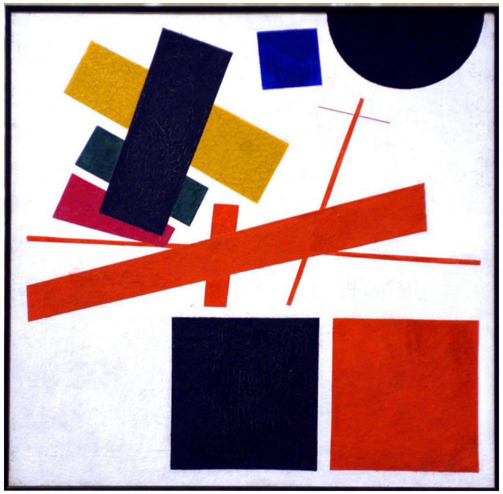
CPEN 422
Software Testing and Analysis



Data Flow Analysis

Structural Testing Beyond Control Flow

- Motivation to go beyond CF:
  - Node and edge coverage don't test interactions
  - Path-based criteria require impractical number of test cases
    - And only a few paths uncover additional faults, anyway
  - Need to distinguish "important" paths
- Intuition: Statements interact through *data flow*
  - Value computed in one statement, used in another
  - Bad value computation revealed only when it is used

Data Flow Anomaly

- Anomaly: It is an abnormal way of doing something.
  - Example 1: The second definition of x overrides the first.

```
x = f1(y);
x = f2(z);
```

- Three types of abnormal situations with using variable.
  - Type 1: Defined and then defined again
  - Type 2: Undefined but referenced
  - Type 3: Defined but not referenced

Anomaly Type 1: Defined and then defined again

- Example 1:

  ```
  x = f1(y);
  x = f2(z);
  ```

- Interpretations of Example 1
  - The first statement is redundant.
  - The first statement has a fault -- the intended one might be: w = f1(y).
  - The second statement has a fault – the intended one might be: v = f2(z).
  - There is a missing statement in between the two: v = f3(x).

Note: It is for the programmer to make the desired interpretation.

Anomaly Type 2: Undefined but referenced

- Example:

```
x = 23; y = 8;
x = x – y + w;     /* w not defined yet */
```

- Two interpretations
  - The programmer made a mistake in using w.
  - The programmer wants to use the runtime assigned value of w.

Anomaly Type 3: Defined but not referenced

– Example: Consider

```
x = foo(23)
y = bar(45)
z = y + 2
```

– If x is not used subsequently, we have a Type 3 anomaly.

13.1
13.2

Definition and Uses (DU)

- Data flow testing: **follow** the data

Def-use (DU)

- Variable **definition:** the variable is assigned a value
- Variable **use:** the variable's value is actually used

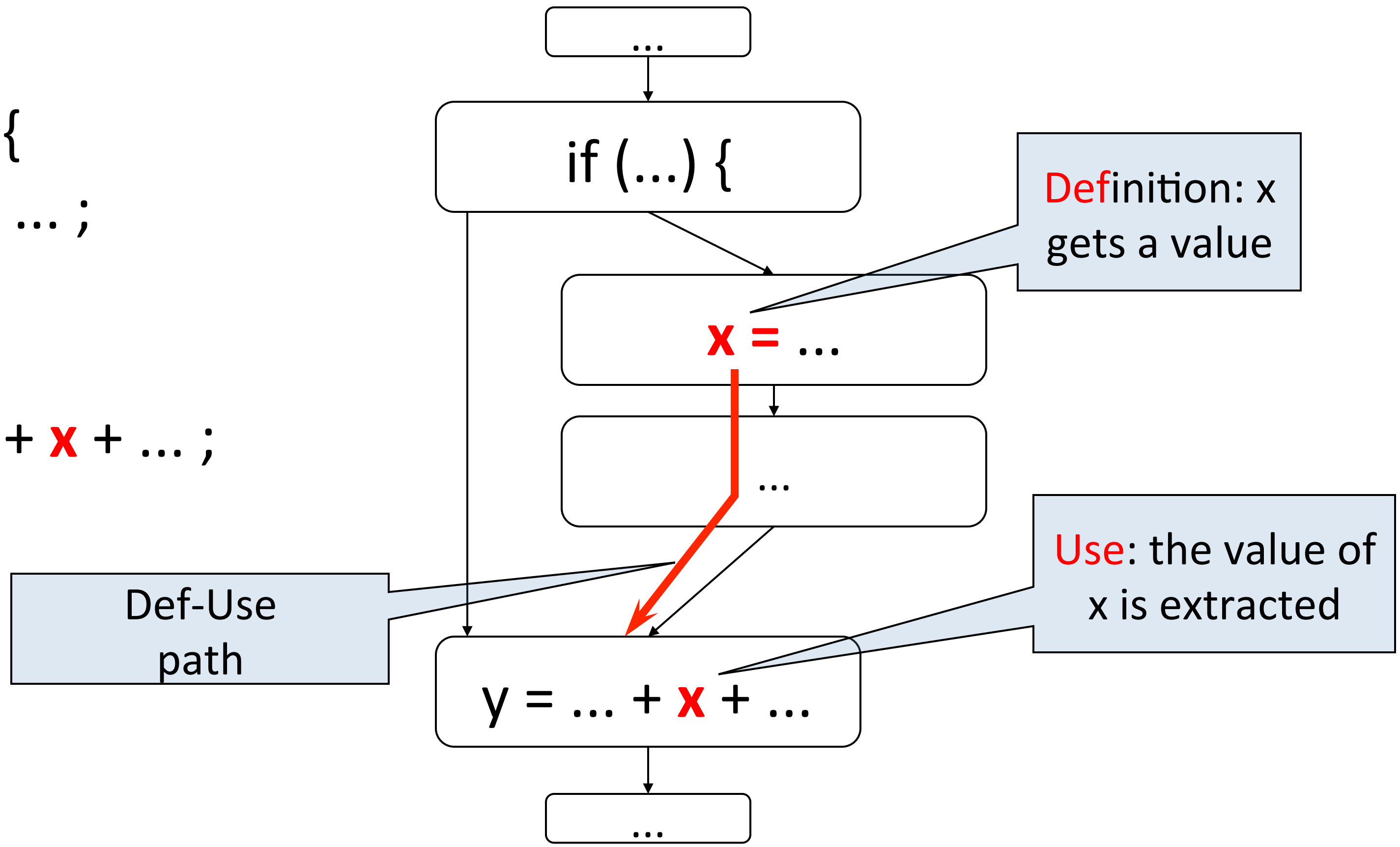
- Goal: search for **def-use pairs**
  - Along **definition-clear** paths

Def-Use Pairs (1)

- A **def-use (du) pair** associates a point in a program where a value is produced with a point where it is used
- **Definition**: where a variable gets a value
  - Variable declaration (often the special value "uninitialized")
  - Variable initialization
  - Assignment
  - Values received by a parameter, e.g., foo(23);
  - Value increments, e.g., x++;
- **Use**: extraction of a value from a variable
  - Expressions
  - Conditional statements
  - Parameter passing
  - Returns

Def-Use Pairs

```
...
if (...) {
    x = ... ;
...
}
y = ... + x + ... ;
```



Identify Definitions and Uses
Draw CFG

```
public int gcd(int x, int y) {
  int tmp;              // A: def x, y, tmp
  while (y != 0) {      // B:
      tmp = x % y;      // C:
      x = y;            // D:
      y = tmp;          // E:
  }
  return x;             // F:
}
```

```
/**  Euclid's algorithm */

public int gcd(int x, int y) {
     int tmp;              // A: def x, y, tmp
     while (y != 0) {      // B: use y
        tmp = x % y;       // C: def tmp; use x, y
        x = y;             // D: def x; use y
        y = tmp;           // E: def y; use tmp
     }
     return x;             // F: use x
  }
```

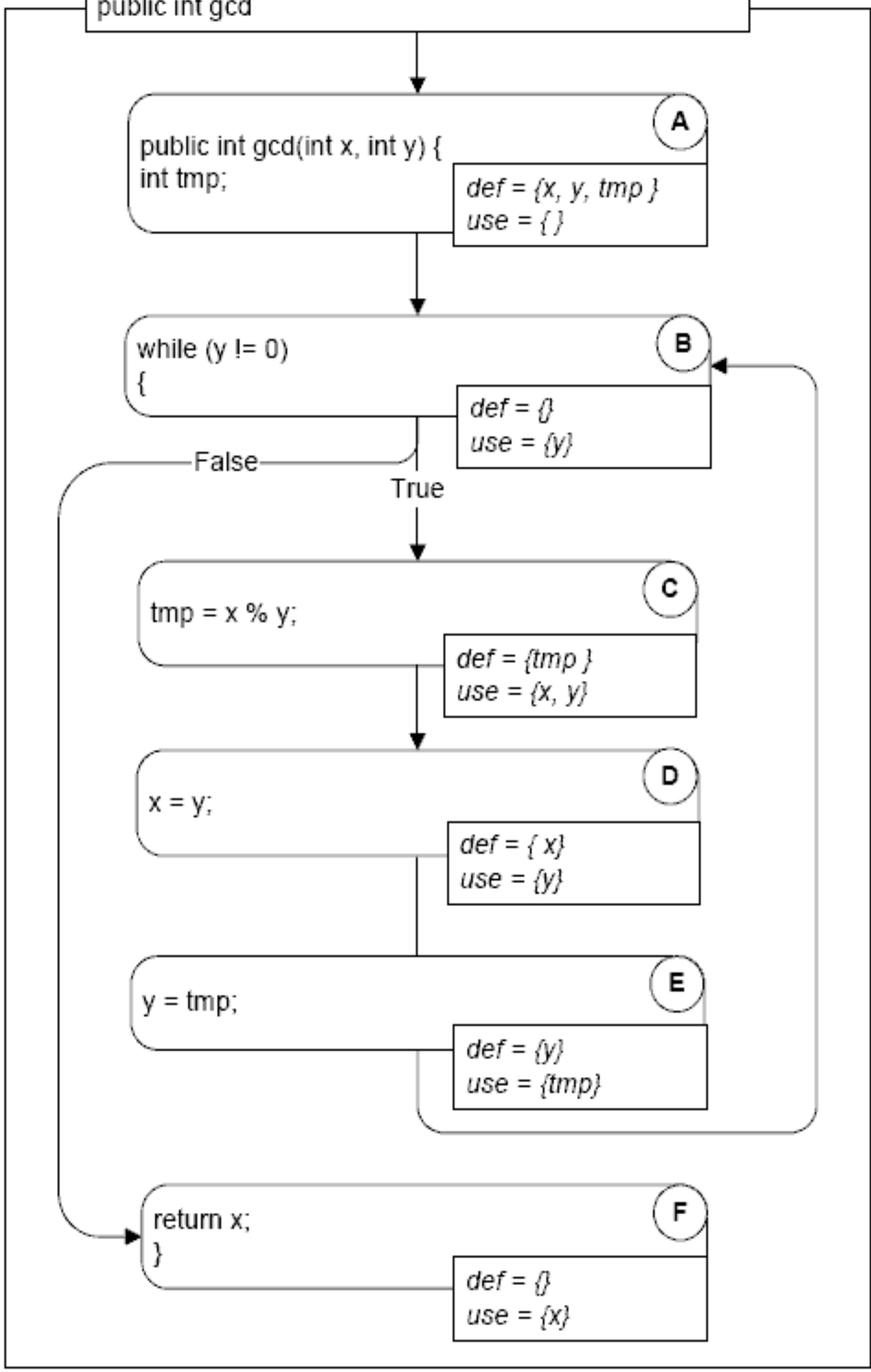


Figure 6.2, page 79

Formal Definition

- DU pair: a pair of *definition* and *use* for some variable, such that at least one DU path exists from the definition to the use

  x = ... is a *definition* of x

  = ... x ... is a *use* of x

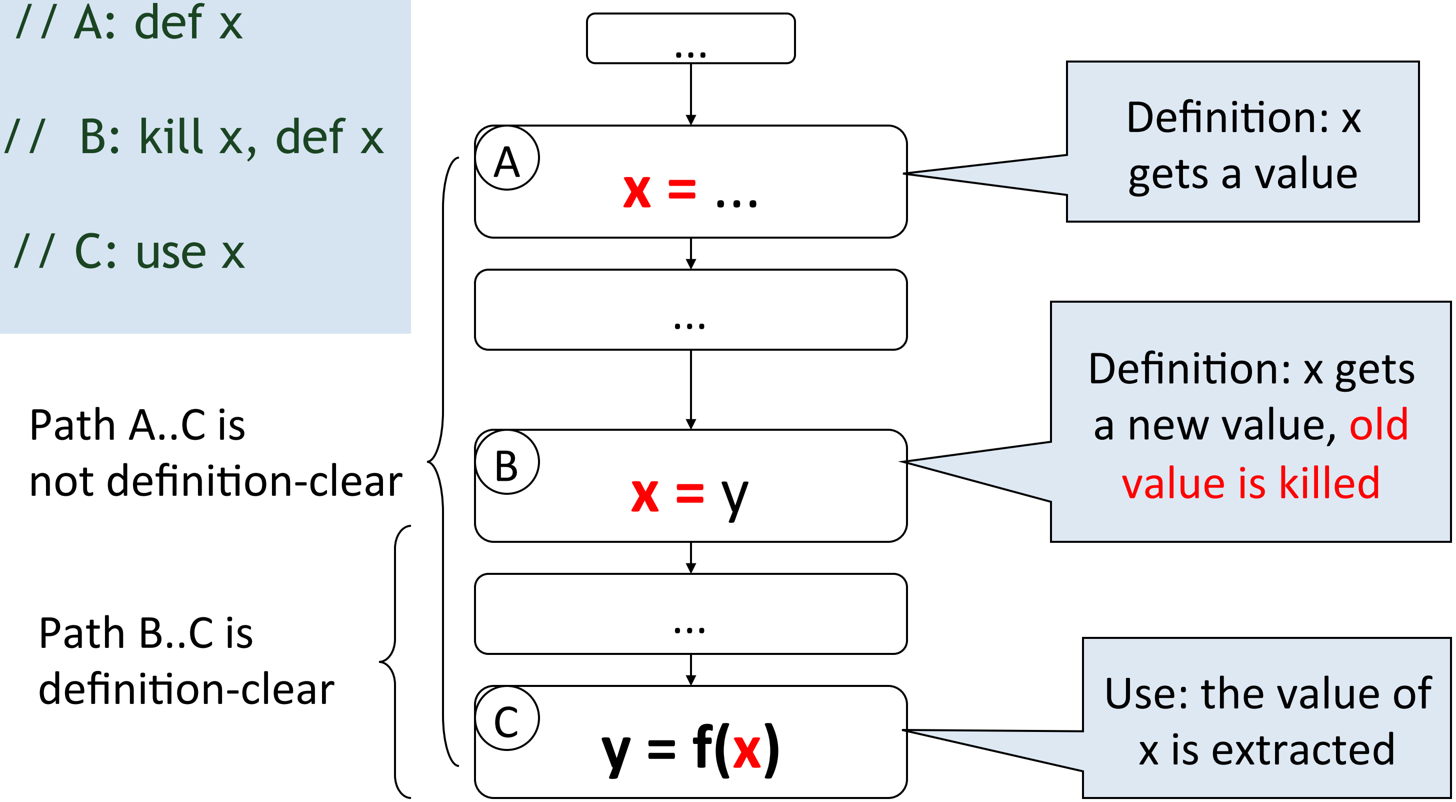
- **Definition-clear path** for a variable X
  - A path containing only **one** definition for X, and
  - X must be used **at least once** in that path
  - **Definition clear**: Value is not replaced on path

- DU path: a **definition-clear** path on the CFG starting from a definition of a variable to a use of the same variable
  - Note: loops could create infinite DU paths between a def and a use

6.1

Definition-Clear or Killing

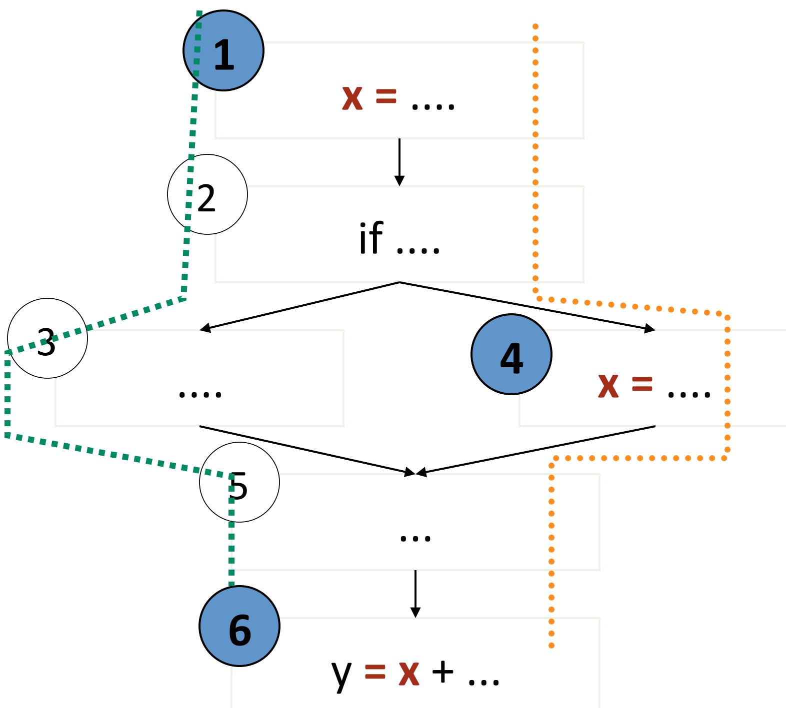
```
x = ...      // A: def x
q = ...
x = y;      //  B: kill x, def x
z = ...
y = f(x);  // C: use x
```



How many DU-pairs for x? Which?

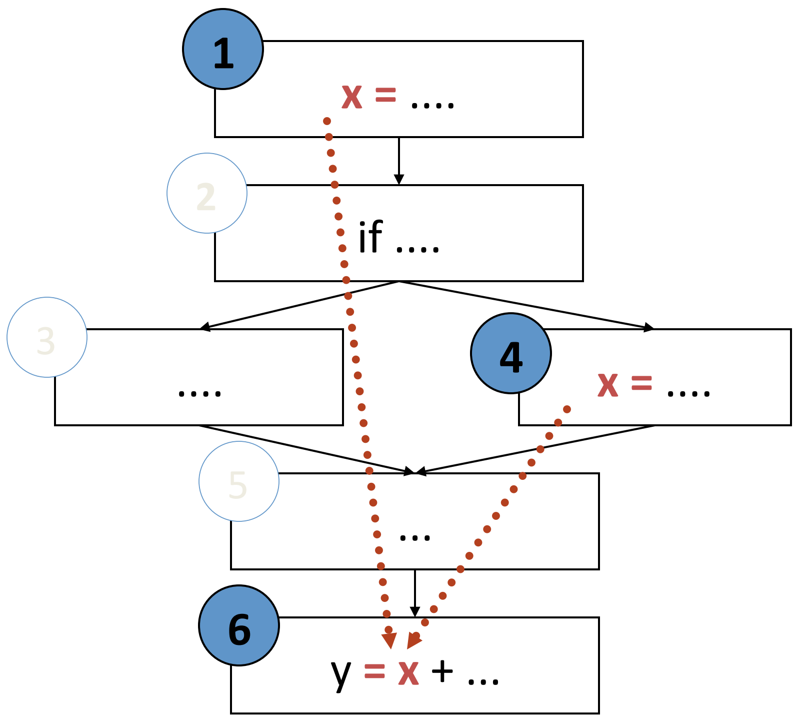
```
x = 2;
if (condition) {
  x = foo() * 3;
}
else {
  z = z + 6;
}
w = z + 12;
y = x + 8;
```

Definition-clear path



- 1,2,3,5,6 is a definition-clear path from 1 to 6
  - x is not re-assigned between 1 and 6
- 1,2,4,5,6 is not a definition-clear path from 1 to 6
  - the value of x is "killed" (reassigned) at node 4
- (1,6) is a DU pair because 1,2,3,5,6 is a definition-clear path

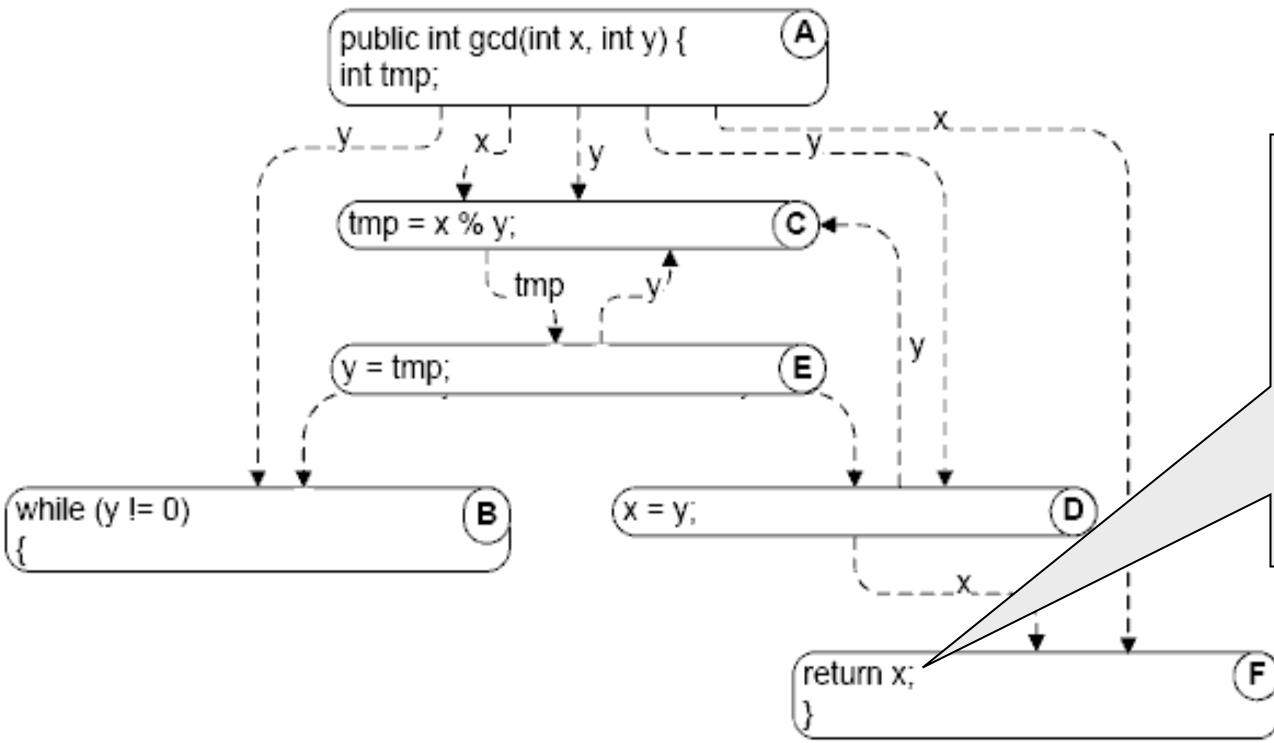
Data flow concept



- Value of x at 6 could be computed at 1 or at 4
- Bad computation at 1 or 4 could be revealed only if they are used at 6
- (1,6) and (4,6) are *def-use (DU) pairs*
  - defs at 1,4
  - use at 6

(Direct) Data Dependence Graph

- A direct data dependence graph is:
  - Nodes: as in the control flow graph (CFG)
  - Edges: def-use (du) pairs, labelled with the variable name



Dependence edges show this x value could be the unchanged parameter or could be set at line D

“where could that value returned in line F come from?”

Scope of Data Flow Analysis

- Intraprocedural
  - Within a single method or procedure
    - as described so far
- Interprocedural
  - Across several methods (and classes) or procedures

Dataflow Adequacy

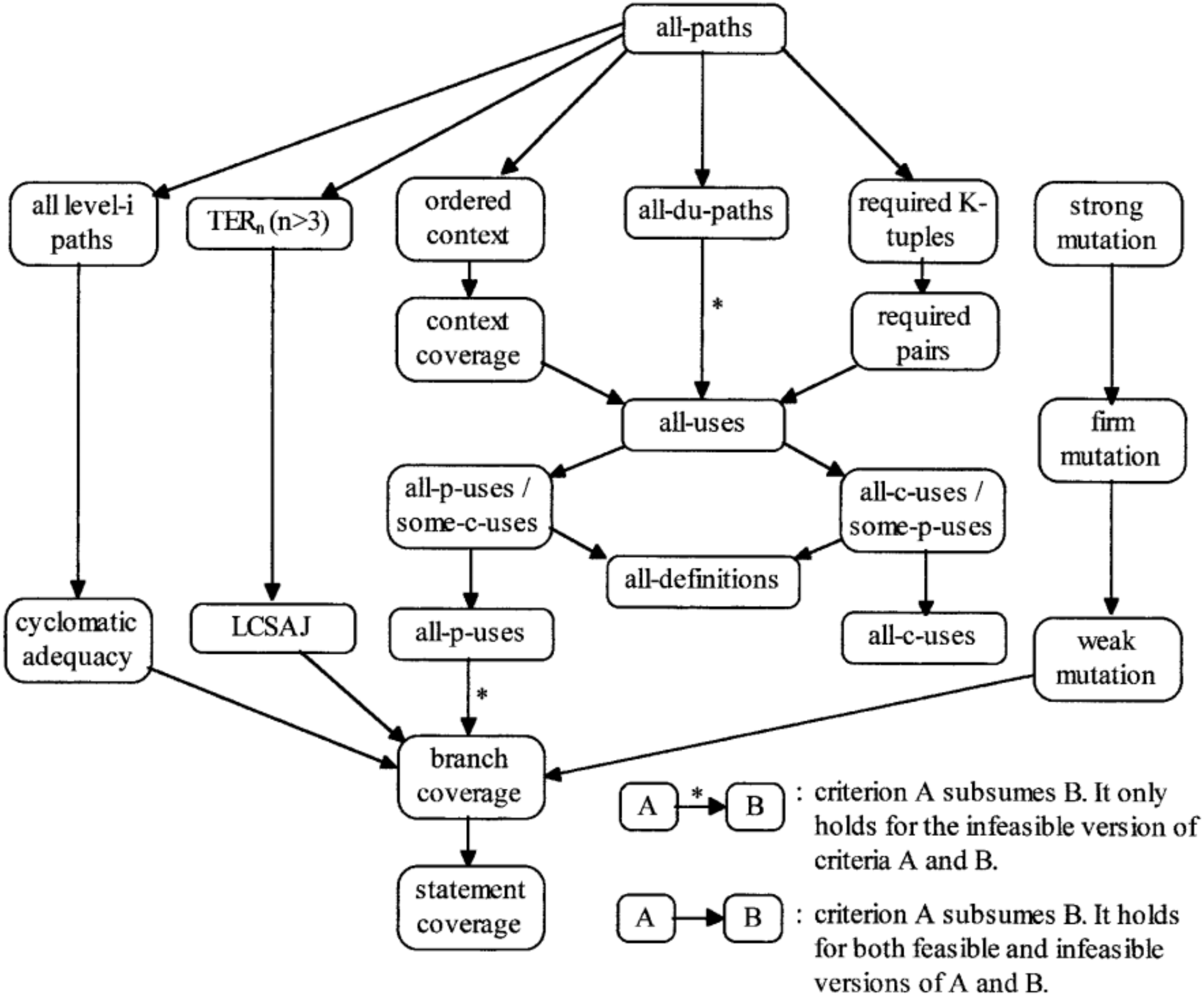
- Finding def-use pairs:
  - Along control-flow paths: **intra**-procedural
  - Along call chains: **inter**-procedural
- "All definitions coverage"
  - Test case for each definition and one possible use
- "All DU-pairs coverage"
  - Every du-pair exercised at least once
- "All DU paths coverage"
  - All control paths containing du-pairs exercised

Testing Security: *Taint Mode*

- Distinguish *safe* versus *tainted* data
- *Taint* all inputs, and follow data flow
- Web security testing?
  - Example: "Taint" analysis to prevent SQL injection attacks (unscaped user input)
- Measure coverage in terms of data flow

Do it Yourself

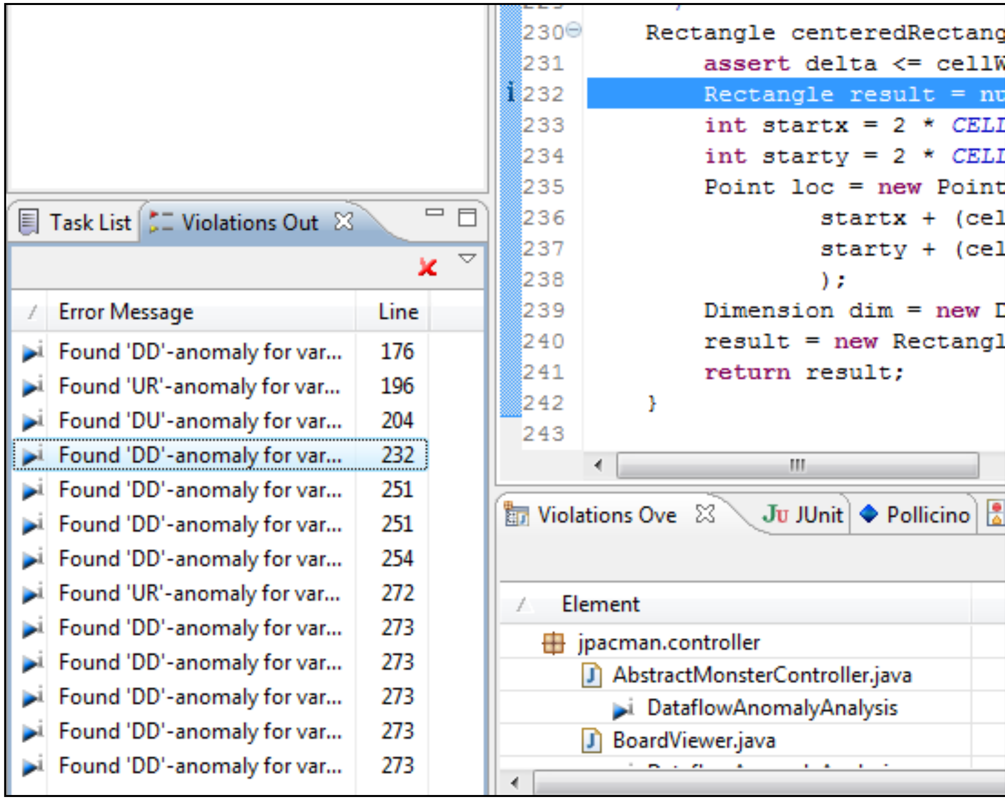
- Few dataflow based test tools available
- Conduct manual analysis when testing critical functionality
  - All definitions / all DU-pairs
- Add test cases to ensure that
  - all definitions for a given use are covered
  - all uses of a given definition are covered


all-paths
all level-i paths
TER_n (n>3)
ordered context
all-du-paths
required K-tuples
strong mutation
context coverage
*
required pairs
all-uses
firm mutation
all-p-uses / some-c-uses
all-c-uses / some-p-uses
all-definitions
cyclomatic adequacy
LCSAJ
all-p-uses
all-c-uses
weak mutation
*
branch coverage
statement coverage
A
*
B
: criterion A subsumes B. It only holds for the infeasible version of criteria A and B.
A
B
: criterion A subsumes B. It holds for both feasible and infeasible versions of A and B.

Intermezzo: Dataflow Analysis in PMD

PMD *Controversial* Ruleset:

- UR-anomaly (Type 2): Undefined-Read
- DD-anomaly (Type 3): Redefinition without use.



Excercise

$N = \{1, 2, 3, 4, 5, 6\}$

$N0 = \{1\}$

$Nf = \{6\}$

$E = \{(1,2), (2,3), (3,4), (3,5), (4,5), (5,2), (2,6)\}$

$def(1) = def(4) = use(3) = use(5) = use(6) = \{x\}$

- (a) Draw the graph.
- (b) List all of the du-paths with respect to x.
- (c) List a minimal test set that satisfies all defs coverage with respect to x.
- (d) List a test set that satisfies all du-paths coverage with respect

- (a) Draw the graph.
- (b) List all of the du-paths with respect to x.
  - 126, 123, 1235, 45, 4526, 4523
- (c) List a minimal test set that satisfies all defs coverage with respect to x.
  - t1: [1, 2, 3, 4, 5, 2, 6]

- (d) List a test set that satisfies all du-paths coverage with respect to x.
  - {t1, t2:[1, 2, 3, 5, 2, 6], t3:[1,2,6], t4:[1,2,3,4,5,2,3,5,2,6]}

  Du-paths **covered**:
  - t1 covers: 45, t2 covers: 123, 1235, t3 covers: 126, t4 covers: 4526, 4523