Lecture 10: Structural Testing - Paths and Data Flow

Gregory Gay

DIT636/DAT560 - February 15, 2023

Test Adequacy Criteria

Compromise between the impossible and the inadequate

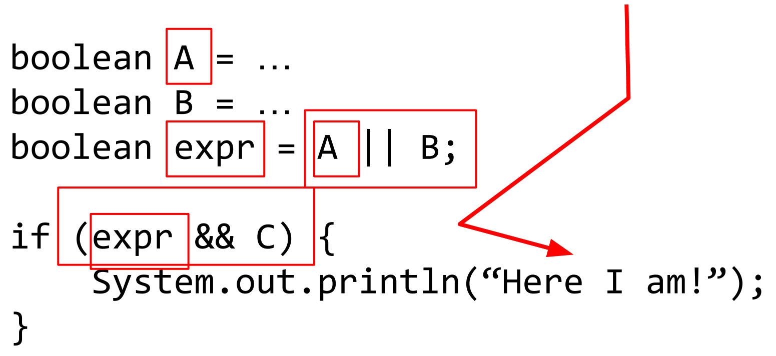
- Can we measure "good testing"?
- **Test adequacy criteria** "score" tests by measuring completion of **test obligations**.
  - Checklists of properties that must be met by test cases.

Structural Coverage Criteria

- Criteria based on exercising:
  - Statements (nodes of CFG)
  - Branches (edges of CFG)
  - Decisions and Conditions
  - Paths
  - … and many more
- Measurements used as adequacy criteria

Elements Vs. Paths

- Statement, Branch, Condition Coverage all focus on one element at a time.
- A test executes a path, not a single element.
- Each element on that path is dependent on the others.



```
boolean A = …
boolean B = …
boolean expr = A || B;

if (expr && C) {
    System.out.println("Here I am!");
}
```

Elements Vs. Paths

- There are different control paths through a program…
- … And different ways that data passed along paths can influence execution.
- Important to examine not just elements, but paths.

```
boolean A = …
boolean B = …
boolean expr = A || B;

if (expr && C) {
    System.out.println(“Here I am!”);
}
```

Fault in definition

Corrupts definition of expr if B = False

expr can corrupt outcome if C = True

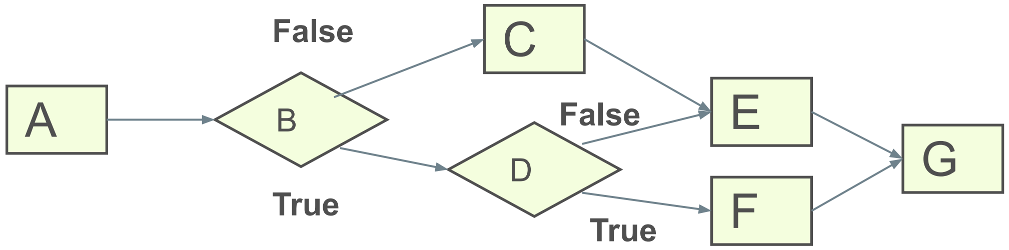
Today's Goals

- Introduce Path Coverage
- Data Flow Coverage Criteria
  - Focus on how information spreads through a program.
  - Based on Definition-Use Pairs
    - (Where is X defined? Where is each definition of X used?)

Path Coverage

Path Coverage

- Path coverage requires that all paths through the CFG are covered.

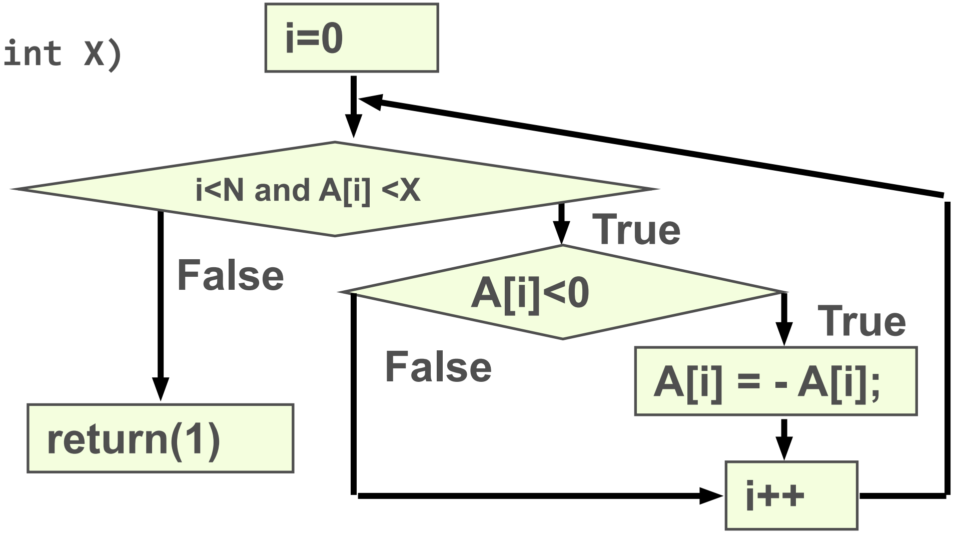


Paths:
A, B, C, E, G
A, B, D, E, G
A, B, D, F, G

- $$\text{Coverage} = \frac{\text{Number of Paths Covered}}{\text{Number of Total Paths}}$$

Path Coverage

```
public int flipSome(int[] A, int N, int X)
{
    int i=0;
    while (i<N and A[i] <X)
    {
        if (A[i]<0)
            A[i] = - A[i];
        i++;
    }
    return A;
}
```



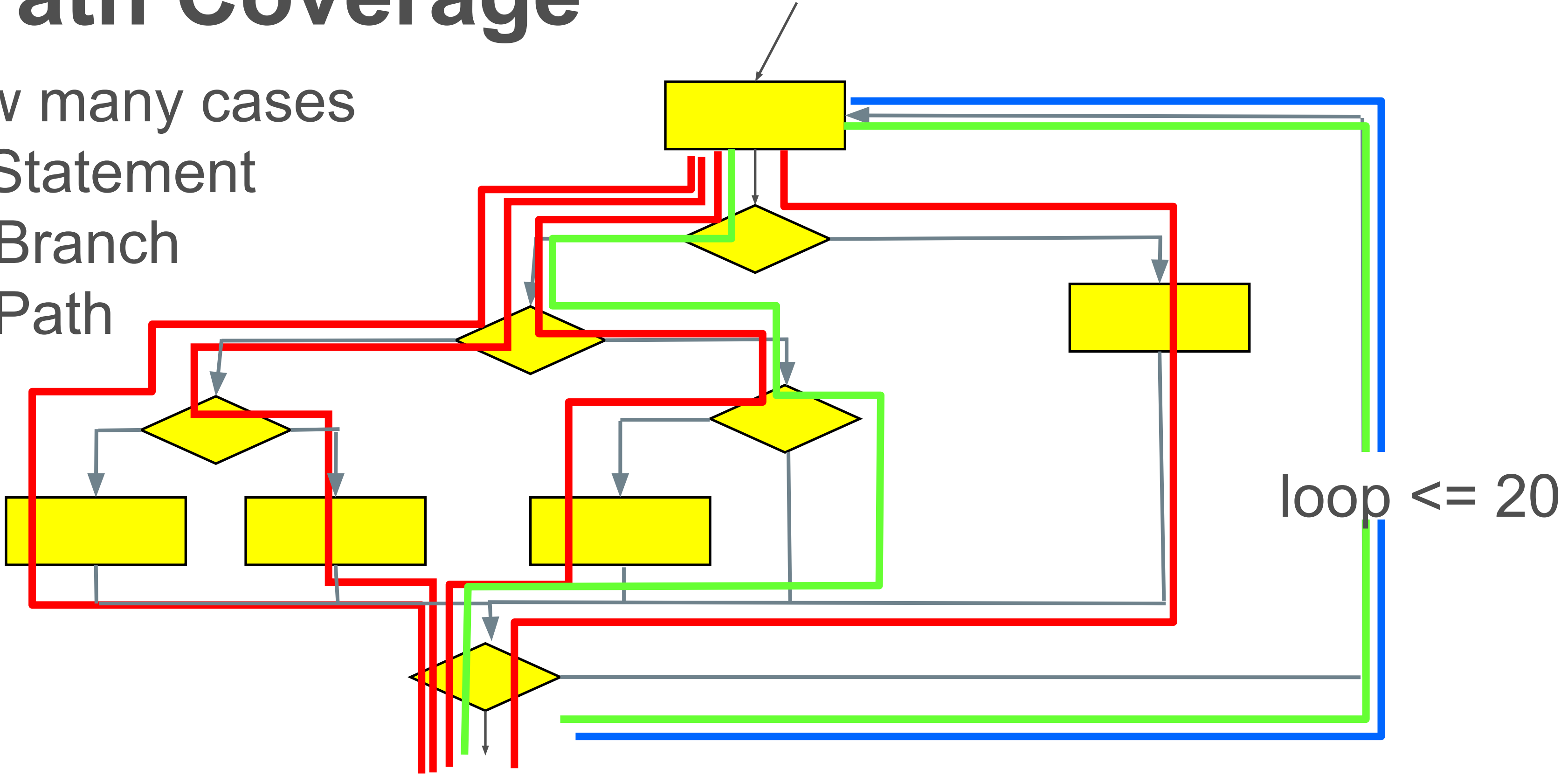
Path coverage is a powerful coverage metric, but is often impractical.

- How many paths does this have?
- Each loop cycle is a separate path!

Path Coverage

How many cases for Statement

- Branch
- Path



Path coverage with (loop <= 20) requires: **3,656,158,440,062,976** test cases

If you run 1000 tests per second, this will take **116,000 years**.

However, there are ways to get some of the benefits of path coverage without the cost...

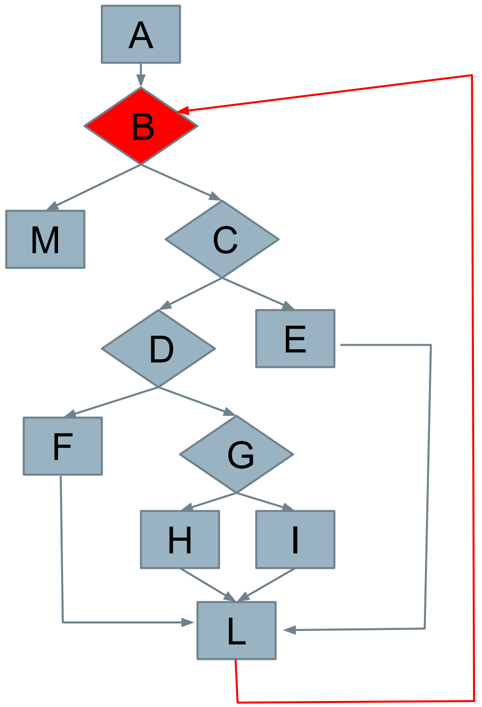
Path Coverage

- Theoretically, a very strong coverage metric.
  - Many faults emerge through sequences of interactions.
- But… Generally impossible to achieve.
  - Loops result in an infinite number of path variations.
  - Even bounding number of loop executions leaves an infeasible number of tests.

Boundary Interior Coverage

- Groups paths that differ only in the subpath they follow when repeating the body of a loop.
  - Executing loop 20 times is different than executing it twice, but same *subpaths* repeat over and over.
  - Unroll loop in CFG into distinct subpaths, and cover those instead of worrying about loop cycles.

Boundary Interior Coverage



A -> B -> M
A -> B -> C -> E -> L -> B
A -> B -> C -> D -> F -> L -> B
A -> B -> C -> D -> G -> H -> L -> B
A -> B -> C -> D -> G -> I -> L -> B

Boundary Interior Coverage

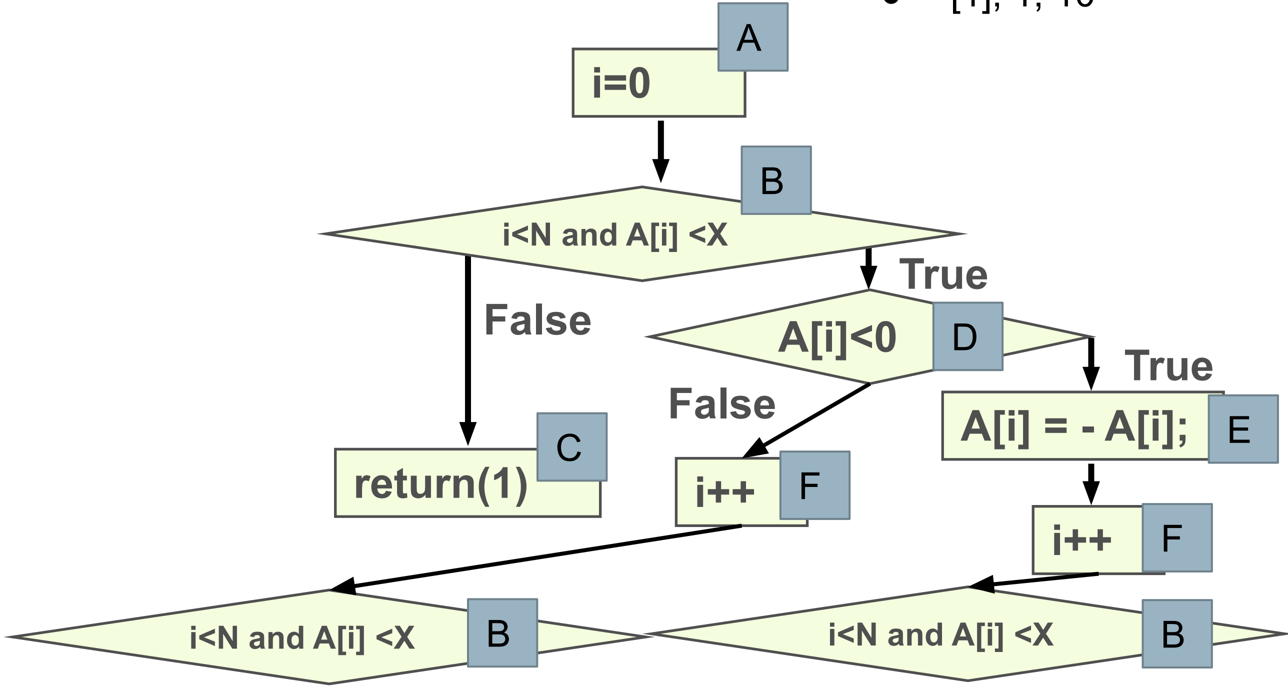
```
public int flipSome(int[] A, int N, int X)
{
      int i=0;
      while (i<N and A[i] <X)
      {
            if (A[i]<0)
                  A[i] = - A[i];
            i++;
      }
      return A;
}
```

Paths:

- A, B, C
- A, B, D, F, B
- A, B, D, E, F, B

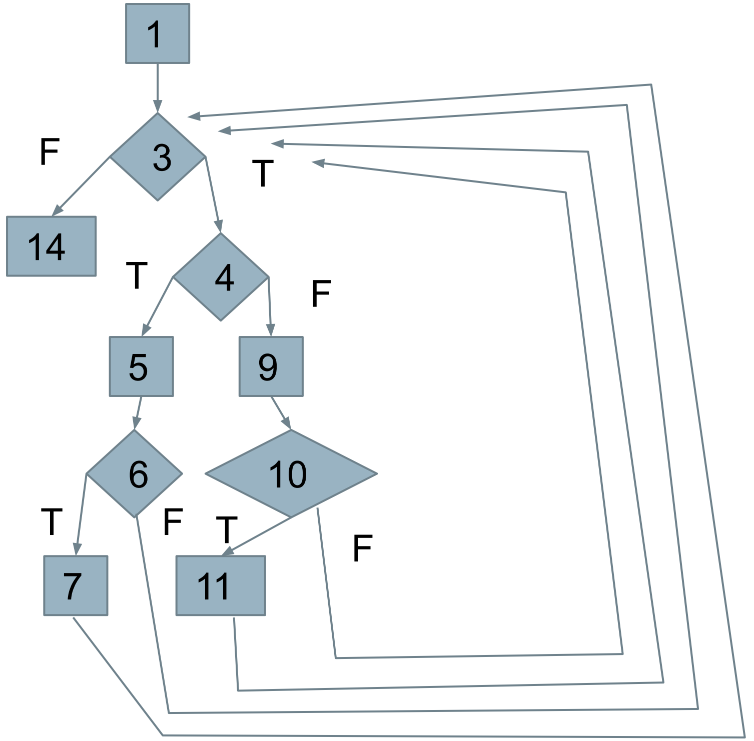
Test Input

- [ ], 0, 10
- [-1], 1, 10
- [1], 1, 10



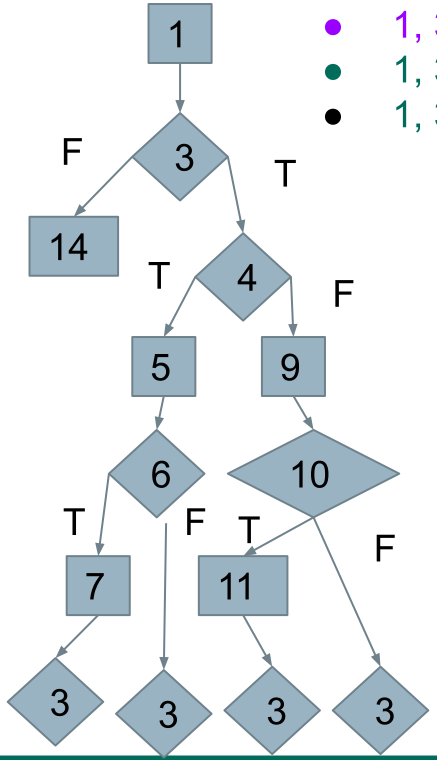
Boundary Interior Example

```
1. public int doSomething(int x, int y)
2. {
3.    while(y > 0) {
4.          if(x > 0) {
5.                y = y - x;
6.                if (y > 0)
7.                      System.out.println(“Y: “ + y);
8.          }else {
9.                x = x + 1;
10.               if (x <= 0)
11.                     System.out.println(X: “ + x);
12.         }
13.   }
14.   return x + y;
15. }
```



Boundary Interior Example

```
1. public int doSomething(int x, int y)
2. {
3.    while(y > 0) {
4.          if(x > 0) {
5.                y = y - x;
6.                if (y > 0)
7.                      System.out.println("Y: " + y);
8.          }else {
9.                x = x + 1;
10.               if (x <= 0)
11.                     System.out.println(X: " + x);
12.         }
13.   }
14.   return x + y;
15. }
```



Paths:
- ● 1, 3-F, 14
- ● 1, 3-T, 4-T, 5, 6-T, 7, 3
- ● 1, 3-T, 4-T, 5, 6-F, 3
- ● 1, 3-T, 4-F, 9, 10-T, 11, 3
- ● 1, 3-T,4-F, 9, 10-F, 3

Test Input:
- ● 10, -1
- ● 3, 4
- ● -1, 1

Number of Paths

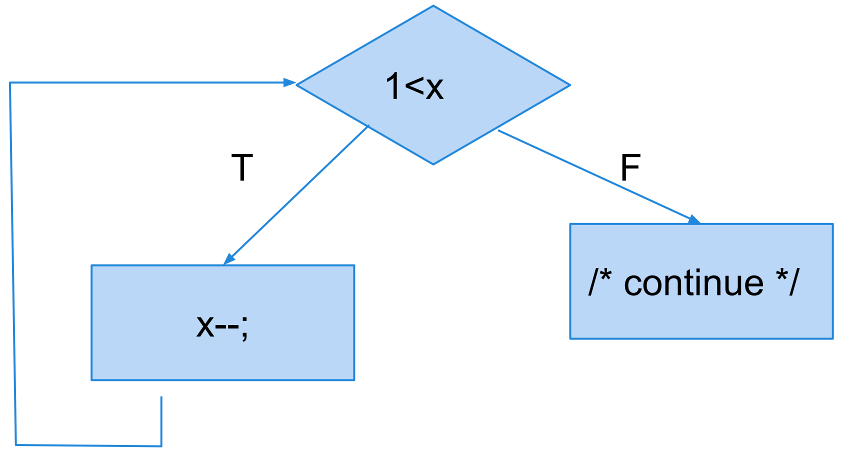
- Boundary Interior Coverage removes bounds number of loop paths.
  - However, number of paths can still be exponential.
  - N non-loop branches results in 2^N paths.
- Additional limitations may need to be imposed.

```
if (a)     S1;
if (b)     S2;
if (c)     S3;
…
if (x)     SN;
```

Data Flow

Control Flow

- Capture how execution navigates between blocks of statements.
- We care about a statement's effect **only when it affects the path**.
  - Deemphasizes information being transmitted.



Data Flow

- Program statements compute and transform data…
- Reason about data dependence
  - A variable is used here.
    - Where does its value come from?
  - Is this value ever used?
  - Is this variable properly initialized?
  - If the expression assigned to a variable is changed what else would be affected?

Data Flow

- Basis of the optimization performed by compilers.
- Used to derive test cases.
  - Have we covered the dependencies?
- Used to detect faults and other anomalies.
  - When can we cache result of a calculation instead of recalculating it?
  - Can we eliminate a variable definition?

Definition-Use Pairs

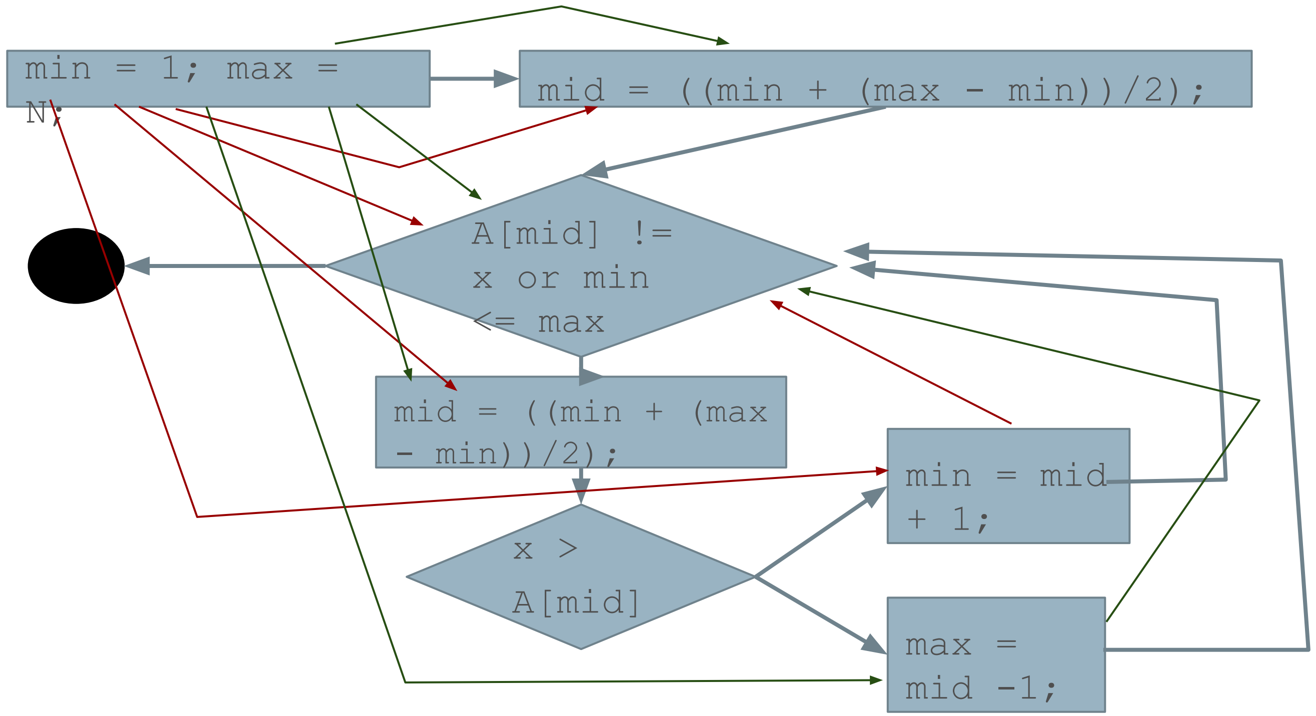
- Data is defined.
  - … and data is used.
- Pairs of definitions and uses capture flow of information through the program.
  - Definitions occur when variables are declared, initialized, assigned values, or received as parameters.
  - Uses occur in expressions, conditional statements, parameter passing, return statements.

Definitions and Uses

```
 1.  min = 1;
 2.  max = N;
 3.  mid = ((min + (max - min))/2);
 4.  while (A[mid] != x or min <= max){
 5.      mid = ((min + (max - min))/2);
 6.      if (x > A[mid]){
 7.          min = mid + 1
 8.      } else {
 9.          max = mid - 1;
10.      }
11.  }
```

1. **def** - min
2. **def** - max, **use** - N
3. **def** - mid, **use** - min, max
4. **use** - A[mid], mid, x, min, max
5. **def** - mid, **use** - min, max
6. **use -** x, A[mid], mid
7. **def -** min, **use** - mid
8. -
9. **def -** max, **use -** mid

Definitions and Uses



min = 1; max = N;
mid = ((min + (max - min))/2);
A[mid] != x or min <= max
mid = ((min + (max - min))/2);
x > A[mid]
min = mid + 1;
max = mid -1;

1. **def** - min
2. **def** - max, **use** - N
3. **def** - mid, **use** - min, max
4. **use** - A[mid], mid, x, min, max
5. **def** - mid, **use** - min, max
6. **use -** x, A[mid], mid
7. **def -** min, **use** - mid
8. -
9. **def -** max, **use -** mid

Definition-Use (DU) Pairs

- We can say there is a **DU pair** when:
  - There is a **def** (definition) of variable x at location A.
  - Variable x is **used** at location B.
  - A control-flow **path** exists from A to B.
  - and the path is **definition-clear** for x from A to B.
- If variable is redefined, original def is **killed** and pair is now between new definition and its use in B.

Example - Definition-Use Pairs

```
 1.  min = 1;
 2.  max = N;
 3.  mid = ((min + (max - min))/2);
 4.  while (A[mid] != x or min <= max){
 5.      mid = ((min + (max - min))/2);
 6.      if (x > A[mid]){
 7.          min = mid + 1
 8.      } else {
 9.          max = mid - 1;
10.      }
11.  }
```

DU Pairs
min: (1, 3), (1, 4), (1, 5), (7, 4), (7, 5)
max: (2, 3), (2, 4), (2, 5), (9, 4), (9, 5)
N: (0, 2)
mid: (3, 4), (5, 6), (5, 7), (5, 9), (5, 4)
x: (0, 4), (0, 6)
A: (0, 4), (0, 6)

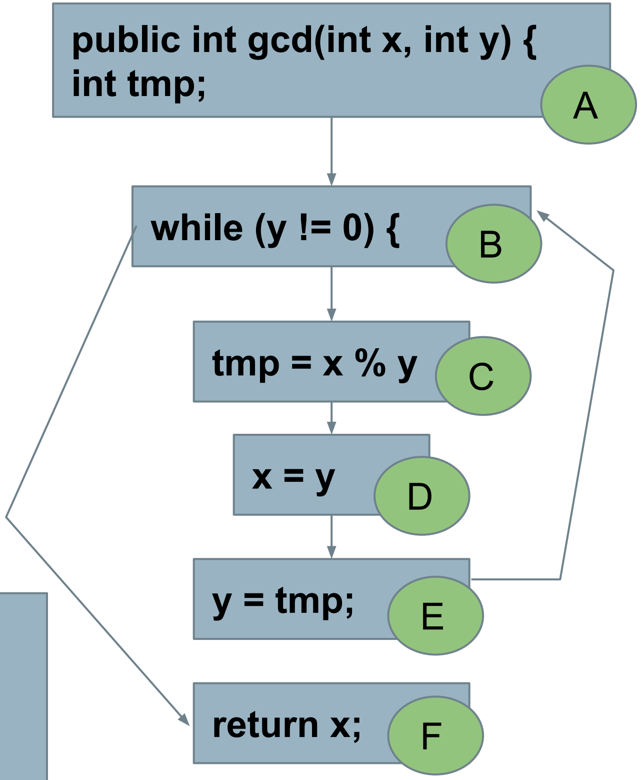
Example - GCD

```
1.  public int gcd(int x, int y){
2.      int tmp;
3.      while(y!=0){
4.          tmp = x % y;
5.          x = y;
6.          y = tmp;
7.      }
8.      return x;
9.  }
```

1. def: x, y
2. def: tmp
3. use: y
4. use: x, y
   def: tmp
5. use: y
   def: x
6. use: tmp
   def: y
7. -
8. use: x

Example - GCD

```
1.  public int gcd(int x, int y){
2.      int tmp;
3.      while(y!=0){
4.          tmp = x % y;
5.          x = y;
6.          y = tmp;
7.      }
8.      return x;
9.  }
```



Def-Use Pairs

x: (1, 4), (5, 4), (5, 8), (1, 8)

y: (1, 3), (1, 4), (1, 5), (6, 3), (6, 4), (6, 5)

tmp: (4, 6)

Example - collapseNewlines

```
7. public static String collapseNewlines(String argStr)
8. {
9.    char last = argStr.charAt(0);
10.   StringBuffer argBuf = new StringBuffer();
11.
12.   for(int cldx = 0; cldx < argStr.length(); cldx++)
13.   {
14.        char ch = argStr.charAt(cldx);
15.        if(ch != ‘\n’ || last != ‘\n’)
16.        {
17.             argBuf.append(ch);
18.             last = ch;
19.        }
20.   }
21.
22.   return argBuf.toString();
23. }
```

Variable	**D-U Pairs**
argStr	(7, 9), (7,12), (7, 14)
last	(9, 15), (18, 15)
argBuf	(10,22), (17, 22)
cldx	(12, 12), (12, 14)
ch	(14, 15), (14, 17), (14, 18)

Let’s Take a Break

Dealing With Arrays and Pointers

- Arrays and pointers (including object references and arguments) introduce issues.
  - It is not possible to determine whether two access refer to the same storage location.

    -
      ```
      a[x] = 13;
      k = a[y];
      ```
      - Are these a def-use pair?

    -
      ```
      a[2] = 42;
      i = b[2];
      ```
      - Are these a def-use pair?

Aliasing

- Two names refer to the same memory location.

  -
    ```
    int[] a = new int[3];
    int[] b = a;
    a[2] = 42;
    i = b[2];
    ```
- Worse in C:
  ```
  p = &b;
  *(p + i) = k;
  ```

Uncertainty

- Aliasing introduces uncertainty.
  - Instead of definition or use of one variable, may have a potential def or use of a set of variables.
- Safest: treat **any** use of a potential alias of V as a use of V.
  - Creates more def-use pairs (some may not be real), but avoids missed pairs.

Dealing With Uncertainty

- Treat all potential aliases as definitions and uses:

```
a[1] = 13;
k = a[2];
```

Def of **a[1]**, use of **a[2]**.

```
a[x] = 13;
k = a[y];
```

Def and use of **array a**.

- Can be very imprecise.
  - They are only the same if x and y are the same.

Dealing With Uncertainty

- Option 2: Treat uncertainty about aliases like uncertainty about control flow.

```
a[x] = 13;
k = a[y];
```

$\longrightarrow$

```
a[x] = 13;
if(x == y)     k = a[x];
else           k = a[y];
```

- Rewrite code to make references explicit.
- In transformed code, all array references are distinct.

Situational Def-Use Pairs

- `++counter, counter++, counter+=1`
  `counter = counter + 1`
  - Use of `counter` then a new definition.
- `char *ptr = *otherPtr`
  - Definition of string `*ptr`
  - Use of memory index `ptr`, string `*otherPtr`, and memory index `otherPtr`.
  - `ptr++`
    - Use of memory index ptr, definition of both memory index and string *ptr (change to index moves pointer to a new location).

Data Flow Coverage Criteria

Overcoming Limitations of Path Coverage

- We can potentially expose many faults by targeting particular paths of execution.
- What are the important paths to cover?
    - Some methods impose heuristic limitations.
    - Use data flow to select paths based on how one element can affect the computation of another.

Choosing the Paths

- Computing the wrong value leads to a failure **only when that value is used**.
  - Pair definitions with usages.
  - Ensure that definitions are actually used by covering paths from definitions to uses.
  - All DU Pair Coverage, All DU Paths Coverage, All Definitions Coverage
    - Varying power and cost.

All DU Pair Coverage

- Requires each DU pair be exercised in at least one program execution.
  - Cover **any path** between a definition and its use.
- Coverage = $\dfrac{\text{number exercised DU pairs}}{\text{number of DU pairs}}$
- Can easily achieve structural coverage without covering all DU pairs.

All DU Pairs Coverage Example

```
1. public int doSomething(int x, int y)
2. {
3.   while(y > 0) {
4.         if(x > 0) {
5.               y = y - x;
6.               if (y > 0)
7.                   System.out.println(“Y: “ + y);
8.         }else {
9.               x = x + 1;
10.              if (x <= 0)
11.                  System.out.println(X: “ + x);
12.        }
13.  }
14.  return x + y;
15. }
```

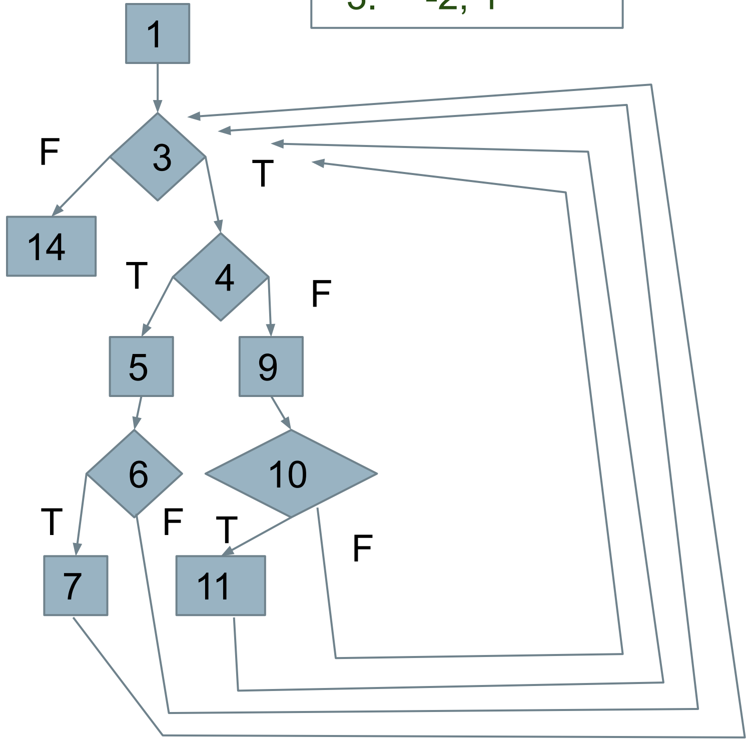
X:
(1, 4), (1, 5), (1, 9), (1, 14)
(9, 10), (9, 11), (9, 4), (9, 5), (9, 9), (9, 14)

Y:
(1, 3), (1, 5), (1, 14)
(5, 6), (5, 7), (5, 3), (5, 5), (5, 14)

X: (1, 4), (1, 5), (1, 9), (1, 14), (9, 10), (9, 11), (9, 5), (9, 9), (9, 14)
Y: (1, 3), (1, 5), (1, 14), (5, 6), (5, 7), (5, 3), (5, 5), (5, 14)

Test Input:
1. -1, 1
2. 3, 7
3. -2, 1

```
1. public int doSomething(int x, int y)
2. {
3.   while(y > 0) {
4.         if(x > 0) {
5.               y = y - x;
6.               if (y > 0)
7.                   System.out.println(“Y: “ + y);
8.         }else {
9.               x = x + 1;
10.              if (x <= 0)
11.                  System.out.println(X: “ + x);
12.        }
13.  }
14.  return x + y;
15. }
```



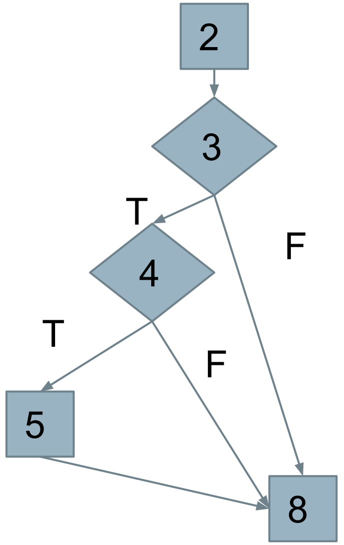
All DU Paths Coverage

- A use may be reachable along several paths from the definition.
- Cover all simple (non-looping) paths at least once.
  - Can reveal faults where a path is exercised that should use a certain definition but doesn't.

$$\text{Coverage} = \frac{\text{number of exercised DU paths}}{\text{number of DU paths}}$$

All DU Paths Example

```
1.  ...
2.  int x = 1;
3.  if (y > 7) {
4.      if (z > 5) {
5.          z = x + 5;
6.      }
7.  }
8.  y = x + 7;
9.  ...
```



DU Pair (2, 8) for X can be reached along multiple paths.

- 2, 3T, 4T, 5, 8
- 2, 3T, 4F, 8
- 2, 3F, 8

Test Input:

- y = 10, z = 6
- y = 10, z = 3
- y = 2, z = (anything)

Path Explosion Problem

- Even without looping paths, number of DU paths can be exponential.
  - Code between definition and use can be irrelevant to that variable, but contains many paths.

```
public void countBits(char ch){
     int count = 0;
     if (ch & 1)   ++count;
     if (ch & 2)   ++count;
     if (ch & 4)   ++count;
     if (ch & 8)   ++count;
     if (ch & 16)  ++count;
     if (ch & 32)  ++count;
     if (ch & 64)  ++count;
     if (ch & 128) ++count;
     System.out.println(ch + “ has ” +
count + “bits set to 1”);
}
```

All Definitions Coverage

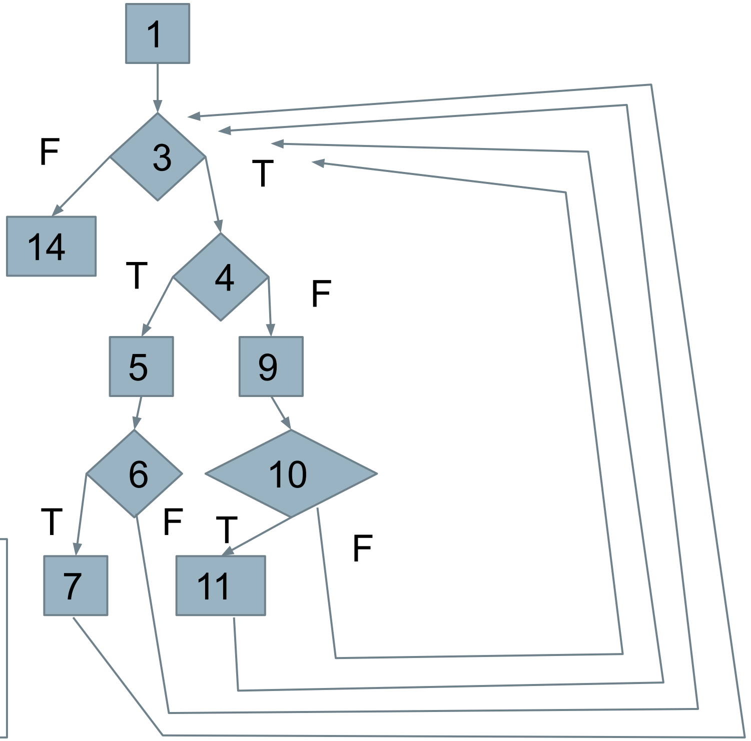
- All DU Pairs/All DU Paths may be too expensive in some situations.
- Pair each definition with at least one use.
  - Skips many DU pairs, but ensures each definition tried.

$$\text{Coverage} = \frac{\text{number of covered definitions}}{\text{number of definitions}}$$

X: (1, 4), (1, 5), (1, 9), (1, 14), (9, 10), (9, 11), (9, 5), (9, 9), (9, 14)
Y: (1, 3), (1, 5), (1, 14), (5, 6), (5, 7), (5, 3), (5, 5), (5, 14)

X: Definitions on lines 1, 9
Y: Definitions on lines 1, 5

```
1.  public int doSomething(int x, int y)
2.  {
3.    while(y > 0) {
4.         if(x > 0) {
5.              y = y - x;
6.              if (y > 0)
7.                  System.out.println(“Y: “ + y);
8.         }else {
9.              x = x + 1;
10.             if (x <= 0)
11.                 System.out.println(X: “ + x);
12.        }
13.   }
14.   return x + y;
15. }
```



- Any input covers (1, -) pairs.
- Reaching lines 5, 9 covers (5,14) and (9,14) pairs.

Infeasibility Problem

- Metrics may ask for impossible test cases.
- Path-based metrics may require infeasible combinations of feasible elements.
  - Alias analysis may add additional infeasible paths.
- All Definitions, All DU-Pairs Coverage reasonable.
  - All DU-Paths is much harder!

Activity - DU Pair Coverage

- Identify all DU pair
- Write **your own** test input to achieve All DU Pair Coverage.
  - e.g., Input (1, 1) For x, covers pairs: (1,4), …

```
1. int doSomething(int x, int y)
2. {
3.      while(y > 0) {
4.           if(x > 0) {
5.                y = y - x;
6.           }else {
7.                x = x + 1;
8.           }
9.      }
10.     return x + y;
11. }
```

Activity - DU Pairs

```
1. int doSomething(int x, int y)
2. {
3.      while(y > 0) {
4.          if(x > 0) {
5.              y = y - x;
6.          }else {
7.              x = x + 1;
8.          }
9.      }
10.     return x + y;
11. }
```

Variable	Defs	Uses
x	1, 7	4, 5, 7, 10
y	1, 5	3, 5, 10

Variable	D-U Pairs
x	(1, 4), (1, 5), (1, 7), (1, 10), (7, 4), (7, 5), (7, 7), (7, 10)
y	(1, 3), (1, 5), (1, 10), (5, 3), (5, 5), (5, 10)

Activity - DU Pairs

```
1.  int doSomething(int x, int y)
2.  {
3.          while(y > 0) {
4.                  if(x > 0) {
5.                          y = y - x;
6.                  }else {
7.                          x = x + 1;
8.                  }
9.          }
10.         return x + y;
11. }
```

Variable	Defs	Uses
x	1, 7	4, 5, 7, 10
y	1, 5	3, 5, 10

Variable	D-U Pairs
x	~~(1, 4), (1, 5), (1, 7), (1, 10), (7, 4), (7, 5), (7, 7), (7, 10)~~
y	~~(1, 3), (1, 5), (1, 10), (5, 3), (5, 5), (5, 10)~~

Test Input 1: (x = 1, y = 2)
Covers lines 1, 3, 4, 5, 3, 4, 5, 3, 10
Test Input 2: (x = -1, y = 1)
Covers lines 1, 3, 4, 6, 7, 3, 4, 6, 7, 3, 4, 5, 3, 10
Test Input 3: (x = 1, y = 0)
Covers lines 1, 3, 8

We Have Learned

- Control-flow and data-flow capture important paths in program execution.
- Analysis of how variables are defined and then used and the dependencies between definitions and usages can help us reveal important faults.
- Many forms of analysis can be performed using data flow information.

We Have Learned

- If there is a fault in a computation, we can observe it by looking at where the computation is used.
- By identifying DU pairs and paths, we can create tests that trigger faults along those paths.
  - All DU Pairs coverage
  - All DU Paths coverage
  - All Definitions coverage

Next Time

- Exercise Session - Structural Testing
  - Using Meeting Planner code.
- Next Tuesday - Fault-Based Testing
  - Pezze & Young - Ch 16
- Assignment 2
  - Due February 26! We have covered everything on it.

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY