

Dataflow-based Coverage Criteria

W. Eric Wong
Department of Computer Science
The University of Texas at Dallas
ewong@utdallas.edu
<http://www.utdallas.edu/~ewong>

Speaker Biographical Sketch

- Professor & Director of International Outreach
Department of Computer Science
University of Texas at Dallas
- Guest Researcher
Computer Security Division
National Institute of Standards and Technology (NIST)
- Vice President, IEEE Reliability Society
- Secretary, ACM SIGAPP (Special Interest Group on Applied Computing)
- Principal Investigator, NSF TUES (Transforming Undergraduate Education in Science, Technology, Engineering and Mathematics) Project
– *Incorporating Software Testing into Multiple Computer Science and Software Engineering Undergraduate Courses*
- Founder & Steering Committee co-Chair for the SERE conference
(*IEEE International Conference on Software Security and Reliability*)
(<http://paris.utdallas.edu/sere13>)



Basic Concepts

- We will now examine some test adequacy criteria based on the **flow of “data”** in a program. This is in contrast to criteria based on the **flow of “control”** that we have examined so far.
- Test adequacy criteria based on the flow of data are useful in improving tests that are adequate with respect to controlflow-based criteria.
- Let us look at an example.

Example: Test Enhancement using Dataflow (1)

```
1  begin
2    int x, y; float z;
3    input (x, y);
4    z=0;
5    if (x== 0)
6      z=z+y;
7    else z=z-y;
8    if (y!=0)
9      z=z/x;
10   else z=z*x;
11   output(z);
12  end
```

Question: Does the following test set reveal the bug?

Test	x	y	z
t_1	0	0	0.0
t_2	1	1	1.0

Example: Test Enhancement using Dataflow (2)

- Neither of the two test cases forces the use of z defined on line 6, at line 9. To do so one requires a test that causes conditions at lines 5 and 8 to be true (i.e., need to satisfy $x == 0$ and $y != 0$)
- The test which we have does not force the execution of this path and hence the *divide by zero* error is not revealed.

```

1  begin
2  int x, y; float z;
3  input (x, y);
4  z=0;
5  if (x==0)
6  z=z+y;
7  else z=z-y;
8  if (y!=0)
9  z=z/x;
10 else z=z*x;
11 output(z);
12 end

```

Annotations in the code:

- Line 5: **define z here** (with an arrow pointing to the `if (x==0)` condition)
- Line 8: `if (y!=0)` ← This condition should be ($y! = 0$ and $x! = 0$)
- Line 9: **use z here** (with an arrow pointing to `z=z/x;`)

Example: Test Enhancement using Dataflow (3)

- Verify that the following test set covers all **def-use pairs** of z and reveals the bug.

Test	x	y	z	*def-use pairs covered
t_1	0	0	0.0	(4,6), (6,10), (10, 11)
t_2	1	1	-1.0	(4,7), (7,9), (9, 11)
t_3	0	1	...	(4,6), (6,9)
t_4	1	0	0.0	(4,7), (7,10), (10, 11)

Annotations in the table:

- Arrows point from the 'def-use pairs covered' column to the following pairs: (4, 5, 6, 8, 10), (4, 5, 7, 8, 9), (4, 5, 6, 8, 9) (reveal the bug), (4, 5, 7, 8, 10).
- A red circle highlights the row for t_3 and the first two rows.
- A blue box highlights the row for t_3 .

*In the pair (t_1, t_2), z is defined in t_1 and used in line t_2 .

def-use pairs with respect to the variable z

Definitions and Uses (1)

- A program written in a procedural language, such as C and Java, contains variables.
- Variables are defined by assigning values to them and are used in expressions.
 - Statement $x = y + z$ defines variable x and uses variables y and z
 - Statement `scanf (“%d %d”, &x, &y)` defines variables x and y
 - Statement `printf (“Output: %d \n”, x + y)` uses variables x and y

Definitions and Uses (2)

- A parameter x passed as *call-by-value* to a function, is considered as *a use* of (or a reference to) x
- A parameter x passed as *call-by-reference*, can serve as *a definition* and *use* of x

Definitions and Uses: Pointers

- Consider the following sequence of statements that use pointers.

```
z=&x;  
y=z+1;  
*z=25;  
y=*z+1;
```

- The first defines a pointer variable z
- the second defines y and uses z
- the third defines x through the pointer variable z , and
- the last defines y and uses x accessed through the pointer variable z

Variable z is a pointer pointing to variable x and contains the memory address of variable x .
 $*z$ retrieves the value at the memory address pointed by variable z . Consequently, $*z = 25$ is to assign 25 to the memory address pointed by variable z . That is, to assign 25 to variable x .

$y = *z + 1$ is to define y as the sum of 1 and the value at the memory address pointed by variable z , i.e., the value of x

Definitions and Uses: Arrays

- Arrays are also tricky. Consider the following declaration and two statements in C:

```
int A[10];  
A[i]=x+y;
```

- The first statement defines variable A .
The second statement defines A and uses i , x , and y .

Alternate: *second statement defines $A[i]$ and not the entire array A .*
The choice of whether to consider the entire array A as defined or the specific element *depends upon how stringent the requirement for coverage analysis is.*

C-Use

- Uses of a variable that occurs within an expression as part of an *assignment statement*, in an *output statement*, as a *parameter within a function call*, and in *subscript expressions*, are classified as **c-use**, where the “c” in c-use stands for **computational**.

- How many **c-uses** of x can you find in the following statements?

```
z=x+1;  
A[x-1]= B[2];  
foo(x*x)  
output(x);
```

- Answer = ?

P-Use

- The occurrence of a variable in an expression used as a *condition in a branch statement* such as an *if* and a *while*, is considered as a **p-use**. The “p” in p-use stands for **predicate**.

- How many **p-uses** of z and x can you find in the following statements?

```
if(z>0){output (x)};  
while(z>x){...};
```

- Answer = ?

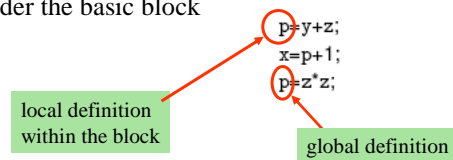
P-Use: Possible Confusion

- Consider the statement:

```
if(A[x+1]>0){output(x)};
```
- The use of **A** is clearly a p-use.
- Is the use of **x** in the subscript a c-use or a p-use?

C-Uses Within a Basic Block

- Consider the basic block



- While there are two definitions of **p** in this block, *only the second definition will propagate to the next block*. The first definition of **p** is considered **local** to the block while the second definition is **global**. *We are only concerned with global definitions and uses.*
- Note that **y** and **z** are **global uses**; their definitions flow into this block from some other block.

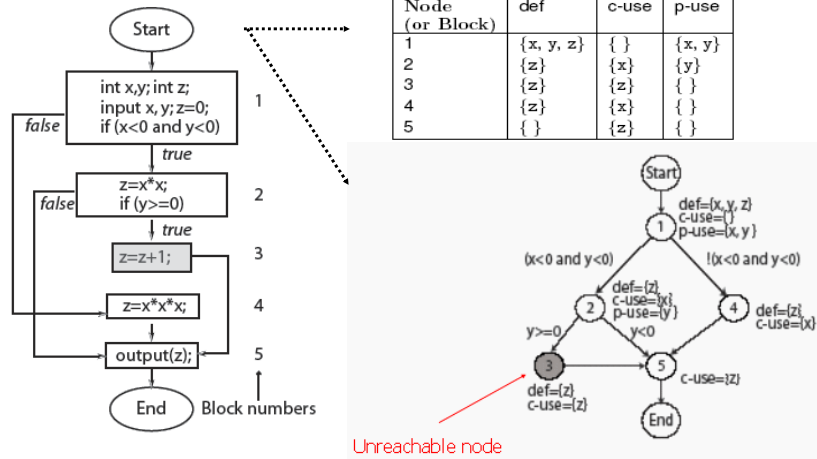
Dataflow Graph

- A *dataflow graph of a program*, also known as *def-use graph*, captures the flow of definitions (also known as defs) and uses across basic blocks in a program.
- *It is similar to a control flow graph* of a program in that the nodes, edges, and all paths in the control flow graph are preserved in the data flow graph. An example follows.

Dataflow Graph: Example (1)

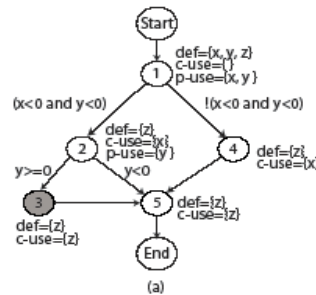
- Given a program, find its basic blocks, compute **defs**, **c-uses** and **p-uses** in each block. *Each block becomes a node in the def-use graph (this is similar to the control flow graph).*
- *Attach defs, c-use and p-use to each node in the graph.*
Label each edge with the condition which when true causes the edge to be taken.
- We use $d_i(x)$ to refer to the definition of variable x at node i . Similarly, $u_i(x)$ refers to the use of variable x at node i .

Dataflow Graph: Example (2)



Def-Clear Path

- Any path starting from a node at which variable x is **defined** and ending at a node at which x is **used**, *without redefining x anywhere else along the path*, is a **def-clear path** for x
- Path 2-5 is def-clear for variable z defined at node 2 and used at node 5.
- Path 1-2-5 is **NOT def-clear** for variable z defined at node 1 and used at node 5.
- Thus definition of z at node 2 is **live** at node 5 while that at node 1 is not live at node 5.



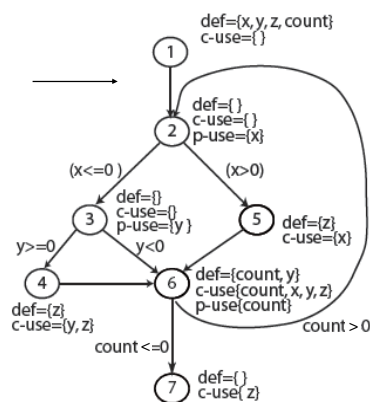
Def-Use Pairs

- Definition of a variable at line l_1 and its use at line l_2 constitute a *def-use pair*. l_1 and l_2 can be the same.
 - $dcu(d_i(x))$ denotes the set of all nodes where $d_i(x)$ is live and c-used.
 - $dpu(d_i(x))$ denotes the set of all edges (k, l) such that there is a def-clear path from node i to edge (k, l) and x is p-used at node k .
- We say that a def-use pair $(d_i(x), u_j(x))$ is covered when a def-clear path that includes nodes i to node j is executed.
- If $u_j(x)$ is a p-use then all edges of the kind (j, k) must also be taken during some executions.

Def-Clear Path (Another Example) (1)

```

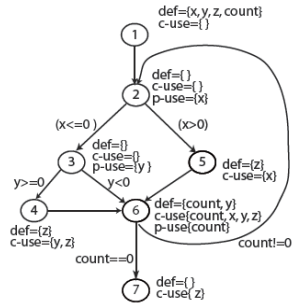
1 begin
2   float x, y, z=0.0;
3   int count;
4   input (x, y, count);
5   do {
6     if (x<=0) {
7       if (y>=0) {
8         z=y*z+1;
9       }
10    }
11    else{
12     z=1/x;
13    }
14    y=x*y+z
15    count=count-1
16    while (count>0)
17     output (z);
18  end
    
```



Node	Lines
1	1, 2, 3, 4
2	5, 6
3	7
4	8, 9, 10
5	11, 12, 13
6	14, 15, 16
7	17, 18

Find def-clear paths for defs and uses of x and z .
Which definitions are live at node 4?

Def-Clear Path (Another Example) (2)



Variable (v)	Defined in node (n)	dpu (v, n)	dpu (v, n)
x	1	{5, 6}	{{(2, 3), (2, 5)}
y	1	{4, 6}	{{(3, 4), (3, 6)}
y	6	{4, 6}	{{(3, 4), (3, 6)}
z	1	{4, 6, 7}	{ }
z	4	{4, 6, 7}	{ }
z	5	{4, 6, 7}	{ }
count	1	{6}	{{(6, 2), (6, 7)}
count	6	{6}	{{(6, 2), (6, 7)}

Infeasible! Why?

Def-Use Pairs: Minimal Set (1)

- Def-use pairs are items to be covered during testing. However, in some cases, coverage of a def-use pair implies coverage of another def-use pair. Analysis of the data flow graph can reveal a minimal set of def-use pairs whose coverage implies coverage of all def-use pairs.
- Exercise: Analyze the def-use graph shown on slide 20 to determine
 - Which def-uses are infeasible?
 - A minimal set of def-uses to be covered
 - corresponding to “set covering”
 - in theory, this is NP-complete
 - χSuds/ATAC provides a good “approximate” solution (will be further explained when we discuss “Regression Testing”)

Def-Use Pairs: Minimal Set (2)

- What will be also covered if we have a test case which covers $(d_1(z), u_4(z))$?
- How about $(d_4(z), u_4(z))$?

C-Use Coverage

- The c-use coverage of a test set T with respect to (P, R) is computed as

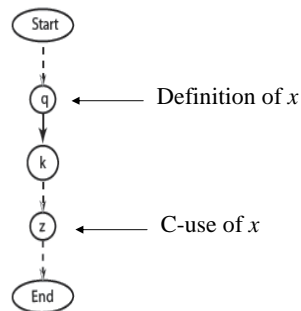
$$\frac{CU_c}{(CU - CU_f)}$$

where CU is the total number of c-uses, CU_c is the number of c-uses covered by test cases in T , and CU_f is the number of infeasible c-uses.

- T is considered adequate with respect to the c-use coverage criterion if its c-use coverage is 1.

C-Use Coverage: Path Traversed

- Path (Start, .. q, k, \dots, z, \dots End) covers the c-use at node z of x defined at node q given that $(k \dots, z)$ is def-clear with respect to x
- **In-class Exercise:** Find the c-use coverage when the code on slide 20 is executed against the test case $\langle x = 5, y = -1, \text{count} = 1 \rangle$



P-Use Coverage

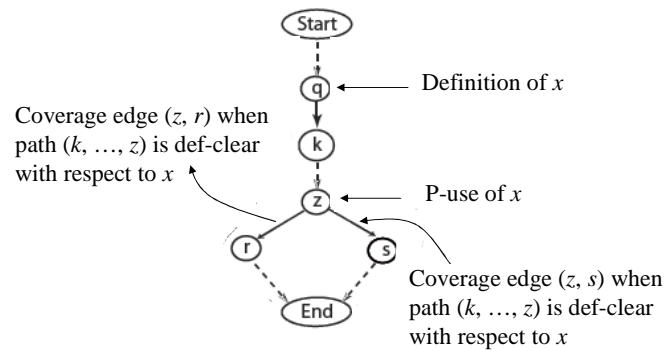
- The p-use coverage of a test set T with respect to (P, R) is computed as

$$\frac{PU_c}{(PU - PU_f)}$$

where PU is the total number of p-uses, PU_c is the number of p-uses covered by test cases in T , and PU_f is the number of infeasible p-uses.

- T is considered adequate with respect to the p-use coverage criterion if its p-use coverage is 1.

P-Use Coverage: Paths Traversed



In-class Exercise: Find the p-use coverage when the code on slide 20 is executed against the test case $\langle x = -2, y = -1, \text{count} = 3 \rangle$

All-Uses Coverage

- The all-uses coverage of a test set T with respect to (P, R) is computed as

$$\frac{(CU_c + PU_c)}{((CU + PU) - (CU_f + PU_f))}$$

where CU , CU_c and CU_f are defined on slide 24, and PU , PU_c and PU_f are defined on slide 26.

- T is considered adequate with respect to the all-uses coverage criterion if its all-uses coverage is 1.

All-Uses Coverage: Example

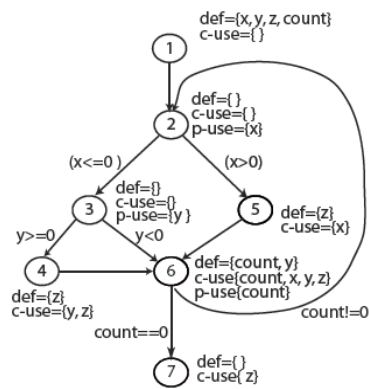
- **In-class Exercise:** Referring to the code on slide 20, is a test set $T = \{ \langle x = 5, y = -1, \text{count} = 1 \rangle, \langle x = -2, y = -1, \text{count} = 3 \rangle \}$ adequate with respect to the all-uses coverage?

Infeasible P- and C-Uses

- Coverage of a c-use or a p-use requires a path to be traversed through the program. However, *if this path is infeasible, then some c-uses and p-uses that require this path to be traversed might also be infeasible.*
- Infeasible c-uses and p-uses are often difficult to determine.

Infeasible C-Use: Example

- Consider the c-use at node 4 of z defined at node 5.
- Explain why this c-use is infeasible.



Subsumes Relation

- Given a test set T that is *adequate* with respect to a criterion C_1 , what can we conclude *about the adequacy* of T with respect to another criterion C_2

Effectiveness of an Adequate Test Set

- Given a test set T that is *adequate* with respect to a criterion C , what can we expect *regarding its effectiveness in revealing bugs*?

Effectiveness Relation

- Given a test set T_1 that is *adequate* with respect to a criterion C_1 and a test set T_2 that is *adequate* with respect to another criterion C_2 . Assume criterion C_1 subsumes criterion C_2 , what can we conclude *about the fault detection effectiveness* of T_1 with respect to the *fault detection effectiveness* of T_2 ?