Testing: Dataflow Coverage

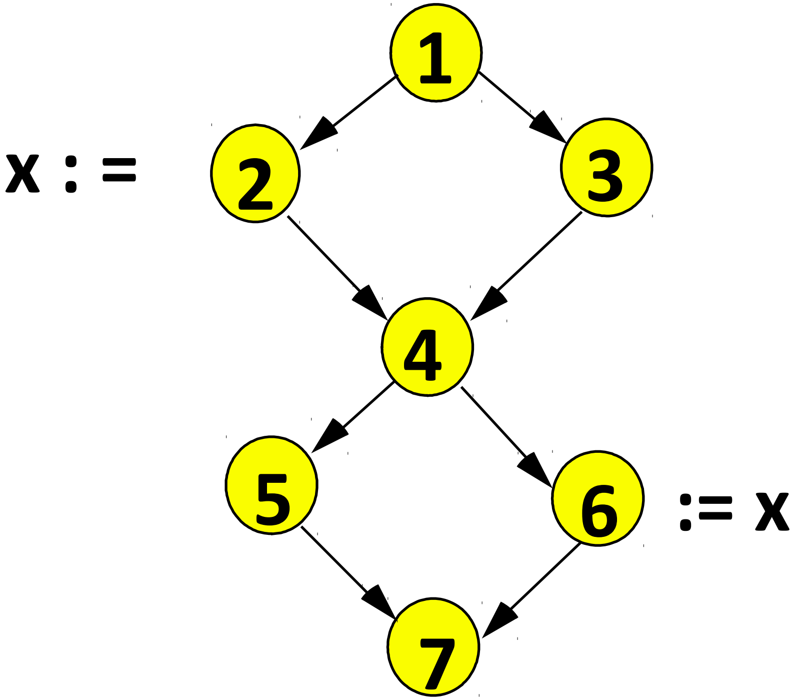
Testing, Quality Assurance, and Maintenance
Winter 2017

Thibaud Lutellier

based on slides by Prof. Arie Gurfinkel, Marsha Chechik
and Prof. Lin Tan

UNIVERSITY OF
WATERLOO

Non-looping Path Selection Problem



All branches **1, 2, 4, 5, 7**

1, 3, 4, 6, 7

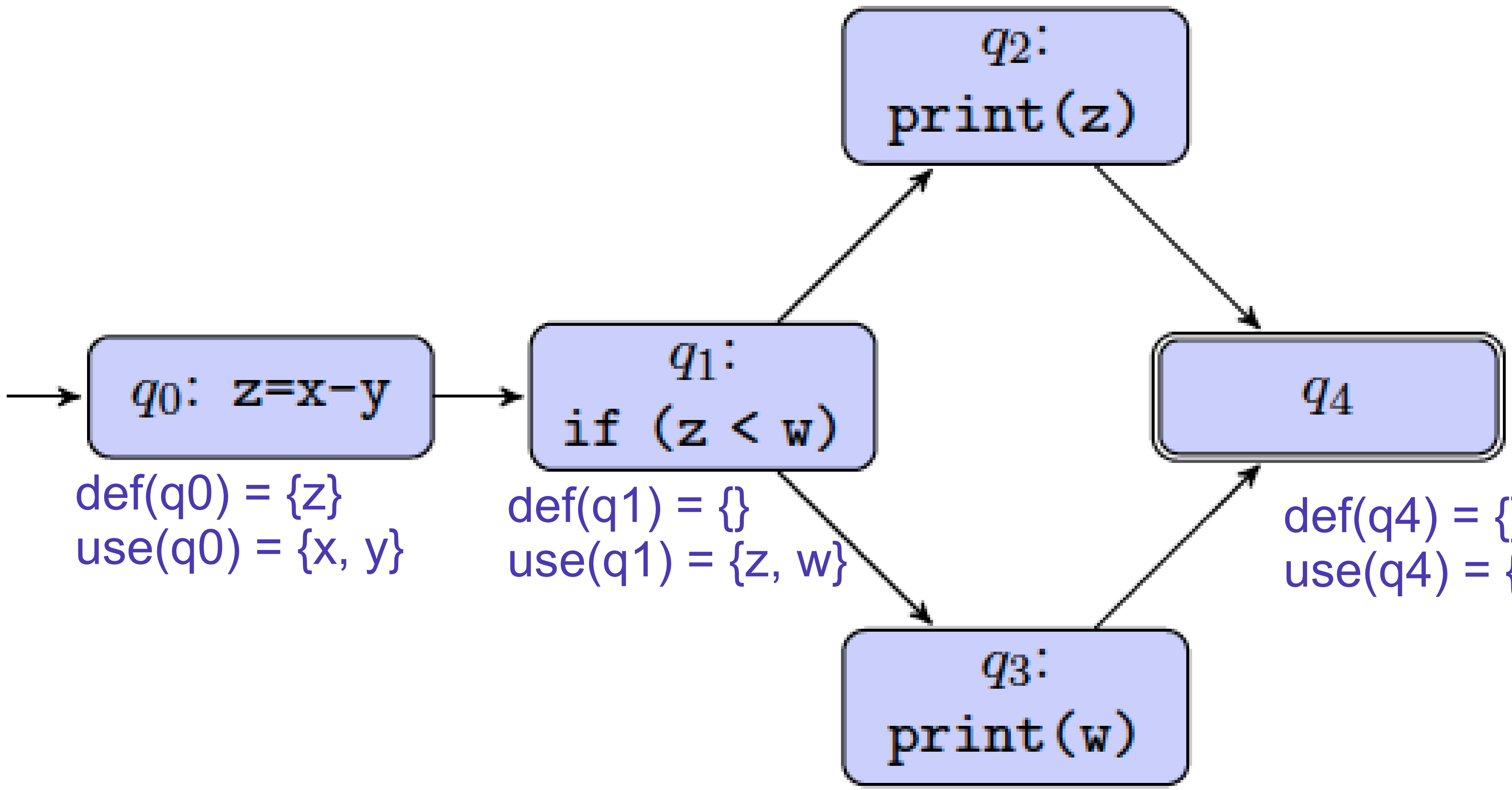
does not exercise the relationship between the definition of X in statement 2 and the reference to X in statement 6.

Def, Use

Goal: Try to ensure that values are computed and used correctly.

- **def**: a location where a value for a variable is stored in memory.
- **use**: a location where a variable's value is accessed.
- **def(n)**: The set of variable defined by node n.
- **use(n)**: the set of variable used by node n

Def-Use CFG



def(q2) = {}
use(q2) = {z}

def(q0) = {z}
use(q0) = {x, y}

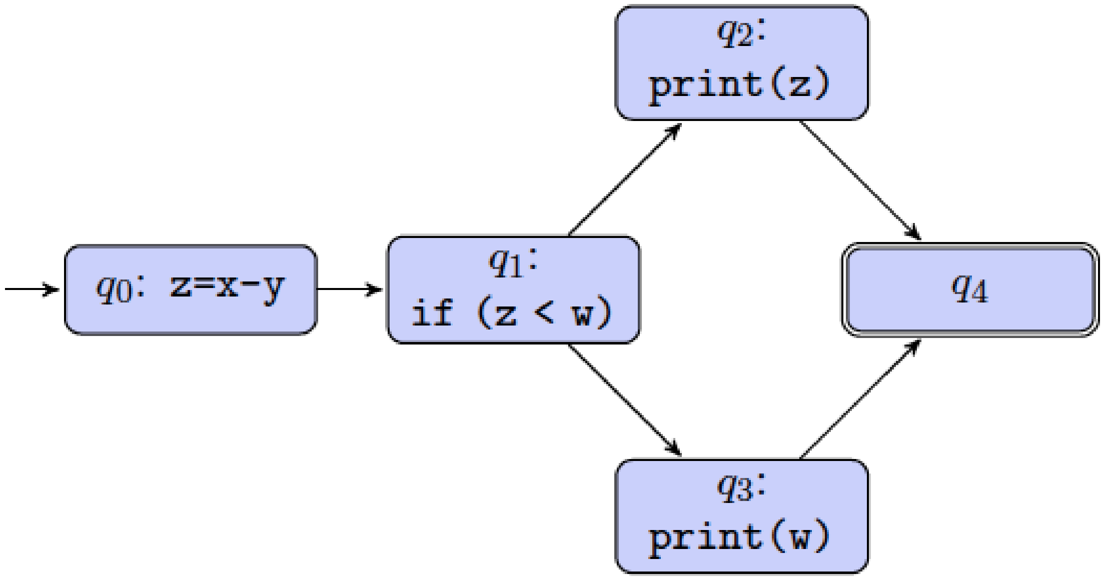
def(q1) = {}
use(q1) = {z, w}

def(q4) = {},
use(q4) = {}

def(q3) = {},
use(q3) = {w}

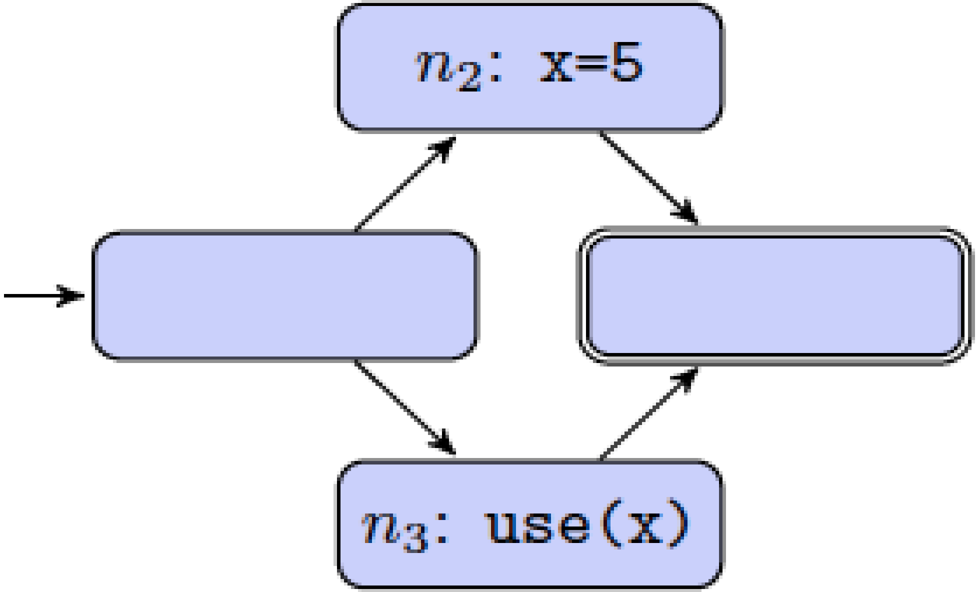
DU-path, Def-clear and Reach

- A **definition-clear path (def-clear)** p with respect to x is a sub-path where x is not defined at any of the nodes in p
- A **du-path** is a simple path where the initial node of the path is the only defining node of x in the path.
- **Reach**: if there is a def-clear path from the nodes m to p with respect to x, then the def of x at m reaches the use at p.



Example for z.:
- du-path: q0,q1,q2
- def-clear: q1,q2,q4
- q0 reaches q2.

Does the def at n2 reach the use at n3?



Assumptions (1/2)

no edges of the form (n,n_{start}) or (n_{finish} ,n)

no edges of the form (n,n)

there is at most one edge (m,n) for all m,n

every control graph is well-formed

- Connected
- Single start and single final node

every loop has a single entry and a single exit

Assumptions (2/2)

at least one variable is associated with a node representing a predicate

no variable definitions are associated with a node representing a predicate

- no predicates of the form: `(x++ > 3)`

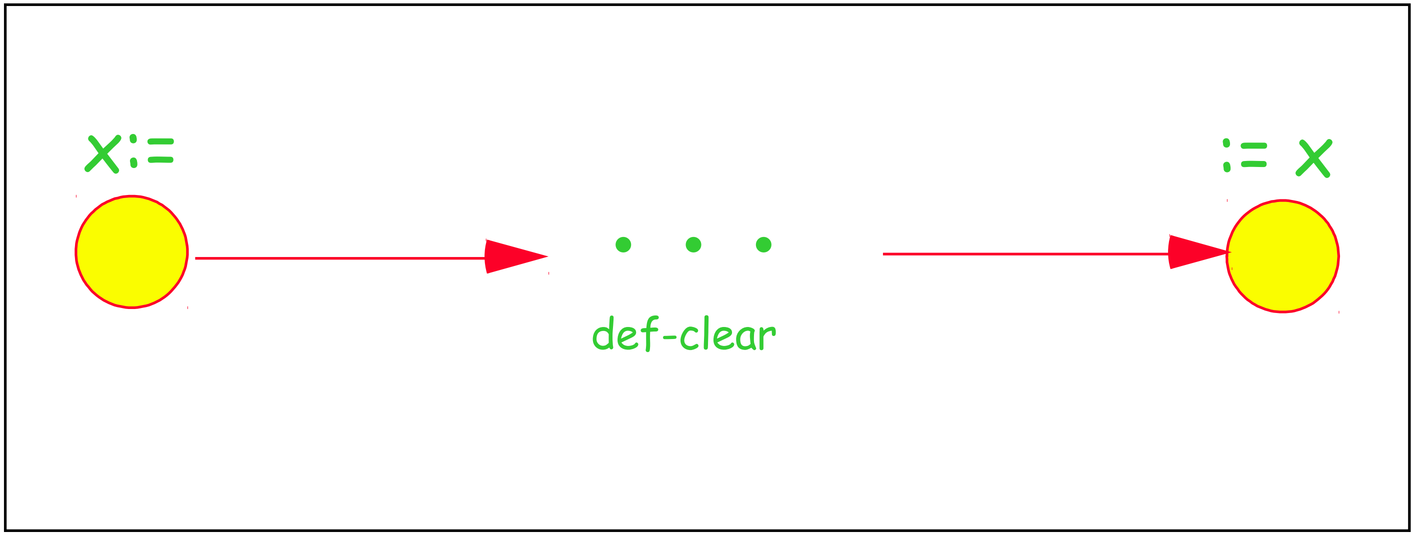
every definition of a variable reaches at least one use of that variable

every use is reached by at least one definition

every control graph contains at least one variable definition

All-Defs-Coverage (ADC)

- **All-Defs-Coverage (ADC):**
  Some def-clear sub-path from **each definition** to **some** use reached by that definition



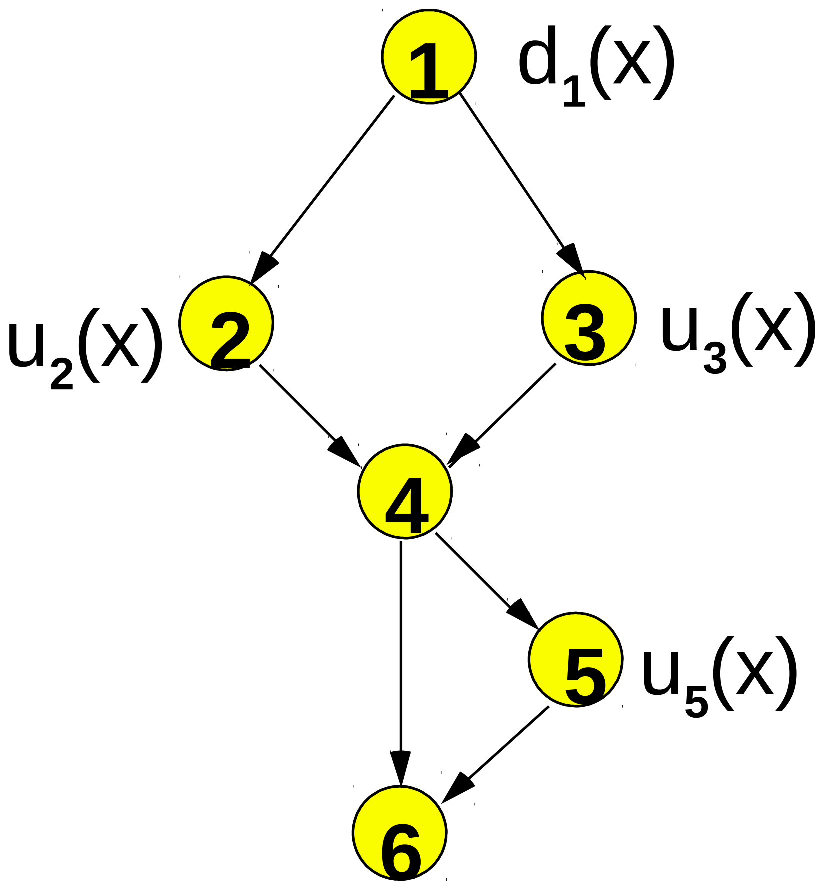
All-Defs Coverage: Example

Requires:

$d_1(x)$ to a use

Satisfactory Path:

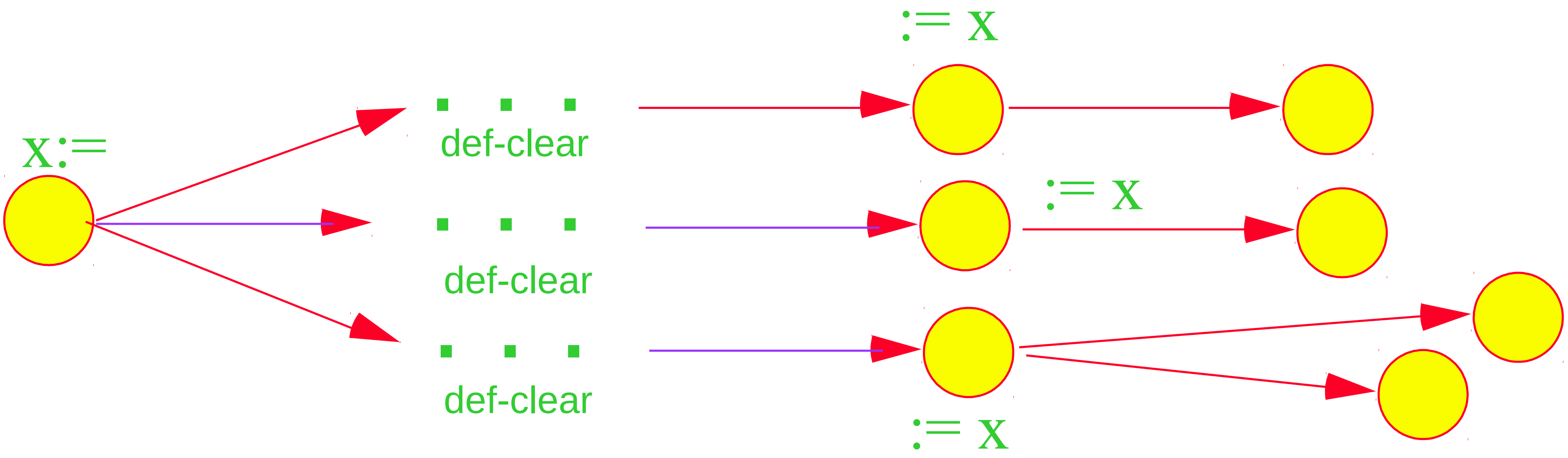
[1, 2, 4, 6]



All-Uses Coverage (AUC)

All-Uses Coverage (AUC)

Some definition-clear subpath from **each** definition to **each** use reached by that definition and each successor node of the use



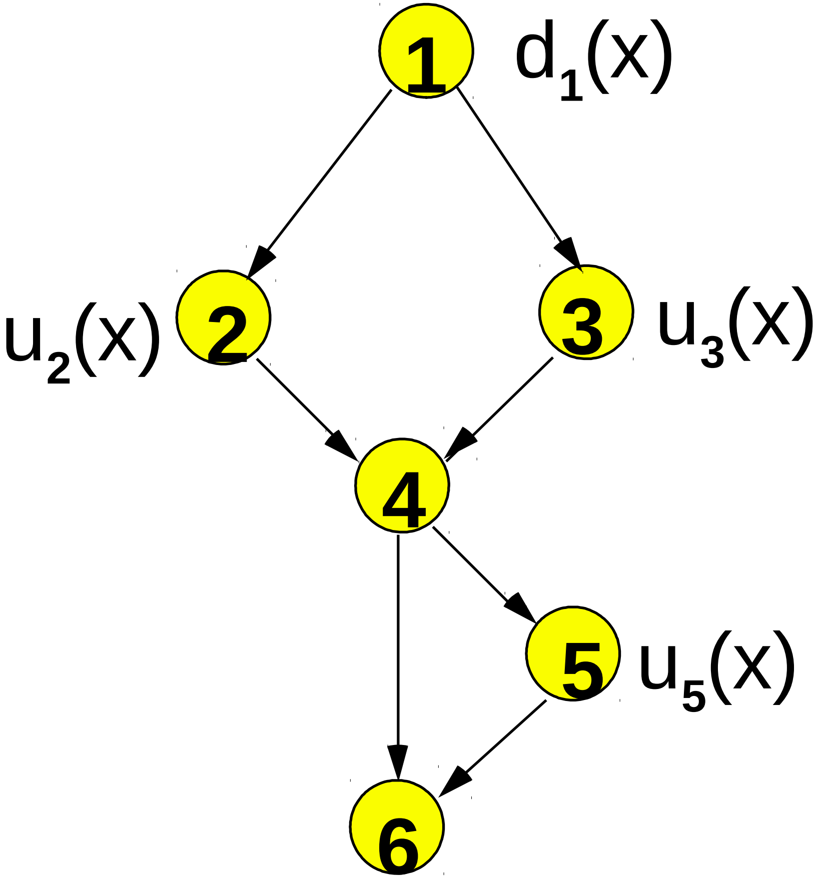
All-Uses Coverage: Example

Requires:

- d1(x) to u2(x)
- d1(x) to u3(x)
- d1(x) to u5(x)

Satisfactory Paths:

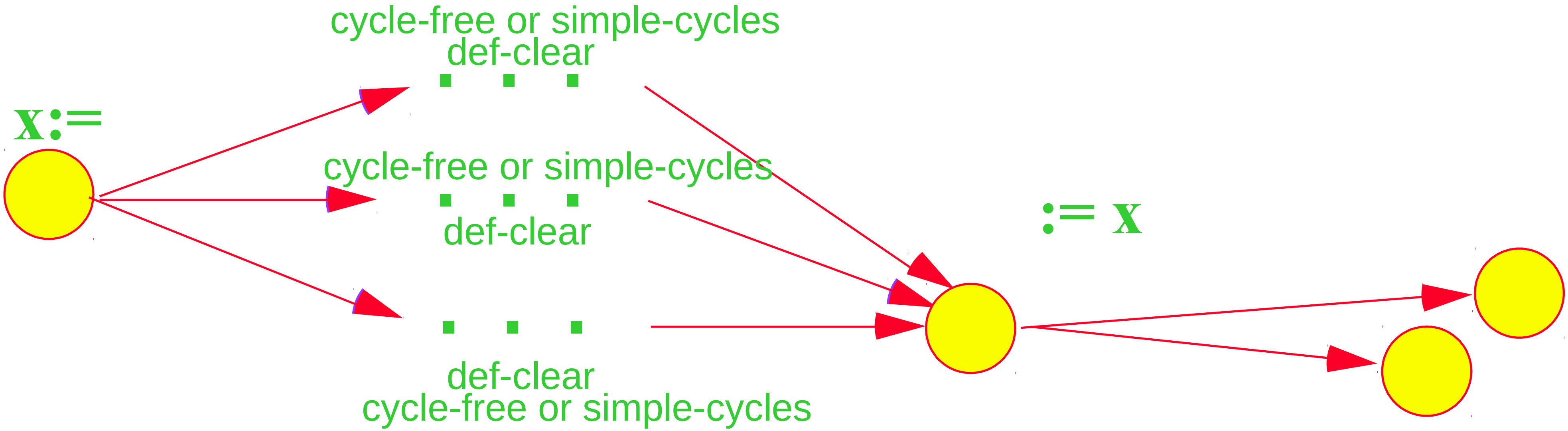
- [1, 2, 4, 5, 6]
- [1, 3, 4, 6]



All-Du-Paths Coverage (ADUPC)

- **All-Du-Paths Coverage (ADUPC):**

  All def-clear sub-paths that are cycle-free or simple-cycles from **each definition** to **each use** reached by that definition and each successor node of the use



cycle-free or simple-cycles
def-clear
x:=
cycle-free or simple-cycles
def-clear
:= x
def-clear
cycle-free or simple-cycles

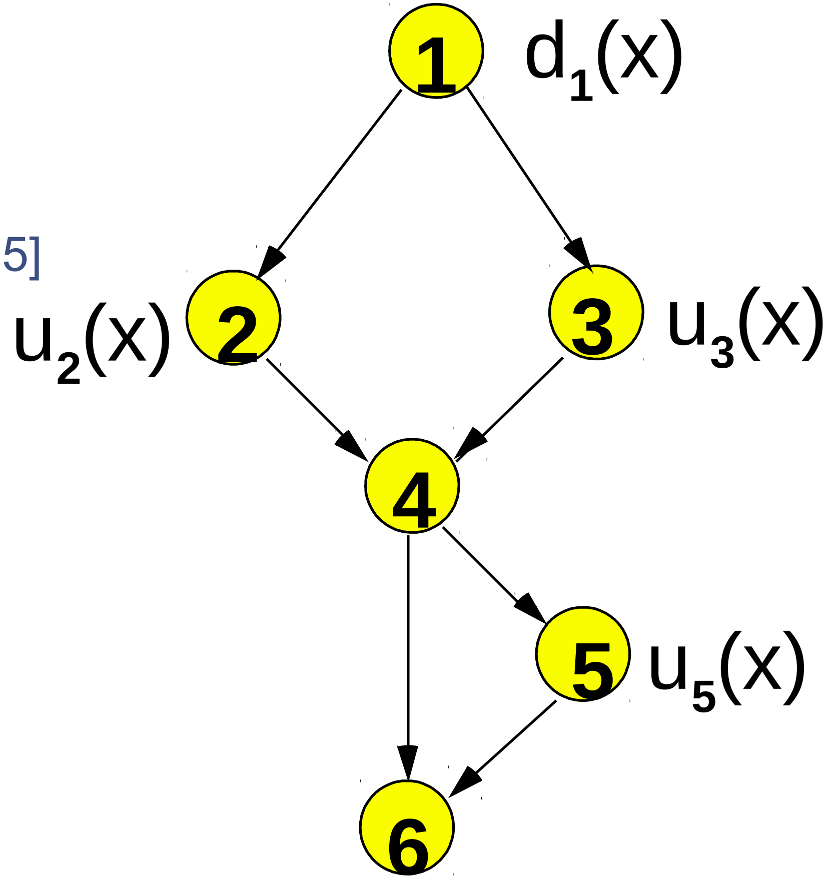
All-Du-Paths Coverage: Example

Requires:

- All d1(x) to u2(x): [1,2]
- All d1(x) to u3(x): [1,3]
- All d1(x) to u5(x): [1,2,4,5], [1,3,4,5]

Satisfactory Paths:

- [1, 2, 4, 5, 6]
- [1, 3, 4 ,5, 6]



Data Flow Test Criteria

Assume definitions occur on nodes only, for the following definitions.

First, we make sure every def reaches a use:

All-Defs Coverage (ADC) : For each set of du-paths S = du (n, v), TR contains at least one path d in S.

Then we make sure that every def reaches all possible uses:

All-Uses Coverage (AUC) : For each set of du-paths to uses S = du (ni, nj, v), TR contains at least one path d in S.

Finally, we cover all the du-paths between defs and uses:

All-du-Paths Coverage (ADUPC) : For each set S = du (ni, nj, v), TR contains every path d in S.

x

Problems with data flow coverage criteria

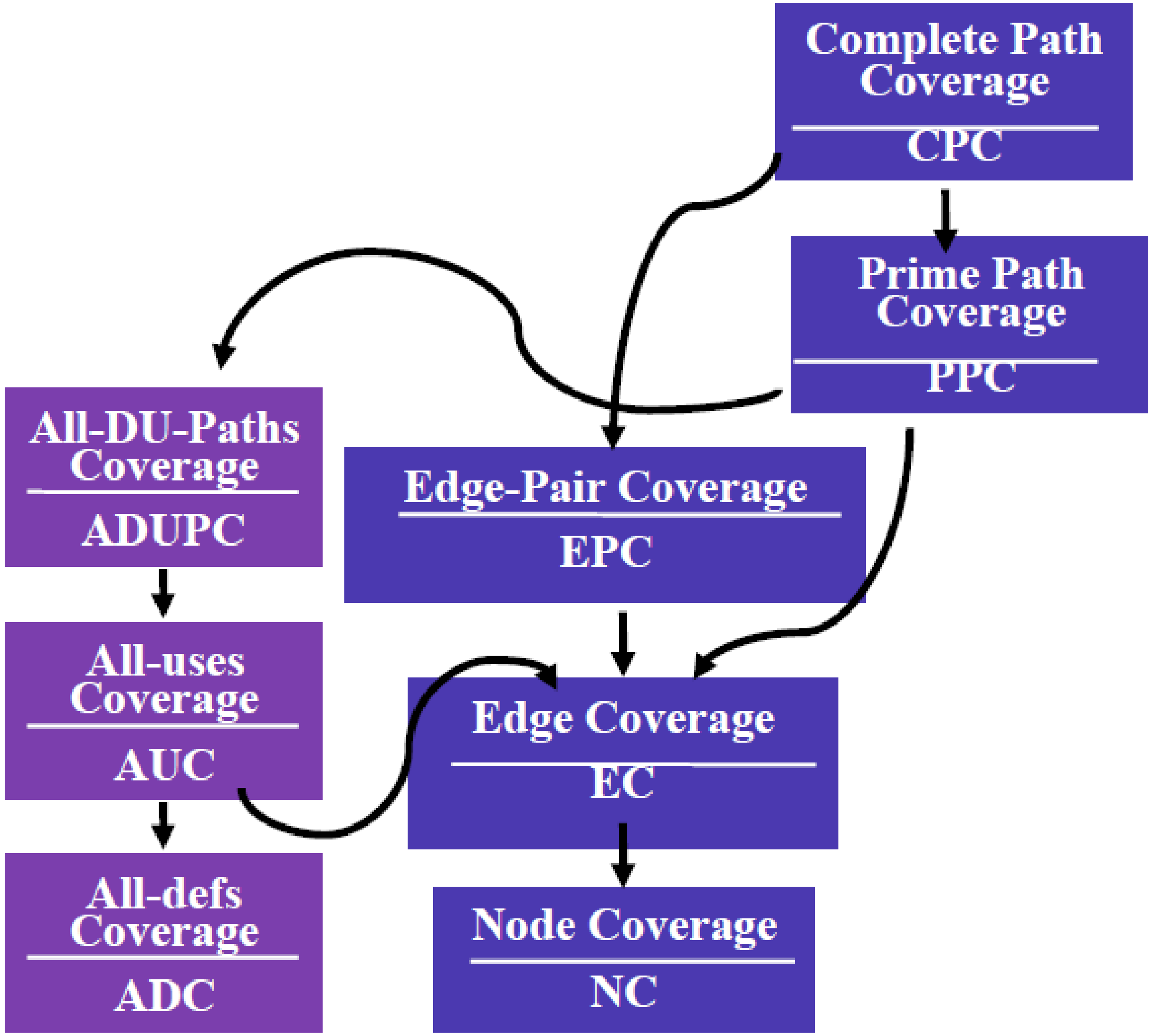
Infeasible paths

- Don’t usually get 100% coverage

Need to understand fault detection ability

Artificially combines control with data flow

Graph Coverage Criteria Subsumption



AUC & EC

Under what additional assumptions, AUC subsumes EC?

- For every node with multiple outgoing edges, at least one variable is used on each out edge, (and the same variables are used on each out edge).

Reasoning (key points, not a full proof):

- Assume test set T satisfies AUC.
- Each edge e1 must be either a branch edge or not.
- If e1 is a branch edge, then it has a use, thus it must be covered by T.
- If e1 is not a branch edge, the closest branch edge before e1, denoted as e2, should be covered T, and if e2 is covered by a test, e1 must also be. If the closed branch edge doesn't exit, then the program has no branch edges; therefore e1 must be covered by T.

Compiler tidbit

In a compiler, we use intermediate representations to simplify expressions, including definitions and uses.

For instance, we would simplify:

- x = foo(y + 1, z * 2)

To:

- a = y+1
- b = z*2
- x = foo(a,b)

Exercise (1/3)

Answer questions (a)-(f) for the graph defined by the following sets:

- N ={0, 1, 2, 3, 4, 5, 6, 7}
- N0 = {0}
- Nf = {7}
- E = {(0,1), (1, 2), (1, 7), (2, 3), (2, 4), (3, 2), (4, 5), (4, 6), (5, 6), (6, 1)}
- def(0)=def(3)=use(5)=use(7) = {X}

Also consider the following test paths:

- t1 = [0,1,7]
- t2 = [0,1,2,4,6,1,7]
- t3 = [0,1,2,4,5,6,1,7]
- t4 = [0,1,2,3,2,4,6,1,7]
- t5 = [0,1,2,3,2,3,2,4,5,6,1,7]
- t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]

Exercise (2/3)

(a) Draw the graph.

(b) List all of the du-paths with respect to x. (Note: Include all du-paths, even those that are subpaths of some other du-paths).

(c) For each test path, determine which du-paths that test path du-tours. Consider direct touring only. *Hint: A table is a convenient format for describing the relationship.*

Exercise (3/3)

(d) List a minimal test set that satisfies *all-defs* coverage with respect to x (Direct tours only). Use the given test paths.

(e) List a minimal test set that satisfies *all-uses* coverage with respect to x. (Direct tours only). Use the given test paths.

(f) List a minimal test set that satisfies *all-du-paths* coverage with respect to x. (Direct tours only). Use the given test paths.

Exercise - cont.

(b) du (0, 5, x): i: [0,1,2,4,5]

du (0, 7, x): ii: [0,1,7]

du (3, 5, x): iii: [3,2,4,5]

du (3, 7, x): iv: [3,2,4,6,1,7]

v: [3,2,4,5,6,1,7]

(c)

Test Path	**Du-tour directly**
t1 = [0,1,7]	ii
t2 = [0,1,2,4,6,1,7]	/
t3 = [0,1,2,4,5,6,1,7]	i
t4 = [0,1,2,3,2,4,6,1,7]	iv
t5 = [0,1,2,3,2,3,2,4,5,6,1,7]	iii, v
t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]	/

Exercise - Partial Solutions.

(d) List a minimal test set that satisfies *all-defs* coverage with respect to x (Direct tours only). Use the given test paths.

- {t1, t4} or {t1, t5} or {t3, t4} or {t3, t5}
- Why isn’t {t1, t6} a correct answer?
  - t6 does not du-tour any path in du(3, x).

(e) List a minimal test set that satisfies *all-uses* coverage with respect to x. (Direct tours only). Use the given test paths.

- {t1, t3, t5}

(f) List a minimal test set that satisfies *all-du-paths* coverage with respect to x. (Direct tours only). Use the given test paths.

- {t1, t3, t4, t5}

Interprocedural Data Flow

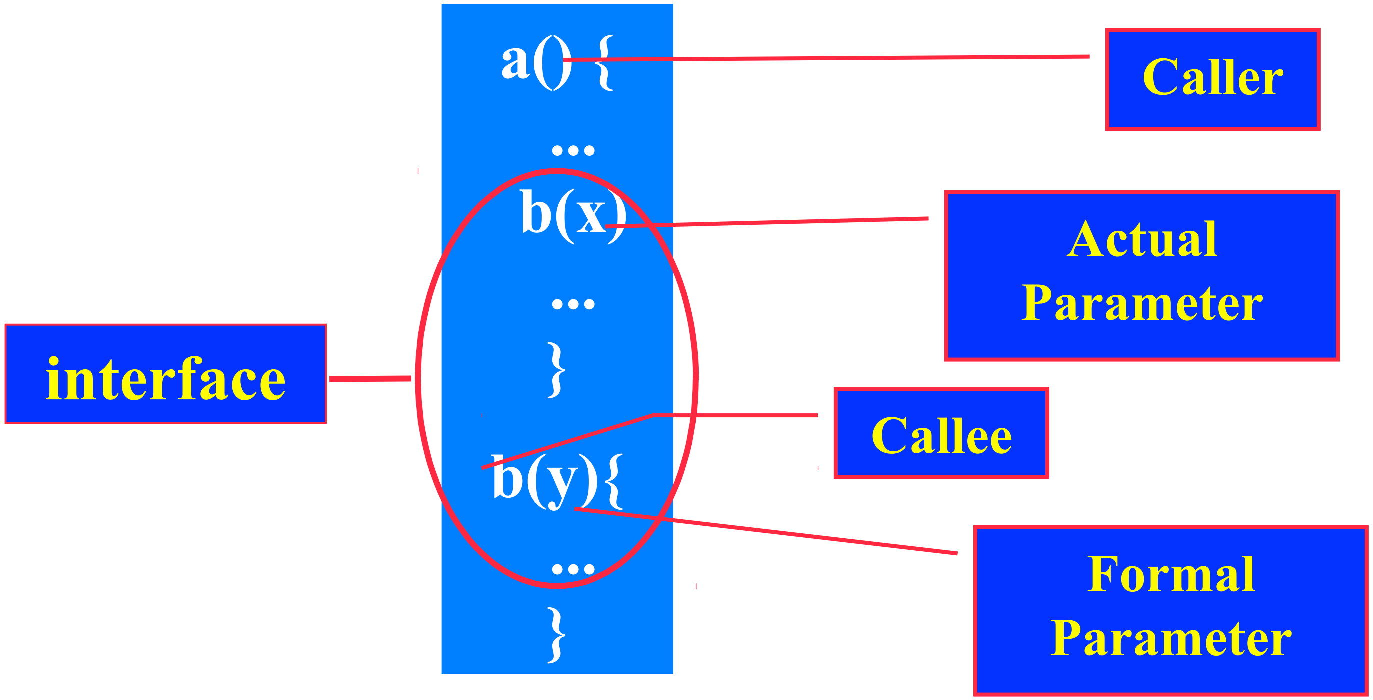
Data flow couplings among units are more complicated than control flow couplings.

When values are passed, they "change names".

Finding which uses a def can reach is very difficult.

- **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

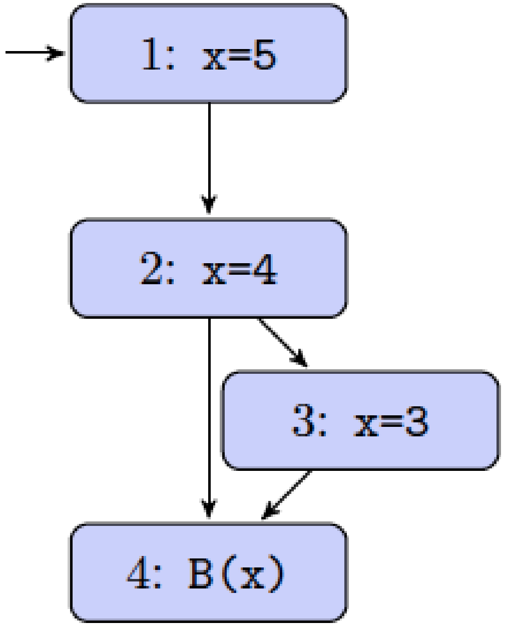
If we focus on the interface, then we just need to consider the last definitions of variables before calls and returns and first uses inside units and after calls.

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 call site to the nodes.

Last-defs & First-uses



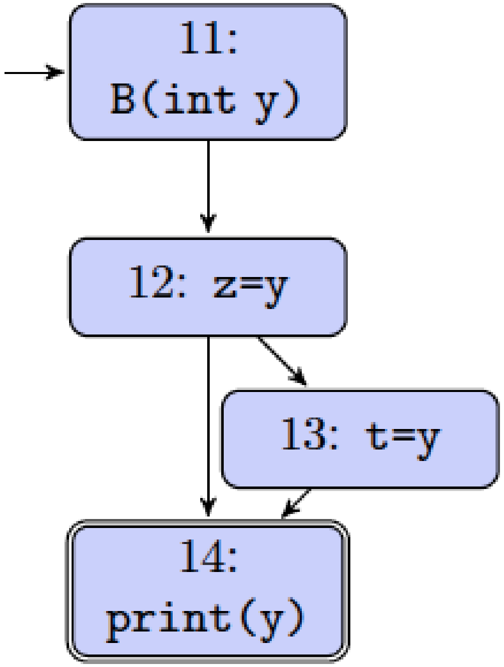
1: x=5

2: x=4

3: x=3

4: B(x)

last-defs are 2, 3



11: B(int y)

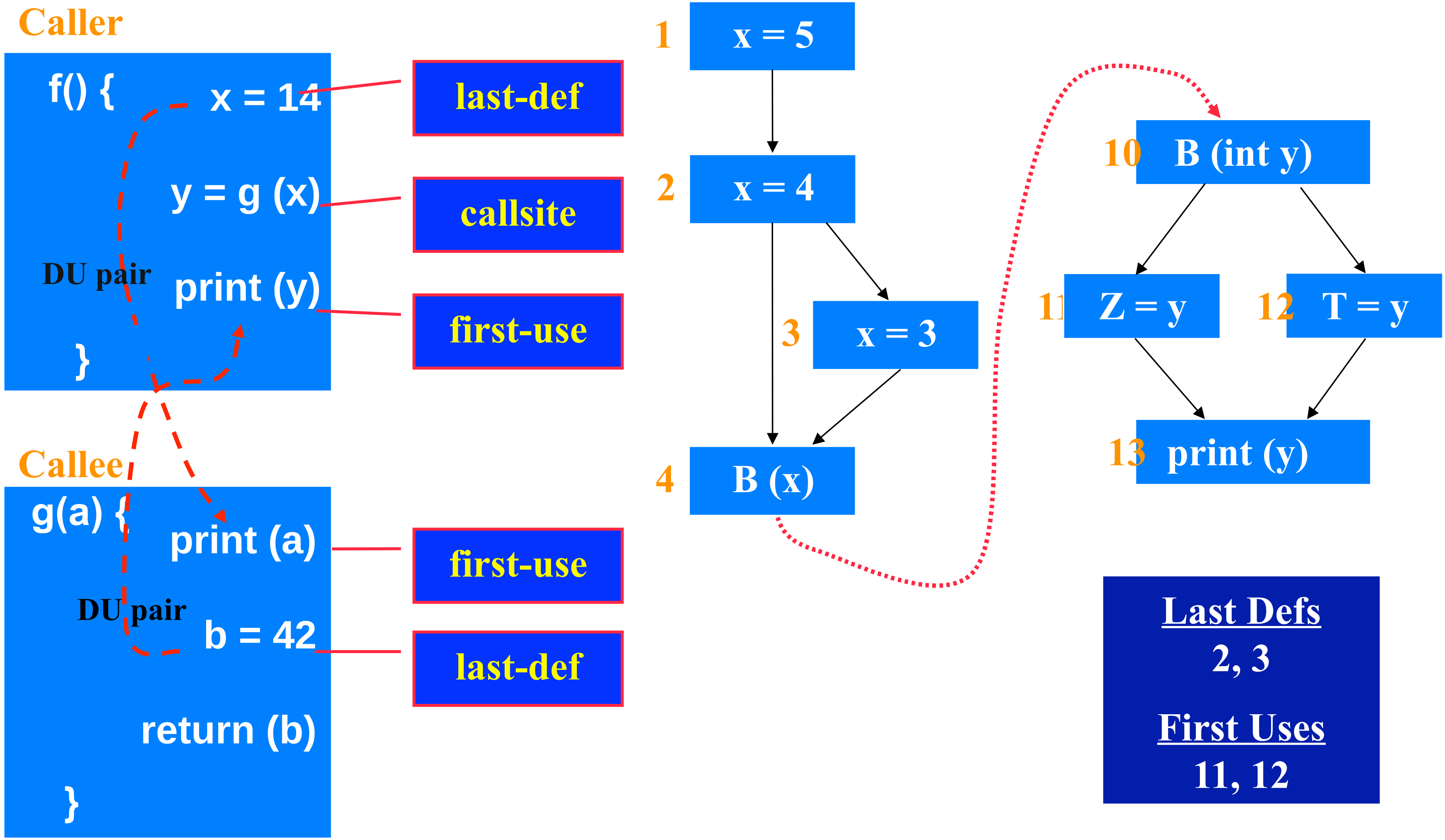
12: z=y

13: t=y

14: print(y)

the first-use is 12

Example Inter-procedural DU Pairs



Caller

```
f() {
    x = 14
    y = g (x)
    print (y)
}
```

Callee

```
g(a) {
    print (a)
    b = 42
    return (b)
}
```

Last Defs
2, 3

First Uses
11, 12

Example: 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;
     }
          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)
     {
        float D;
        boolean Result;
        D = (float) Math.pow ((double)B,

             (double2-4.0)*A*C );
        if (D < 0.0)
        {
           Result = false;

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

     } // End class Quadratic
```

Example: 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;
      }
          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)
 {
     float D;
     boolean Result;
     D = (float) Math.pow ((double)B,
          (double2-4.0)*A*C );
     if (D < 0.0)
     {
        Result = false;
        return (Result);
     }
     Root1 = (float) ((-B + Math.sqrt(D))/(2.0*A));
     Root2 = (float) ((-B – Math.sqrt(D))/(2.0*A));
     Result = true;
     return (Result);
 } / /End method Root

 } // End class Quadratic
```

Example: 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;
     }
         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)
 {
    float D;
    boolean Result;
    D = (float) Math.pow ((double)B,
          (double2-4.0)*A*C );
    if (D < 0.0)
    {
      Result = false;
      return (Result);
    }
    Root1 = (float) ((-B + Math.sqrt(D))/(2.0*A));
    Root2 = (float) ((-B – Math.sqrt(D))/(2.0*A));
    Result = true;
    return (Result);
 } / /End method Root

 } // End class Quadratic
```

first-use

last-defs

first-uses

last-defs

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)

```
1 // Prog
two nun
2 impor
3
4 class Q
5 {
6  privat
7
8  publi
9  {
10      int X, Y, Z;
11     boolean ok;
12     int controlFlag = Integer.parseInt (argv[0]);
13     if (controlFlag == 1)
14     {
15         X = Integer.parseInt (argv[1]);
16         Y = Integer.parseInt (argv[2]);
17         Z = Integer.parseInt (argv[3]);
18      }
19     else
20     {
21          X = 10;
22          Y = 9;
23          Z = 12;
24      }
```

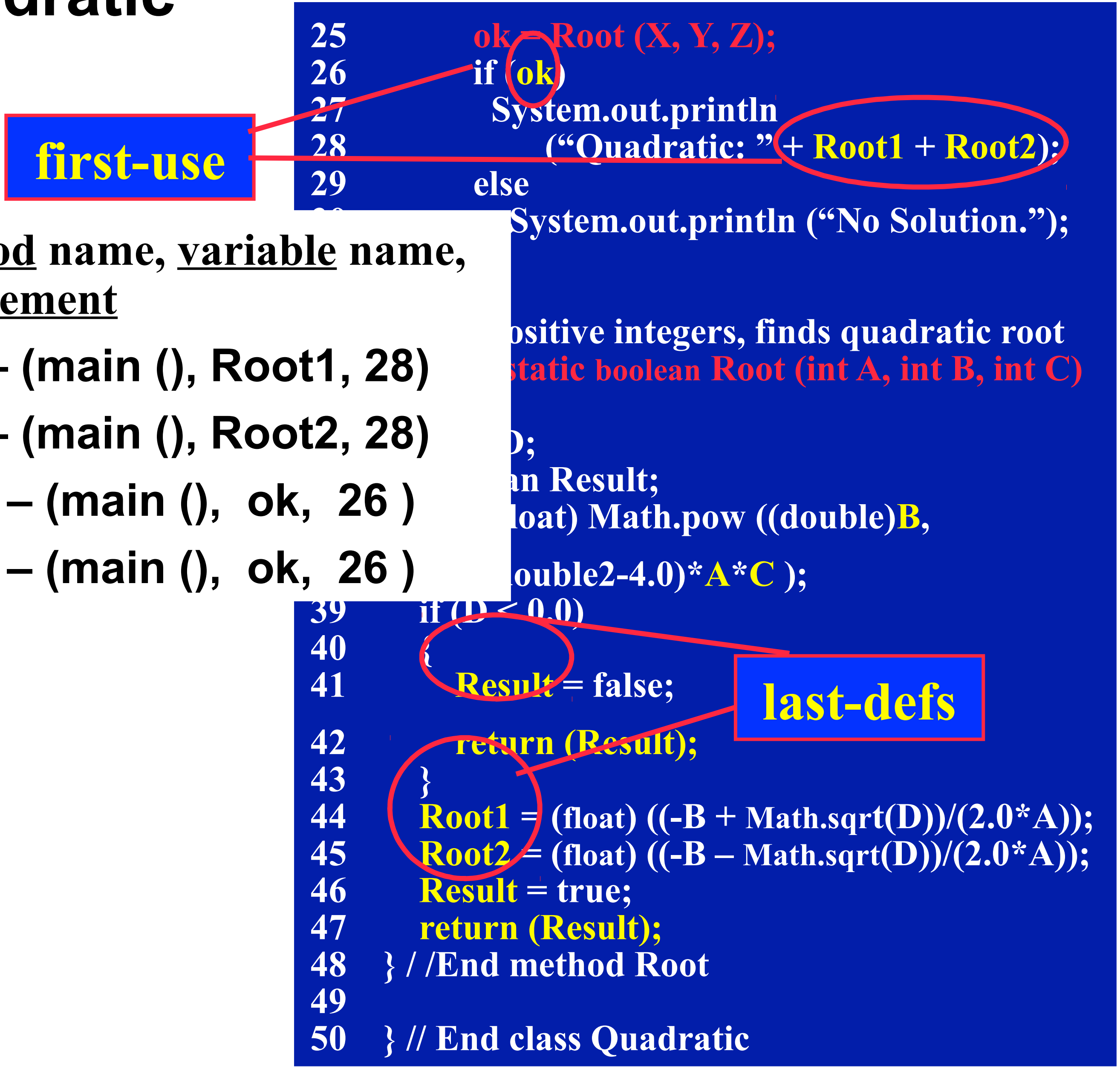
last-defs

```
+ Root2);
olution.”);
dratic root
int B, int C)
36        float D;
37        boolean Result;
38        D = (float) Math.pow ((double)B,
              (double2-4.0)*A*C );
39        if (D < 0.0)
40        {
41           Result = false;
42           return (Result);
43        }
44        Root1 = (float) ((-B + Math.sqrt(D))/(2.0*A));
45        Root2 = (float) ((-B – Math.sqrt(D))/(2.0*A));
46        Result = true;
47        return (Result);
48    } / /End method Root
49
50    } // End class Quadratic
```

first-uses

UNIVERSITY OF WATERLOO

Example: Quadratic



```
25          ok = Root (X, Y, Z);
26          if (ok)
27            System.out.println
28                ("Quadratic: " + Root1 + Root2);
29          else
              System.out.println ("No Solution.");


            ositive integers, finds quadratic root
            static boolean Root (int A, int B, int C)

            D;
            an Result;
            loat) Math.pow ((double)B,

            ouble2-4.0)*A*C );
39        if (D < 0.0)
40        {
41          Result = false;
42          return (Result);
43        }
44        Root1 = (float) ((-B + Math.sqrt(D))/(2.0*A));
45        Root2 = (float) ((-B – Math.sqrt(D))/(2.0*A));
46        Result = true;
47        return (Result);
48    } / /End method Root
49
50    } // End class Quadratic
```

Pairs of locations: method name, variable name, statement

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

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

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

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

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)

Strengths and Weaknesses of Graph Coverage:

Must create graph

Node coverage is usually easy, but cycles make it hard to get good coverage in general.

Incomplete node or edge coverage point to deficiencies in a test set.

Summary

Summarizing Structural Coverage:

- Generic; hence broadly applicable
- Uses no domain knowledge

Summarizing Dataflow Coverage:

- Definitions and uses are useful but hard to reason.

Miscellaneous other notes:

- Control-flow graphs are manageable for single methods, but not generally more than that.
- Use call graphs to represent multiple methods, hiding details of each method.
- When we want to test du-paths across multiple elements, use first-use/last-def heuristic.