Introduction to Software Testing
(*2nd edition*)
Chapter 7.4

Graph Coverage for Design Elements

Paul Ammann & Jeff Offutt

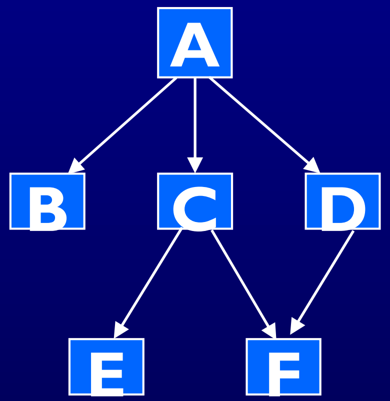
http://www.cs.gmu.edu/~offutt/softwaretest/

OO Software and Designs

- Emphasis on modularity and reuse puts complexity in the design connections
- Testing design relationships is more important than before
- Graphs are based on the connections among the software components
  - Connections are dependency relations, also called couplings

Call Graph

- The most common graph for structural design testing
- Nodes : Units (in Java – methods)
- Edges : Calls to units



Example call graph

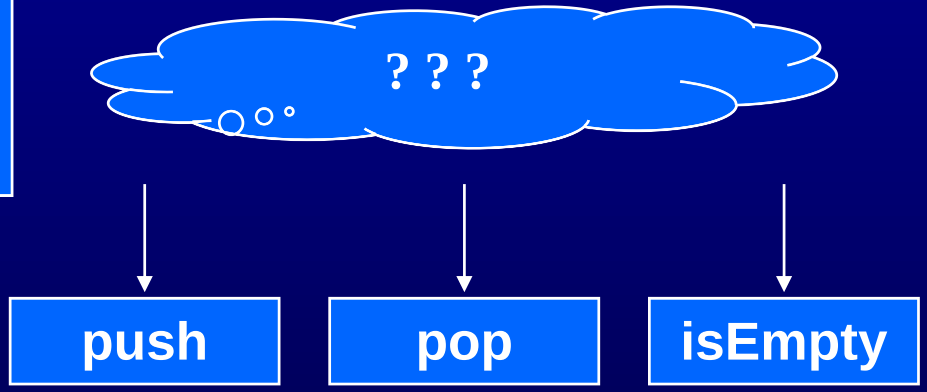
<u>Node coverage</u> : call every unit at least once (method coverage)

<u>Edge coverage</u> : execute every call at least once (call coverage)

Call Graphs on Classes

- Node and edge coverage of class call graphs often do not work very well
- Individual methods might not call each other at all!

<u>Class stack</u>
public void push (Object o)
public Object pop ( )
public boolean isEmpty (Object o)



Other types of testing are needed – do <u>not</u> use graph criteria

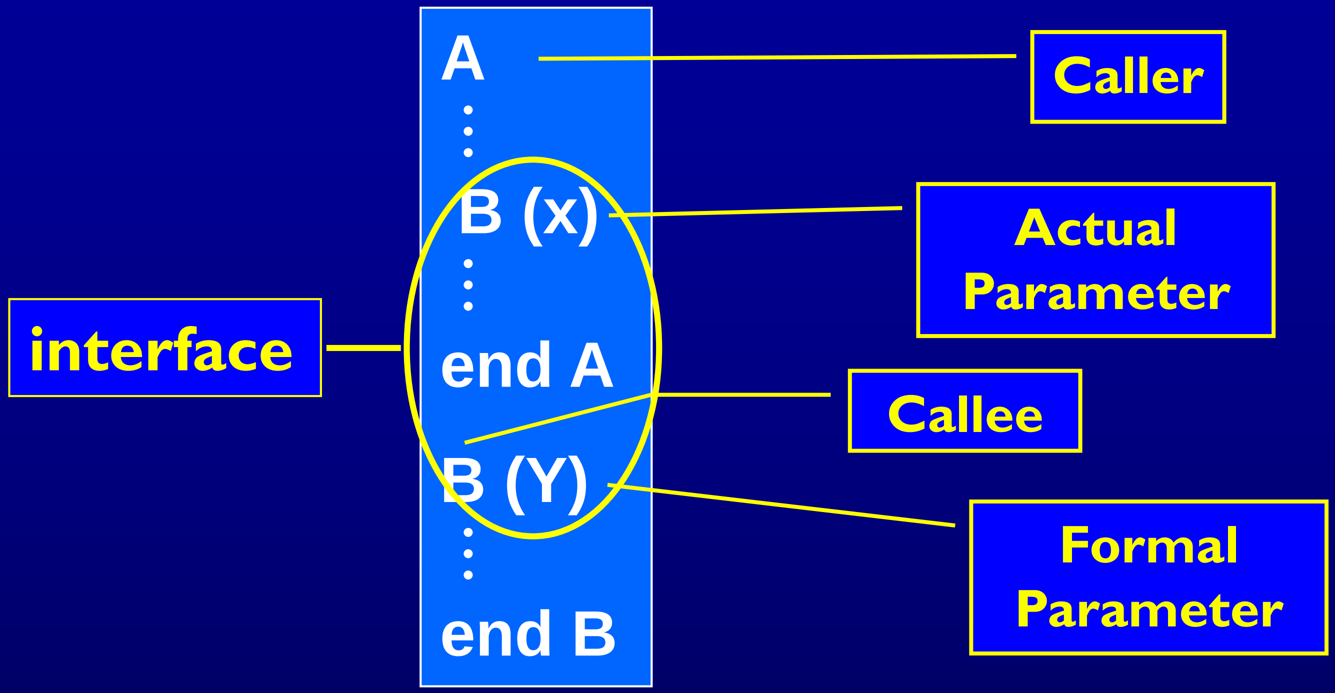
Data Flow at the Design Level

- Data flow couplings among units and classes are more complicated than control flow couplings
  - When values are passed, they "change names"
  - Many different ways to share data
  - Finding defs and uses can be difficult – finding which uses a def can reach is very difficult
- When software gets complicated … testers should get interested
  - That's where the faults are!

Preliminary Definitions

- Caller : A unit that invokes another unit
- Callee : The unit that is called
- Callsite : Statement or node where the call appears
- Actual parameter : Variable in the caller
- Formal parameter : Variable in the callee

Example Call Site

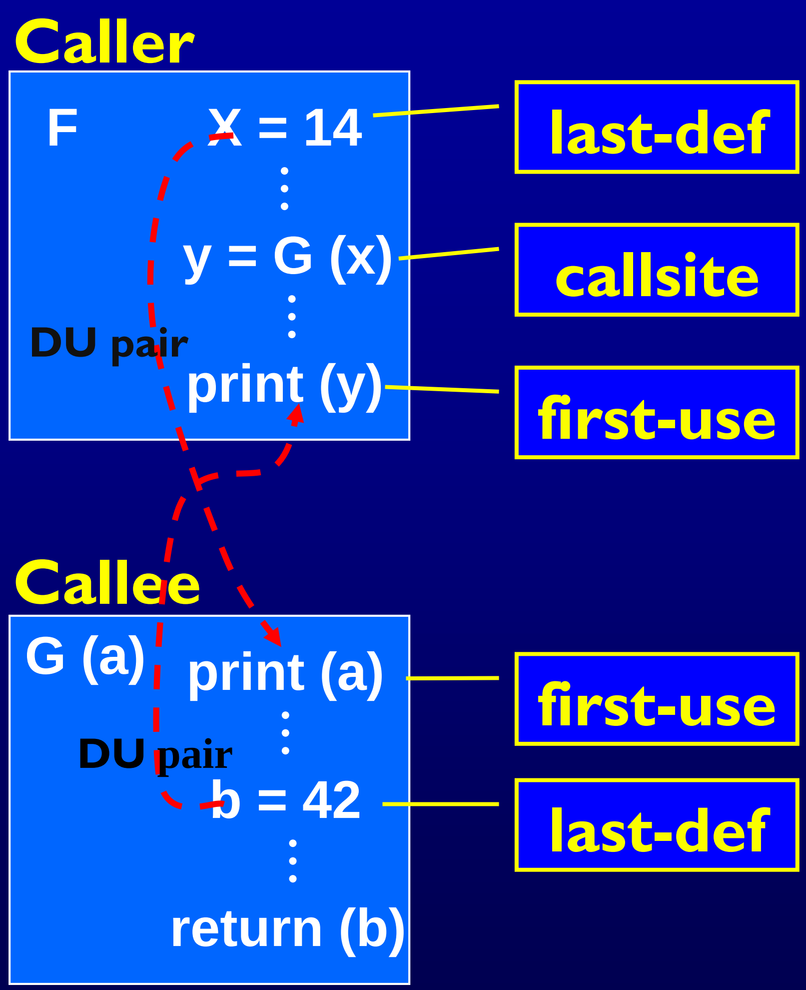


- Applying data flow criteria to def-use pairs between units is too expensive
- Too many possibilities
- But this is integration testing, and we really only care about the interface …

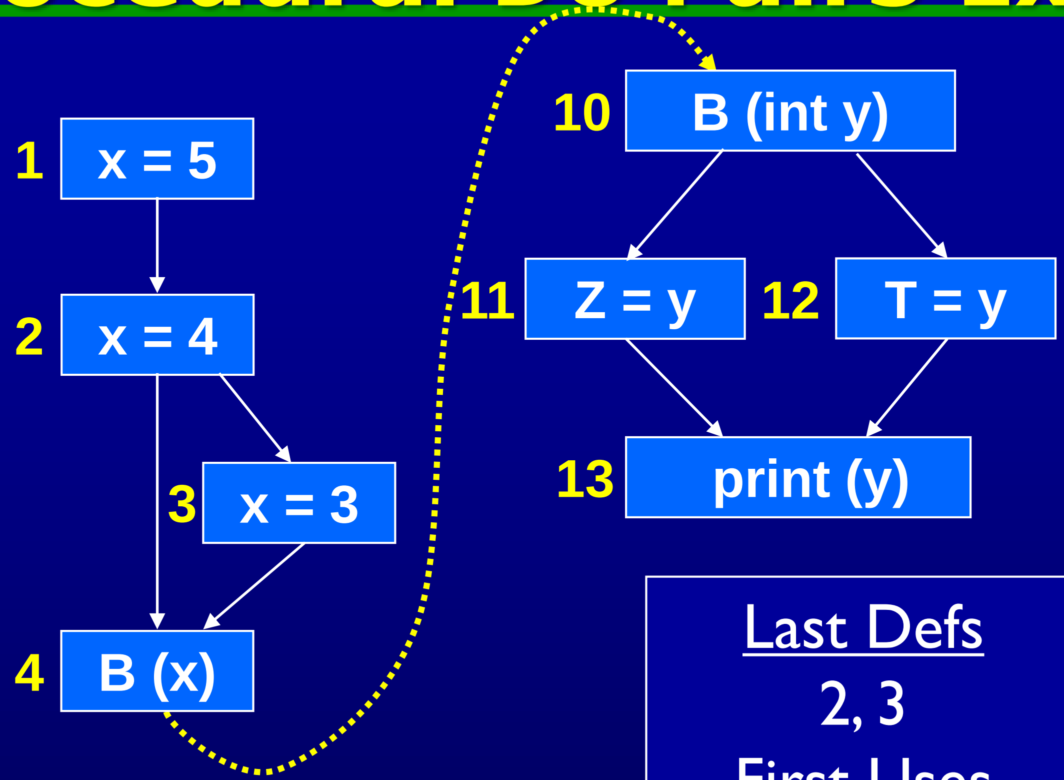
Inter-procedural DU Pairs

- If we focus on the interface, then we just need to consider the last definitions of variables before calls and returns and first uses inside units and after calls
- Last-def : The set of nodes that define a variable *x* and has a def-clear path from the node through a callsite to a use in the other unit
  – Can be from caller to callee (parameter or shared variable) or from callee to caller as a return value
- First-use : The set of nodes that have uses of a variable *y* and for which there is a def-clear and use-clear path from the callsite to the nodes

Inter-procedural DU Pairs Example



Inter-procedural DU Pairs Example



Last Defs

2, 3

First Uses

11, 12

DU Pairs

(A, x, 2)—(B, y, 11)

(A, x, 2)—(B, y, 12)

(A, x, 3)—(B, y, 11)

(A, x, 3)—(B, y, 12)

Example – Quadratic

```
// Program to compute the quadratic root for three numbers
import java.lang.Math;

class Quadratic
{
 private static float Root1, Root2;

 public static void main (String[] argv)
 {
    int X, Y, Z;
    boolean ok;
    int controlFlag = Integer.parseInt (argv[0]);
    if (controlFlag == 1)
    {
        X = Integer.parseInt (argv[1]);
        Y = Integer.parseInt (argv[2]);
        Z = Integer.parseInt (argv[3]);
     }
    else
    {
         X = 10;
         Y = 9;
         Z = 12;
     }
            ok = Root (X, Y, Z);
            if (ok)
              System.out.println
                    (“Quadratic: ” + Root1 + Root2);
            else
                System.out.println (“No Solution.”);
  }

// Three positive integers, finds quadratic root
  private static boolean Root (int A, int B, int C)
  {
      double D;
      boolean Result;
      D = (double) (B*B) -  (double) (4.0*A*C );
      if (D < 0.0)
      {
          Result = false;
          return (Result);
      }
      Root1 = (double) ((-B + Math.sqrt(D))/(2.0*A));
      Root2 = (double) ((-B – Math.sqrt(D))/(2.0*A));
      Result = true;
      return (Result);
  } // End method Root
} // End class Quadratic
```

```
// Program to compute the quadratic root for two numbers
import java.lang.Math;

class Quadratic
{
 private static float Root1, Root2;

 public static void main (String[] argv)
 {
     int X, Y, Z;
    boolean ok;
    int controlFlag = Integer.parseInt (argv [0]);
    if (controlFlag == 1)
    {
       X = Integer.parseInt (argv [1]);
       Y = Integer.parseInt (argv [2]);
       Z = Integer.parseInt (argv [3]);
    }
    else
    {
       X = 10;
       Y = 9;
       Z = 12;
    }
```

shared variables

last-defs

first-use

first-use

last-def

last-defs

```
          ok = Root (X, Y, Z);
          if (ok)
            System.out.println
                (“Quadratic: ” + Root1 + Root2);
          else
              System.out.println (“No Solution.”);
    }

    // Three positive integers, finds the quadratic root
    private static boolean Root (int A, int B, int C)
    {
        double D;
        boolean Result;
        D = (double) (B*B) - (double) (4.0*A*C);
        if (D < 0.0)
        {
            Result = false;

            return (Result);
        }
        Root1 = (double) ((-B + Math.sqrt (D)) / (2.0*A));
        Root2 = (double) ((-B – Math.sqrt (D)) / (2.0*A));
        Result = true;
        return (Result);
    } / /End method Root
} // End class Quadratic
```

Quadratic – Coupling DU-pairs

Pairs of locations: method name, variable name, statement

(main (), X, 15) – (Root (), A, 38)

(main (), Y, 16) – (Root (), B, 38)

(main (), Z, 17) – (Root (), C, 38)

(main (), X, 21) – (Root (), A, 38)

(main (), Y, 22) – (Root (), B, 38)

(main (), Z, 23) – (Root (), C, 38)

(Root (), Root1, 44) – (main (), Root1, 28)

(Root (), Root2, 45) – (main (), Root2, 28)

(Root (), Result, 41) – (main (), ok, 26)

(Root (), Result, 46) – (main (), ok, 26)

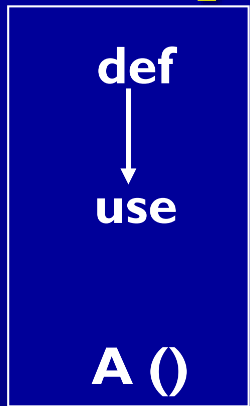
Coverage criteria for coupling graphs

- The data flow criteria can now be applied to coupling graphs:
- All-Defs coverage requires that a path be executed from every last-def to at least one first-use
  - *All-Coupling-Def* coverage
- All-Uses coverage requires that a path be executed from every last-def to every first-use
  - *All-Coupling-Use* coverage
- All-du-Paths coverage requires that we tour every simple path from every last-def to every first-use
  - *AllCoupling-du-Paths* coverage

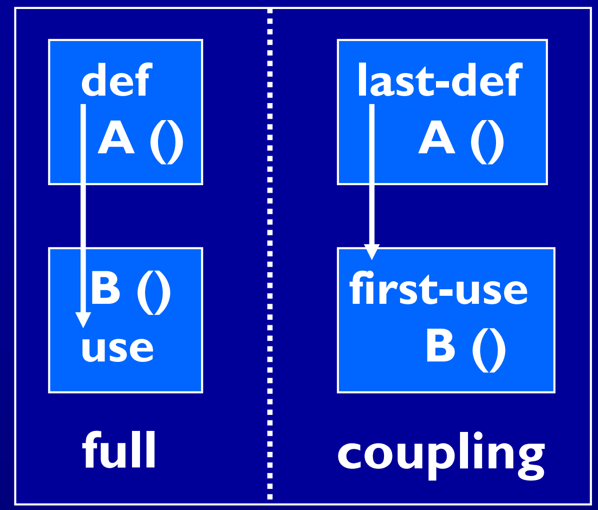
Coupling Data Flow Notes

- Only variables that are used or defined in the callee
- Implicit initializations of class and global variables
- Transitive DU-pairs are too expensive to handle
  - A calls B, B calls C, and there is a variable defined in A and used in C
- Arrays : a reference to one element is considered to be a reference to all elements

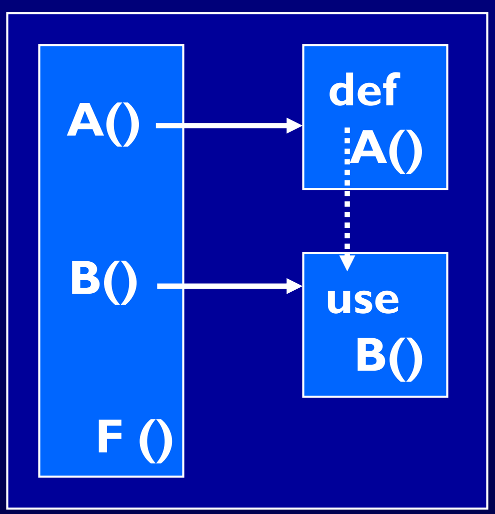
Types of Def-Use Pairs



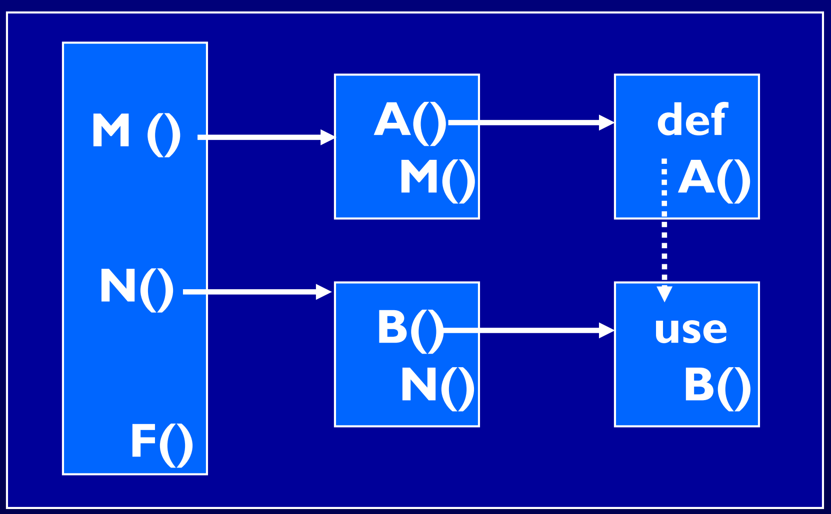
intra-procedural data flow
(within the same unit)



inter-procedural
data flow



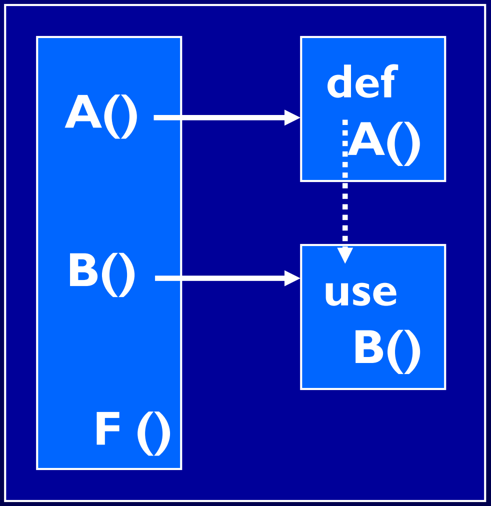
object-oriented direct
coupling data flow



object-oriented indirect
coupling data flow

Def-use pairs in object-oriented software

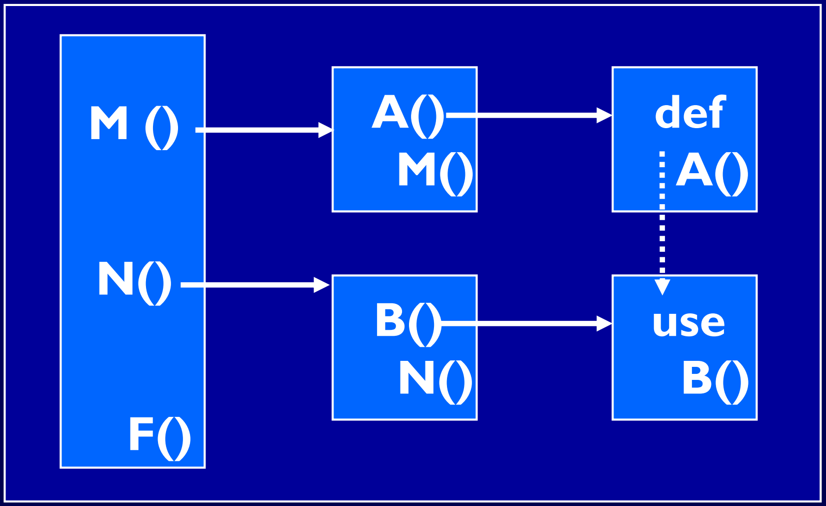
- DU pairs are usually based on the class variables
  - Class (State) Variables : The variables declared at the class level, often private
- **Direct coupling**:
  - A coupling method, F(), calls two methods, A() and B()
  - A() defines a variable and B() uses it
  - A() and B() could be in the same class,
  - Or they could be in different classes and accessing the same global variables
  - A() and B() may be called through the same object instance or not
    - o.A() and o.B()
    - o1.A() and o2.B()



object-oriented direct coupling data flow

Def-use pairs in object-oriented software

- **Indirect coupling:**
  - F() calls two methods, M() and N() which in turn call two other methods, A() and B()
  - A() defines a variable and B() uses it
  - So the reference is indirect
  - The analysis for indirect du-pairs is considerably more complicated than for direct du-pairs

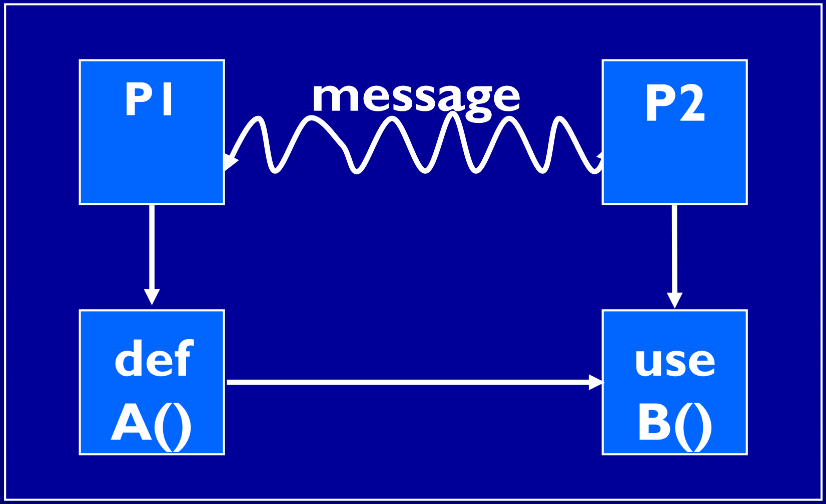


object-oriented indirect coupling data flow

OO Data Flow Summary

- The defs and uses could be in the same class, or different classes
- Researchers have applied data flow testing to the direct coupling OO situation
  – Has not been used in practice
  – No tools available
- Indirect coupling data flow testing has not been tried either in research or in practice
  – Analysis cost may be prohibitive

Web Applications and Other Distributed Software



distributed software data flow

- "message" could be HTTP, RMI, or other mechanism
- A() and B() could be in the same class or accessing a persistent variable such as in a web session
- Beyond current technologies

Summary—What Works?

- Call graphs are common and very useful ways to design integration tests
- Inter-procedural data flow is relatively easy to compute and results in effective integration tests
- The ideas for OO software and web applications are preliminary and have not been used much in practice