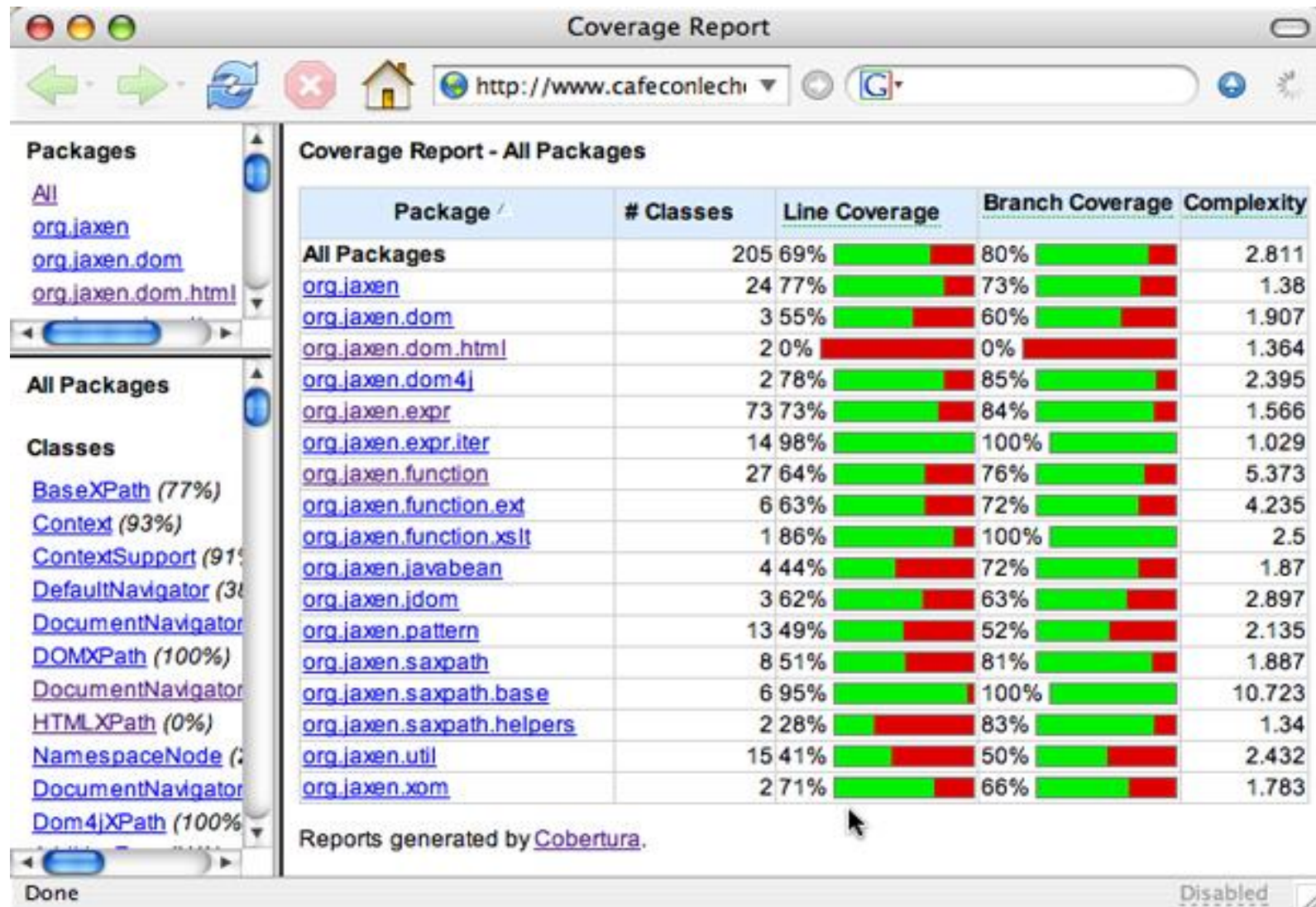
- Statement coverage
- Decision coverage
- Path coverage

Statement coverage report



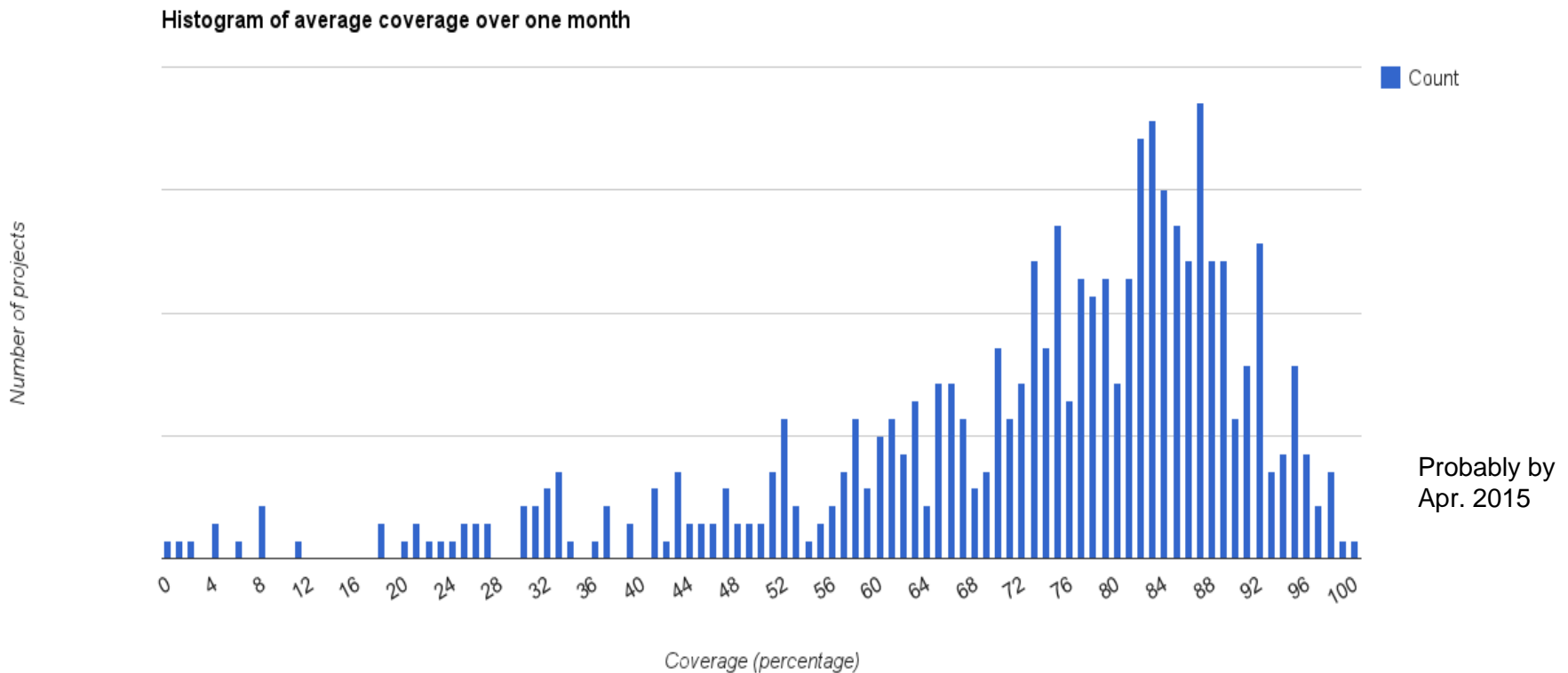
You can get test coverage report with [GitLab CI](#)

Coverage report



Test coverage at Google

- Per-project **statement** coverage
- **1.5 years worth of data across hundreds of projects**



The median is 78%, the 75th percentile is 85% and 90th is percentile 90%.

Decision coverage

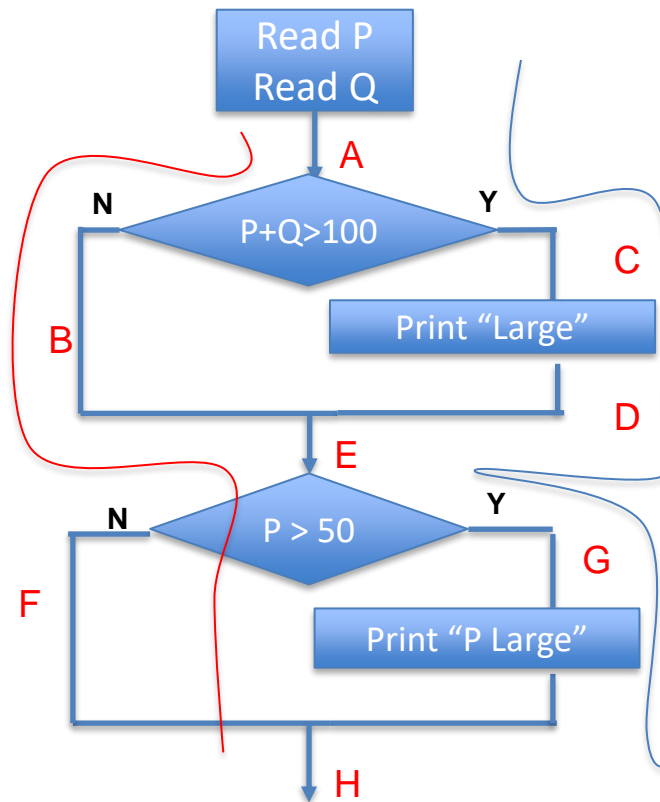
- Each one of the possible branch (true/false or switch-case) from each **decision** point is executed at least once

*

Read P
Read Q

If $P+Q > 100$ THEN
 Print "Large"
ENDIF

IF $P > 50$
 Print " P Large"
ENDIF



Statement coverage:
A, C, D, E, G, H

Decision coverage:
A, C, D, E, G, H
+
A, B, E, F, H

Branch coverage

- Count branches covered
- Slightly different from decision coverage

If 3 out of 4 branches of a switch statement are executed

- Branch coverage 75%
- Decision coverage 0% (a decision is considered covered only if all its branches are covered)

Path coverage

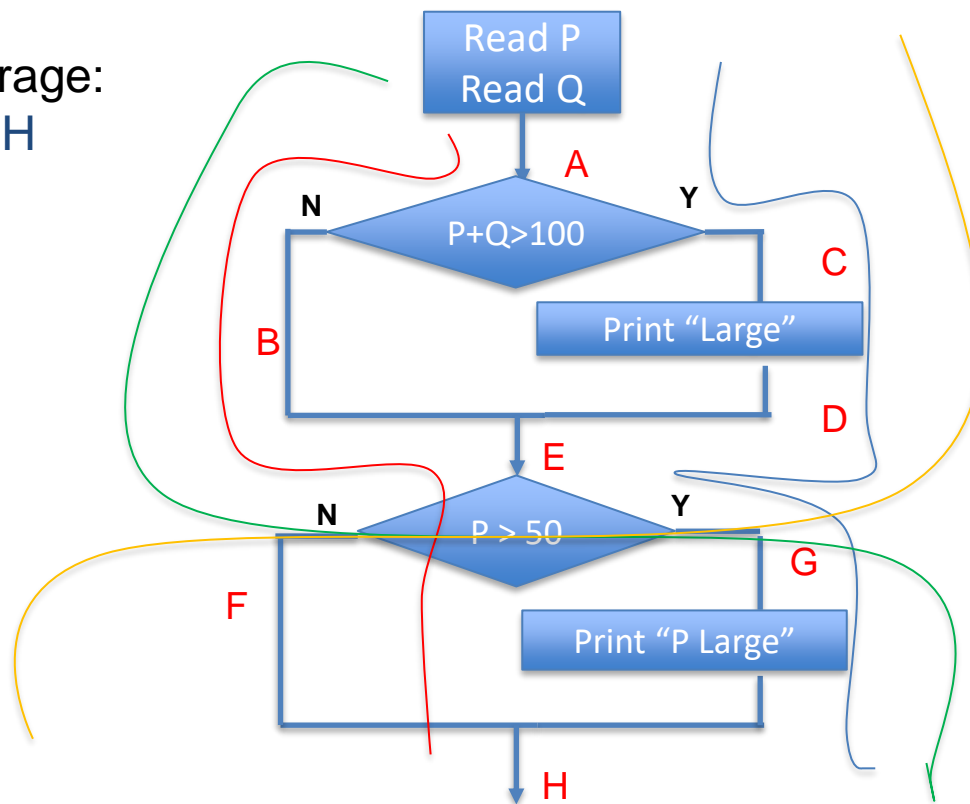
- All possible **paths** of the software should be tested

Decision coverage:

A, C, D, E, G, H

+

A, B, E, F, H



Path coverage:

A, C, D, E, G, H

+

A, B, E, F, H

+

A, B, E, G, H

+

A, C, D, E, F, H

White-box testing methods

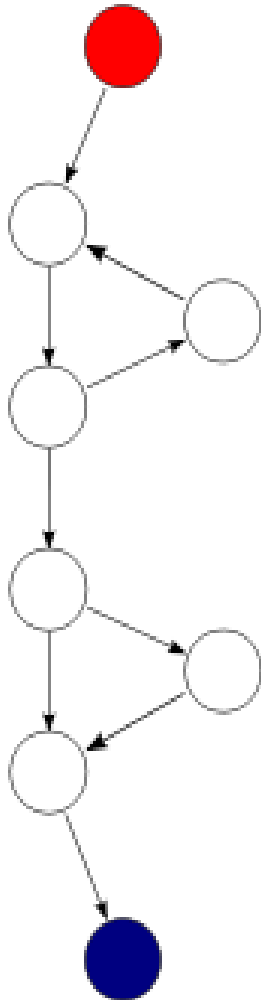
- Usually need to generate control/data flow graph first using some tools, e.g., Java parser (Front End)*
- Then, use different test inputs to achieve different coverages
- Path coverage is more difficult to achieve
 - How many test cases will be sufficient?
 - McCabe's cyclomatic complexity can help
 - How to systematically create test cases?
 - Decision table can help

* <http://www.semanticdesigns.com/Products/FrontEnds/JavaFrontEnd.html>

McCabe's cyclomatic complexity

- From control flow graph of the program
- $v(G) = E - N + 2P$
- $v(G)$ = cyclomatic complexity
- E = the number of edges of the graph
- N = the number of nodes of the graph
- P = the number of connected components

Graph example

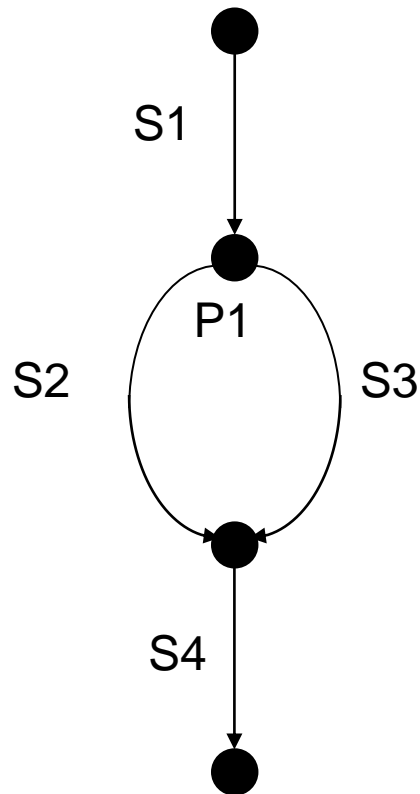


- $v(G) = E - N + 2P$
- E: edge, N:node, P: connected component
- $E = 9$
- $N = 8$
- $P = 1$
- $V(G) = 9 - 8 + 2 \times 1 = 3$

Using $v(G)$ to count number of paths

- The minimum number of paths through the code is $v(G)$
- As long as the code graph is a **DAG – Directed Acyclic Graph** – the maximum number of paths is $2^{|\{\text{predicates}\}|}$ (when all predicates have only two possible values)
- Thus, we have that
$$V(G) \leq \text{number of paths} \leq 2^{|\{\text{predicates}\}|}$$

Simple case - 1

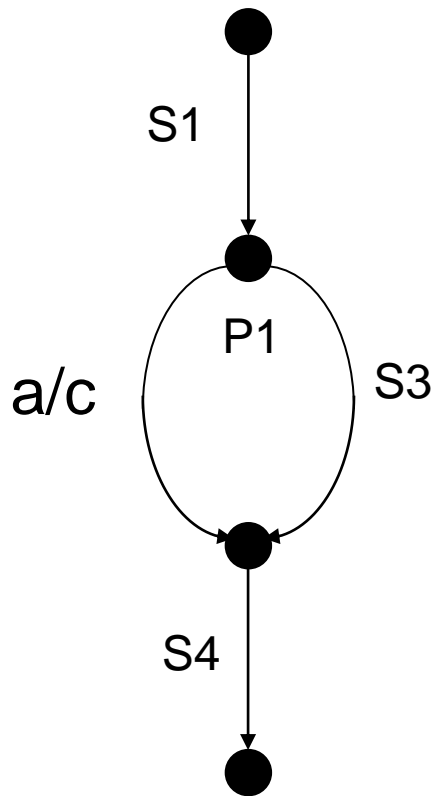


S1;
IF P1 THEN S2 ELSE S3
S4;

$$E = 4, N = 4, P = 1, E - N + 2P = 2$$

- $V(G) = 2$,
- One predicate (P1)
- Two test cases can cover all paths

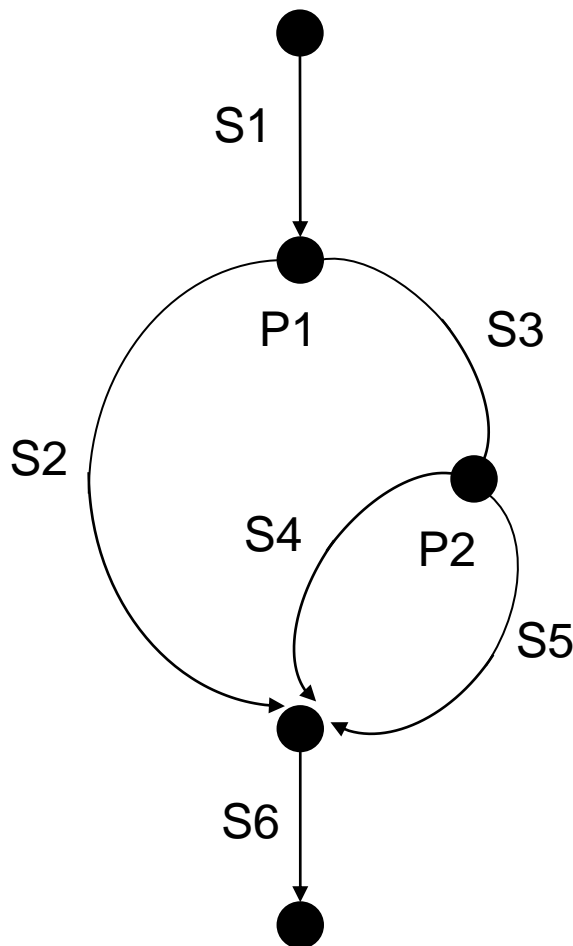
Simple case – 2



S1;
IF P1 THEN $X := a/c$ ELSE S3;
S4;

- $V(G) = 2$,
- One predicate (P1)
- Two test cases will cover all paths but not all cases. What about the case $c = 0$?

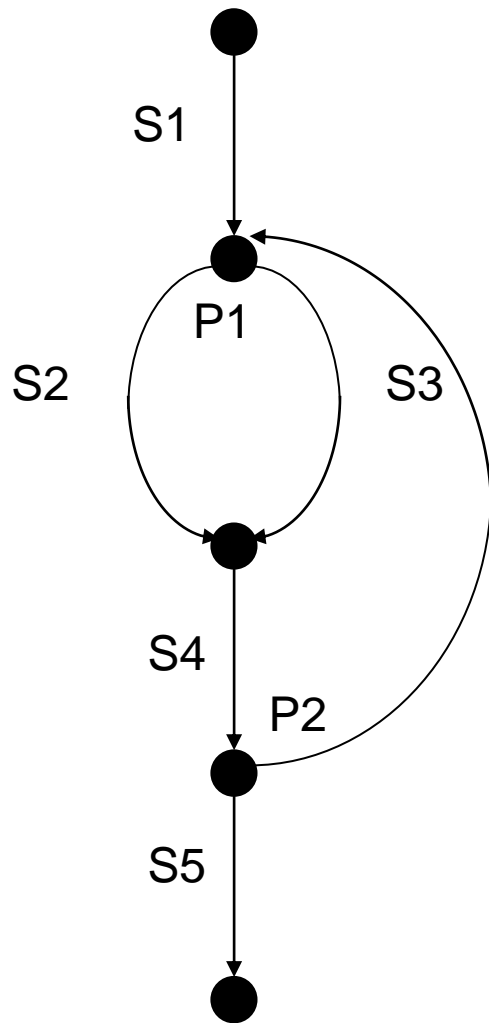
Nested decisions



```
S1;  
IF P1 THEN S2 ELSE  
    S3;  
    IF P2 THEN S4 ELSE S5  
FI  
S6;
```

$v(G) = 3$, while $\text{Max} = 4$.
Three test case will cover all paths.

Problem – the loop



```
S1;  
DO  
    IF P1 THEN S2 ELSE S3;  
    S4  
OD UNTIL P2  
S5;
```

Not a **Directed Acyclic Graph**
 $v(G) = 3$ and Max
is 4 but there is an “infinite”
number of paths

Problem – the loop (cont')

- Loops are the great problem in white box testing
- It is common practice to test the system going through each loop
 - 0 times – loop code never executed
 - 1 time – loop code executed once
 - 5 times – loop code executed several times
 - 20 times – loop code executed “many” times

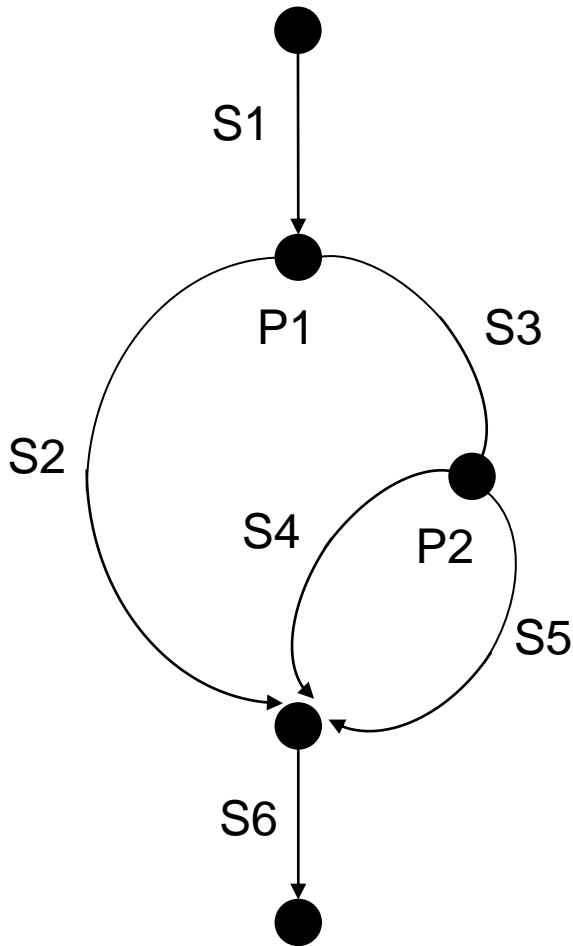
Use decision table to generate test cases for path coverage

- Make a table of all predicates
- Insert all combinations of True / False/Switch – 1 / 0/many – for each predicate
- Construct a test for each combination
- A general technique used to achieve full path coverage
- However, in many cases, it may lead to over-testing

Using a decision table

P1	P2	P3	Test description or reference
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

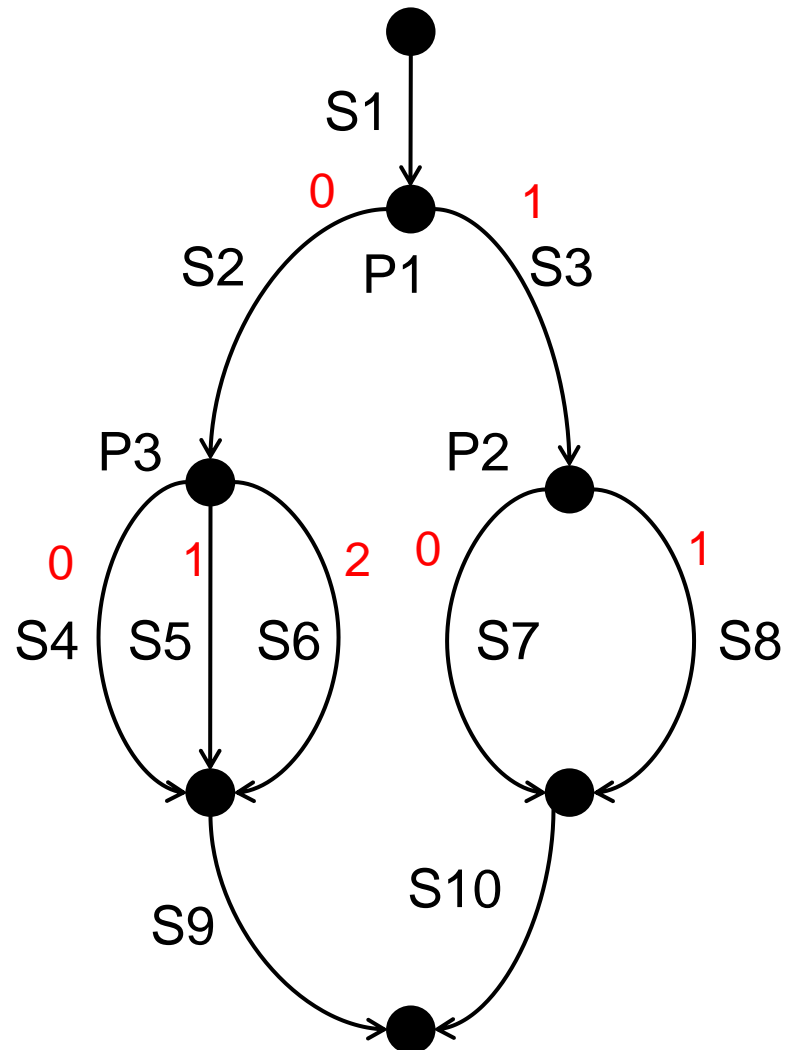
Decision table example – binary



P1	P2	Test description or reference
0	0	S1, S3, S5, S6
0	1	S1, S3, S4, S6
1	0	S1, S2, S6
1	1	S1, S2, S6

Go to **www.menti.com** and use the code **5729 3680**

Exercise



Outline

- Control flow testing and coverage
- Data flow testing and coverage
- Mutation testing and coverage

Data flow testing

- Data flow testing uses the **control flow graph** to explore the **unreasonable** things that can happen to data (data flow anomalies)
- Data flow anomalies are detected based on the associations between definition, initialization, and use of variables, e.g.,
 - Variables are used without being initialized

Variable lifecycle

- Variables have a life cycle
 - Defined and initialized
 - A value is bound to the variable
 - E.g., $x = \dots$
 - Used
 - The value of the variable is referred
 - Predicate use (**p-use**), E.g., if ($x > 10$)
 - Computational use (**c-use**). E.g., $y = x + 1$;
 - Killed (destroyed)

Variable lifecycle (cont')

```
{           // begin outer block
  int x = 1; // x is defined as an integer within this outer block
  ...;      // x can be accessed here
  {         // begin inner block
    int y = 2; // y is defined within this inner block
    ...;      // both x and y can be accessed here
  }         // y is automatically destroyed at the end of this block
  ...;      // x can still be accessed, but y is gone
}           // x is automatically destroyed
```


Data flow anomalies and normal use

- **dd: define and then define again – suspicious**
- **dk: define and then kill – potential bug**
- **ku: kill and then used – serious defect**
- **kk: kill and then kill again – potential bug**
- du: define and then use – OK
- kd: kill and then redefine – OK
- ud: use and then redefine – OK
- uk: use and then kill – OK
- uu: use and then use – OK
- ...

Why data-flow testing?

- To uncover possible bugs in data usage during the execution of the code
- Test cases are created to trace every definition to each of its use, and every use is traced to each of its definition
- Various coverage strategies are employed for the creating of the test cases

Data-flow test strategies

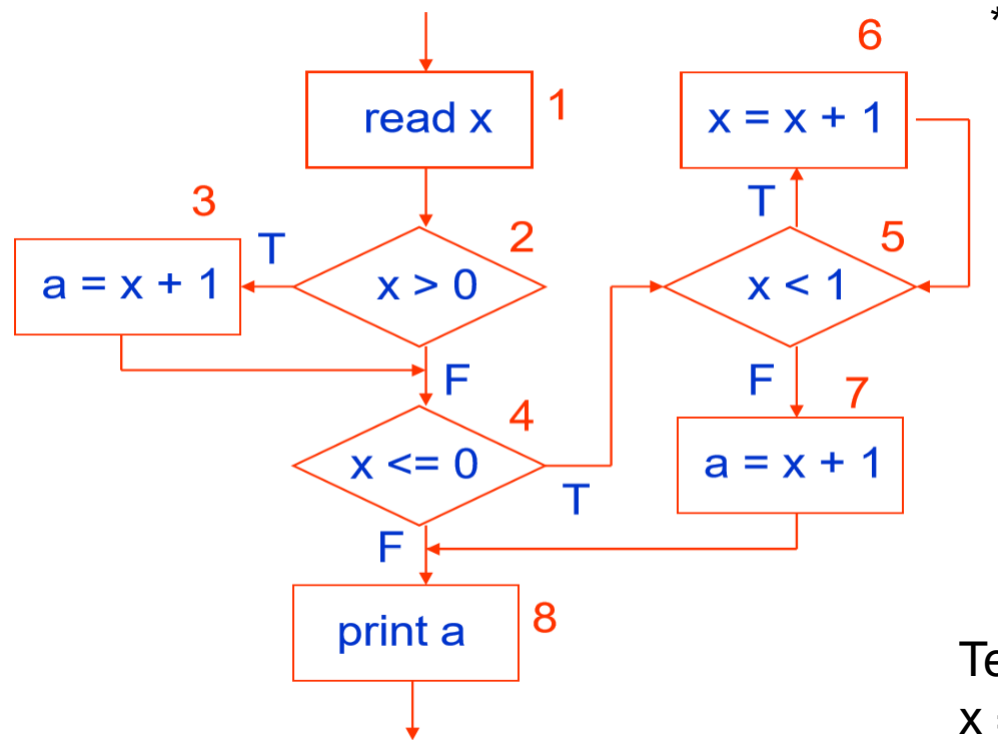
- **All definitions (AD)**: every definition of every variable should be covered by **at least one use** (c-use or p-use) of that variable
- **All computational uses (ACU)**: for every variable, there is a path from **every definition** to **every c-use** of that definition
- **All predicate-uses (APU)**: for every variable, there is a path from **every definition** to **every p-use** of that definition
- **All c-uses/some p-uses (ACU+P)**: for every variable and **every definition** of that variable, at least one path from the definition to **every c-use** should be included. If there are definitions of the variable with **no c-use** following it, then **add p-use** test cases to cover every definition

Data-flow test strategies (cont')

- **All p-use/some c-use** (APU+C): for every variable and **every definition** of that variable, at least one path from the definition to **every p-use** should be included. If there are definitions of the variable with **no p-use** following it, then **add c-use** test cases to cover every definition
- **All uses** (AU): for every variable, there is a path from **every definition** to **every use** of that definition, i.e., APU + ACU
- **All du paths** (ADUP): test cases **cover every simple sub-path** from each variable definition to **every p-use** and **c-use** of that variable
- Note that the **“kill”** usage is not included in any of the test strategies

All definitions

- All definitions (AD): **every definition** of every variable should be covered by at least **one use** (c-use/p-use) of that variable.



AD

- (1, 2, x)
- (3, 8, a)
- (7, 8, a)
- (6, 5, x)

Test cases for AD coverage

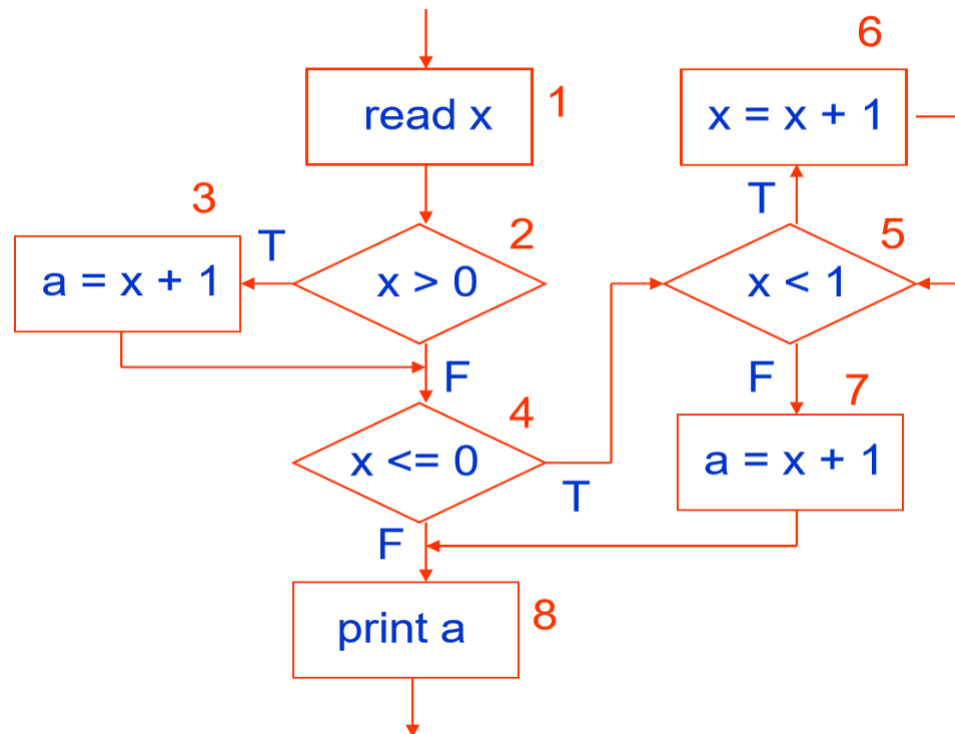
x = 0, a = any (1, 2, 4, 5, 6, 5, 7, 8)

x = 1, a = any (1, 2, 3, 4, 8)

*<https://www.cs.ccu.edu.tw/~naiwei/cs5812/st5.pdf>

All computational uses

- All computational uses (ACU): for **every** variable, there is a path from **every** definition to **every c-use** of that definition

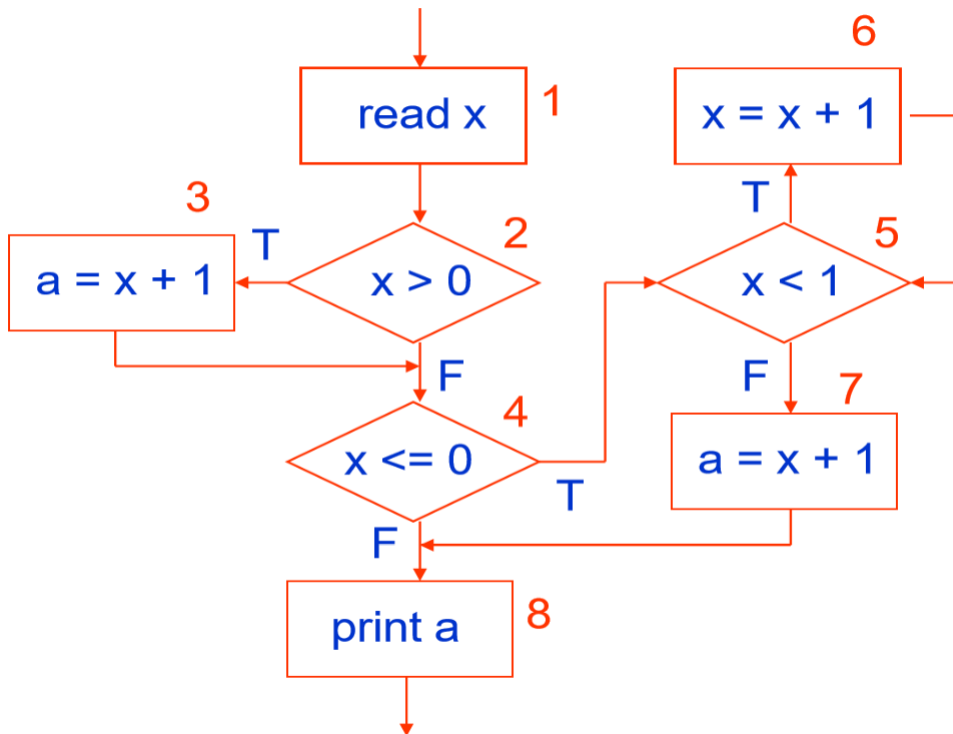


ACU

- (1, 3, x)
- (1, 6, x)
- (1, 7, x)
- (6, 6, x)
- (6, 7, x)
- (3, 8, a)
- (7, 8, a)

All predicate-uses

- All predicate-uses (APU): for **every** variable, there is a path from **every definition** to **every p-use** of that definition

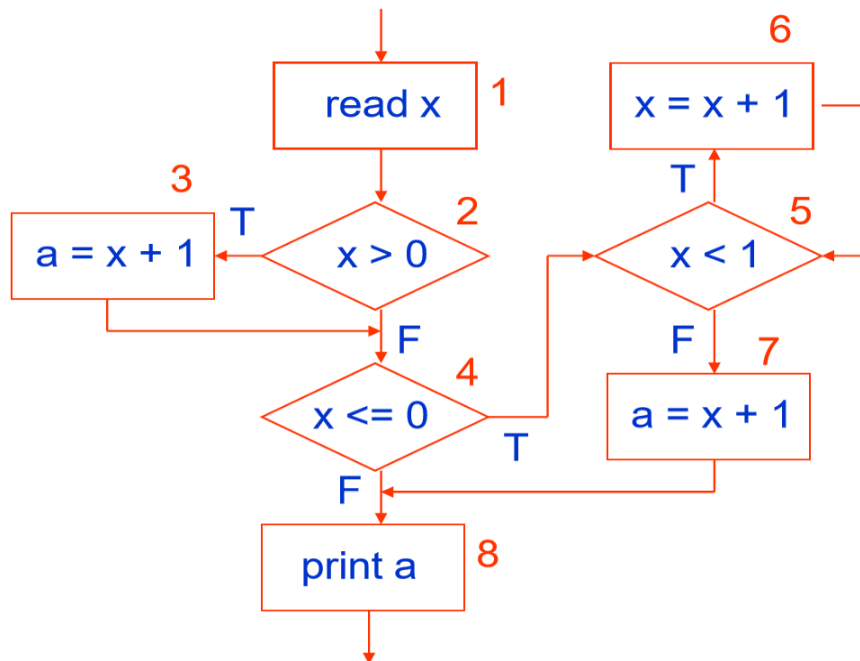


APU

- (1, 2, x)
- (1, 4, x)
- (1, 5, x)
- (6, 5, x)

All c-uses/some p-uses

- All c-uses/some p-uses (ACU+P): for **every** variable and **every definition** of that variable, at least one path from the definition to **every c-use** should be included. If there are definitions of the variable with **no c-use** following it, then **add p-use** test cases to cover every definition



ACU

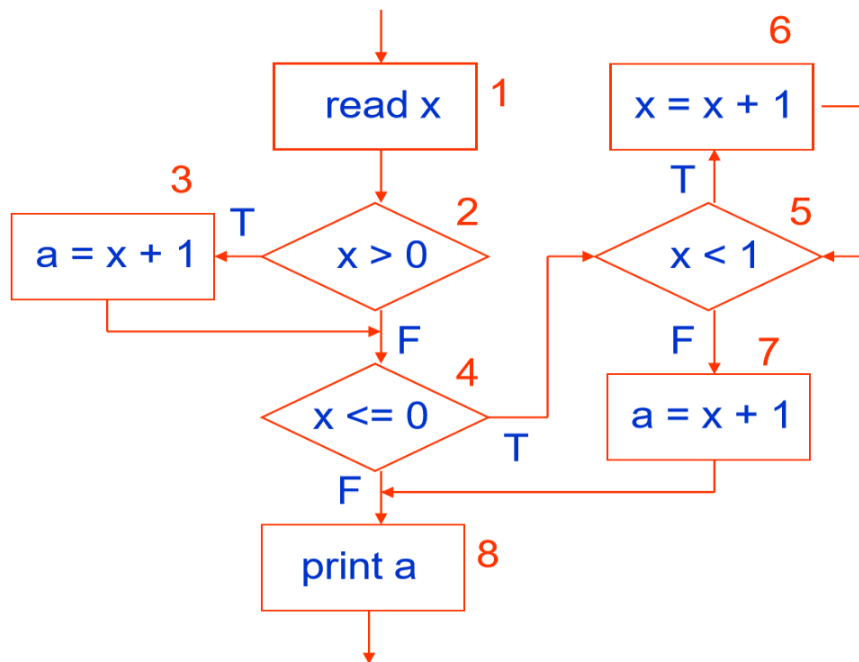
- (1, 3, x)
- (1, 6, x)
- (1, 7, x)
- (6, 6, x)
- (6, 7, x)
- (3, 8, a)
- (7, 8, a),

ACU + P

- (1, 3, x)
- (1, 6, x)
- (1, 7, x)
- (6, 6, x)
- (6, 7, x)
- (3, 8, a)
- (7, 8, a)

All p-use/some c-use

- All p-use/some c-use (APU+C): for every variable and **every definition** of that variable, at least one path from the definition to **every p-use** should be included. If there are definitions of the variable with **no p-use** following it, then **add c-use** test cases to cover every definition



APU

- (1, 2, x)
- (1, 4, x)
- (1, 5, x)
- (6, 5, x)

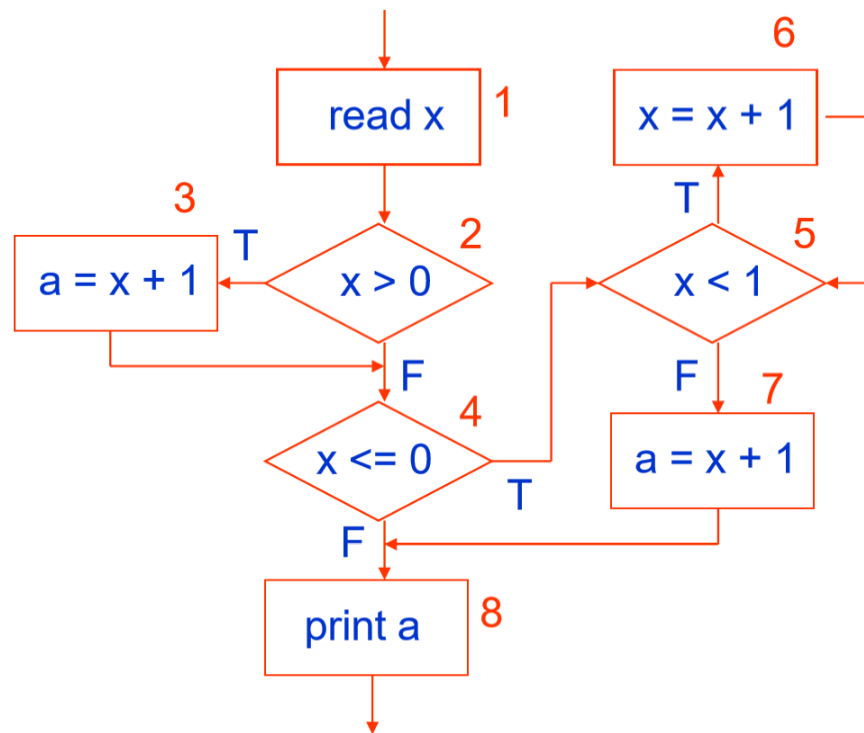
VS.

APU + C

- (1, 2, x)
- (1, 4, x)
- (1, 5, x)
- (6, 5, x)
- (3, 8, a)
- (7, 8, a)

All uses

- All uses (AU): for **every** variable, there is a path from **every definition** to **every use** of that definition, i.e., APU + ACU



AU

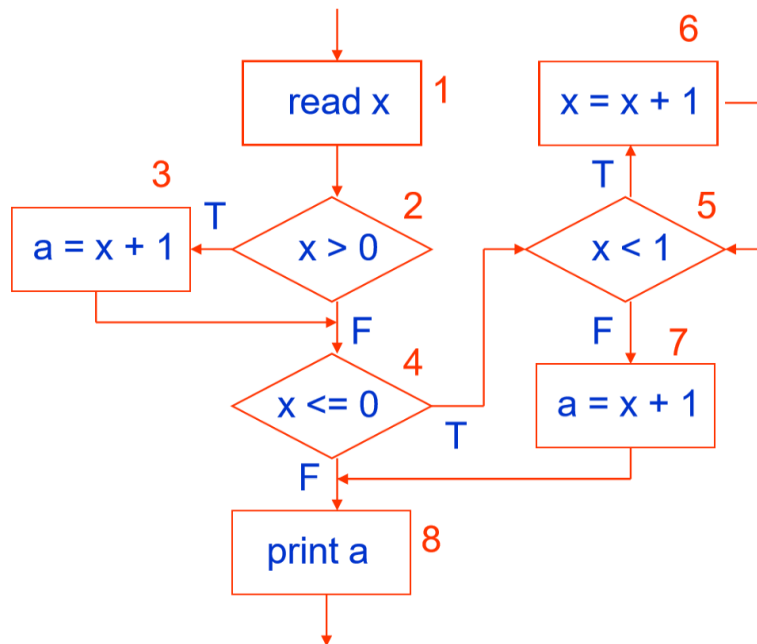
- (1, 2, x)
 - (1, 4, x)
 - (1, 5, x)
 - (6, 5, x)
- } APU

- (1, 3, x)
 - (1, 6, x)
 - (1, 7, x)
 - (6, 6, x)
 - (6, 7, x)
- } ACU

- (3, 8, a)
- (7, 8, a)

All du paths

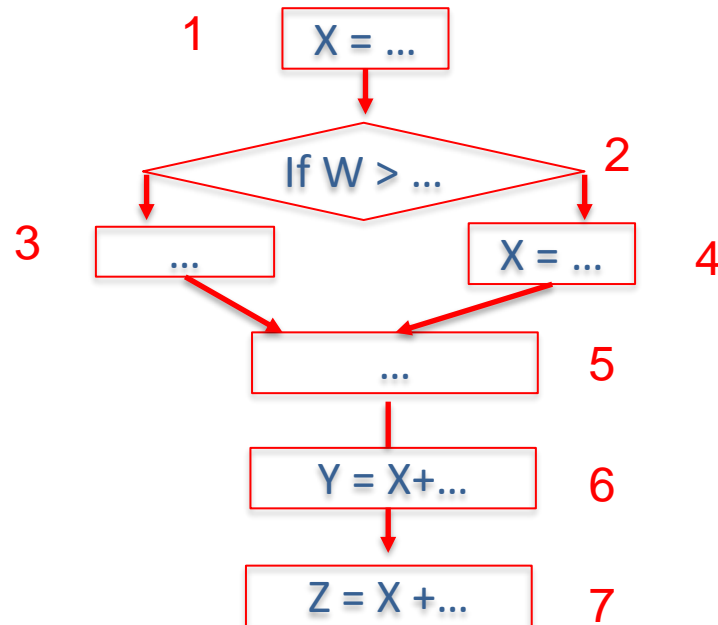
- All du paths (ADUP): More demanding than All uses (AU). If there are **multiple paths** between a given definition and a use, **they must all be included**
- However, ADUP includes loop just once



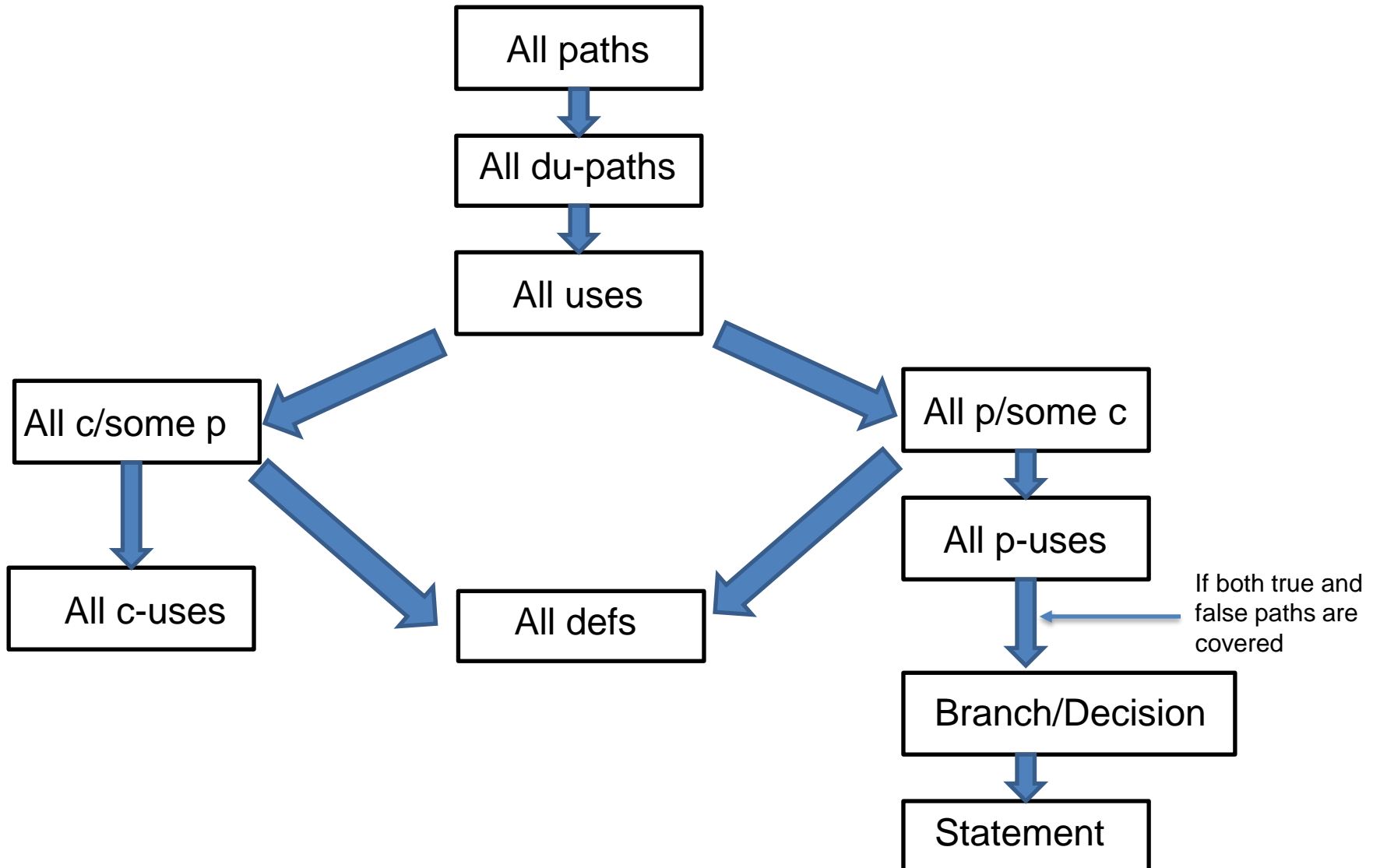
- In AU, we include path (3, 8, a), meaning either (3, 4, 5, 7, 8) **OR** (3, 4, 8) **OR** (3, 4, 5, 6, 5, 7, 8) is ok
- In ADUP, since there are multiple paths from 3 to 8, we need to include (3, 4, 5, 7, 8) **AND** (3, 4, 8) **AND** (3, 4, 5, 6, 5, 7, 8)

Go to **www.menti.com** and use the code **5729 3680**

Exercise



Relationship between strategies

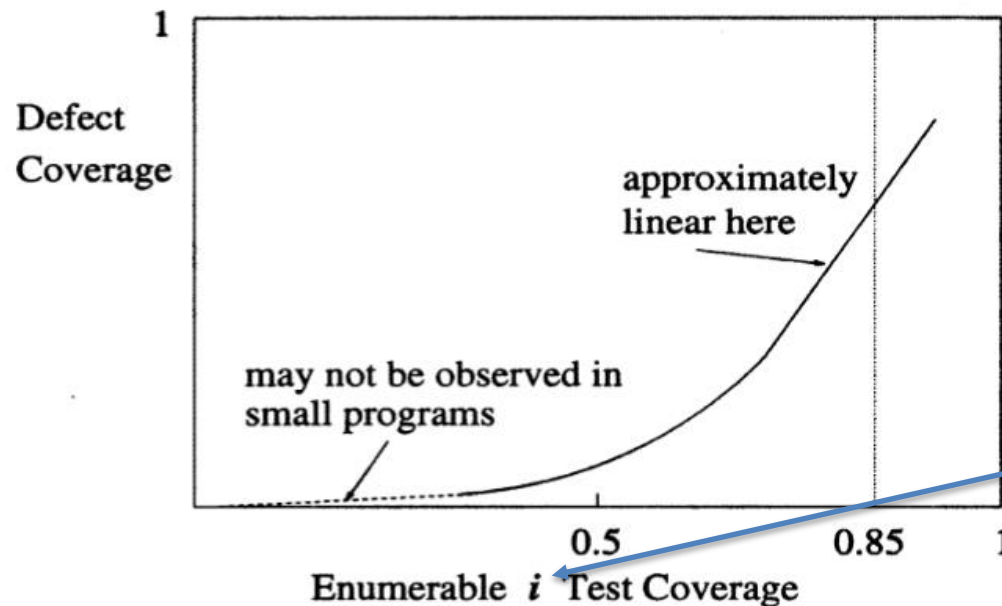


Why do we measure test coverage?

- One way of using statement/branch/path/dataflow coverage is to use it as **a test acceptance criteria**
 - Run test cases
 - Have we reached, e.g., 85% statement coverage
 - Yes – stop testing
 - No – coverage measure will help us directly identify untested code and indirectly help develop/select new test cases

Use test coverage to estimate defect coverage

- Defect coverage: the fraction of actual defects initially present that would be detected by a given test set



Enumerable i

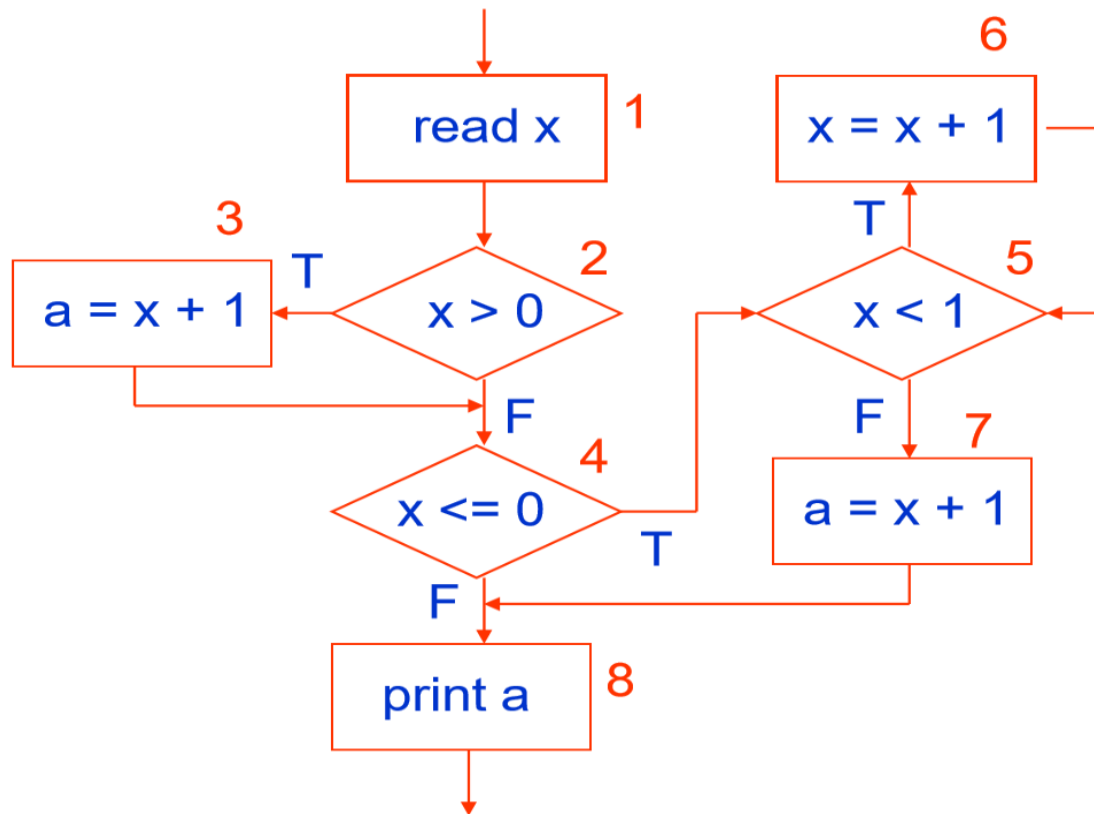
- Blocks/statements
- Branches
- C-uses
- P-uses

Logarithmic-exponential relationship between test coverage and defect coverage

Use coverage for program slicing

- Static backward slicing
 - A slice with respect to a variable v at a certain point p in the program is **the set of statements** that **contributes** to the value of the variable v at p
 - $S(v, p)$ denotes the set of **nodes** in the control flow graph that **contributes to the value of the variable v at point p**

Static backward slicing



$S(x, 1) = \{1\}$

$S(x, 2) = \{1\}$

$S(x, 3) = \{1, 2\}$

$S(x, 4) = \{1, 2\}$

$S(x, 5) = \{1, 2, 4\}$

$S(x, 6) = \{1, 2, 4, 5, 6\}$

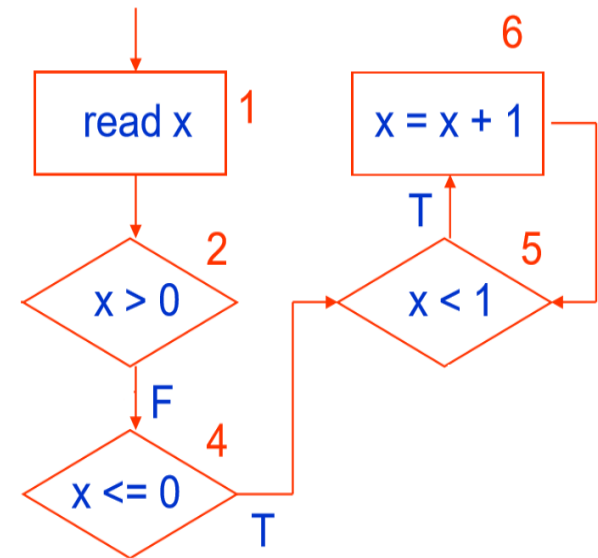
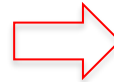
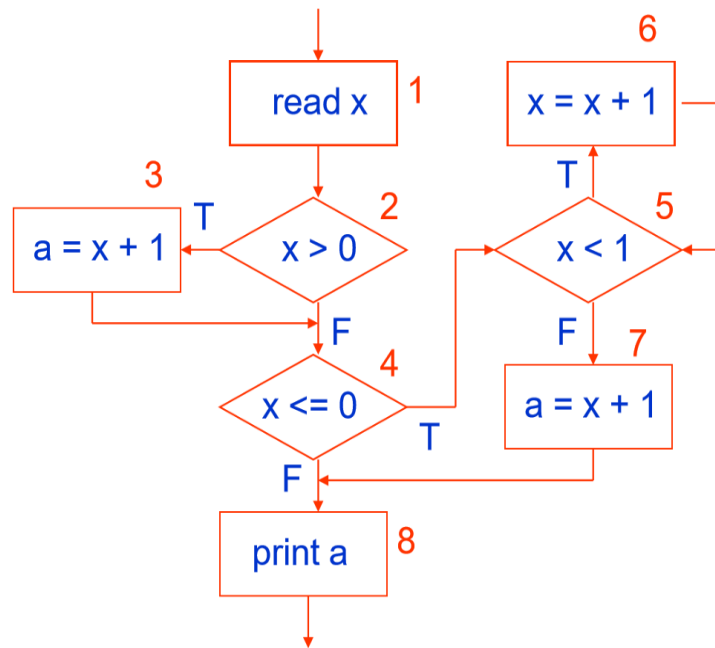
$S(x, 7) = \{1, 2, 4, 5, 6\}$

All **Defs** and All **P-uses** of a variable **v** before a certain point **p** are used for slicing*

* <http://www.cs.cmu.edu/afs/cs/academic/class/17654-f01/www/refs/Binkley.pdf>

Program slicing can help many activities

- Facilitate debugging (Finding and localizing errors)
- Facilitate programming understanding



$$S(x, 6) = \{1, 2, 4, 5, 6\}$$

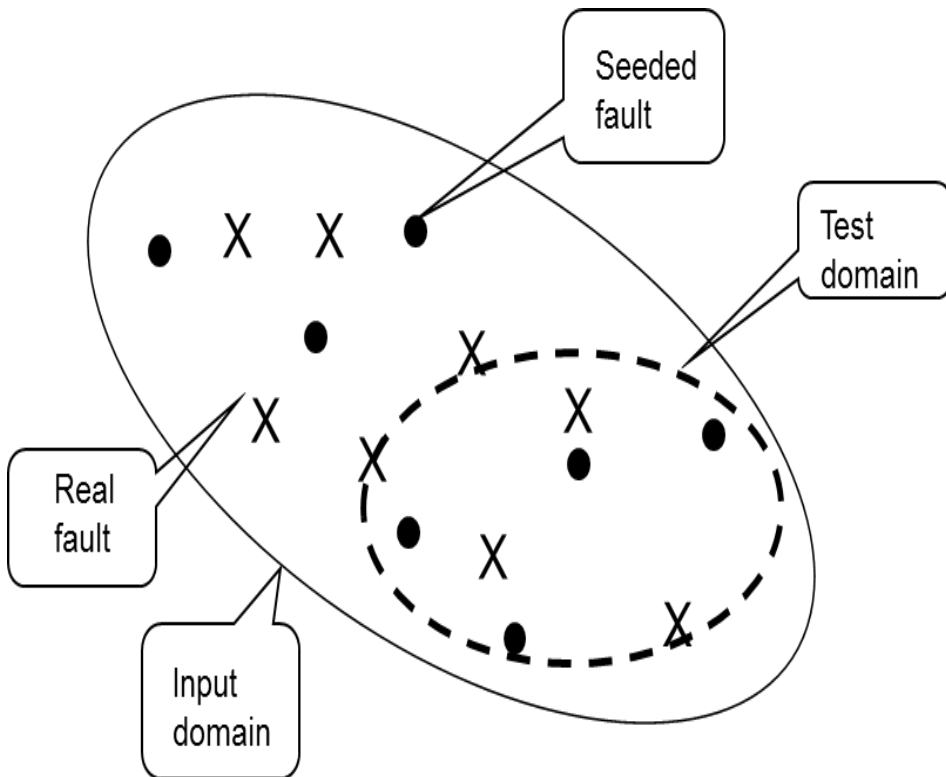
Outline

- Control flow testing and coverage
- Data flow testing and coverage
- Mutation testing and coverage

Mutation testing and coverage

- The tester can also measure **coverage of seeded faults** and use it as an indicator to measure whether the test set is adequate
 - Seed faults in the code
 - Have we found all the seeded faults
 - Yes – test set is adequate
 - No – One or more seeded faults are not found.
This tells us which **locations and types of defects** have not been covered by test cases sufficiently.
Then we need to develop/select more test cases.

Fault seeding and estimation



- We assume
$$N_0 / N = S_0 / S$$
- Thus
$$N_0 = N * S_0 / S,$$
- N_0 : number of faults in the code
- N : number of faults found using a specified test set
- S_0 : number of seeded faults
- S : number of seeded faults found using a specified test set

Typical mutants

- Where and how to seed the faults
 - Save and seed faults identified during earlier project activities
 - Draw faults to seed from an experience database containing typical faults
 - Arithmetic operator replacement
 - Logical connector replacement
 - Relational operator replacement
 - Absolute value insertion, etc.
- More info. can be found at
 - <http://www.uio.no/studier/emner/matnat/ifi/INF4290/v10/undervisningsmateriale/INF4290-Mutest.pdf>
 - <https://www.softwaretestinghelp.com/what-is-mutation-testing/>
 - https://pitest.org/java_mutation_testing_systems/

Code to be tested

- This is what the code might look like:
 - 1) Read Age*
 - 2) If age>14*
 - 3) Doctor= General Physician()*
 - 4) End if*

Seed faults (mutants)

- Mutation type 1: Relational operator replacement

Mutant #1:

- 1) *Read Age*
- 2) *If age<14 'Changing the > with <'*
- 3) *Doctor= General Physician()*
- 4) *End if*

Mutant #2:

- 1) *Read Age*
- 2) *If age=14 'Changing the > with ='*
- 3) *Doctor= General Physician()*
- 4) *End if*

Mutation coverage

- If our test inputs are 14 and 15. Which of these mutants will be killed?

Changing
> with <

Changing >
with =

Test inputs	Expected result	Mutant 1	Mutant 2
14	GP not assigned	Succeeds - GP not assigned	Fails - GP assigned
15	GP assigned	Fails - GP not assigned	Fails - GP not assigned

- 14 finds a failure when runs again mutant 2, not effective against mutant 1
- 15 finds failures when runs again mutants 1 and 2

Summary

- We studied
 - Control flow test and related coverage measurement
 - Data flow test and related coverage measurement
 - Mutation test and coverage measurement

Next lecture

- Black-box unit testing