Software Testing

Lecture 3 – Clear Testing Goals

The Goals of Testing

Question – what goals might you have when testing?

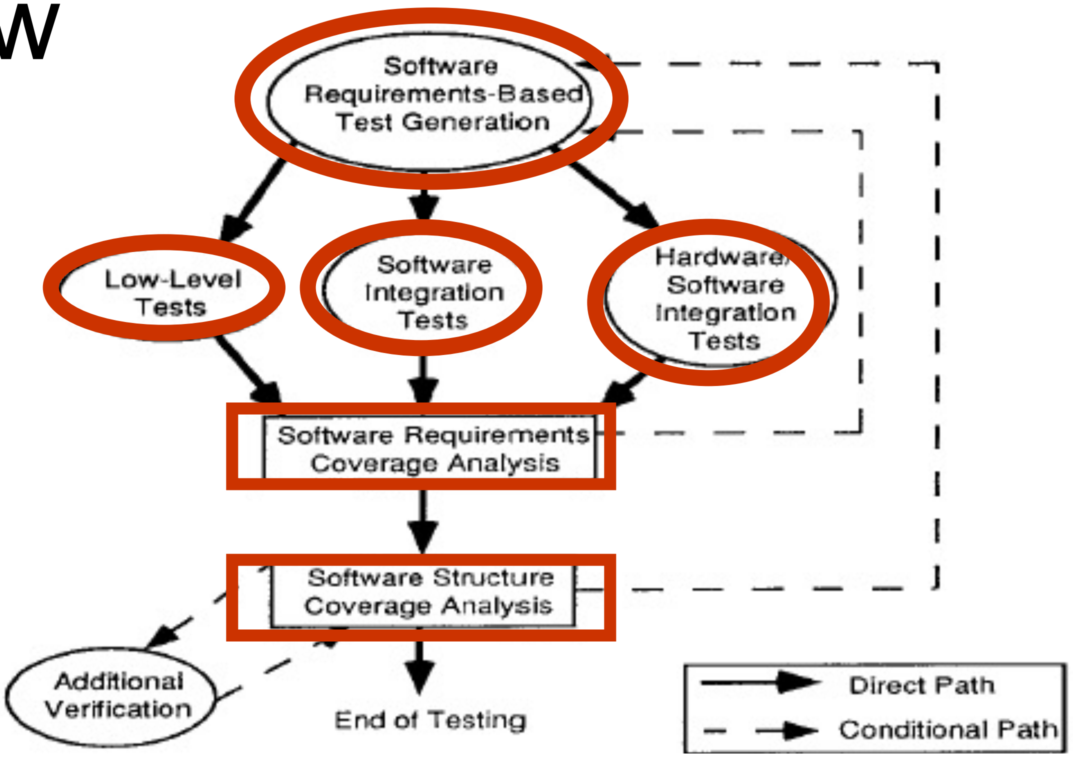
- Find the maximum number of bugs
- Know whether you have undiscovered bugs
- Comply with regulator-set standards
- Have a compelling defence in a court case
- Do any of the above in the minimum time and cost
- …

Industrial perspective: airborne software

- DO178C: Software Considerations in Airborne Systems and Equipment Certification

- Two complementary objectives for testing
  - To demonstrate that the software satisfies its requirements
  - To demonstrate with **a high degree of confidence** that errors which could lead to unacceptable failure conditions, as determined by the system safety assessment process, have been removed

Testing Overview


Software Requirements-Based Test Generation
Low-Level Tests
Software Integration Tests
Hardware/ Software Integration Tests
Software Requirements Coverage Analysis
Software Structure Coverage Analysis
Additional Verification
End of Testing
Direct Path
Conditional Path

[DO-178]

Typical Testing Requirements

- **Testing Levels**
  - Low-level testing: Ensures software components satisfy low-level and derived requirements
  - Software integration testing: Ensures software components interact correctly with each other and satisfy software requirements and software architecture
  - Hardware/software integration testing: Ensures correct operation of software in target computer environment

- **Test Cases**
  - Normal range
  - Robustness

- **Coverage Analysis**
  - Requirements-based coverage analysis: how well requirements-based testing verified implementation of software requirements
  - Structural coverage analysis: which code structures not exercised by the requirements-based test procedures

When is enough testing enough?

- Enough implies judgement
- Reliance on structural coverage as one measure of the “completeness” of testing
- Coverage is a measure, not a method or test technique

[Beizer 1983]

Software Testing Metrics

Definition

Software Metrics

A system or standard of measurement of software, which gives a numerical value to some property of interest.

Example Metrics

- LOC (ugh)
- Lead time: time to deliver a new feature idea in working software.
- Cycle time: time to implement a change in production software.
- Velocity: how many “software units” (e.g., classes, features) are completed in an iteration or sprint.
- Code churn: LOC modified/added/deleted in a specified period of time.
- MTBF: mean time between failures
- MTTR: mean time to repair (e.g., a security breach)
- Defect Removal Efficiency (after delivery).

Exhaustive testing is best

- Exhaustive coverage of possible inputs
  - (and input *sequences*)
- Exhaustive coverage of possible context states
  - E.g. operating system state
  - E.g. hardware state
  - E.g. (for reactive systems) external world state

Exhaustive Testing

- For exhaustive testing, *2^n tests are required* for a decision with *n inputs.*
- *Question 1 : Consider an expression with 36 inputs. How much time would it take to* execute all of the test *cases required for exhaustive testing* of this expression if you *could run 100 test cases per second?*
- Answer 1: (2^{36} tests)*(1 sec/100 tests)*(1 minute/60 sec)*(1 hour/60 min)*(1 day/24 hour)*(1 year/365 day) = ***Approximately 21.79 years***

Definition

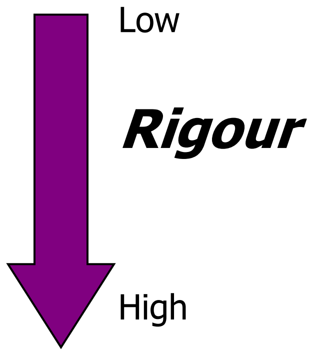
Coverage

*A measure of the **proportion** of a **structure** or a **domain** that a program, a test case, a test suite exercises.*

Assurance & Software Coverage

- Rather than directly demonstrating the satisfaction of the requirements to the required ***failure rate***, testing is used to provide ***confidence*** that the these requirements have been ***sufficiently*** satisfied
  - Rigour of testing is proportionate to failure rate
    - Lower failure rates → more rigorous testing

- Code test coverage is typically used as a criterion for sufficiency of testing, e.g.
  - Statement coverage
  - Decision coverage
  - Condition coverage
  - Condition/decision coverage
  - Modified condition/decision coverage



Assurance and test coverage for airborne software

Level D
(28 objectives)

Planning
CM
QA
HL req Coverage
HL req robustness
Code target
Cert liaison
Tool qual

Level C
(57 objectives)
Level D
+
More planning
Verif req, design,
integ processes
Test LL req
Verif test plan, proc
& results
LL req coverage
Statement coverage
Data & control
coverage

Level B
(65 objectives)
Level C
+
Artefact compatibility
Verifiability
Independence
Decision coverage
Transition

Level A
(66 objectives)
Level B
+
MC/DC coverage
More independence
Source to object

Assurance and test coverage for automotive applications

- Objective of testing: to demonstrate that the software units fulfill the software unit specifications and do not contain undesired functionality

- Analysis of structural coverage can reveal shortcomings in requirement-based test cases, inadequacies in requirements, dead code, deactivated code or unintended functionality

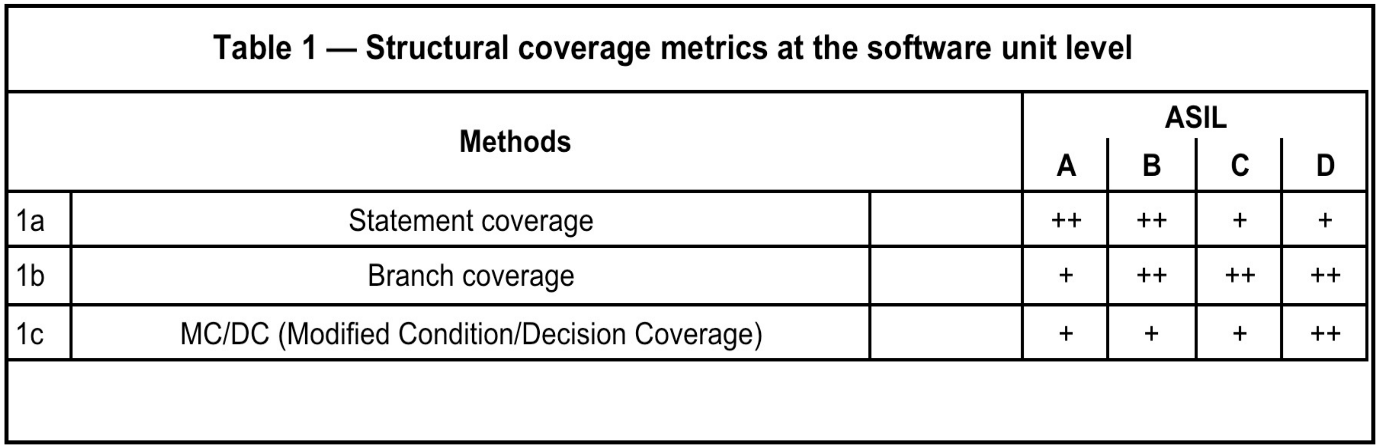


Table 1 — Structural coverage metrics at the software unit level

Methods			ASIL			
			A	B	C	D
1a	Statement coverage		++	++	+	+
1b	Branch coverage		+	++	++	++
1c	MC/DC (Modified Condition/Decision Coverage)		+	+	+	++

[ISO26262]

Let’s look at a selection of coverage criteria (from the perspective of measurement)

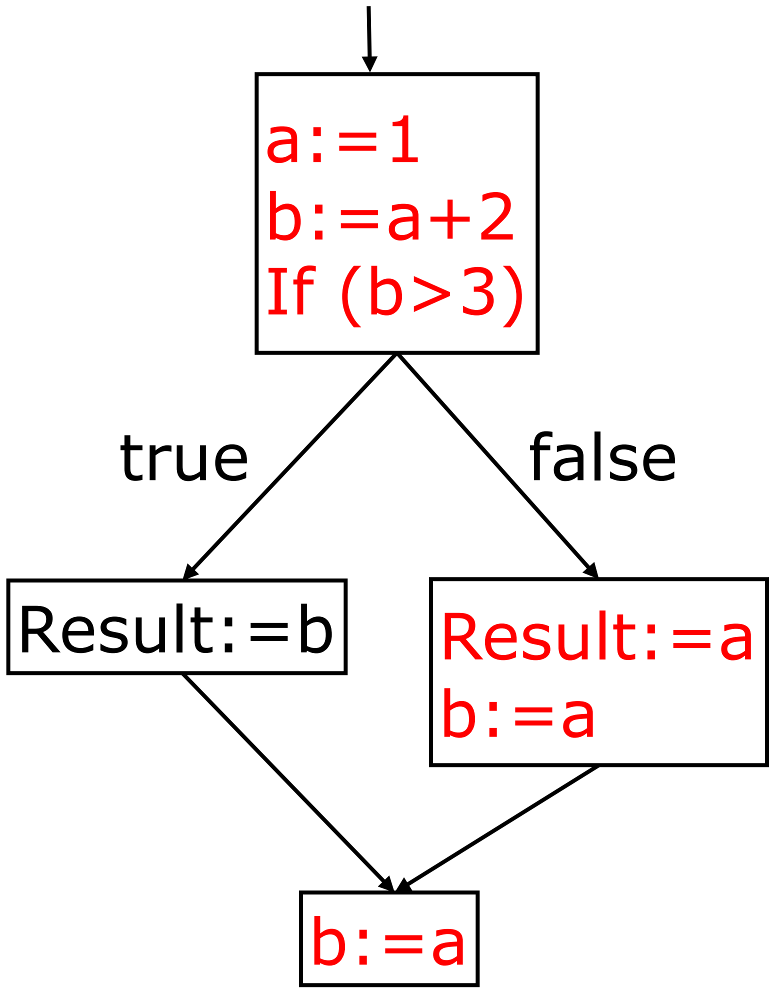
What’s a desirable coverage criterion?

Path Coverage

A test suite achieves path coverage if it results in each path through the program being taken at least once.

Example

Coverage of the red path:
50%
(1/2 paths)



Exercise: minimum number of test cases to cover all paths

```
public static int max(boolean considerA, int a,
int b, int c) {
    if (considerA && a>b) {
        if (a>c)
            return a;
        else
            return c;
    } else
        if (b>c)
            return b;
        else
            return c;
}
```

Question - Is path coverage the same as exhaustive testing?

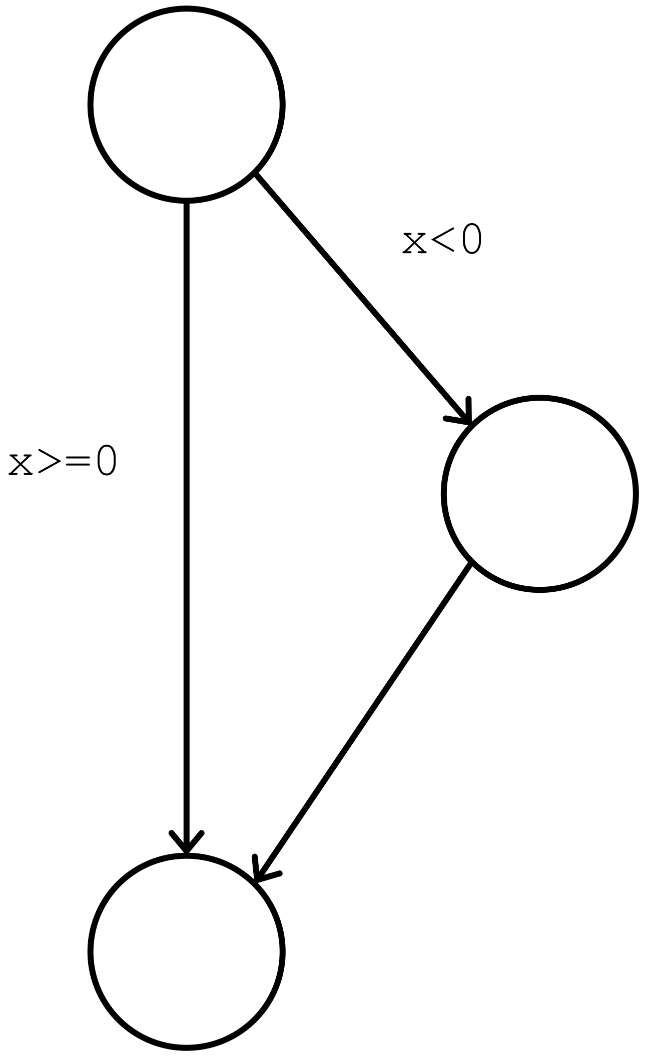
- No
- Path coverage ensures that each control flow path has been seen at least once
- **But** it doesn’t ensure that each path is tried with *all possible variable values*

Let’s scale our ambitions *right down*

Statement Coverage

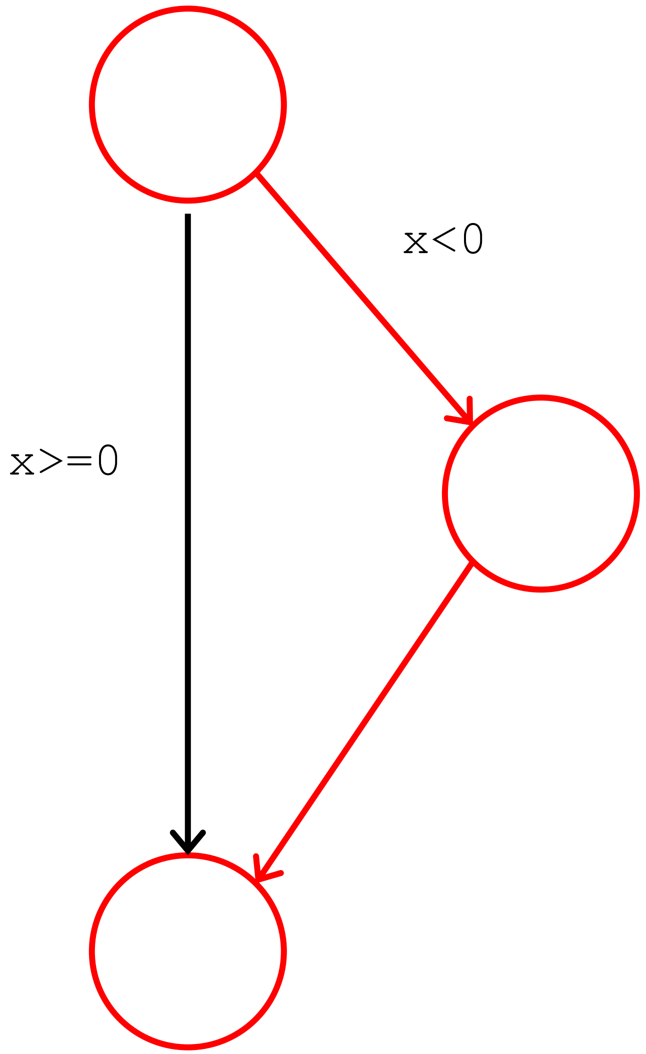
A test suite achieves statement coverage if it results in each statement in the program being run at least once

Statement coverage is very weak



```
int abs(int x) {
    int ans = x;
    if (x < 0)
        ans = -x;
    return ans;
}
```

Statement coverage is very weak

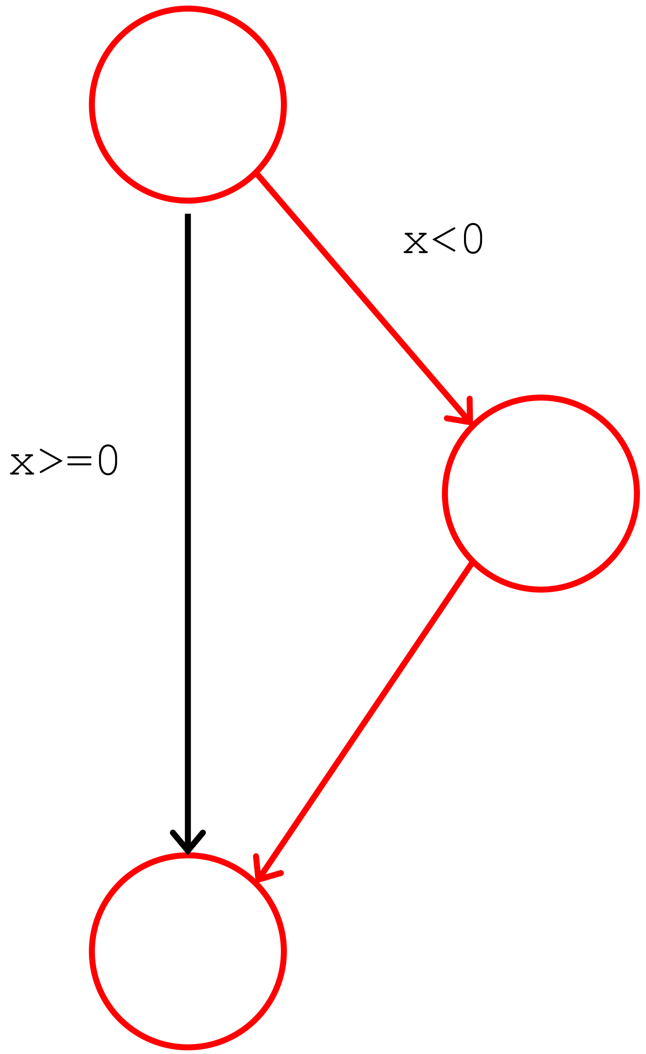


```
int abs(int x) {
    int ans = x;
    if (x < 0)
        ans = -x;
    return ans;
}
```

`Test case: abs(-4)`

Achieved statement coverage

Statement coverage is very weak



```
int abs(int x) {
    int ans = -x;
    if (x < 0)
        ans = -x;
    return ans;
}
```

`Test case: abs(-4)`

Achieved statement coverage

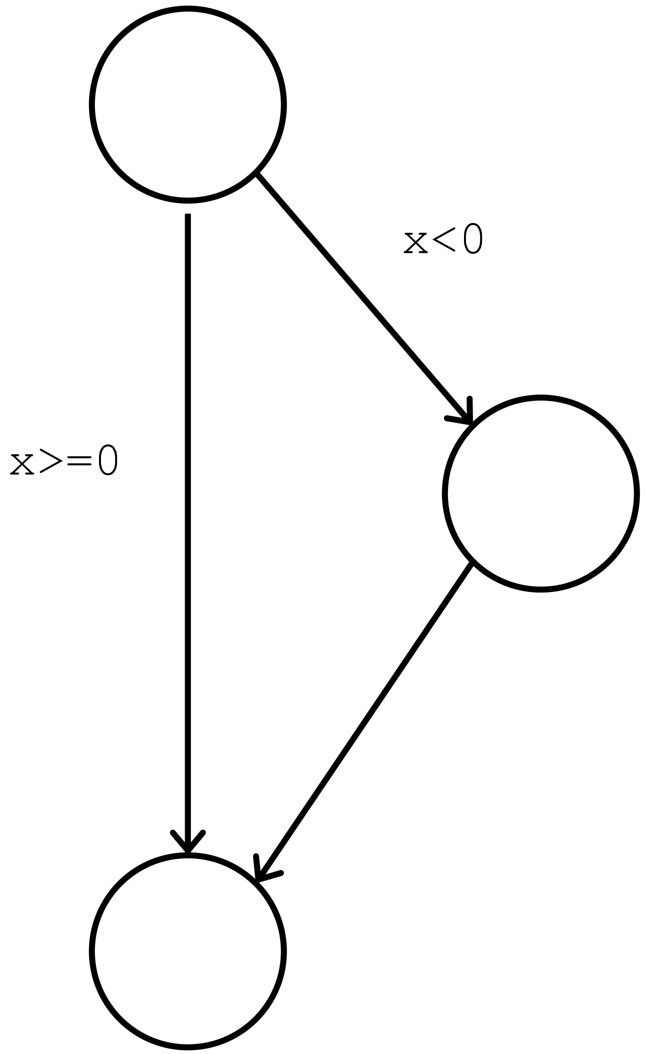
Didn’t test half the inputs at all

A compromise position

Branch Coverage

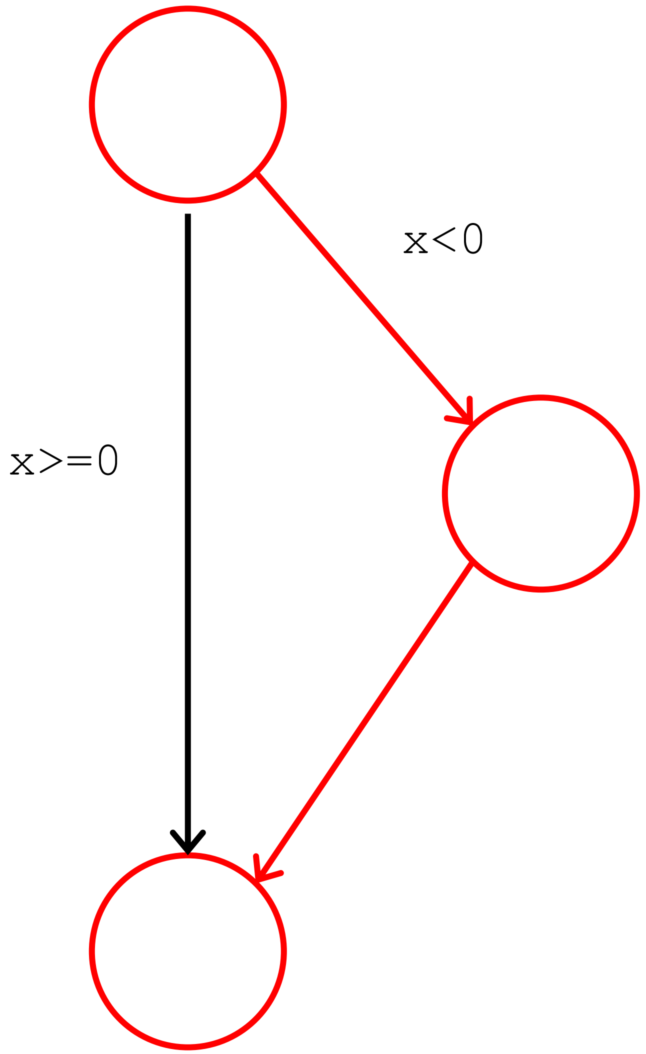
A test suite achieves branch coverage if it results in each branch in the program being taken at least once

Branch Coverage Example



```
int abs(int x) {
    int ans = x;
    if (x < 0)
        ans = -x;
    return ans;
}
```

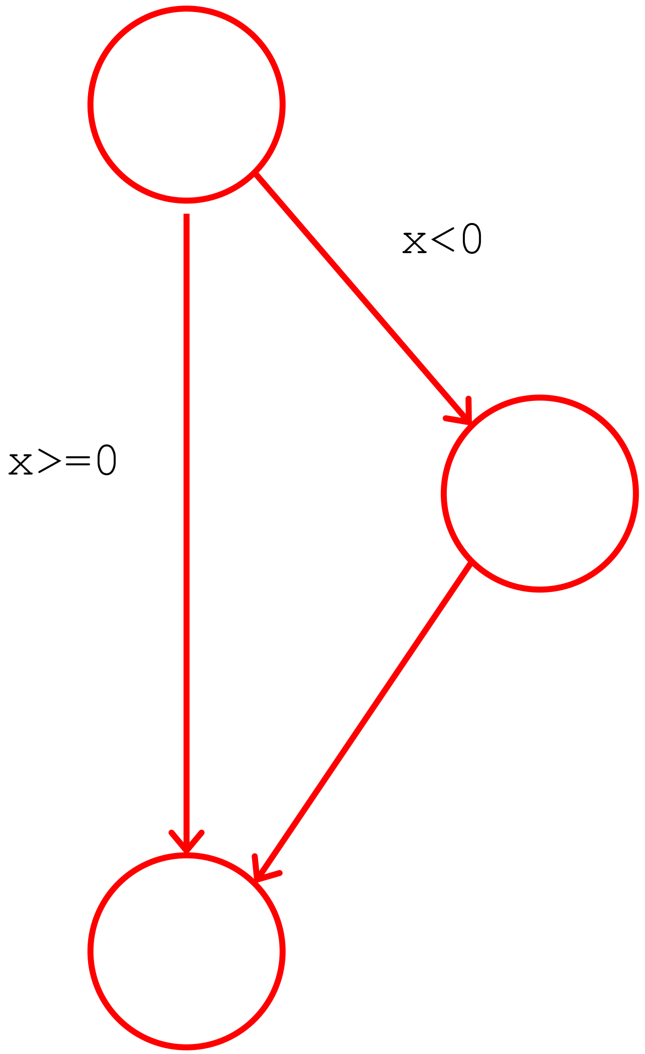
Branch coverage often beats statement coverage



```
int abs(int x) {
    int ans = x;
    if (x < 0)
        ans = -x;
    return ans;
}
```

Test case 1: abs(-4)

Branch coverage often beats statement coverage

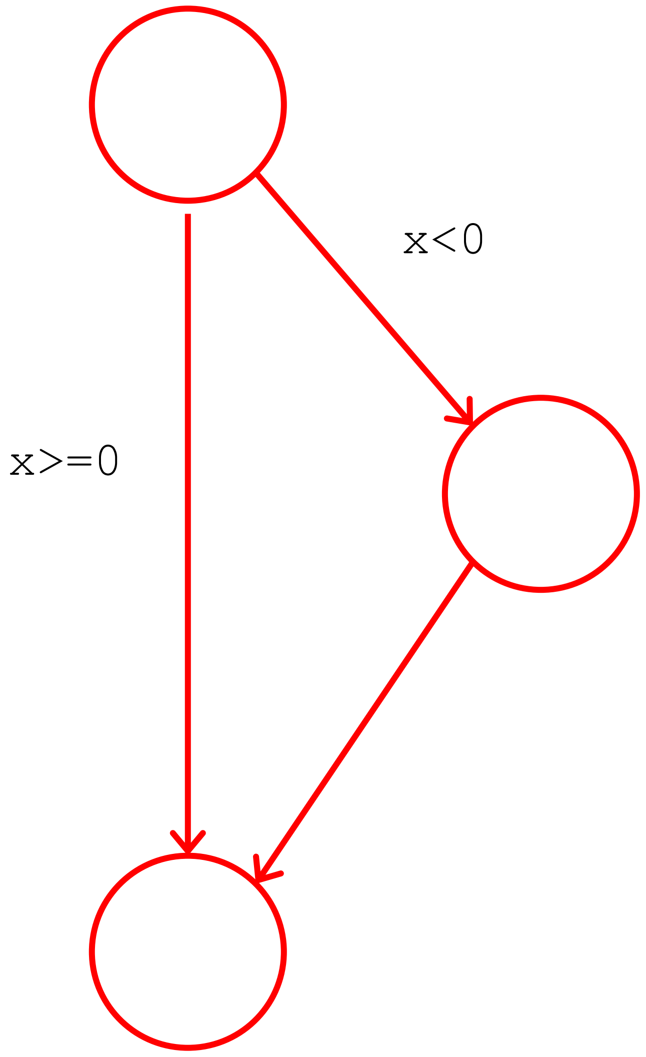


```
int abs(int x) {
    int ans = x;
    if (x < 0)
        ans = -x;
    return ans;
}
```

```
Test case 1: abs(-4)
Test case 2: abs(4)
```

Achieved branch coverage

Branch coverage often beats statement coverage



```
int abs(int x) {
    int ans = -x;
    if (x < 0)
        ans = -x;
    return ans;
}
```

```
Test case 1: abs(-4)
Test case 2: abs(4)
```

Achieved branch coverage

Question - Is branch coverage the same as path coverage?

- No

- Path coverage ensures that each control flow path has been seen at least once
- Branch coverage ensures that each individual *branch* has been taken
- …**but** there may be combinations of branches that haven’t been.

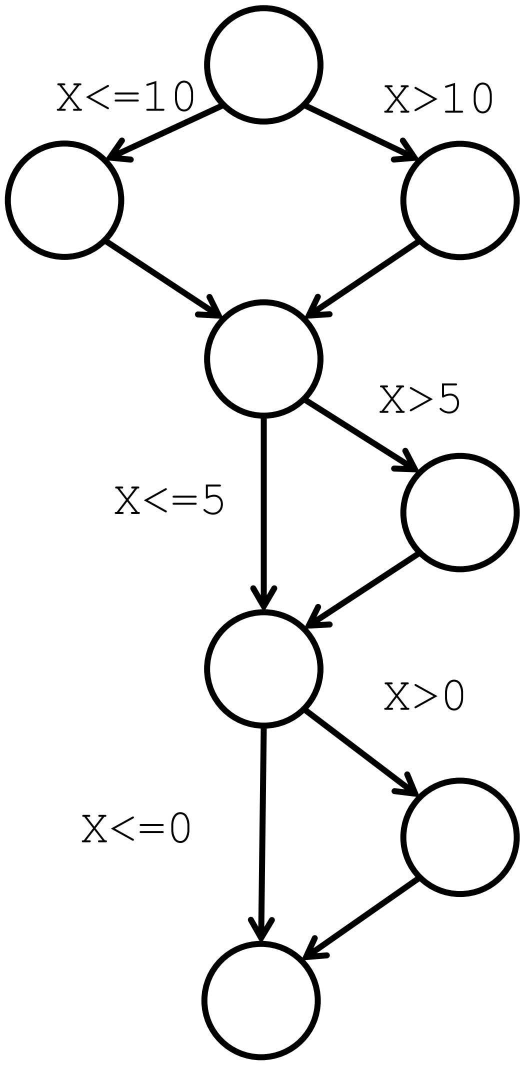
- Any questions so far?

Quick Quiz

1. Draw the CFG for this code
2. Find a test suite that achieves statement coverage for this code
3. Find a test suite that achieve branch coverage for this code
4. Can you find a test suite that achieves statement coverage but not branch coverage?
5. Is there a test suite that achieves branch coverage but not statement coverage?

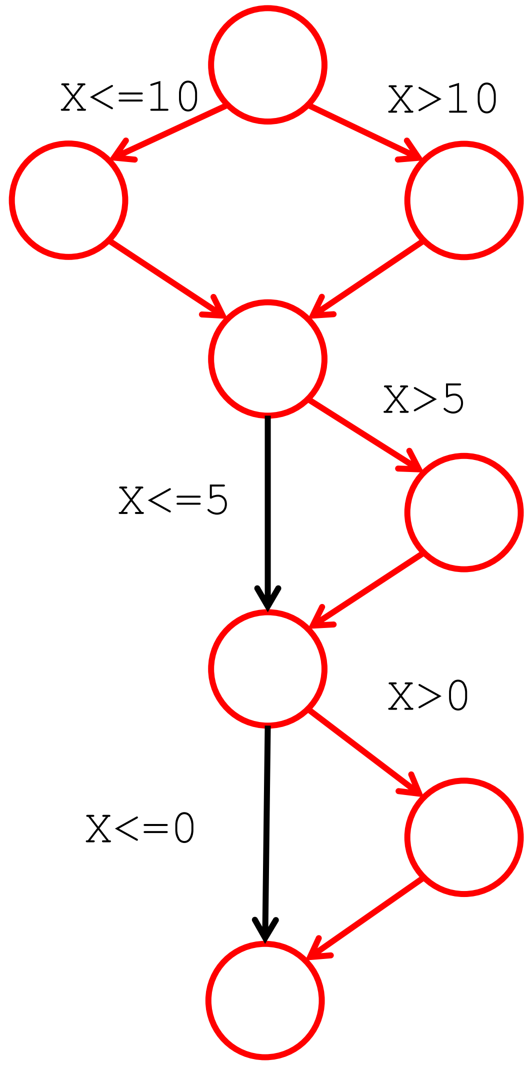
```
int function(int x){
        int y = 0;
        if (x > 10)
              y = 1;
        else
              y = 2;
        if (x > 5)
              y = y*2;
        if (x > 0)
              y = y*3;
        return y;
}
```

Quick Quiz – An Answer



```
int function(int x){
        int y = 0;
        if (x > 10)
                y = 1;
        else
                y = 2;
        if (x > 5)
                y = y*2;
        if (x > 0)
                y = y*3;
        return y;
}
```

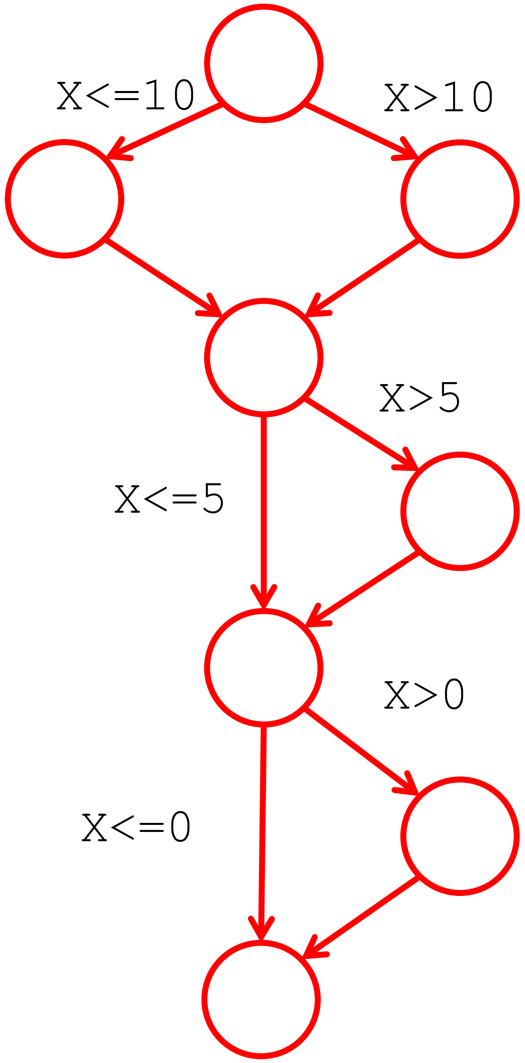
Quick Quiz – An Answer



```
int function(int x){
        int y = 0;
        if (x > 10)
                y = 1;
        else
                y = 2;
        if (x > 5)
                y = y*2;
        if (x > 0)
                y = y*3;
        return y;
}
function(20)
function(7)
```

Statement coverage but not branch coverage

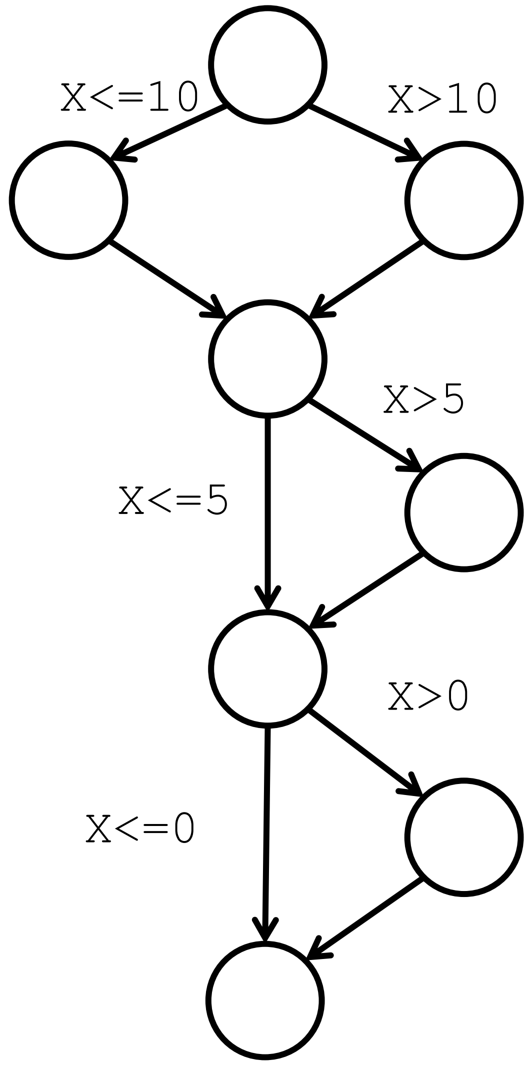
Quick Quiz – An Answer



```
int function(int x){
      int y = 0;
      if (x > 10)
             y = 1;
      else
             y = 2;
      if (x > 5)
             y = y*2;
      if (x > 0)
             y = y*3;
      return y;
}
function(20)
function(-3)
```

Statement coverage and branch coverage

Quick Quiz – An Answer



```
int function(int x){
      int y = 0;
      if (x > 10)
            y = 1;
      else
            y = 2;
      if (x > 5)
            y = y*2;
      if (x > 0)
            y = y*3;
      return y;
}
```

Impossible to achieve branch coverage without achieving statement coverage

A problem with branch coverage

```
if (averageMark >= 50 && hardFailCount == 0){
      grade = "pass";
} else {
      grade = "fail";
}
```

Branch coverage met by:

- (50,0) → “pass”
- (49,1) → “fail”

A problem with branch coverage

```
if (averageMark >= 50 || hardFailCount == 0){
        grade = "pass";
} else {
        grade = "fail";
}
```

Branch coverage met by:

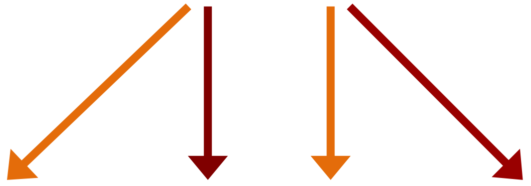
- (50,0) → “pass”
- (49,1) → “fail”

Let’s try something slightly different

Condition Coverage

A test suite achieves condition coverage if it results in each condition in each branching instruction being both true and false

```
if (a || b)
```



100% obtained with (a,b) = (true, true) and (false,false)

What does Condition Coverage require here?

```
if (averageMark >= 50 || hardFailCount == 0){
     grade = "pass";
} else {
     grade = "fail";
}
```

Condition coverage met by:

- (50,0) → “pass” (true, true)
- (49,1) → “fail”   (false, false)

Let’s add some more constraints

Combinatorial Coverage

A test suite achieves combinatorial coverage if it results in all the conditions in each branching instruction being tried in all combinations of truth values

What does Combinatorial Coverage require here?

```
if (averageMark >= 50 || hardFailCount == 0){
      grade = "pass";
} else {
      grade = "fail";
}
```

Combinatorial coverage met by:

- (50,0) → "pass" (true, true)
- (49,1) → "fail"   (false, false)
- (50,1) → "pass" (true, false)   **wrong behaviour – we found a bug!**
- (49,0) → "pass" (false, true)   **wrong behaviour – we found a bug!**

What does Combinatorial Coverage Require *here?*

```
if (averageMark >= 50 OR
  hardFailCount == 0 OR
  elephantCount > 3){
```

…

…and *here?*

```
if ( averageMark >= 50 OR
  hardFailCount == 0 OR
  elephantCount > 3 AND
  specialString.equals(“Treats”)
  ){
…
```

So, let’s try another compromise

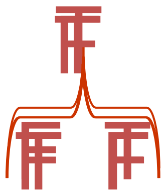
Modified Condition/Decision Coverage (MC/DC)

A test suite achieves MC/DC if

- It achieves 100% branch coverage
- It achieves 100% condition coverage
- Each entry/exit point is exercised
- Each condition affects the behaviour independently

Modified Condition/Decision Coverage (MC/DC)

- Every decision in program has taken all possible outcomes at least once
- Every condition in a decision in program has taken all possible outcomes at least once
- Every condition in a decision has been shown to independently affect that decision's outcome



  - If (A or B) then do something; end if;
  - Test cases ($\underline{T}$F), ($\underline{F}$T), and ($\underline{F}$F) meet coverage criterion

Another example

```
if ( (A || B) && C )
{
        /* super fun stuff */
}
else
{
        /* even more super fun stuff */
}
```

- *Condition coverage:* each of A, B, C evaluated true and false at least once. Tests: (1) A=true, B=true, C=true; (2) A=false, B=false, C=false
- *Decision coverage:* condition ((A||B)&&C) should be evaluated to true at least once and to false at least once. Same tests work.
- *MCDC:* changing value of only one condition changes outcome.
  - A = false, B = false, C = true ---> decision is evaluated to false
  - A = false, B = true, C = true ---> decision is evaluated to true
  - A = false, B = true, C = false ---> decision is evaluated to false
  - A = true, B = false, C = true ---> decision is evaluated to true

MC/DC

- For decisions with a large number of inputs, MC/DC requires considerably more test cases (typically *n+1* tests for *n* inputs)

- Meeting MC/DC is expensive and may constitute half testing cost
  - Some claim that it finds relatively few problems
  - Evidence for value of MC/DC in not clear

- MC/DC is sometimes used in a “check box” manner

Source vs. Object Code Coverage

- Structural coverage achieved at <u>source code</u> level can differ from that achieved at <u>object code</u> level
  - Depending on language and compiler features, multiple object code statements can be generated from a single source code statement
- Achieving MC/DC at source code level does not guarantee MC/DC at object code level, and vice versa
- In general, structural coverage analysis may be performed on the source code if equivalence in coverage can be shown
  - But for the highest level software, additional verification should be performed if the compiler generates object code not directly traceable to the source code statements

[DO248B]

Structural Coverage vs. Structural Testing

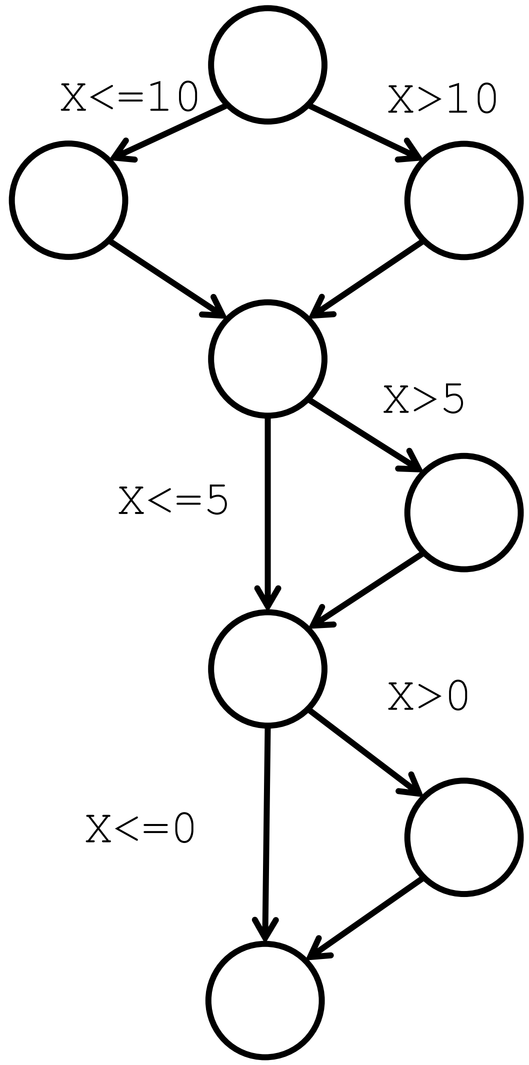
- There is a distinction between structural coverage analysis and structural testing:
  - The purpose of structural coverage analysis is to determine which code structure was not exercised by the requirements-based test procedures
  - Structural testing is the process of exercising software with test scenarios written from the source code, not from the requirements
- Structural testing provides no information about whether the code is doing what it is supposed to be doing as *specified* in the requirements
  - Also fails to assure that there are no unintended functions
    - Critical for safety-critical systems

[DO248B]

- Any questions?

Metric Subsumption

Quick Quiz – An Answer



```
int function(int x){
      int y = 0;
      if (x > 10)
            y = 1;
      else
            y = 2;
      if (x > 5)
            y = y*2;
      if (x > 0)
            y = y*3;
      return y;
}
```

Impossible to achieve branch coverage without achieving statement coverage

Criteria Subsumption

- Criterion A subsumes criterion B if every test suite that satisfies criterion A also satisfies criterion B

- Branch coverage subsumes statement coverage
  - If a test suite achieves branch coverage it must also achieve statement coverage

Criteria Subsumption – A Warning

- If criterion A subsumes criterion B it means all ***test suites*** satisfying criterion A satisfy criterion B
- However, it does not mean that ***all failures found by a test suite*** satisfying criterion A will be found be a test suite satisfying criterion B
- In general, test suites meeting subsuming criteria *do* find more failures
  – But no guarantee

Criteria Subsumption Problem

```
int abs(int x) {
    int ans = x;
    if (x < 2)
        ans = -x
    return ans;
}
```

Test suite 1:

abs(10)

abs(-10)

Achieves branch and statement coverage

Test suite 2:

abs(1)

Achieves statement coverage only

Criteria Subsumption Problem

```
int abs(int x) {
    int ans = x;
    if (x < 2)
        ans = -x
    return ans;
}
```

Test suite 1:

abs(10) = 10 (passed)

abs(-10) = 10 (passed)

Achieves branch and statement coverage

Test suite 2:

abs(1)

Achieves statement coverage only

Criteria Subsumption Problem

```
int abs(int x) {
    int ans = x;
    if (x < 2)
        ans = -x
    return ans;
}
```

Test suite 1:

abs(10) = 10 (passed)

abs(-10) = 10 (passed)

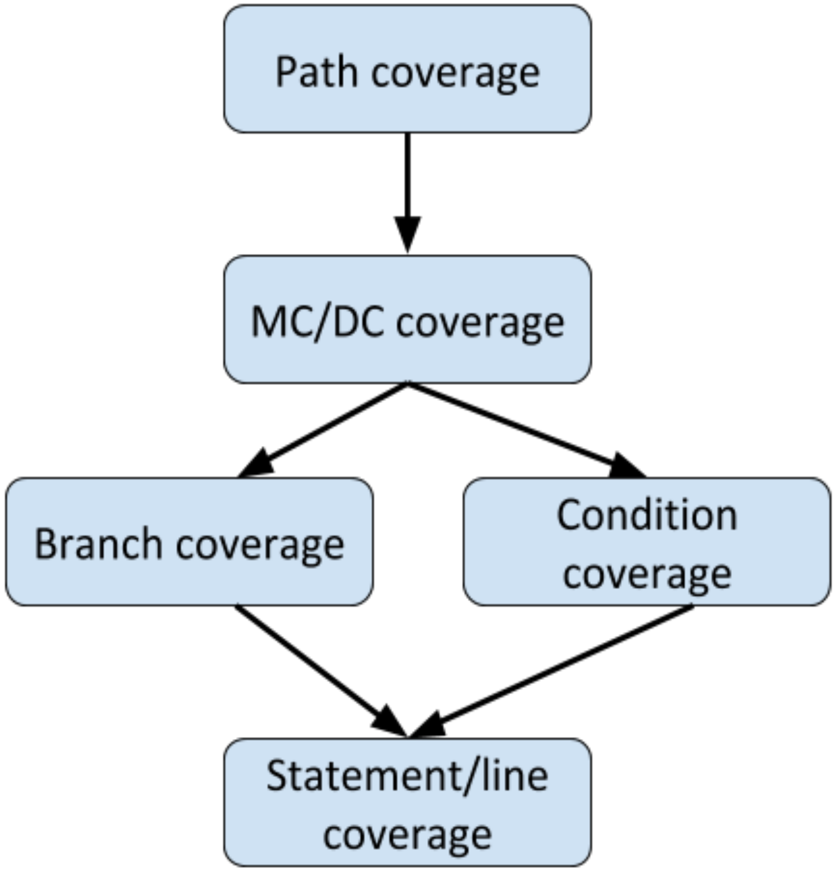
Achieves branch and statement coverage

Test suite 2:

abs(1) = -1 (failed)

Achieves statement coverage only

Criteria Subsumption – Summary



Metrics and Method Evaluation

Question – You’ve evaluated your testing on a project, and you’ve got good coverage. Does that mean that the approach you used is a good method for *all* projects?

- No!
- Different projects will have different testing requirements (remember “different worlds”)
- Rigorous evaluation requires confidence *over multiple projects*

Some Metrics Aren’t Directly About Testing, But Have Implications for Testing

So, you want to know how much testing you have to do, and how much it will cost you…

- You can use *software metrics* to assess the size of your system

Size-related metrics

- Lines of Code (LOCs)
- Number of classes or header files
- Number of methods per class
- Number of attributes per class
- Size of compiled code
- Memory footprint

Not all parts of a program deserve equal testing effort. How do we decide what to focus on?

(for now, assume you *don't* have any test results yet)

One strategy – you can take parts of a program and calculate complexity metrics

- Criticality
- Complexity
- Number of states
- Coupling metrics

- Any questions?