Cpt S 422: Software Engineering Principles II

White-box testing – Part 3

Dr. Venera Arnaoudova

WASHINGTON STATE UNIVERSITY

Outline

- ✓ Control flow coverage
  - ✓ Statement, Edge, Condition, Path coverage
- ✓ Data flow coverage
  - ✓ Definitions-Usages of data
- ❑ Analyzing coverage data
- ❑ Integration testing
  - ➢ Coupling-based criteria
- ❑ Conclusions
  - ➢ Generating test data, Marick's Recommendations

Testing Productivity

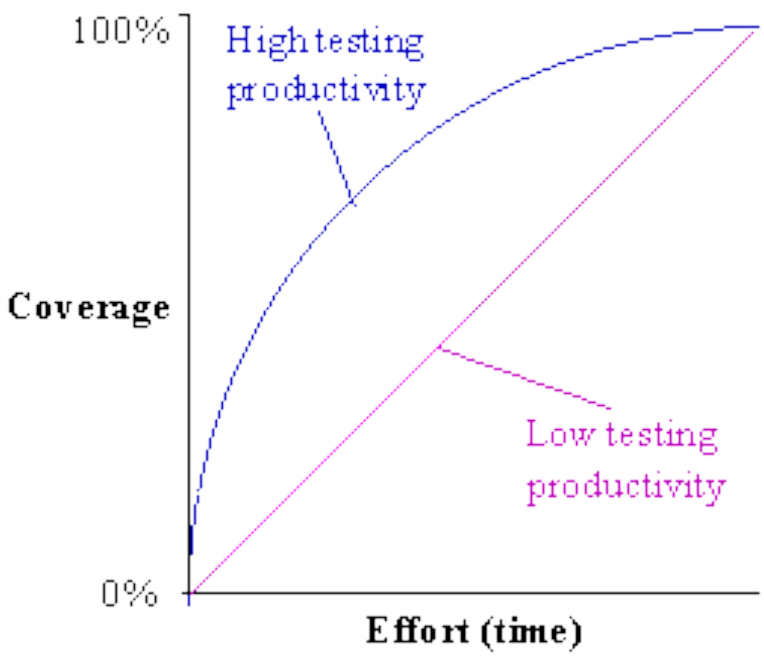


Figure 1: Coverage rate

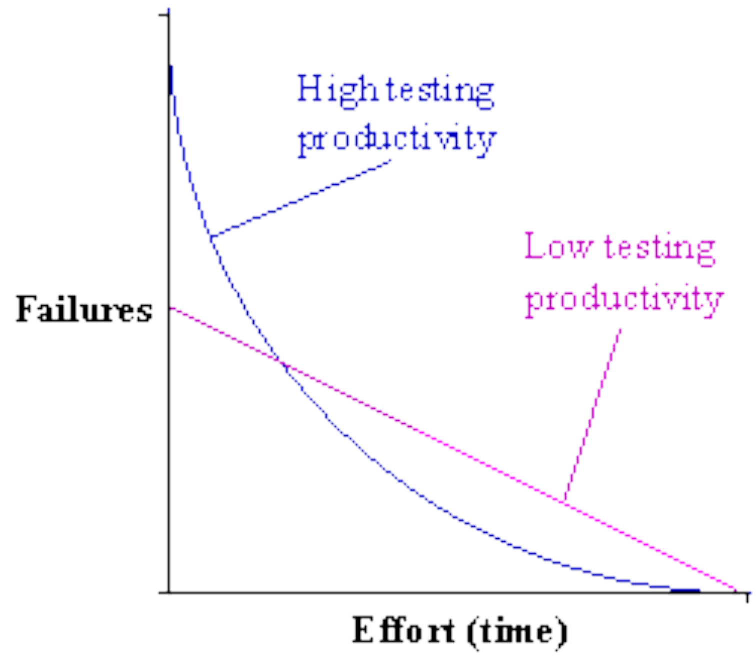
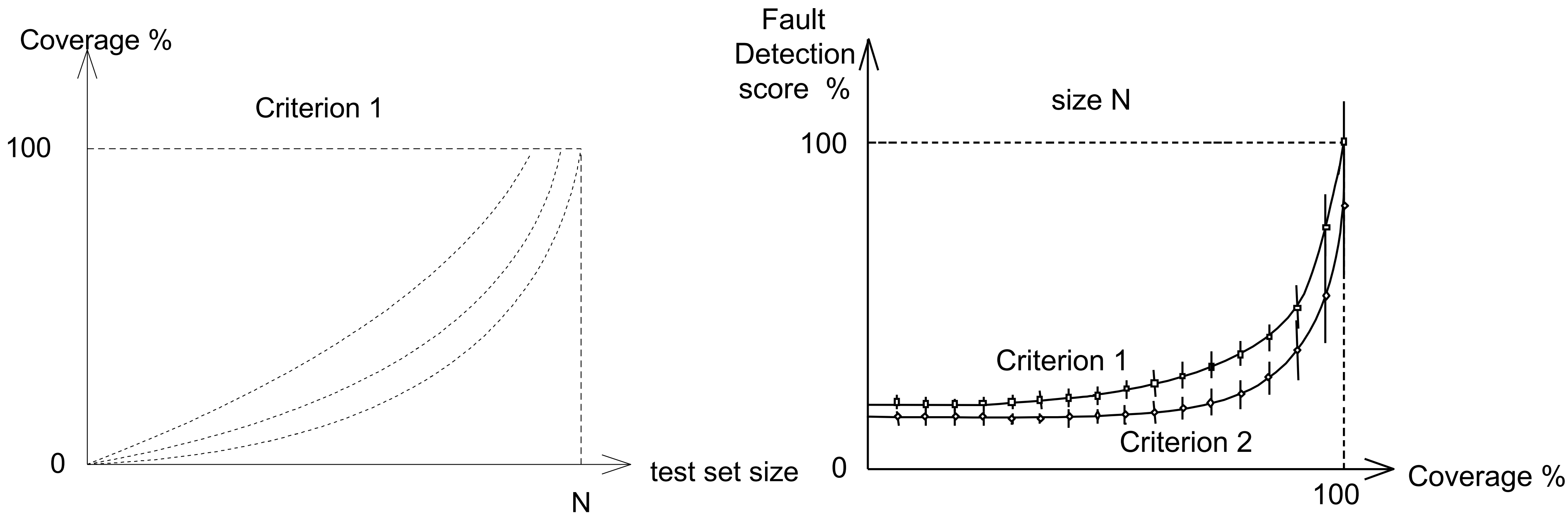


Figure 2: Failure discovery rate

Typical Analyses



Experiment of Hutchins et al.

- Goal: Compare the effectiveness of control flow and data flow coverage (All-Edges versus All-DU coverage criteria)
- Object systems: 130 versions derived from 7 C programs (141-512 LOC) by seeding 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.
- They examined the relationship between fault detection and test set coverage/size

M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proceedings of International Conference on Software Engineering (ICSE)*, 1994, pp. 191-200.

One Program Example

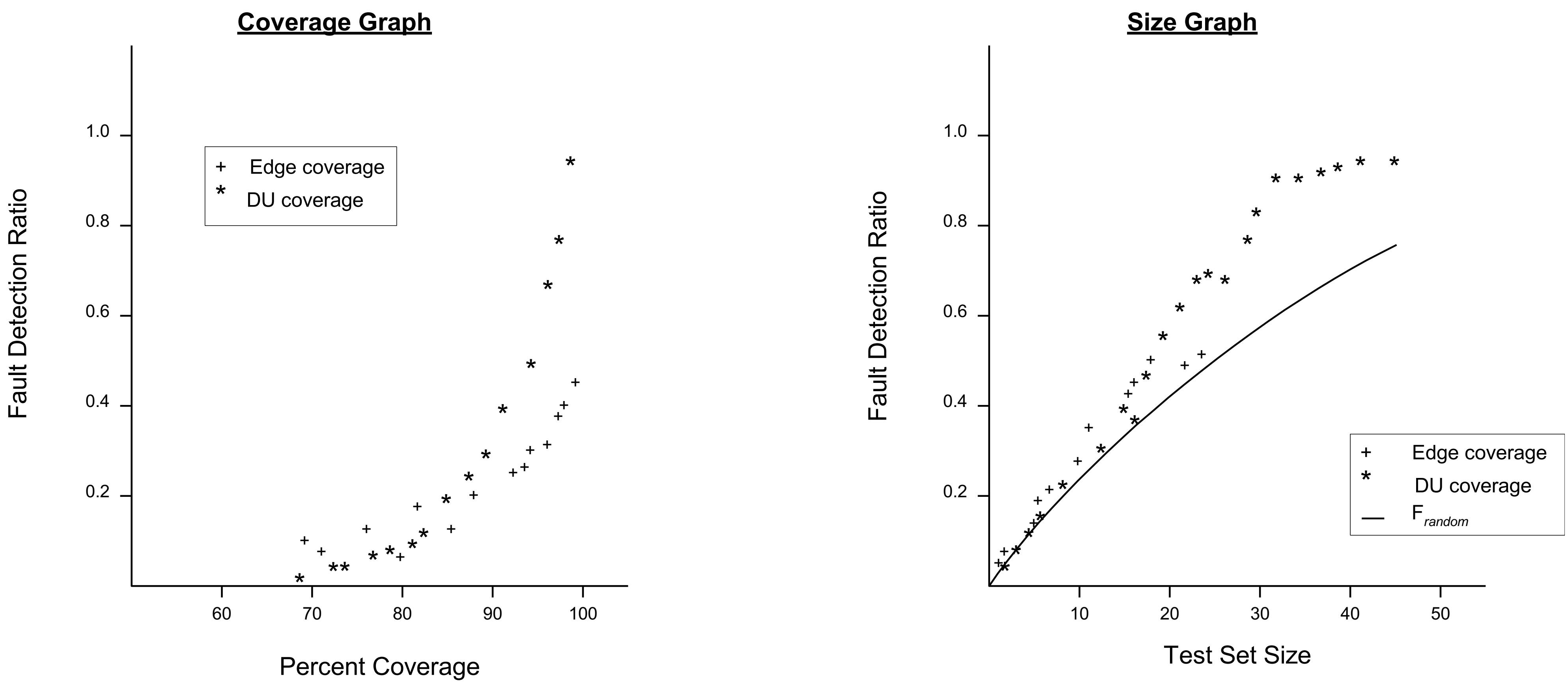


Figure: Fault Detection Ratios for One Faulty Program

Results

- Both coverage criteria performed better than random test selection
- 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
- As expected, on average, achieving all-DU coverage required significantly larger test sets compared to all-Edge coverage

Outline

- ✓ Control flow coverage
  - ✓ Statement, Edge, Condition, Path coverage
- ✓ Data flow coverage
  - ✓ Definitions-Usages of data
- ✓ Analyzing coverage data
- ❑ Integration testing
  - ➢ Coupling-based Criteria
- ❑ Conclusions
  - ➢ Generating test data, Marick's Recommendations

Coupling-based Criteria

- Refer to the testing of interfaces between units/modules to assure that they have consistent assumptions and communicate correctly.

  - Coupling between two units measures the dependency relations between two units by reflecting the interconnections between units; faults in one unit may affect the coupled unit (Yourdon and Constantine, 1979)

  - Coupling-based Coverage Criteria, proposed by Jin and Offutt (1998), specifically aimed at integration testing in a non-OO context, based on data flow analysis

Basic Definitions

- The interface between two units is the mapping of *actual* to *formal* parameters

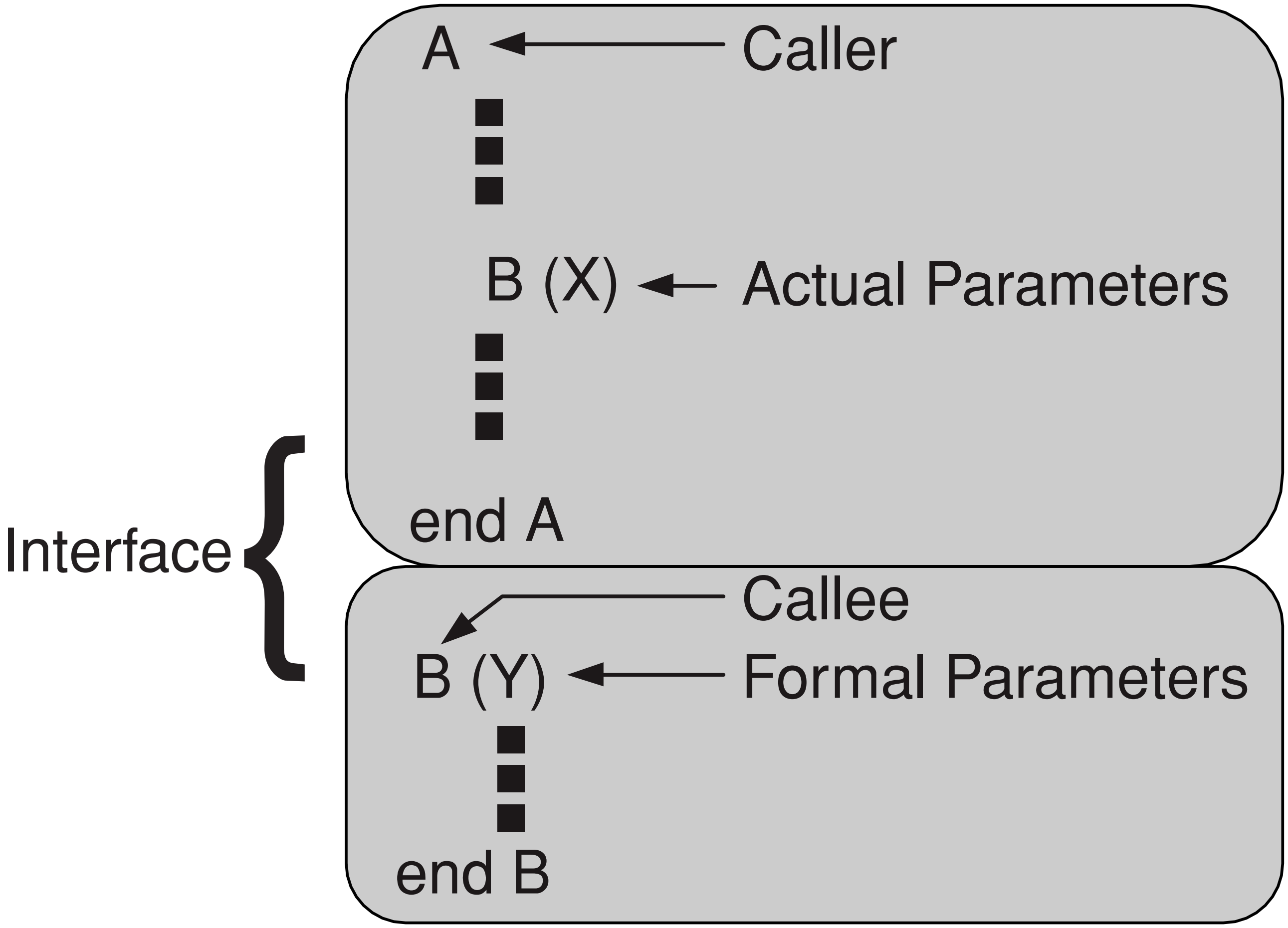
- During integration testing, we want to look at *definitions* and *uses* <u>across</u> different units

- To increase our confidence in interfaces, we want to ensure that variables defined in *caller* units are appropriately used in *callee* units

- Look at variables <u>definitions</u> *before* calls and returns to other units, and <u>uses</u> of variables just *after* calls and returns from the called unit

- We refer to *coupling variables* for variables that are defined in one unit and used in another.

Example


A
Caller
B (X)
Actual Parameters
end A
Interface
Callee
B (Y)
Formal Parameters
end B

Basic Definitions

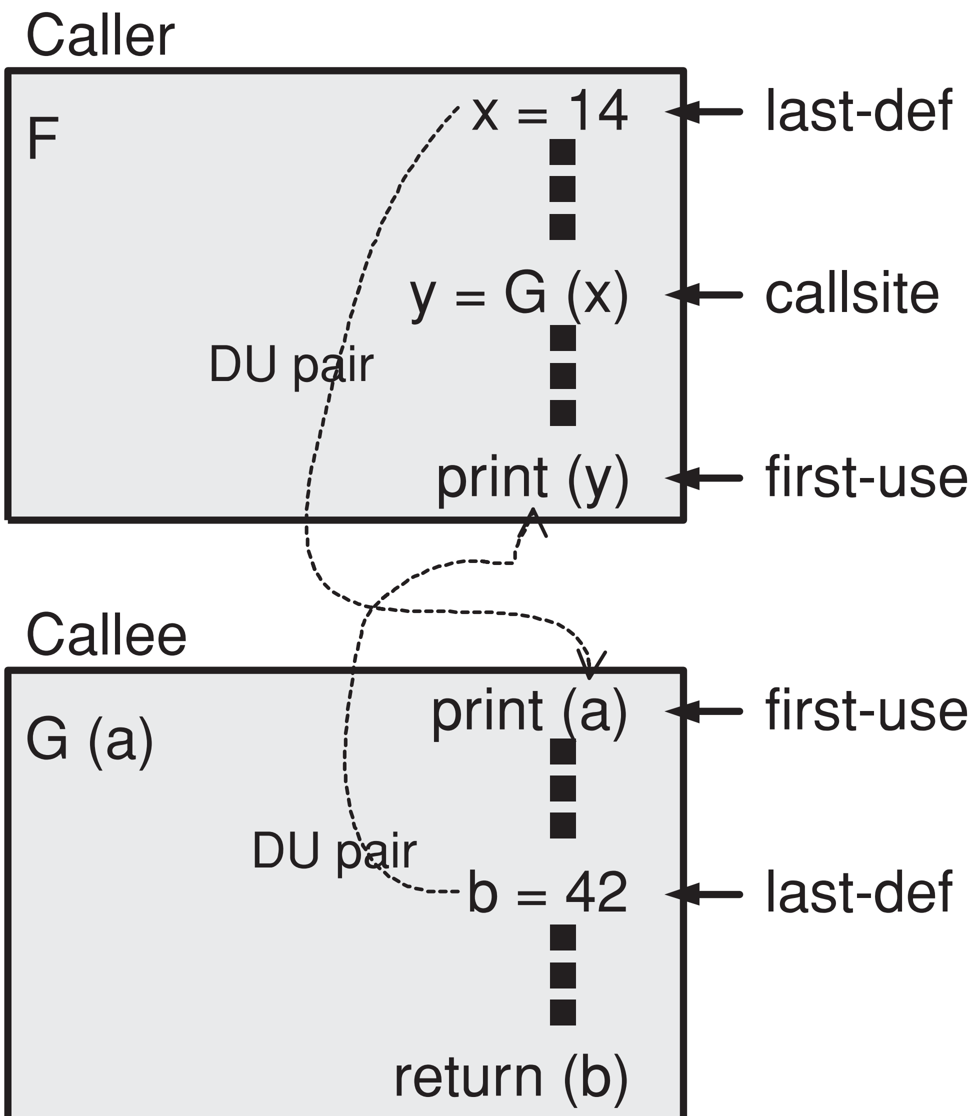
- We differentiate three types of coupling between units:
  - Parameter coupling
  - Shared data coupling (e.g., global variables)
  - External device coupling (e.g., files)

- *Call sites*: statements in caller (A) where callee (B) is invoked

- *Last-Defs*: The set of nodes that define x for which there is a def-clear path from the node through the callsite / return to a use in the other unit.

- *First-Uses*: The set of nodes that have uses of y and for which there exists a def-clear and use-clear path between the callsite (if the use is in the caller) or the entry point (if the use is in the callee) and the nodes.

Example



Caller
F
x = 14
last-def
y = G (x)
callsite
DU pair
print (y)
first-use
Callee
G (a)
print (a)
first-use
DU pair
b = 42
last-def
return (b)

Coupling Paths & Criteria

- A coupling du-path is a path from a last-def to a first-use
- List of criteria (apply to the 3 types of coupling):
  - call-coupling:
    - ✓ requires execution of all call sites in the caller.
  - all-coupling-defs:
    - ✓ requires that for each coupling definition at least one coupling path to **at least one** reachable coupling use is executed.
  - all-coupling-uses:
    - ✓ requires that for each coupling definition at least one coupling path to **each** reachable coupling use is executed.
  - all-coupling-paths:
    - ✓ requires that **all** loop-free coupling paths be executed.

Parameter Coupling

- Subsumption:

All-Coupling-Paths

$\downarrow$

All-Coupling-Uses

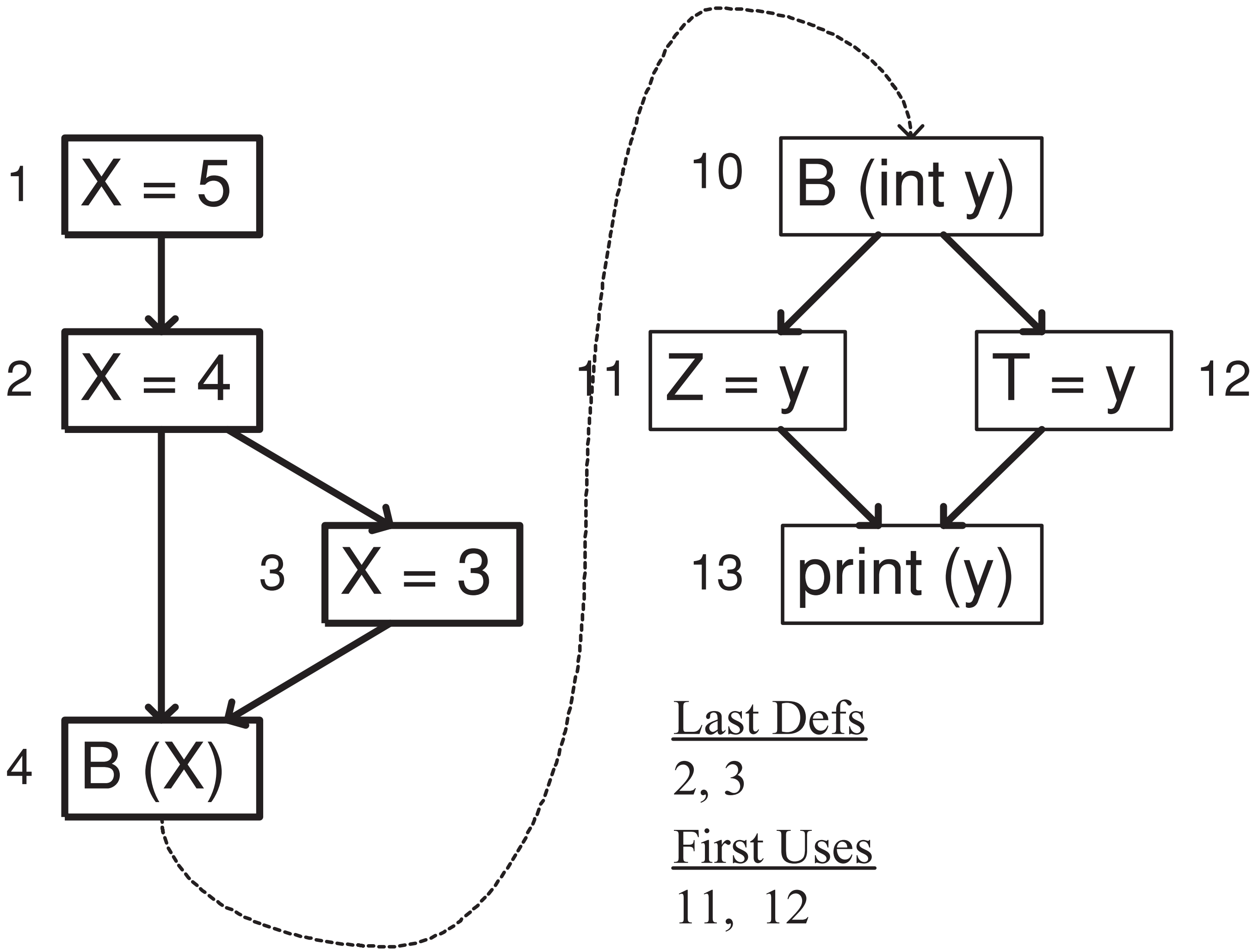
$\downarrow$

All-Coupling-Defs

$\downarrow$

Call Coupling

Example with two CFG’s



Last Defs

2, 3

First Uses

11, 12

Coupling du-pairs for X:

- 2,11
- 2,12
- 3,11
- 3,12

```
1  // Program to compute the quadratic root for two numbers
2  import java.lang.Math;
3
4  class Quadratic
5  {
6  private static double Root1, Root2;
7
8  public static void main (String[] argv)
9  {
10    int X, Y, Z;
11    boolean ok;
12    if (argv.length == 3)
13    {
14       try
15       {
16          X = Integer.parseInt (argv[1]);
17          Y = Integer.parseInt (argv[2]);
18          Z = Integer.parseInt (argv[3]);
19       }
20       catch (NumberFormatException e)
21       {
22          System.out.println ("Inputs not integers, using 8, 10, -33.");
23          X = 8;
24          Y = 10;
25          Z = -33;
26       }
27    }
28    else
29    {
30       X = 8;
31       Y = 10;
32       Z = -33;
33    }
34    ok = Root (X, Y, Z);
35    if (ok)
36       System.out.println
37          ("Quadratic: Root 1 = " + Root1 + ", Root 2 = " + Root2);
38    else
39       System.out.println ("No solution.");
40 }
41
42 // Finds the quadratic root, A must be non-zero
43 private static boolean Root (int A, int B, int C)
44 {
45    double D;
46    boolean Result;
47    D = (double)(B*B) - (double)(4.0*A*C);
48    if (D < 0.0)
49    {
50       Result = false;
51       return (Result);
52    }
53    Root1 = (double) ((-B + Math.sqrt(D)) / (2.0*A));
54    Root2 = (double) ((-B - Math.sqrt(D)) / (2.0*A));
55    Result = true;
56    return (Result);
57 } // End method Root
58
59 } // End class Quadratic
```

Example

Task: Identify all coupling du-pairs using the following format:

- ❑ (unit1(), variable1, line1) -- (unit2(), variable2, line2)
- ❑ E.g., (main(), X, 16) -- (Root(), A, 47)

Coupling du-pairs

1. (main(), X, 16) -- (Root(), A, 47)
2. (main(), Y, 17) -- (Root(), B, 47)
3. (main(), Z, 18) -- (Root(), C, 47)
4. (main(), X, 23) -- (Root(), A, 47)
5. (main(), Y, 24) -- (Root(), B, 47)
6. (main(), Z, 25) -- (Root(), C, 47)
7. (main(), X, 30) -- (Root(), A, 47)
8. (main(), Y, 31) -- (Root(), B, 47)
9. (main(), Z, 32) -- (Root(), C, 47)

10. (Root(), Root1, 53) -- (main(), Root1, 37)
11. (Root(), Root2, 54) -- (main(), Root2, 37)
12. (Root(), Result, 50) -- (main(), ok, 35)
13. (Root(), Result, 55) -- (main(), ok, 35)

Case Study

- Comparison of All-coupling-uses criterion with Category Partition

- Mistix program, C, 31 function units, 65 function calls, 533 LOCs, 21 faults seeded (but 12 could be detected), test cases devised manually

- The faults, category-partition tests, and all-coupling-uses tests were created by different people

Results

- The coupling-based technique performed better (11/12 vs 7/12) than category-partition with half as many test cases (37 vs 72).

- Threats to validity: fault sample, small program, comparisons with other integration test techniques is missing

  - Harrold and Soffa, Selecting and Using Data for Integration Testing, IEEE Software, March 1991.
  - Leung and White, A Study of Integration Testing and Software Regression at the Integration Level, in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 1990.

Remarks

- Empirical studies are rare but they are crucial as theory is of little help to evaluate testing strategies

- But general issues are:

  - How representative are the faults seeded (type, size)?
  - How representative is the program (size, complexity)?
  - Were test cases derived independently of faults? (automation preferred)
  - Results' uncertainty boundaries (faults seeded are samples from a distribution) – it helps determine if observed differences can be obtained by chance

Outline

- ✓ Control flow coverage
  - ✓ Statement, Edge, Condition, Path coverage
- ✓ Data flow coverage
  - ✓ Definitions-Usages of data
- ✓ Analyzing coverage data
- ✓ Integration testing
  - ✓ Coupling-based Criteria
- ❑ Conclusions
  - ➢ Generating test data, Marick's Recommendations

Generating Code-based Tests

- 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, it is sufficient to solve a set of simultaneous inequalities in the input variables of the program, each inequality describing the proper path through one conditional

- Conditionals may be expressed in local variable values derived from the inputs and those must figure in the inequalities as well

Example

```
int z;
scanf("%d%d", &x, &y);
if (x > 3) {
    z = x+y;
    y+= x;
    if (2*z == y) {
        /* statement to be covered */
…
```

Inequalities:

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

1 Solution:

X = 4

Y= -4

Problems

- The presence of loops and recursion in the code makes it 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
- Coverage may be 100% and the tester may yet miss some functionalities (omission faults)

Marick's Recommendations

Brian Marick recommends the following approach (for large scale testing):

1. Generate functional tests from requirements and design to try every function.
2. Check the structural coverage after the functional tests are all verified to be successful.
3. Where the structural coverage is imperfect, generate functional tests (not structural) that induce the additional coverage.

This works because form (structure) should follow function!

- Uncovered code must have some purpose, and that purpose has not been invoked, so some function is untested