CPEN 422
Software Testing and Analysis

Test Adequacy and Coverage

Coverage Criteria

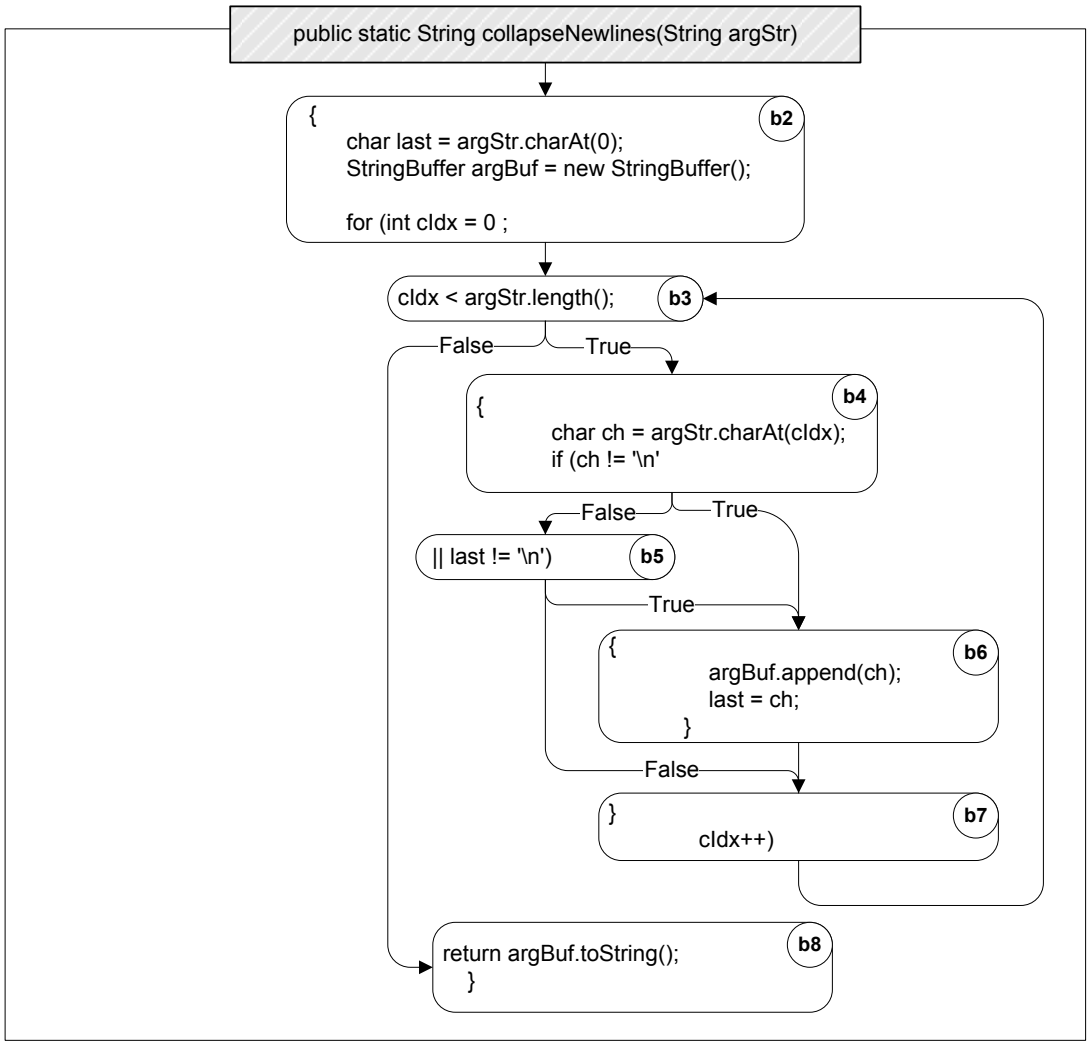
Basic Coverage

Advanced Coverage

- Line coverage
- Statement
- Function/Method coverage
- Branch coverage
- Decision coverage
- Condition coverage
- Condition/decision coverage
- Modified condition/decision coverage
- **Path coverage**
- **Loop coverage**
- Mutation adequacy
- …

Path-based criteria?

- All paths?
- Which paths?
- Loop coverage?



Branch vs Path Coverage

```
if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();
```

How many test cases to achieve branch coverage?

Branch vs Path Coverage

```
if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();
```

How many test cases to achieve branch coverage?

Two, for example:

1. **cond1: true, cond2: true**
2. **cond1: false, cond2: false**

Branch vs Path Coverage

```
if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();
```

How about path coverage?

Branch vs Path Coverage

```
if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();
```

How about path coverage?

Four:

1. **cond1: true, cond2: true**
2. **cond1: false, cond2: true**
3. **cond1: true, cond2: false**
4. **cond1: false, cond2: false**

Branch vs Path Coverage

```
if( cond1 )
   f1();
else
   f2();

if( cond2 )
   f3();
else
   f4();

if( condN )
   fN();
else
   fN();

if( condN )
   fN();
else
   fN();
if( condN )
   fN();
else
   fN();
if( condN )
   fN();
else
   fN();
if( condN )
   fN();
else
   fN();
```

How many test cases for path coverage?

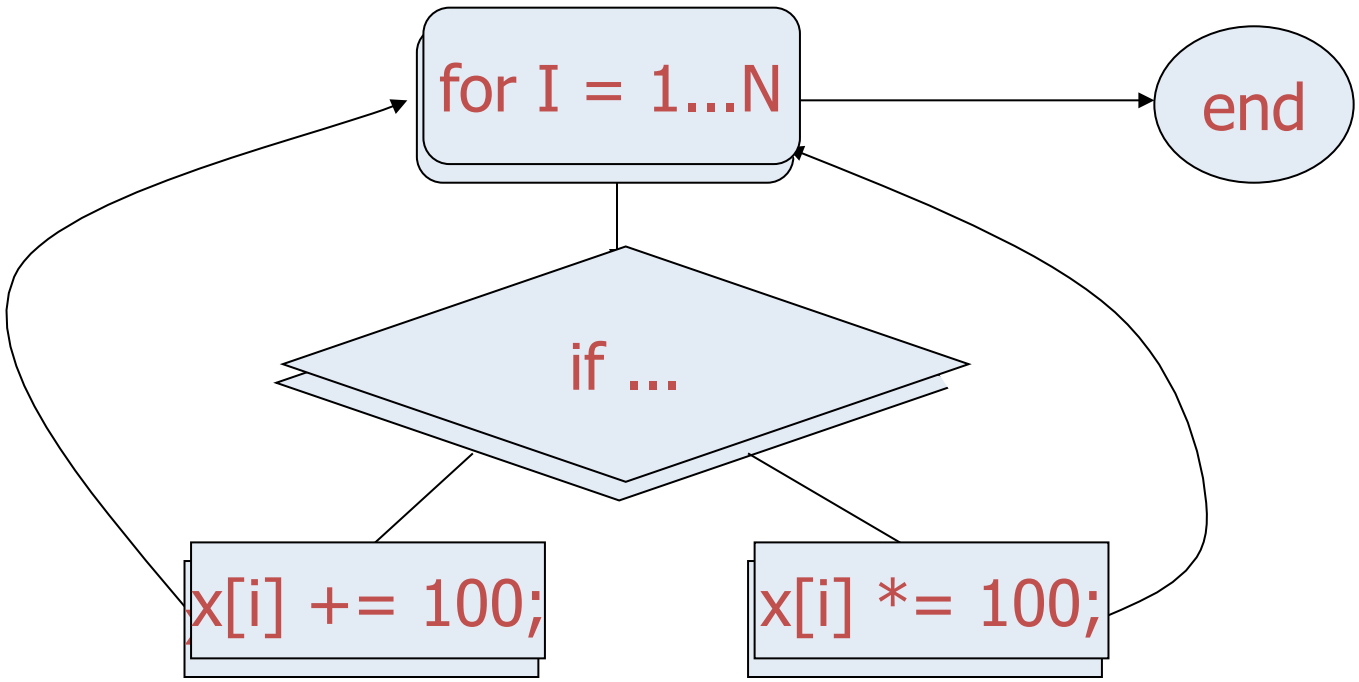
2^n test cases

Path Coverage

Adequacy criterion: each path must be executed at least once

Coverage:

$$\frac{\text{\# executed paths}}{\text{\# paths}}$$



Path Coverage

- "Loop boundary" testing:
  - Limit the number of traversals of loops: Zero, once, many
- "Boundary interior" testing:
  - Unfold loop as tree
- "Linear Code Sequence and Jump", LCSJ
  - Limit the length of the paths to be traversed
- "Cyclomatic complexity" / McCabe
  - "Linearly independent paths"

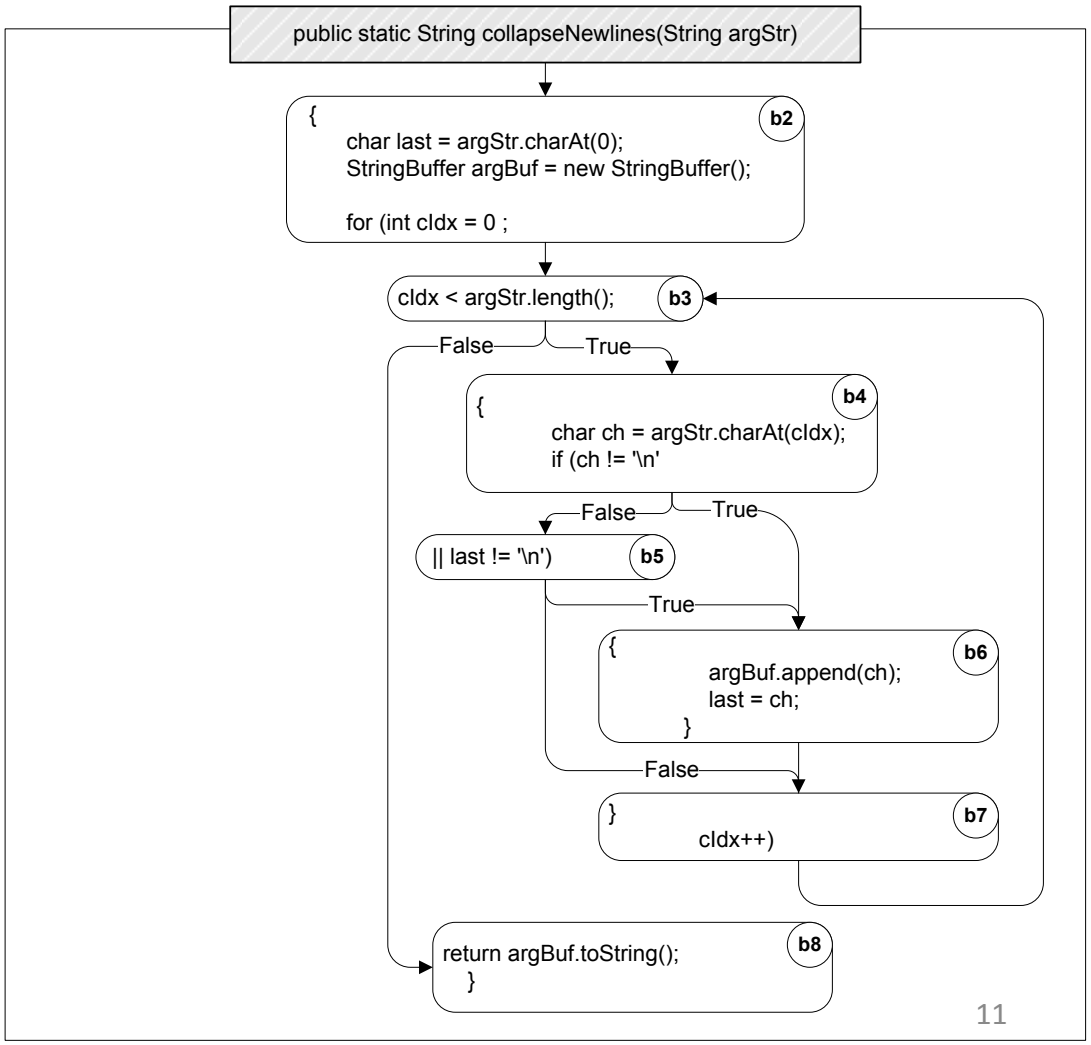
Loop Boundary Adequacy

Execute each loop:

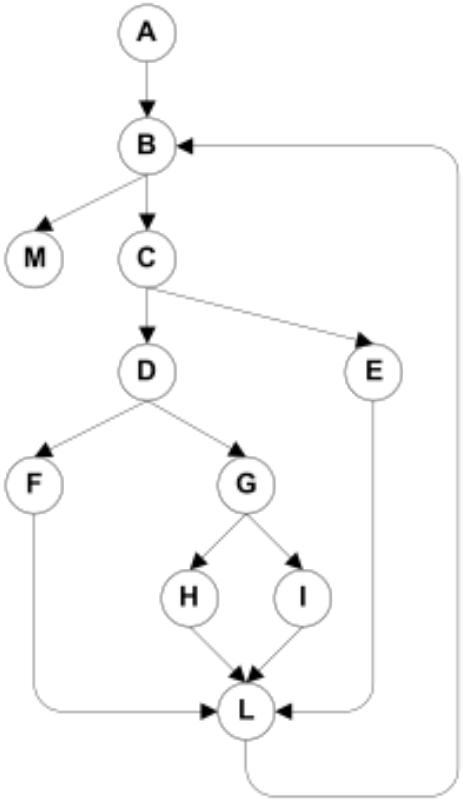
- Zero
- Once
- More than once

Alternatively

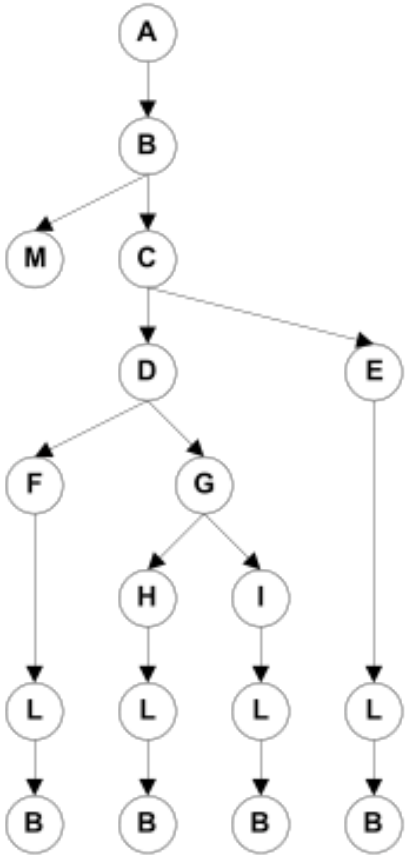
- Min / max



Unfolding a Loop to a Tree: Boundary Interior Coverage



(i)

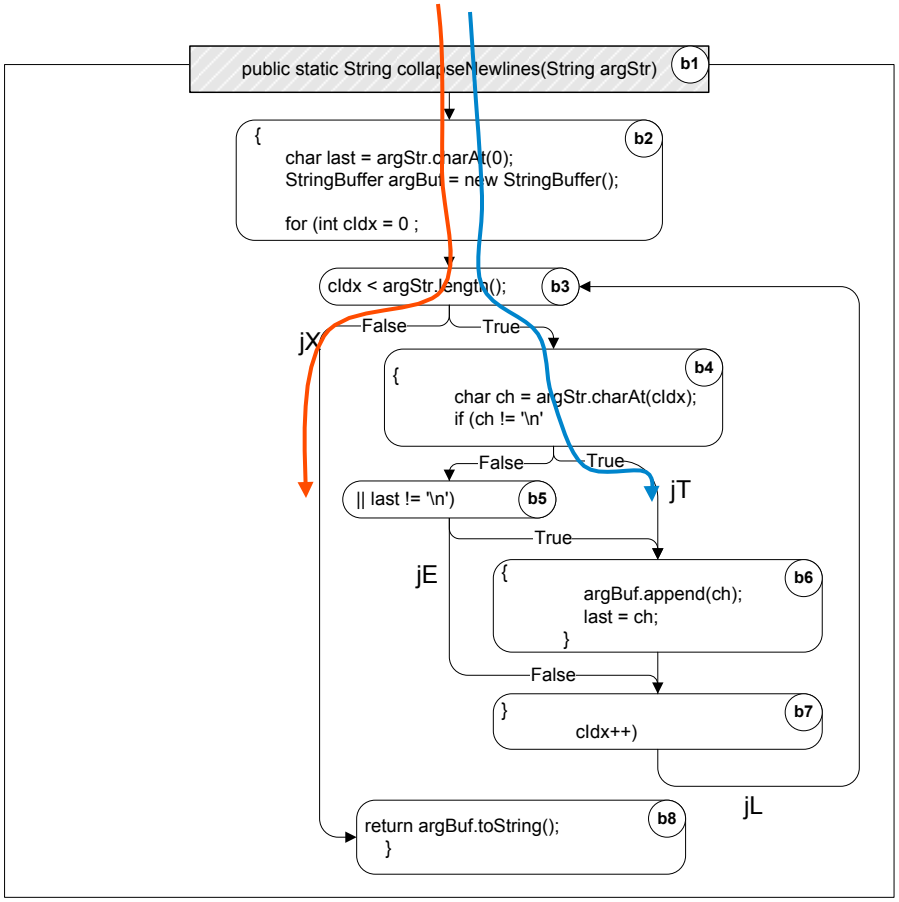


(ii)

Similar to *Transition Tree*

Linear Code Sequence and Jump (LCSJ)

Subpaths of control flow graph from one branch to another



From	Sequence of basic blocs	To
Entry	b1 b2 b3	jX
Entry	b1 b2 b3 b4	jT
Entry	b1 b2 b3 b4 b5	jE
Entry	b1 b2 b3 b4 b5 b6 b7	jL
jX	b8	ret
jL	b3 b4	jT
jL	b3 b4 b5	jE
jL	b3 b4 b5 b6 b7	jL

The Basis-Path Test Model

- Identify "basis paths"
  - **independent** paths that span the graph
  - "*Any path that includes at least one edge that is not included in any of the former independent paths*"
  - Start with the longest one.
- You'll need "C = *e* - *n* + 2" basis paths
  - Every path is a linear combination of basis cycles
  - One test case for each basis path

Watson & McCabe, Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. NIST, 1994 (1983)

McCabe's Cyclomatic Complexity

- C = |E| - |N| + 2
- C = nr of decision points + 1
- C = nr of if-statements + 1
  (+ #switch cases)

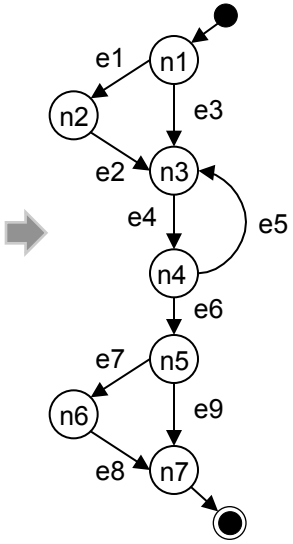
- C > 10: *method too complex* [McCabe, 1976]
- *[C correlated with #lines of code]*

Identify the Linearly Independent Paths (draw the CFG first)

```
if expression1 then
    statement2
end if

do
  statement3
  while expr4
end do

if expression5 then
    statement6
end if
statement7
```

Linearly Independent Paths

```
if expression1 then
    statement2
end if

do
    statement3
  while expr4
end do

if expression5 then
    statement6
end if
statement7
```



V(G) = e – n + 2 = 9 – 7 + 2 = 4

LIPs:

P1 = e3, e4, e6, e7, e8

P2 = e3, e4, e6, e9

P3 = e1, e2, e4, e6, e9

P4 = e3, e4, e5

State Coverage

- Given a state machine of the program
- Have all the states been covered!

- Challenges:
  - How many states are there?
  - Estimations?

Facebook state coverage exercise

- Given a test suite that tests the client-side of Facebook web application
- How would you measure the state coverage of the Facebook application?

Coverage Tools

Cobertura

- Line, branch, complexity
- HTML report generated
- Counters per line
- Byte code transformation

- Hard to get inter-project coverage with maven

Coverage Report - org.jpacman.framework.model

Package	# Classes	Line Coverage		Branch Coverage		Complexity
org.jpacman.framework.model	15	94%	239/253	53%	155/290	1.245

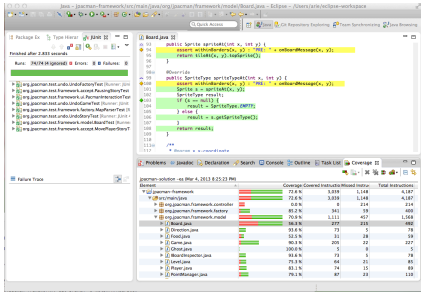
Classes in this Package	Line Coverage		Branch Coverage		Complexity
Board	98%	50/51	55%	51/92	1.769
Direction	100%	11/11	N/A	N/A	1
Food	66%	8/12	25%	4/16	1.2
Game	100%	58/58	73%	25/34	1.389
Ghost	100%	2/2	N/A	N/A	1
IBoardInspector	N/A	N/A	N/A	N/A	1
IBoardInspector$SpriteType	100%	7/7	N/A	N/A	1
IGameInteractor	N/A	N/A	N/A	N/A	1
IPointInspector	N/A	N/A	N/A	N/A	1
Level	76%	13/17	37%	6/16	1
Player	90%	18/20	50%	8/16	1.111
PointManager	100%	19/19	53%	14/26	1.143
Sprite	92%	23/25	52%	25/48	1.143
Tile	96%	28/29	52%	22/42	1.222
Wall	100%	2/2	N/A	N/A	1

Report generated by Cobertura 1.9.4.1 on 2/28/13 4:46 PM.

EclEmma

- Line, branch, complexity
- In Eclipse
- No counters
- On the fly instrumentation
- Exceptions not covered

- Easy inter-project coverage in Eclipse



Is 100% Coverage *Feasible*?

- Mutually exclusive conditions
  - (a < 0 && a < 10)
    - (**T && F)** is not feasible
- Dead code / unreachable code
- "This should never happen" code

Systems in practice:
Statement coverage 85-90% feasible

Infeasible Paths Example

```
int example (int a) {
  int r = OK;
  int a = -1;

  if(a == -1) {
    r = ERROR_CODE;
    ERXA_LOG(r);
  }

  if(a == -2) {
    r = OTHER_ERROR_CODE;
    ERXA_LOG(r);
  }

  return r;
}
```

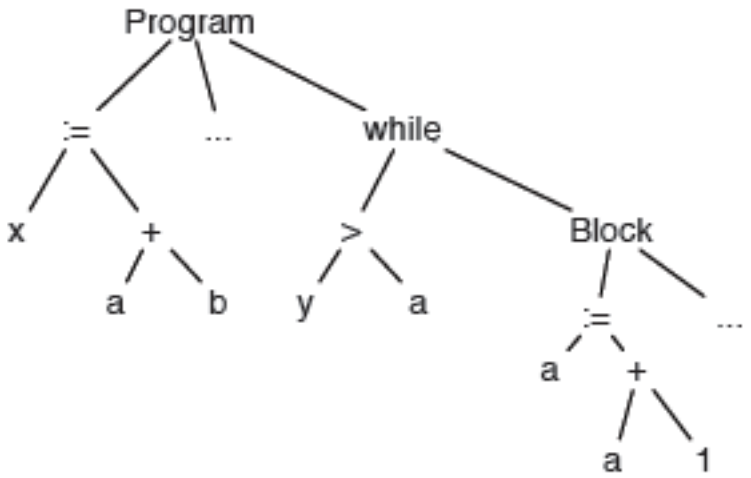
Three feasible paths: a = -1; a = -2 or any other a.
Infeasible path: a == -1 as well as a == -2.

How do we measure coverage?

First:

1. *Parse the source code to build an Abstract Syntax Tree (AST)*
2. *Analyze the AST to build a Control Flow Graph (CFG)*
3. *Count points of interest*
    1. *(total # of statements, branches, etc)*
4. *Instrument the AST using the CFG*
    1. *add tracing statements in the code*

```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```



How do we measure coverage?

Then:

1. *Transform AST back to instrumented code*
2. *Recompile and run the test suite on the recompiled code*
3. *Collect tracing data*
    1. *(line 1 executed, line 3 executed, etc)*
4. *Calculate coverage:*
    1. *# traced points / total # points*

Coverage May Affect Test Outcomes

- *Heisenberg effect*
  - *the act of observing a system inevitably alters its state.*

- Coverage analysis changes the code by adding tracing statements
- Instrumentation can change program behaviour

Enabled In-code Assertions Mess Up Branch Coverage Reporting

assert P

Turned into:

```
if assertions-enabled
then
    if P
    then nothing
    else fail
else nothing
```

- Thus 4 branches!
- Reported as such
- Assertions shouldn't fail
- Resulting branch coverage reports:
  - Not useful with assertion checking enabled
  - Without it, they miss invariants
- Ignore them?
  - Feature of clover
  - Emma feature request

Coverage May Affect Test Outcomes

- *Always run your test suite*
  - *With and without coverage*
  - *(And with and without in-code assertions enabled)*
- Test suite of JUnit itself:
  - *Failed with EclEmma coverage enabled!* (fixed now).

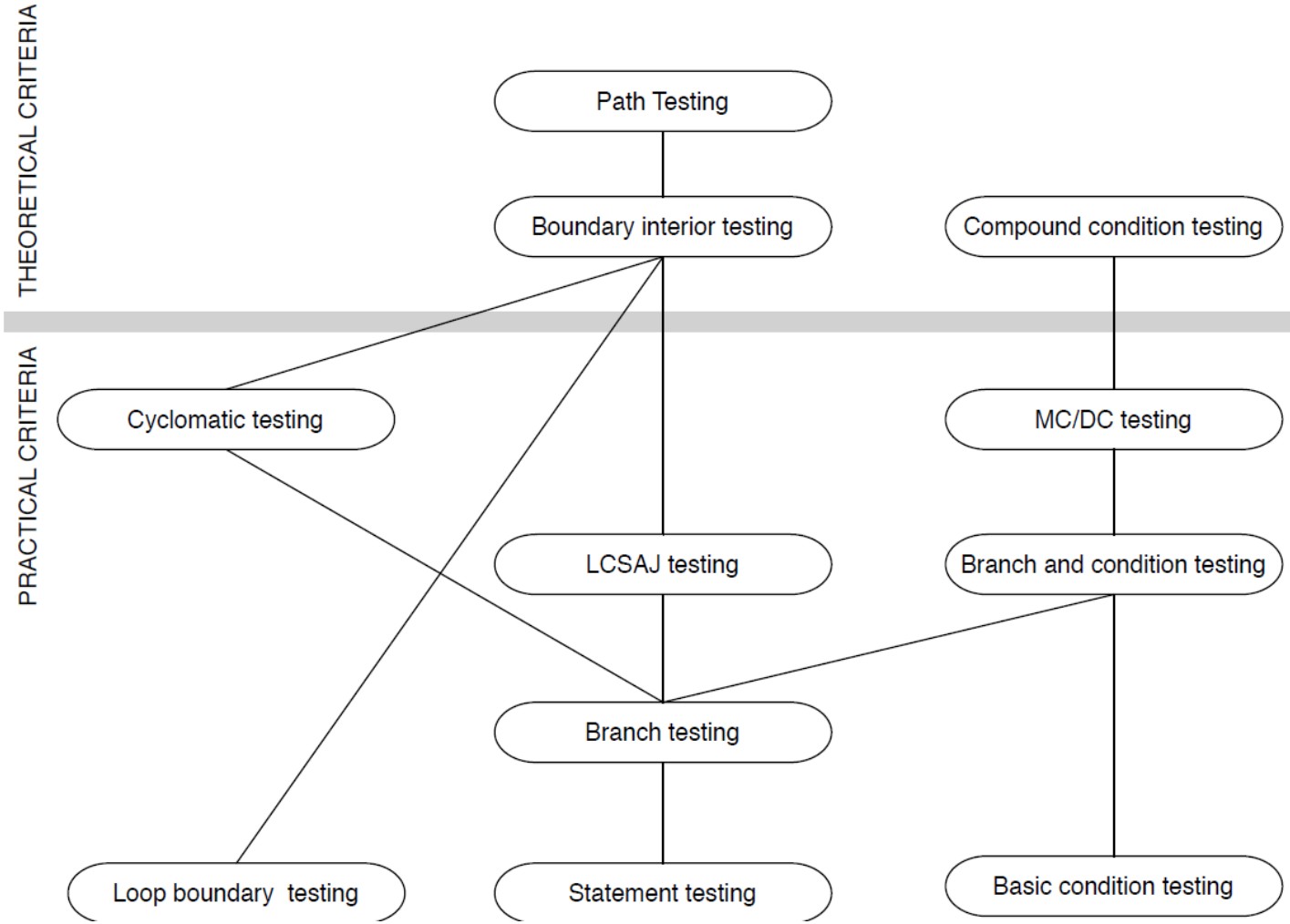
9.4
12.7

Coverage Strategy Subsumption

- Strategy X **subsumes** strategy Y if
  - all elements that Y covers are also covered by X

- Subsumes hierarchy:
  - Analytical ranking of coverage metrics
  - E.g.: MC/DC > all branches > all statements

Subsumption relation



Wrong in the book on p.231!
See http://www.cs.uoregon.edu/~michal/book/errata.html

Coverage: Useful or Harmful?

- Measuring coverage (% of satisfied test obligations) can be a useful indicator ...
  - Of progress toward a thorough test suite, of trouble spots requiring more attention
- ... or a dangerous seduction
  - Coverage is only a proxy for thoroughness or adequacy
  - It's easy to improve coverage without improving a test suite (much easier than designing good test cases)
- The only measure that really matters is **effectiveness**

Is coverage correlated with fault-finding effectiveness of a test suite?

Coverage Is Not Strongly Correlated with Test Suite Effectiveness

Laura Inozemtseva and Reid Holmes
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
{lminozem,rtholmes}@uwaterloo.ca

ABSTRACT

The coverage of a test suite is often used as a proxy for its ability to detect faults. However, previous studies that investigated the correlation between code coverage and test suite effectiveness have failed to reach a consensus about the nature and strength of the relationship between these test suite characteristics. Moreover, many of the studies were done with small or synthetic programs, making it unclear whether their results generalize to larger programs, and some of the studies did not account for the confounding influence of test suite size. In addition, most of the studies were done with adequate suites, which are are rare in practice, so the results may not generalize to typical test suites.

We have extended these studies by evaluating the relationship between test suite size, coverage, and effectiveness for large Java programs. Our study is the largest to date in the literature: we generated 31,000 test suites for five systems consisting of up to 724,000 lines of source code. We measured the statement coverage, decision coverage, and modified condition coverage of these suites and used mutation testing to evaluate their fault detection effectiveness.

We found that there is a low to moderate correlation between coverage and effectiveness when the number of test cases in the suite is controlled for. In addition, we found that stronger forms of coverage do not provide greater insight into the effectiveness of the suite. Our results suggest that coverage, while useful for identifying under-tested parts of a program, should not be used as a quality target because it is

1. INTRODUCTION

Testing is an important part of producing high quality software, but its effectiveness depends on the quality of the test suite: some suites are better at detecting faults than others. Naturally, developers want their test suites to be good at exposing faults, necessitating a method for measuring the fault detection effectiveness of a test suite. Testing textbooks often recommend coverage as one of the metrics that can be used for this purpose (e.g., [29, 34]). This is intuitively appealing, since it is clear that a test suite cannot find bugs in code it never executes; it is also supported by studies that have found a relationship between code coverage and fault detection effectiveness [3, 6, 14–17, 24, 31, 39].

Unfortunately, these studies do not agree on the strength of the relationship between these test suite characteristics. In addition, three issues with the studies make it difficult to generalize their results. First, some of the studies did not control for the size of the suite. Since coverage is increased by adding code to existing test cases or by adding new test cases to the suite, the coverage of a test suite is correlated with its size. It is therefore not clear that coverage is related to effectiveness independently of the number of test cases in the suite. Second, all but one of the studies used small or synthetic programs, making it unclear that their results hold for the large programs typical of industry. Third, many of the studies only compared adequate suites; that is, suites that fully satisfied a particular coverage criterion. Since adequate test suites are rare in practice, the results of these studies

Andrew Glover:
"Don't be fooled by the coverage report"

http://www.ibm.com/developerworks/library/j-cq01316/

- Use coverage to expose code that's ***not*** **adequately tested**
  - Find low values, and understand why they are low.
- Then you can use coverage "where it really counts:"
  - To estimate the time to modify existing code (easier to modify if well-tested)
  - To evaluate code quality (observe coverage trends)

Questions?