Data Flow Testing

Dr. John H Robb, PMP, SEMC
UTA Computer Science and Engineering

What is Data Flow Testing?

- Data Flow testing is a "white box" testing technique that can be used to detect improper use of data values.

- Data Flow testing refers to a category of structural testing techniques that focus on the points of the code variables obtain values (are defined) and the points of the program these variables are referenced (are used)
  - Around faults that may occur when a variable is defined, assigned a value and referenced
  - Parts of a program that constitute a slice – a subset of program statements that comply with a specific slicing criterion (i.e. all program statements that are affected by variable x at point P)

- Data Flow testing involves selecting entry/exit paths with the objective of covering certain data definition and use patterns, commonly known as Data Flow criteria

Examples of Data-Flow Errors

- Some examples of data-flow errors:
  - Assigning an incorrect or invalid value to a variable. These kinds of errors include data-type conversion
  - issues where the compiler allows a conversion but there are side effects that are undesirable.
  - Incorrect input results in the assignment of invalid values.
  - Failure to define a variable before using its value elsewhere.
  - Incorrect path taken due to the incorrect or unexpected value used in a control predicate.
  - Trying to use a variable after it is destroyed or out of scope.
  - Redefining a variable before it is used.

- Those familiar with Static Code Analysis (SCA) tools will see similarities with the checks that SCA tools perform

How is Data Flow Testing Performed?

- Data Flow testing is performed in three steps
  1. Model the program as a CFG
  2. The associations between definitions and usages of variables to be covered is established
  3. Test cases are created using a the paths developed in the previous step

- As opposed to developing test cases solely from the control flow of the program, data flow identifies potential defects by examining the patterns in which that data is used.

- Since data flow testing is performed from a CFG, it provides a means to identify additional test cases that can detect defects. It is a complementary technique to the basis path testing approach, which when both are used together produce a very solid set of test cases.

Data Flow Terminology

- The notation for representing the patterns is:
  - d – defined, created, initialized
  - k – killed, terminated, undefined
  - u – used
  - c – used in a computation
  - p – used in a predicate (decision)
  - ~x - indicates all prior actions are not of interest to x
  - x~ - indicates all post actions are not of interest to x

- The table on the following slide lists the combinations of usage and their corresponding consequences.

- We can see that not all data-flow anomalies are harmful but they are all suspicious and indicate that an error can occur.

- For example, the usage pattern ‘ku’ indicates that a variable is used after it has been killed which is a serious defect.

Definitions

- A program variable is *defined* "def" when it appears:
  - on the left hand side of an assignment statement eg y = 17
  - as an call-by-reference parameter in a subroutine call eg update(x, &y);
- A program variable is *used* "ref" when it appears:
  - on the right hand side of an assignment statement eg y = x+17
  - as an call-by-value parameter in a subroutine or function call eg y = sqrt(x)
  - in the predicate of a branch statement eg if (x > 0) { … }

Definitions (cont.)

- Use in the predicate (decision) is a predicate-use or "p-use"
- Any other use is a computation-use or "c-use"
- A use is either a p-use or a c-use
- For example, in the program fragment:

```
if ( x > 0 ) {
        print(y);
        }
```

  there is a p-use of x and a c-use of y
- A variable can also be used and then re-defined in a single statement when it appears:
  - on both sides of an assignment statement eg y = y + x
  - as an call-by-reference parameter in a subroutine call
    eg increment(&y)
  - Notice that a du path would lead to these nodes since they perform a ref before performing a def

Definitions (cont.)

- A path is definition clear ("def-clear") with respect to a variable v if it has no variable re-definition of v on the path
- A complete path is a path whose initial node is a start node and whose final node is an exit node
- A definition-use pair ("du-pair") with respect to a variable v is a double (d,u) such that
  - d is a node in the program's flow graph at which v is defined,
  - u is a node or edge at which v is used and
  - there is a def-clear path with respect to v from d to u
- Note that the definition of a du-pair does not require the existence of a feasible def-clear path from d to u

General Observations

- Testers can see how these definitions capture the essence if computing stored data.
- Du-paths and dc-paths describe the flow of data across source statements from points at which the values are defined to points at which the values are used.
- Du-paths that are not dc-paths are potential problems.
- This is the way that testers/programmers tend to troubleshoot problems as well. We tend to look for places where the value mis-compared and trace this back to all other places where the data was defined.
- By defining test cases this way we are defining tests that reflect the way we tend to naturally troubleshoot software problems.

Static vs. Dynamic Data Flow Testing

- Static Analysis is analysis performed on source code without actually executing it - like our CFG/basis path test case design - except we do expect to actually execute test later with the basis path

- Dynamic Analysis is performed on a program as it is executing and is based on intermediate values that result from its execution.

- Recent approaches look at hybrid approaches including many commercial testing tools

Set Use Pairs

- This one is sometimes called set-use pair notation.
- We will split the lifecycle of a data variable into three separate patterns:
  - d - the time when the variable is created, defined, or initialized.
  - u - used. The variable may be used in a computation or in a decision predicate.
  - k - killed, destroyed. or has become out of scope.
- These three atomic actions are then combined to show a data flow. A ~ (tilde) is often used to show the first or last action that can occur.
- We then analyze the pairs
  - ~d, or first define. This is the normal way a variable is originated; it is declared.
  - du, or define-use. This is the normal way a variable is used. Defined first and then used in an assignment or decision predicate
  - ku, or kill then used. This is a bug; the variable was killed and then used.

Set Use Pairs - Data Flow Terminology Table

Anomaly		Explanation
~d	first define	Allowed.
du	define – use	Allowed. Normal case.
dk	define – kill	Potential bug. Data is killed without use after definition.
~u	first use	Potential bug. Data is used without definition.
ud	use – define	Allowed. Data is used and then redefined.
uk	use – kill	Allowed.
~k	first kill	Potential bug. Data is killed before definition.
ku	kill – use	Serious Defect. Data is used after being killed.
kd	kill – define	Allowed. Data is killed and then re-defined.
dd	define–define	Potential bug. Double definition.
uu	use – use	Allowed. Normal case.
kk	kill – kill	Potential bug.
d~	define last	Potential bug.
u~	use last	Allowed.
k~	kill last	Allowed. Normal case.

An Example

- Consider the example of a software program that calculates the bill of a cell telephone provider depending upon the usage

- The following rules define how the Bill is charged

Usage (minutes)	Charge	Notes
0	$0.00	Not specified by the table – from code
<=100	$40.00	Base rate
101-200	$0.50 ($/min)	For each additional minute
>200	$0.10 ($/min)	For each additional minute

- If the Bill is >=$100.00 then a 10 % discount is applied to the total Bill

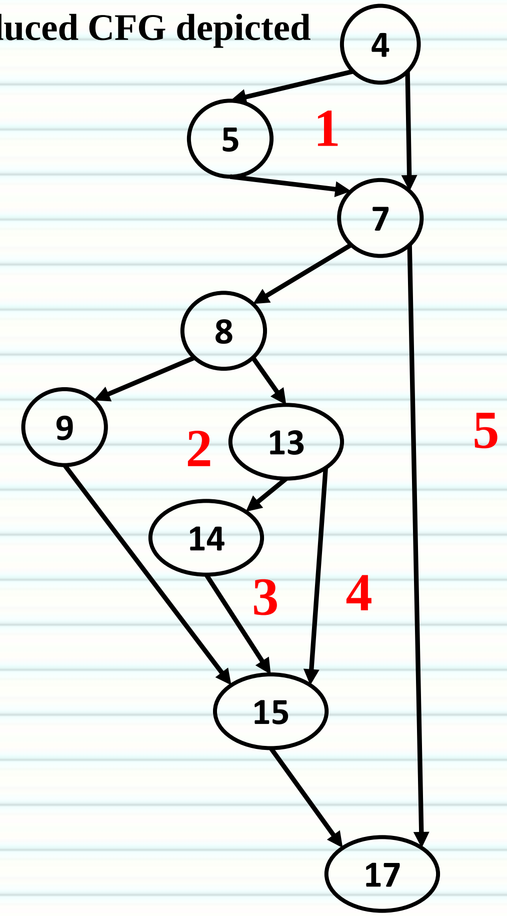
Code for the Example Problem

```
public static double calculateBill (int Usage) {
        double Bill = 0;

        if (Usage > 0)
                Bill = 40;

        if (Usage > 100)
                if (Usage <= 200)
                        Bill += (Usage-100)*0.5;
                else {
                        Bill += 50.0 + (Usage - 200)*0.1; }

                        if (Bill >=100.0)
                                Bill *= 0.9;
                        }

        return Bill;
}
```

Student Exercise - Develop the CFG, Basis paths, and boundary values and expected outputs for the code above

Basis Path & Boundary Value Approach

Reduced CFG depicted



Basis paths :

1) 4-7-17
2) 4-5-7-17
3) 4-5-7-8-9-15-17
4) 4-5-7-8-13-15-17
5) 4-5-7-8-13-14-15-17

Some paths are not feasible - 4-7-8+

Boundary values for basis paths would be:

1) usage=0, exp output=$0.00
2) usage=1, exp output=$40.00
3) usage=101, exp output=$40.50
4) usage=200, exp output=$90.00
5) usage=300, exp output=$90.00

Additional test cases to test all boundary values and extreme ranges would be:

6) usage=100, exp output=$40.00
7) usage=201, exp output=$90.10
8) usage=299, exp output=$99.90
9) usage=1,000, exp output=$153.00

Notice that the ECPs with a non-uniform response would require three additional test cases (nodes 9, 11,14) which are not addressed yet (direct comparison with the referenced paper which does not specifically address them).

0	1 100	101 200	201 299	300 ∞

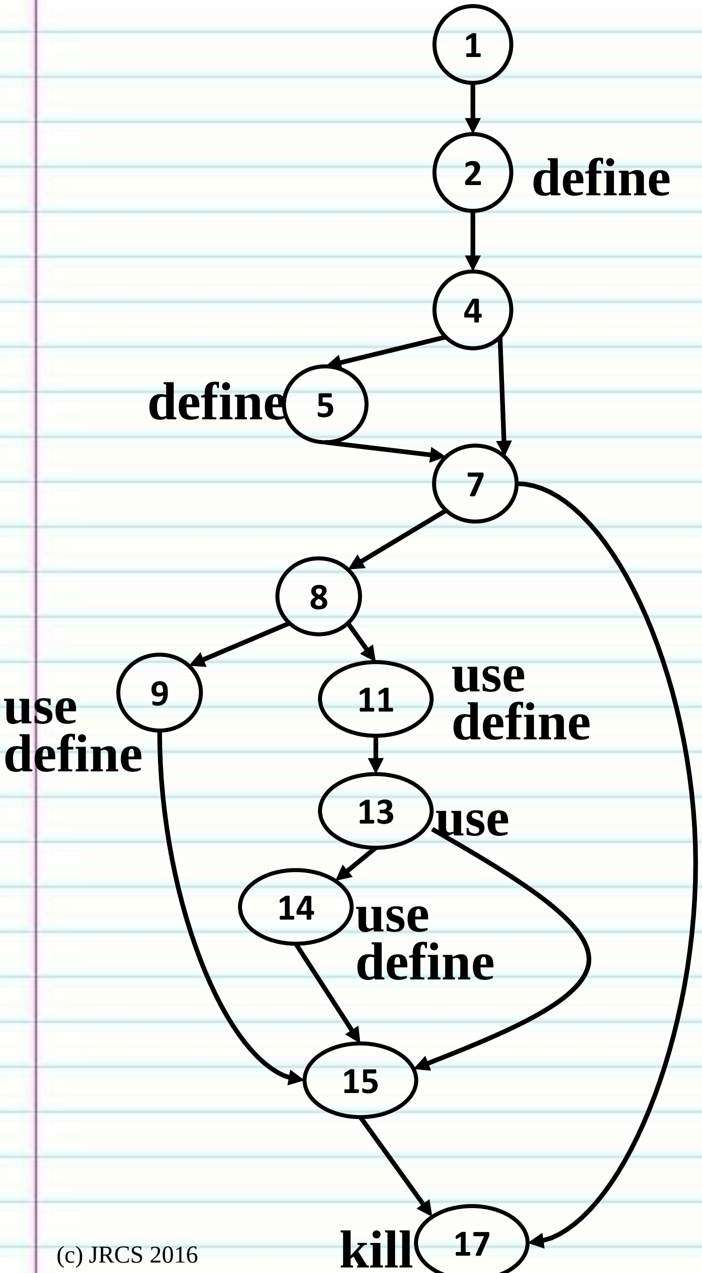
Define Kill Use Information

- For variable ‘*Usage*’, the define-use-kill patterns are
    - ~ define : normal case
    - define-use : normal case
    - use-use : normal case
    - use-kill : normal case

- For variable ‘*Bill*’, the define-use-kill patterns are
    - ~ define : normal case
    - define-define: suspicious
    - define-use : normal case
    - use-define : acceptable
    - use-use : normal case
    - use-kill : normal case

- So the example identified for the variable Bill a potential defect

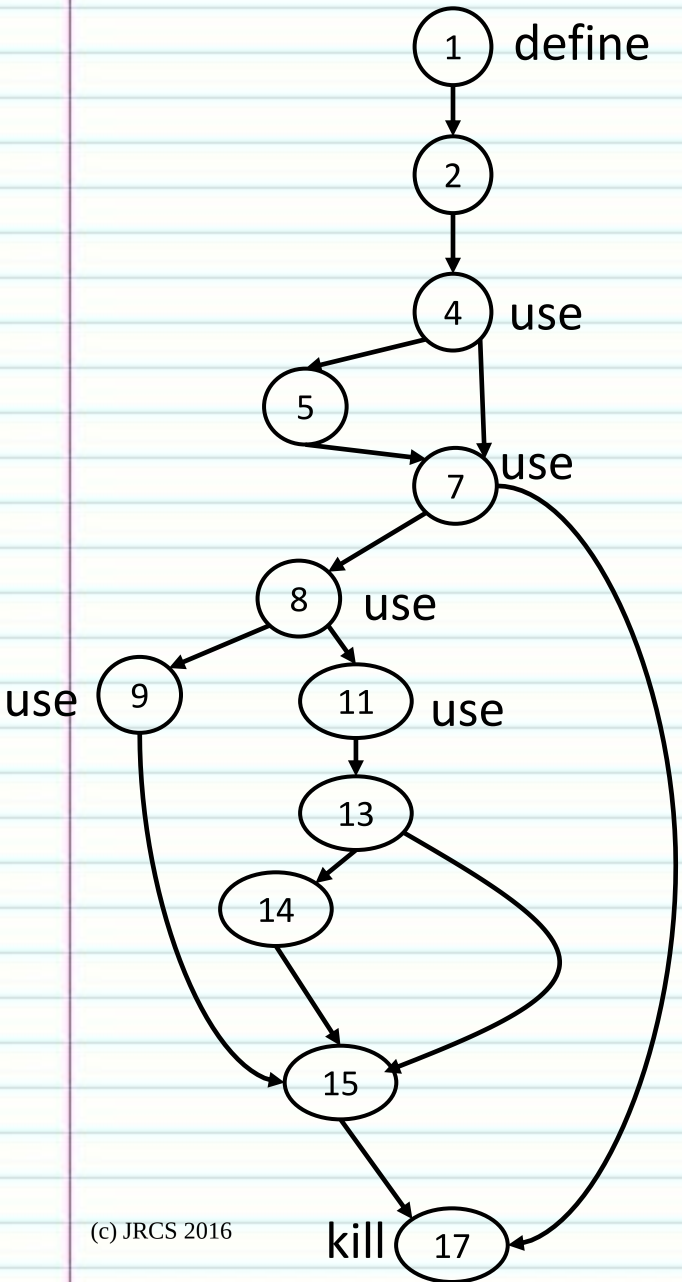
Annotated CFG for Variable Bill



Anomaly		Explanation
~d	1-2	Allowed. Normal case
dd	1-2-4-5	Potential bug. Double definition.
du	5-7-8-11	Allowed. Normal case.
ud	11	Allowed. Data is used and then redefined.
uk	17-18	Allowed. 18 not shown on graph
dd	2-4-5	Potential bug. Double definition.
uu	13-14	Allowed. Normal case.
k~	18	Allowed. Normal case. 18 not shown on graph

The table identifies 0-2-4-5 as a potential defect due the the dd relationship from the static analysis

Annotated CFG for Variable 'Usage'



Anomaly		Explanation
~d	1	Allowed.
du	1-2-4	Allowed. Normal case.
uk	9-17-18	Allowed. 18 not shown on graph
uu	8-11	Allowed. Normal case.
k~	18	Allowed. Normal case. 18 not shown on graph

No potential bugs identified with the variable Usage

Dynamic Data Flow Testing Strategies

- The primary purpose of dynamic data-flow testing is to uncover possible defects in data usage during the execution of the code.
- To achieve this, test cases are created which trace every definition to each of its use and every use is traced to each definition.
- Various strategies are employed for the creation of the test cases as discussed on the next slides.
- The definition of all strategies is followed by an example to explain each.
- This is dynamic data flow testing because we expect to execute the test cases we are developing. Typical dynamic data flow testing would also include results from actual execution - leading research, beyond this example, would allow this data to improve the selection of test cases.
- This example is simply intended to show the direct results of test cases developed from a static data flow example and compare this with our basis path/boundary coverage approach.

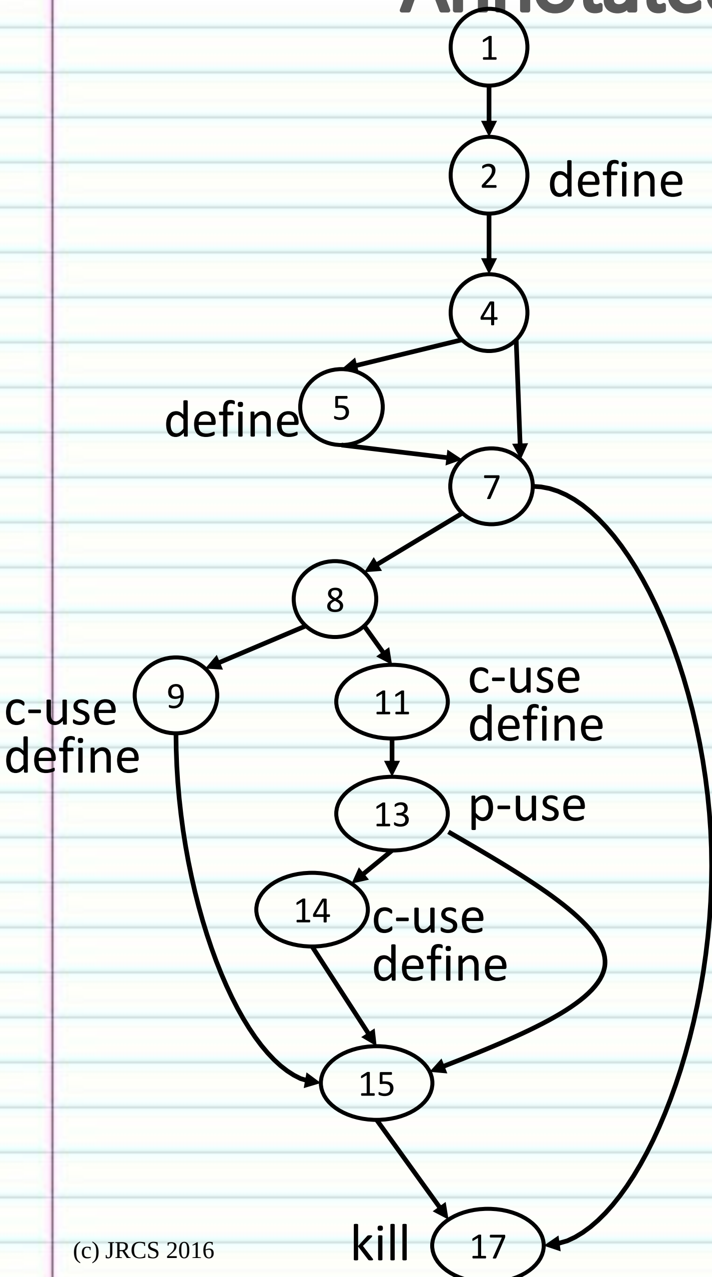
Dynamic Data Flow Testing Strategies (cont.)

- All-definition (AD)
  - Every definition of every variable is covered by at least one use of that variable, whether that use is a computational use or a predicate use.
  - For this testing strategy, there is path from every definition to at least one use of that definition.
- All-du paths (ADUP)
  - For every program variable v, every du-path from every definition of v to every c-use and every p-use of v must be covered. It is the strongest data-flow testing strategy since it is a superset of all other data flow testing strategies. Moreover, this strategy requires greatest number of paths for testing.
- All-Uses (AU)
  - At least one path from every definition of every variable to every use of that can be reached by that definition. For every use of the variable, there is a path from the definition of that variable to the use.

Dynamic Data Flow Testing Strategies (cont.)

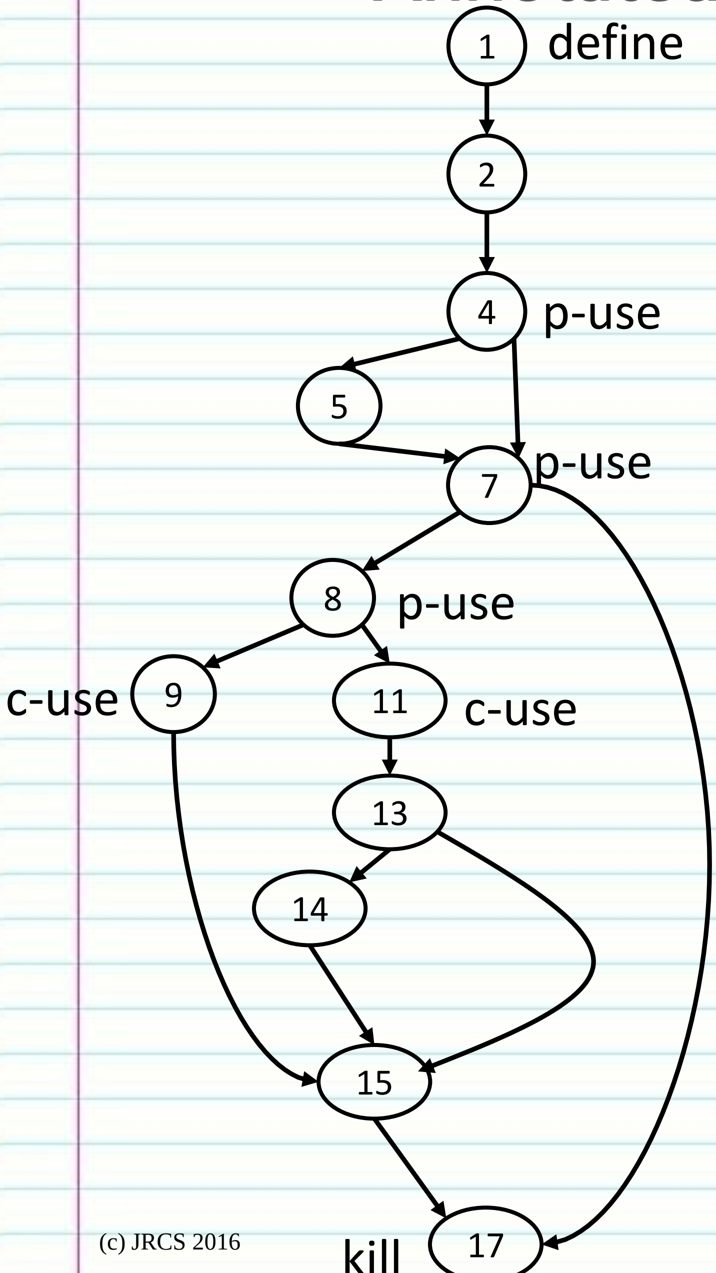
- All-c-uses/Some-p-uses (ACU+P)
  - For this testing strategy, for every variable, there is a path from every definition to every c-use of that definition. If there is a definition with no c-use following it, then a p-use of the definition is considered.
- All-p-uses/Some-c-uses (APU+C)
  - For this testing strategy, for every variable, there is a path from every definition to every p-use of that definition. If there is a definition with no p-use following it, then a c-use of the definition is considered.
- All-p-uses (APU)
  - For this testing strategy, for every variable, there is path from every definition to every p-use of that definition. If there is a definition with no p-use following it, then it is dropped from consideration.
- All-c-uses (ACU)
  - For this testing strategy, for every variable, there is a path from every definition to every c-use of that definition. If there is a definition with no c-use following it, then it is dropped from consideration.

Annotated CFG for Variable 'Bill'




```
public static double calculateBill (int Usage) {
    double Bill = 0;

    if (Usage > 0)
        Bill = 40;

    if (Usage > 100)
        if (Usage <= 200)
            Bill += (Usage-100)*0.5;
        else {
            Bill += 50.0 + (Usage - 200)*0.1;

            if (Bill >=100.0)
                Bill *= 0.9;
        }

    return Bill;
}
```

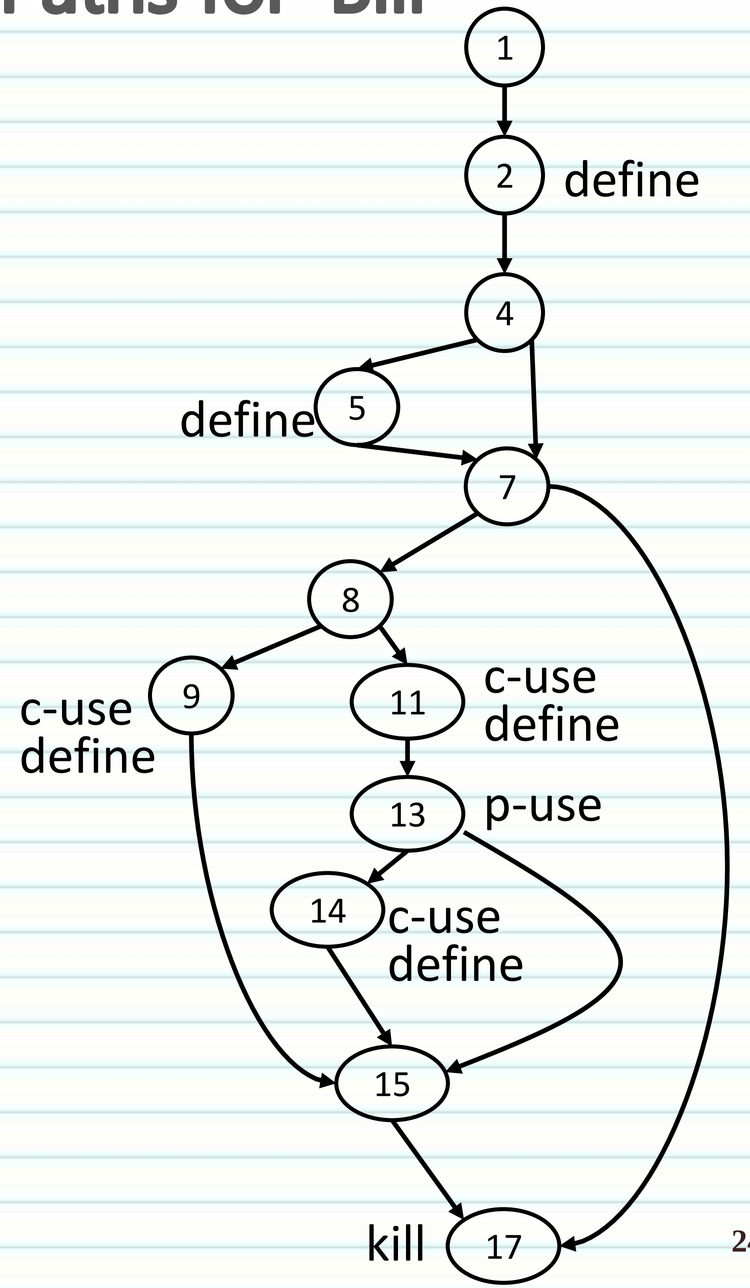
Annotated CFG for Variable 'Usage'




```
public static double calculateBill (int Usage) {
        double Bill = 0;

        if (Usage > 0)
                Bill = 40;

        if (Usage > 100)
                if (Usage <= 200)
                        Bill += (Usage-100)*0.5;
                else {
                        Bill += 50.0 + (Usage - 200)*0.1;

                        if (Bill >=100.0)
                                Bill *= 0.9;
                        }

        return Bill;
}
```

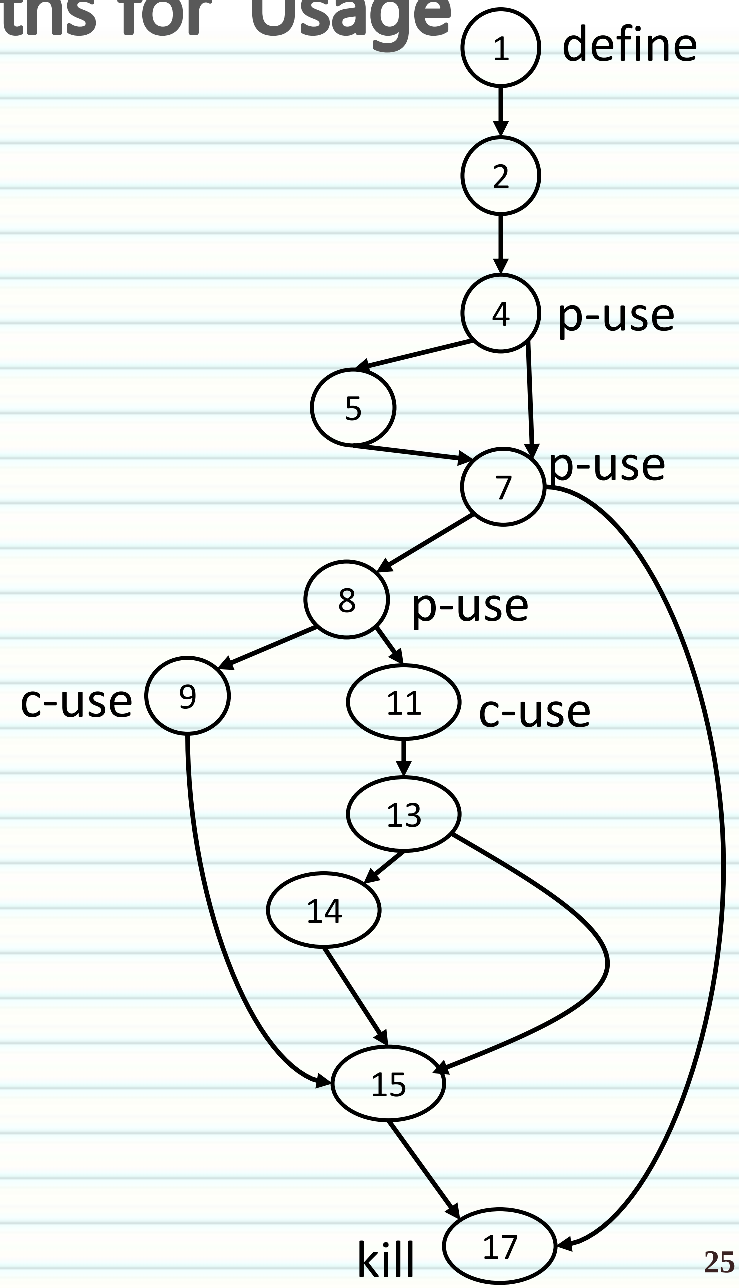
Data Flow Testing Paths for 'Bill'

Strategy	Bill
All uses (AU)	5-7-8-11 11-13 11-13-14 14-17 5-7-8-9
All p – uses (APU)	2-4-5-7-8-11-13 5-7-8-11-13 11-13
All c – uses (ACU)	2-4-17 5-7-8-11 11-13-14 14-17 9-17
All p – use/some c (APU + C)	2-4-5-7-8-11-13 5-7-8-11-13 11-13 14-17 9-17
All c – use/some p (ACU + P)	2-4-17 5-7-8-11 5-7-8-9 11-13-14 14-17 9-17



Data Flow Testing Paths for 'Usage'

Strategy	Usage
All uses (AU)	1-2-4 1-2-4-5-7 1-2-4-5-7-8 1-2-4-5-7-8-11 1-2-4-5-7-8-9
All p – uses (APU)	1-2-4 1-2-4-5-7 1-2-4-5-7-8
All c – uses (ACU)	1-2-4-5-7-8-11 1-2-4-5-7-8-9
All p – use/some c (APU + C)	1-2-4 1-2-4-5-7 1-2-4-5-7-8
All c – use/some p (ACU + P)	1-2-4-5-7-8-11 1-2-4-5-7-8-9

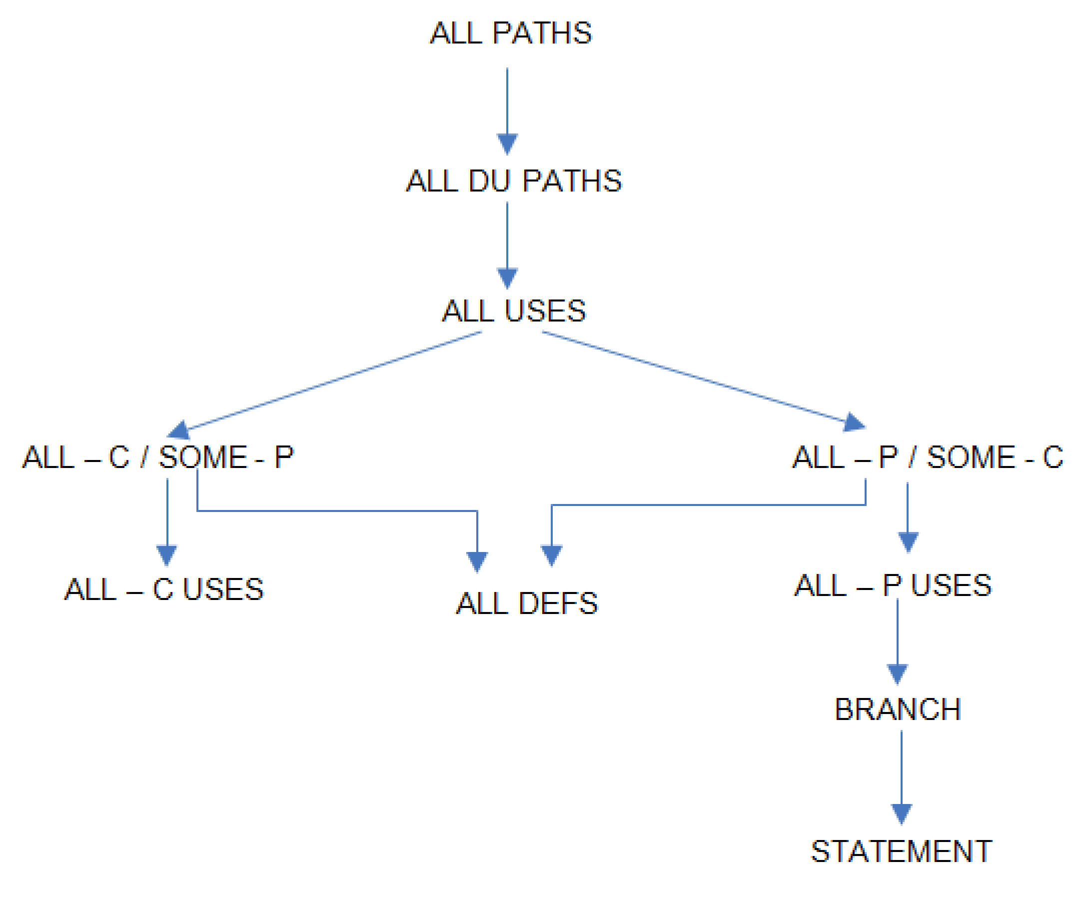


Data Flow Testing Paths for Each Variable

Strategy	Bill	Usage
All du (ADUP)	(ACU+P) + (APU+C)	(ACU+P) + (APU+C)
All definition	2-4-17 5-7-8-11 11-13 14-17 9-17	1-2-4

Ordering of Strategies

- The strength of test cases goes from the highest at the top to the weakest at the bottom



Various Test Suites for Variable 'Bill'

- The table on the right represents the different test strategies, the selected paths, and inputs/expected outputs identified by the authors (path numbers are according to their statement numbering)
- We'll create a comparison with the most rigorous data flow test strategy (all du-paths) and compare this with the basis path approach (developed earlier).

- The authors haven't tested a single boundary value!
- We're going to adjust the input values for Usage to test boundary points

Uses	Bill	Input Usage Value	Expected Value
All definition (AD)	1-2-10 3-4-5-6 6-7 8-10 9-10	0 220 220 350 220	0.0 92.0 92.0 94.5 92.0
All c – use (ACU)	1-2-10 3-4-5-6 6-7-8 8-10 9-10	0 220 350 350 220	0.0 92.0 94.5 94.5 92.0
All p – use (APU)	1-2-3-4-5-6-7 3-4-5-6-7 6-7	220 220 220	92.0 92.0 92.0
All c - use + p (ACU + P)	1-2-10 3-4-5-6 3-4-5-9 6-7-8 8-10 9-10	0 220 170 350 350 170	0.0 92.0 75.0 94.5 94.5 75.0
All p - use + c (APU + C)	1-2-3-4-5-6-7 3-4-5-6-7 6-7 8-10 9-10	220 220 220 350 170	92.0 92.0 92.0 94.5 75.0
All uses (AU)	3-4-5-6 6-7 6-7-8 8-10 3-4-5-9	220 220 350 350 170	92.0 92.0 94.5 94.5 75.0

Developing the Data Flow Tests Cases

- The All du-paths tests are a combination of the
  - All c-use+p (there is a path from every definition to every c-use of that definition. If there is a definition with no c-use following it, then a p-use of the definition is considered.)
  - All p-use+c (for every variable, there is a path from every definition to every p-use of that definition. If there is a definition with no p-use following it, then a c-use of the definition is considered)
- Consideration of the paths for Bill and using the original values for Usage as proposed by the authors (note: no BVs are tested).

Uses	Paths (paper)	CFG paths	Authors choice for Usage	Authors choice for Expected Output
All du -paths (APU+C & ACU+P)	1-2-10	2-4-17	0	$0.00
	3-4-5-6-7	5-7-8-11-13	220	$92.00
	3-4-5-9	5-7-8-9	170	$75.00
	6-7	11-13	220	$92.00
	6-7-8	11-13-14	350	$94.50
	8-10	14-17	350	$94.50
	9-10	9-17	220	$92.00

Developing the Data Flow Test Cases (cont.)

- Consideration of the paths for Usage and using the original values of Usage as proposed by the authors (note: no BVs are tested).

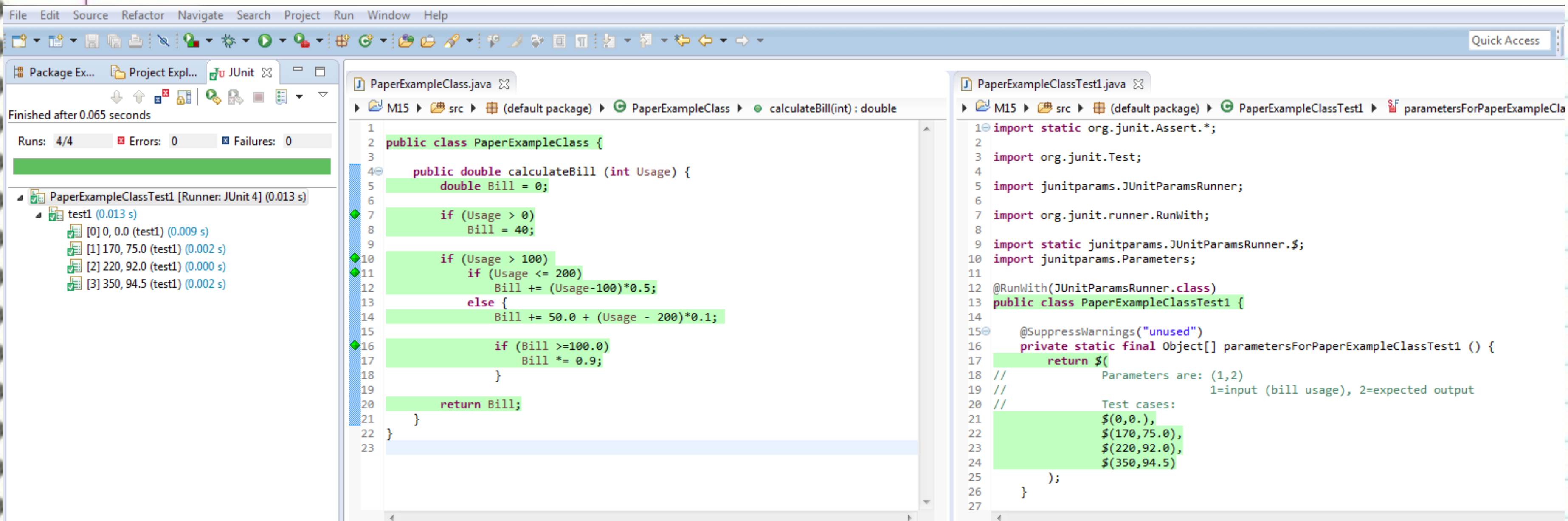
Uses	Paths (paper)	CFG paths	Authors choice for Usage	Authors choice for Expected Output
All du -paths (APU+C & ACU+P)	0-1-2-3-4-5-6	0-2-4-5-7-8-11	220	$92.00
	0-1-2-3-4-5-9	0-2-4-5-7-8-9	170	$75.00
	0-1-2	0-2-4	0	$0.00
	0-1-2-3-4	0-2-4-5-7	170	$75.00
	0-1-2-3-4-5	0-2-4-5-7-8	170	$75.00

- Combine the test cases developed for Bill and Usage into a unique set based on the inputs values and expected outputs (these are the authors test cases)

Authors choice for Usage	Authors choice for Expected Output
0	$0.00
170	$75.00
220	$92.00
350	$94.50

Authors Test Case JUnit & Coverage Results

- The authors tests pass and they achieve full coverage



- Are these good tests? To perform a comparison with our approach we need to complete test case design.

Completing the Basis Path/BV Test Case Design

- We indicated earlier that we need to add three test cases to achieve coverage of the non-uniform ECPs
- We will pick points in the middle of the linear regions at statements 9, 11, and 13

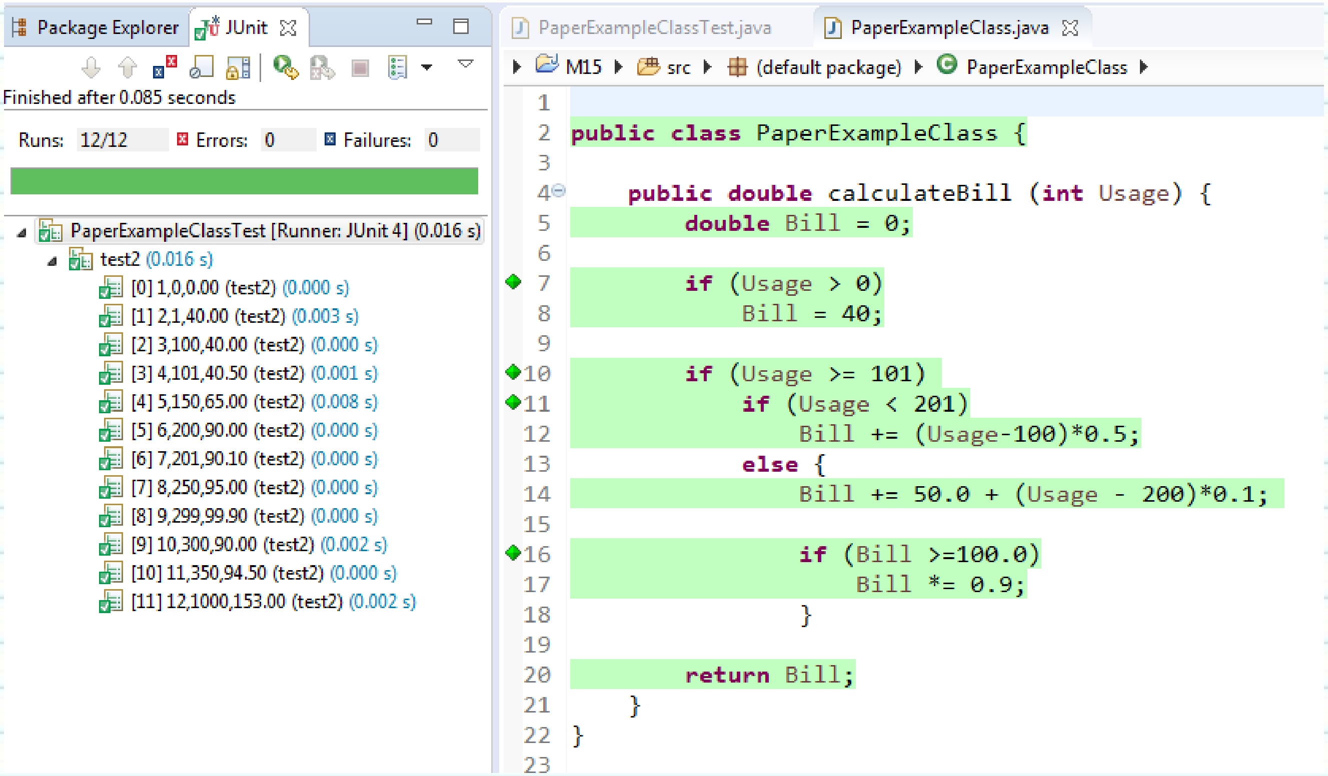
Uses	CFG paths	BV choice for Usage	BV choice for Expected Output
Linear response ECPs	9	150	$65.00
	11	250	$95.00
	13	350	$94.50

- Combining these with the 9 previous test cases the total set is:

Test Case Number	Inputs	Exp Out
	Usage	Bill
1	0	0.00
2	1	40.00
3	100	40.00
4	101	40.50
5	150	65.00
6	200	90.00
7	201	90.10
8	250	95.00
9	299	99.90
10	300	90.00
11	350	94.50
12	1,000	153.00

Completing the Basis Path/BV Test Case Design (cont.)

- Our BP/BV tests pass and achieve full decision level coverage



- But let's compare overall coverage (next slide)

Completing the Basis Path/BV Test Case Design (cont.)

- Here are the two test cases set side-by-side

Authors choice for Usage	Authors choice for Expected Output
0	$0.00
170	$75.00
220	$92.00
350	$94.50

Test Case Number	Inputs	Exp Out
	Usage	Bill
1	0	0.00
2	1	40.00
3	100	40.00
4	101	40.50
5	150	65.00
6	200	90.00
7	201	90.10
8	250	95.00
9	299	99.90
10	300	90.00
11	350	94.50
12	1,000	153.00

- The authors test cases compare to tests 1, 5, 8, and 11
- The authors have never tested any of the 8 boundary values!

Assessing Data Flow Testing

- For this example we are better off using basis path/boundary coverage as our primary tool
- Both approaches would have identified the double writing of data for the variable Bill
  - Data flow testing identified this as a dd-occurence - a potential defect
  - Basis path would have identified this during test cases design
  - Neither uncovered any failure in the software (there was none)
- For this example, data flow testing <u>did not</u> provide additional insight and tools over basis path/boundary value testing,
  - it may be a good secondary tool useful for detecting additional defects. Many software test tools provide static and dynamic dataflow analysis as part of their test suite
  - it is most definitely not a primary testing tool - test cases miss 8 boundary values!
- UTA is doing some promising leading edge research with dynamic data flow testing but this is beyond the scope of this class

Slice Based Testing

- Program slicing is a method used for abstracting behavioral information from programs.
- Start with a subset of a program’s behavior and reduce it to a minimal form that still produces that behavior. The reduced program is called a “slice”.

```
public int sliceMethod (int num) {
int i, product=0, sum = 0;

for(i = 0; i <= num; i++)
  sum += i;
product=sum*num;

return product; }
```

Program slice for variable i

```
public int sliceMethod (int num) {
int i, product=0, sum = 0;

for(i = 0; i <= num; i++)
  sum += i;
}
```

- Static slices are computed statically using a dependence graph.
- Dynamic slices are created from data dependency information is traversed to compute the slices during an execution trace of the program - therefore, it is a set of statements that did affect the value of a variable v given a specific input.

Slice Testing Definitions

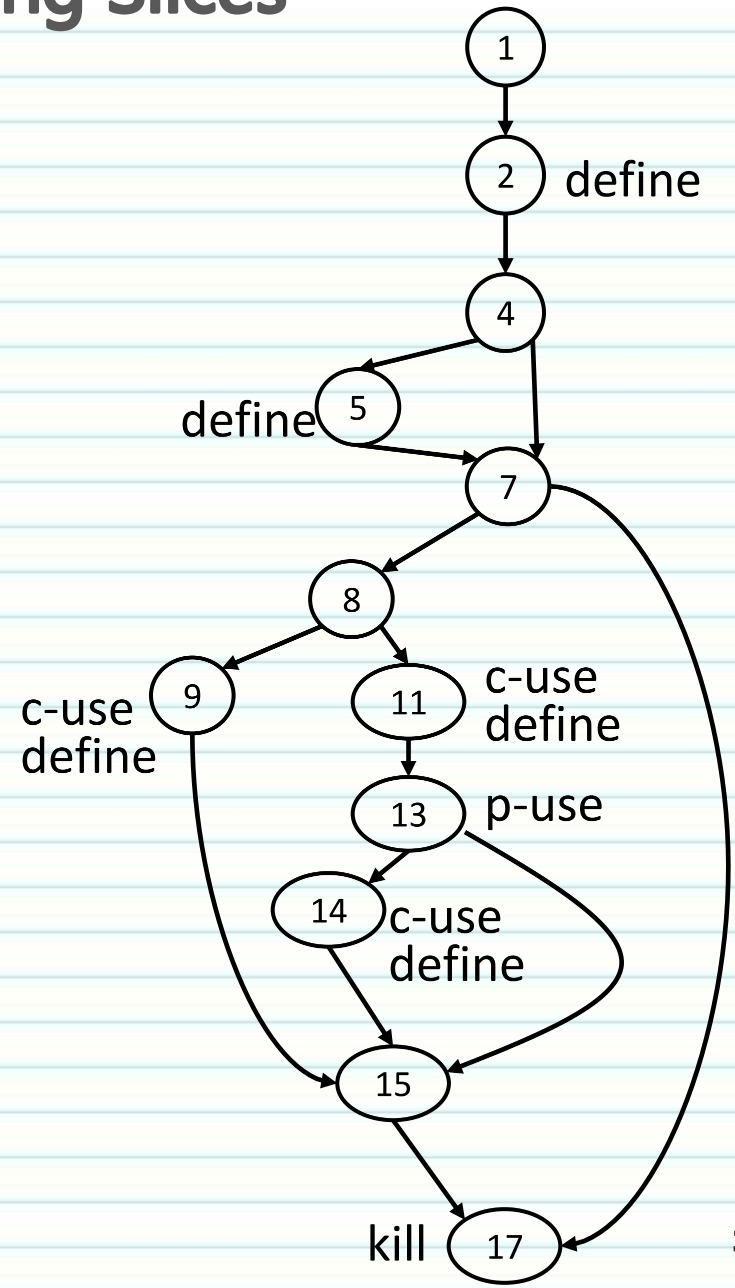
- A slice on the variable set V at statement fragment n, written as C<n, V>, is the set of nodes that affect the variables in V at node n.
  - We can simplify this to a single variable v, such that a slice is C<n, v>.
  - For multiple variables we can take the union of slices on a single variable

- A backward slice, is the set of nodes that contribute to the values of v at node n. For a single variable v (e.g., Bill) this is determined by the def(s) and use(s) of v (Bill).

- Examples of testing slices for the example program are shown on the next slide.

Example Testing Slices

```
1         public static double calculateBill (int Usage) {
2                   double Bill = 0;
3
4                   if (Usage > 0)
5                             Bill = 40;
6
7                   if (Usage > 100) {
8                             if (Usage <= 200)
9                                       Bill += (Usage-100)*0.5;
10                            else {
11                                      Bill += 50.0 + (Usage - 200)*0.1; }
12
13                  if (Bill >=100.0)
14                            Bill *= 0.9;
15                            }
16
17                  return Bill;
18        }
```



C<2,Bill>={2}
C<11,Bill>={2,5,11}
C<17,Bill>={2,5,9,11,13,14,17}