SOFTENG 254:
Quality Assurance

Lecture 4a: Data-flow Testing

Paramvir Singh
School of Computer Science

Potential Assessment Question

Draw the *control flow graph* for the code shown in the figure. Annotate the code as appropriate. (code not provided here)

Agenda

- PAQ
- Admin
  - Assignment 1?
- Coverage criteria from CFGs
- Data-flow testing — choosing tests based on how values are propagated through code

What Am I Testing?

- AKA “If this test fails, where is the fault?”

```
@Test
public void testRoundingAtHalf() {
    Stats stats = new Stats(new int[]{1, 0, 0, 1});
    assertEquals(1, stats.getMean());
}
```

Test Requirements based on paths

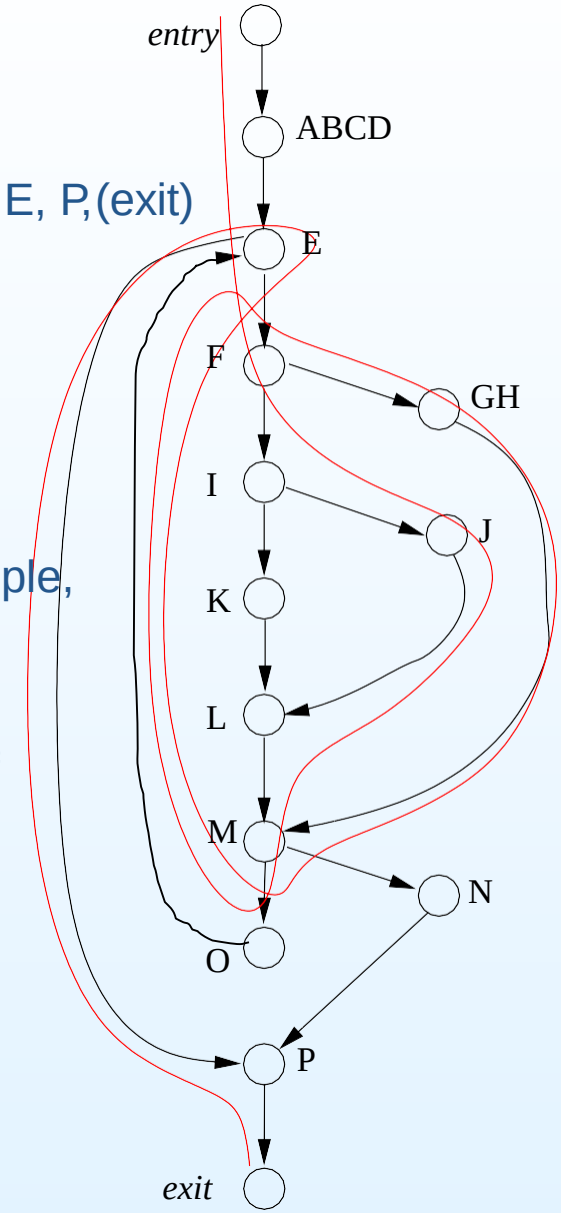
- “all paths” coverage criterion often results in too many test requirements
- ”statement coverage” and “branch coverage” criteria often do not produce enough test requirements, but are subsets of “all paths”
- Are there other subsets of “all paths” that can do better?
- General idea: annotate CFGs with characteristics of interest, determine path subsets based on characteristics

Example Criteria

- A **path** in a CFG is a sequence of vertices in the CFG $(v_0, v_1, v_2, \ldots, v_k)$ such that $\forall 1 \leq i \leq k, (v_{i-1}, v_i)$ is an edge in the CFG
  - Recall: A *test path* is a path where v_0 is the entry vertex and v_k is the exit vertex
- A **subpath** $q = (w_0, w_1, \ldots, w_m)$ of a path $p = (v_0, v_1, v_2, \ldots, v_k)$ when $\exists l, \forall 0 \leq i \leq m, w_i = v_{l+i}$
- A path is **simple** if either every vertex only appears once in the path, or the only vertex that appears twice appears as the first and last vertex in the path.
  - A subpath of a simple path must also be simple (so there can be a *lot* of simple paths in a CFG)
- A **prime** path is a maximal length simple path, that is, path p is prime if it is simple and $\forall$paths q, such that p is a subpath of q, q is not simple.

Examples

- Test Path **TP**:
  (entry), A, E, F, I, J, L, M, O, E, F, G, M, O, E, P, (exit)
- Path **P1**:
  E, F, I, J, L, M, O, E, F
- P1 is a *subpath* of TP, P1 is **not** *simple*
- Path **P2**:
  F, I, J, L, M
- P2 is a subpath of P1 (and TP), P2 is simple,
  P2 is **not** *prime*
- Path **P3**: F, I, J, L, M, O, E, F
- P3 is a subpath of P1, is not a subpath of
  P2, is simple, and is prime



Example Coverage Criteria

- Criterion: Paths of length 2
- Criterion: Simple paths
- Criterion: Prime paths

Data Flow

- computation consists of manipulating values, moving them among different memory locations
- (almost) every failure can be traced to a wrong value being in the wrong place at the wrong time

⇒ look at where the values come from and where they go — “data flow”

- choose a subset of “all paths” based on data flow
- annotate vertices in CFGs with indications as to what variables are “defined” and what variables are “used” at those vertices

Data Flow Actions

- control flow graphs, except vertices are labelled with what happens to data objects at the corresponding statements
- 3 major and 2 minor things happen to data objects

  Defined (d) data value comes into existence, assigned to variable (or similar)

  Killed or undefined (k) data value no longer available

  Usage (u)/Computation(c) used for computation

  Usage (u)/Predicate(p) used in a predicate
- Look at paths on which things happen to the same data object

Data Flow Action Specifics

def: a location where a value is stored into memory

- variable appears on the left side of assignment operator
- variable is a formal parameter of a method (occurs in entry vertex)
- variable is an input to a program (some languages)
- variable is an actual parameter in a method invocation and the method changes its value (some languages)

use: a location where a variable's value is accessed

- the variable appears on the right hand side of an assignment
- the variable appears in a predicate (in a conditional test)
- the variable is an actual parameter in a method invocation
- the variable is in a return statement for a method
- the variable is an output of a program (some languages)

Data Flow Examples


```
int x=3; // Definition of x
x=4; // x defined again
{
  int y=3; // Def of y
}// y is killed
int y=x+1; // Use of x in computation, Def of y
if (x >3) { // Use of x in predicate
}
```

Data Flow Examples


```
{
  MyClass mc = new MyClass(); // mc defined
  YourClass yc;   // yc not defined
  yc = new YourClass(mc); //yc defined, mc used (comp)
  mc.doSomething(42); //mc used (comp)
  if (yc.isSomething()) {//yc used (pred,comp)

  }
}// mc, yc killed
```

Data Flow Examples

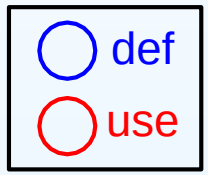

```
{
  MyClass mc = new MyClass(); // mc defined
  YourClass yc;   // yc not defined
  yc = new YourClass(mc); //yc defined, mc used (comp)
  mc.doSomething(42); //mc used (comp)
  if (yc.isSomething()) {//yc used (pred,comp)

  }
}// mc, yc killed
```

Example: Variation B

```
   public static int leadingSpacesCount(String text, int tabstop) {
A    int index =0;
B    int count =0;
C    int interTab = 0;
D    char[] chars =text.toCharArray();
E    while (Character.isWhitespace(chars[index])) {
F      if (chars[index] ==’\t’) {
G        count +=tabstop - interTab;
H        interTab = 0;
       }else {
I        if (interTab ==tabstop - 1) {
J          interTab = 0;
         }else {
K          interTab++;
         }
L        count++;
       }
M      if (index ==chars.length-1){
N        break;
       }
O      index++;
     }
P    return count;
  }
```

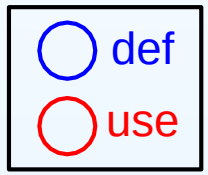
Example: Variation B

```
   public static int leadingSpacesCount(String text, int tabstop) {
A    int index =0;
B    int count =0;
C    int interTab = 0;
D    char[] chars =text.toCharArray();
E    while (Character.isWhitespace(chars[index])) {
F      if (chars[index] ==’\t’) {
G        count +=tabstop - interTab;
H        interTab =0;
       }else {
I        if (interTab ==tabstop - 1) {
J          interTab =0;
         }else {
K          interTab++;
         }
L        count++;
       }
M      if (index ==chars.length-1) {
N        break;
       }
O      index++;
     }
P    return count;
   }
```



Example: Variation B

```
    public static int leadingSpacesCount(String text, int tabstop) {
A     int index =0;
B     int count =0;
C     int interTab = 0;
D     char[] chars =text.toCharArray();
E     while (Character.isWhitespace(chars[index])) {
F       if (chars[index] ==’\t’) {
G         count +=tabstop - interTab;
H         interTab = 0;
        }else {
I         if (interTab ==tabstop - 1) {
J           interTab = 0;
          }else {
K           interTab++;
          }
L         count++;
        }
M       if (index ==chars.length-1) {
N         break;
        }
O       index++;
      }
P     return count;
    }
```



Data Flow Anomalies

- just looking at data actions on a test path can provide useful candidates for test cases
- given a test path and a variable, what data actions take place in what order?

dd (mostly) harmless, but worth checking
dk possible fault
du normal case
kd normal case
kk it's dead already (suspicious)
ku fault (not possible in some languages)
ud probably not a fault
uk normal
uu normal

Data Flow Anomalies

- "–" means that none of the actions of interest have happened to the data object
  - **–k** — variable not defined but then killed
  - **–d** — first definition in path
  - **–u** — used before defined
  - **k–** — killed is the last thing that happens
  - **d–** — defined but never used
  - **u–** — used but never killed, probably ok

Data Flow Anomalies

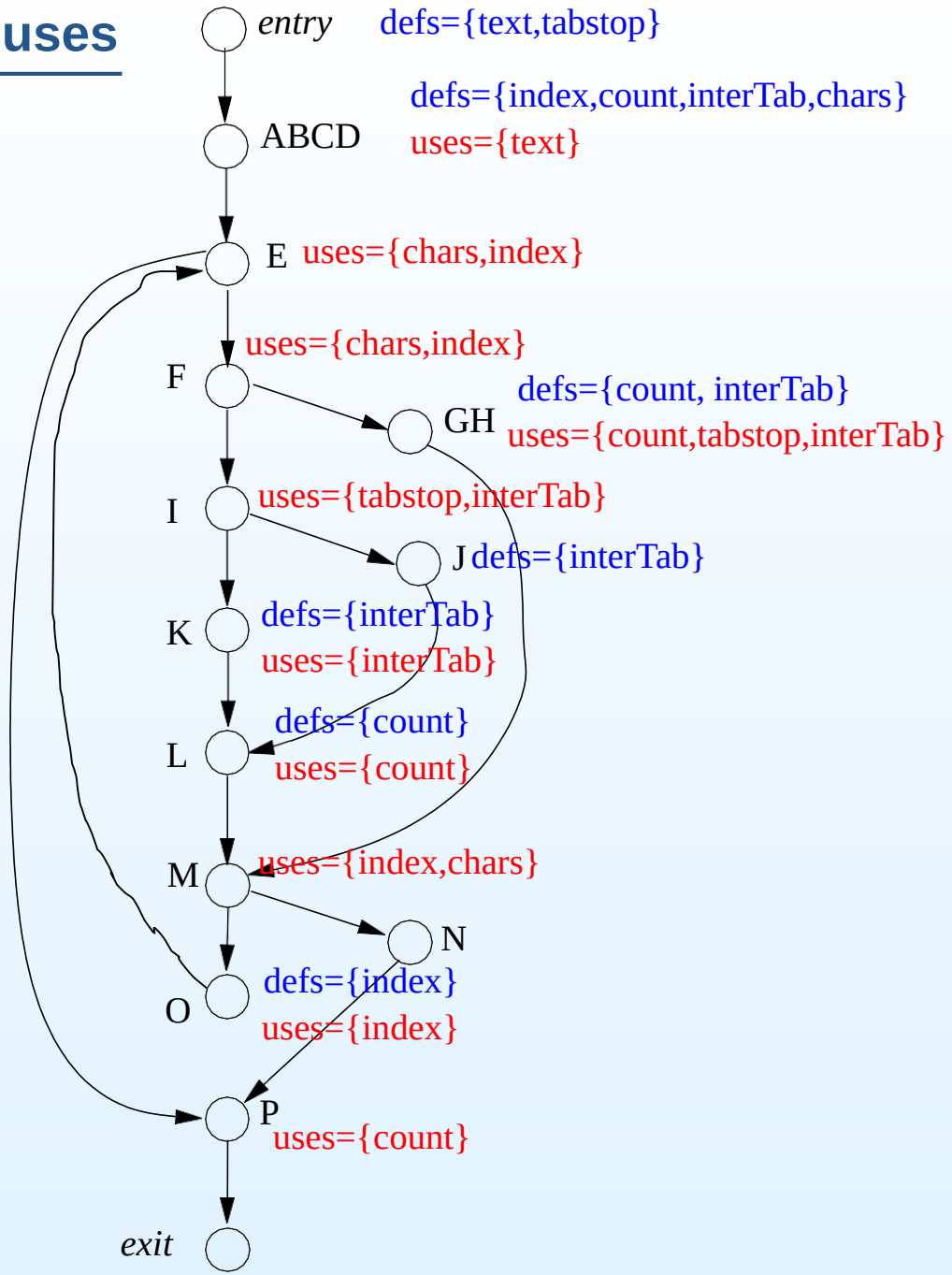
- "–" means that none of the actions of interest have happened to the data object
  - **–k** — variable not defined but then killed
  - **–d** — first definition in path
  - **–u** — used before defined
  - **k–** — killed is the last thing that happens
  - **d–** — defined but never used
  - **u–** — used but never killed, probably ok
- Issues with "global" data

du-paths

- We care about whether a value that is defined is correct for a given use — pair defs with uses

  $\Rightarrow$ we care about which defs "reach" which uses
- The set $defs(v)$ is the set of variables that are defined at vertex v
- The set $uses(v)$ is the set of variables that are used at vertex v
- It could be that $x \in defs(v)$ and $x \in uses(v)$
  - e.g. `i++` — use followed by def
  - e.g. `int i = 0; System.out.println(i);` — basic blocks with multiple statements, def followed by use
- A path $p = (v_0, v_1, \ldots, v_k)$ is def-clear with respect to variable x if $\forall 1 \leq i \leq k-1, x \notin defs(v_i)$
- A du-path with respect to variable x is a simple path $p = (v_0, v_1, \ldots, v_k)$ that is def-clear with respect to x such that $x \in defs(v_0)$ and $x \in uses(v_k)$
  - need special cases for $k = 0$ (def must come before use)
  - there can be other uses of x on p

Example: defs and uses



Example: du-paths for `interTab`

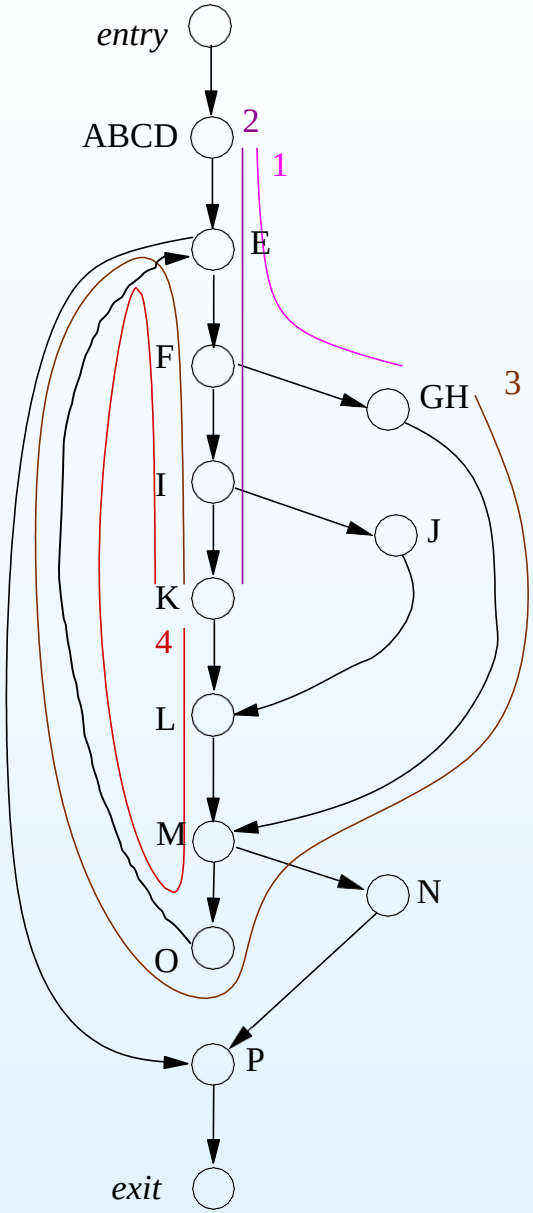
1: A, E, F, G

2: A, E, F, I, K

3: G, M, O, E, F, I, K

4: K, L, M, O, E, F, I, K

(not a complete list)



Key Points

- Need to find “interesting” subsets of paths — coverage criterion = definition of “interesting”
- A failure is often due to the wrong value being used somewhere.
- Finding the fault that caused the failure (“debugging”) involves finding out where that wrong value came from
- ⇒ a good place to look for possible faults is along the paths between the source of the value (definition) and its use
- Procedure
  - Identify *definitions* of values
  - Identify *uses* of those definitions
  - Identify (control flow) paths from definitions of values to their uses