



SENG 438

Software Testing, Reliability & Quality

Chapter 5B: White-box Testing

Data-flow Coverage & Coverage Tools

Department of Electrical & Computer Engineering, University of Calgary

B.H. Far (far@ucalgary.ca)

<http://people.ucalgary.ca/~far>



Contents

- Preliminary
- Control flow
 - Statement/Node/Line Coverage
 - Decision/Edge/Branch Coverage
 - Condition Coverage
 - Path Coverage
- Data flow
- Coverage tools



Our focus in this Chapter





What We've Learnt So Far?

- Statement/Decision/Condition/Path coverage → using control flow graph
- Acceptance criteria:
 - Statement, decision, basic and compound condition adequacy, MC/DC, path adequacy
- **Deficiencies:**
 - 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
- Need to distinguish “*important*” paths



UNIVERSITY OF
CALGARY

Section 4



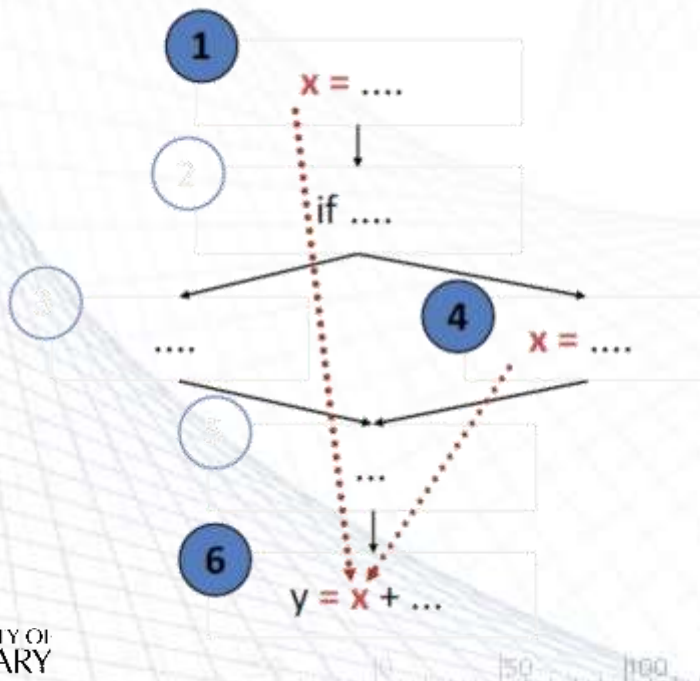
Data Flow Coverage

**How to enhance a test suite by finding
“meaningful” paths to test?**



Motivation

- **Intuition:** Statements interact through *data flow*
 - Value computed in one statement, used in another
 - Bad value computation revealed only when it is used

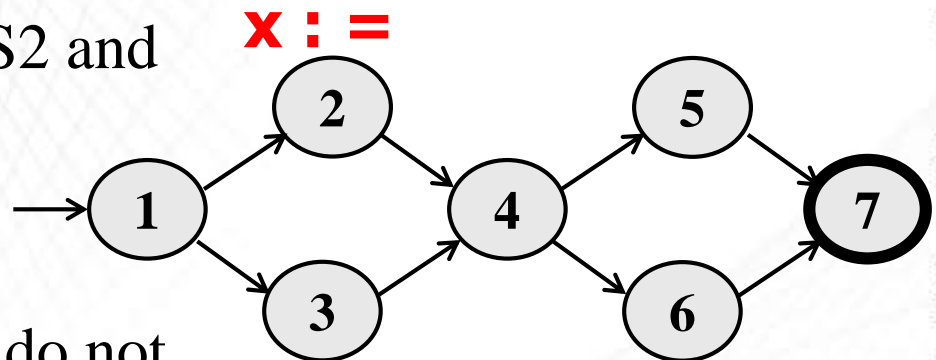


- Value of x at 6 could be computed at 1 or at 4
- Bad computation at 1 or 4 could be revealed only if they are used at 6
- (1,6) and (4,6) are *def-use (DU) pairs*
 - defs at 1,4
 - use at 6



Data Flow: What is?

- Suppose we define x in S2 and have ref to it in S6



- The two branches tested do not exercise the relationship between the definition of x in S2 and the use of x in S6

Branches tested:

1, 2, 4, 5, 7

1, 3, 4, 6, 7

- Does this cause a problem?

We need to know where the value for a given variable is defined and where it is used (correctly)



Data-flow-based Testing

- **Basic idea:** test the connections between variable **definitions** (“write”) and variable **uses** (“read”)
- Starting point: **variation of** the control flow graph annotated with location of defined and used variables
 - Set **def (n)** : contains variables that are defined at node n (i.e., they are written)
 - Set **use (n)** : variables that are read



Example 1

- L7,L8,L9: define objects and variables
- L16 variable i is both defined and used
- L21 day defined
- L22 and L25 day used
- Etc.

There is a possibility in some programs that a “used” value is modified somewhere on the way between “def” and “use” and that may cause a program failure

```
01. import java.util.*;
02. public class CalendarTest
03. {
04.     public static void main(String[] args)
05.     {
06.         // construct d as current date
07.         GregorianCalendar d = new GregorianCalendar();
08.         int today = d.get(Calendar.DAY_OF_MONTH);
09.         int month = d.get(Calendar.MONTH);
10.         // set d to start date of the month
11.         d.set(Calendar.DAY_OF_MONTH, 1);
12.         int weekday = d.get(Calendar.DAY_OF_WEEK);
13.         // print heading
14.         System.out.println("Sun Mon Tue Wed Thu Fri Sat");
15.         // indent first line of calendar
16.         for (int i = Calendar.SUNDAY; i < weekday; i++)
17.             System.out.print(" ");
18.         do
19.         {
20.             // print day
21.             int day = d.get(Calendar.DAY_OF_MONTH);
22.             if (day < 10) System.out.print(" ");
23.             System.out.print(day);
24.             // mark current day with *
25.             if (day == today)
26.                 System.out.print("* ");
27.             else
28.                 System.out.print(" ");
29.             // start a new line after every Saturday
30.             if (weekday == Calendar.SATURDAY)
31.                 System.out.println();
32.             // advance d to the next day
33.             d.add(Calendar.DAY_OF_MONTH, 1);
34.             weekday = d.get(Calendar.DAY_OF_WEEK);
35.         }
36.         while (d.get(Calendar.MONTH) == month);
37.         // the loop exits when d is day 1 of the next month
38.         // print final end of line if necessary
39.         if (weekday != Calendar.SUNDAY)
40.             System.out.println();
41.     }
42. }
```




Need for Data Flow Testing

- Let's see the need for data-flow-based testing

```
1: int calculate(int x, int y)
2: {
3:     double z=0;
4:     if (x<>0)        // x neq 0
5:         z=z+y;
6:     else
7:         z=z-y;
8:     if (y<>0)        // y neq 0
9:         z=z/x;
10:    else
11:        z=z*x;
12:    return z;
13: }
```

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

- Here is a MC/DC-adequate test set
 - i.e., coverage=100%
- **Exercise:** Verify that it is MC/DC-adequate

Note where the value for x is defined and then used may belong to different paths not traversed by the test suite



Need for Data Flow Testing

- The given test suite does not reveal the fault (the chance for division by zero)
- Neither of the two test cases forces the use of x at L9, defined on L1, and with value of $x=0$
- In larger programs this will be more problematic

```
1: int calculate(int x, int y)
2: {
3:     double z=0;
4:     if (x<>0)           // x neq 0
5:         z=z+y;
6:     else
7:         z=z-y;
8:     if (y<>0)           // y neq 0
9:         z=z/x;          ← Use x
10:    else
11:        z=z*x;           ← Use x
12:    return z;
13: }
```

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



Need for Data Flow Testing

- To do so, we need a test case that causes condition at L8 to be true, and $x=0$
- An MC/DC-adequate test does not force the execution of this path with $x=0$ and hence the “divide by zero” defect will not be detected
- Again, consider large systems with $>$ thousands LOC

```
1: int calculate(int x, int y)
2: {
3:     double z=0;
4:     if (x<>0)           // x neq 0
5:         z=z+y;
6:     else
7:         z=z-y;
8:     if (y<>0)           // y neq 0
9:         z=z/x;         ← Use x
10:    else
11:        z=z*x;         ← Use x
12:    return z;
13: }
```

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

Therefore, %100 MC/DC coverage cannot guarantee that the program dose not fail



Data Flow Analysis - Definition

- In **Data Flow Analysis**, we focus on paths that are significant for the data flow in the program
- Focus of testing is on the assignment of values to objects/variables and their uses
- Analysis of occurrences of variables:
 - **Definition occurrence**: a value is written (bound) to a variable
 - **Use occurrence**: value of a variable is **read** (referred)
 - **Predicate use (p-use)**: a variable is used to decide whether a predicate **evaluates** to true or false
 - **Computational use (c-use)**: **compute** a value for defining other variables or output values



Data Flow Analysis - Example

- Let's see an example

```
1. public int factorial(int x) {  
2.     int i, result = 1;  
3.     for (i=2; i<=x; i++) {  
4.         result = result * i;  
5.     }  
6.     return result;  
7. }
```

Def-Use Table



Variable	Def-line	Use-line



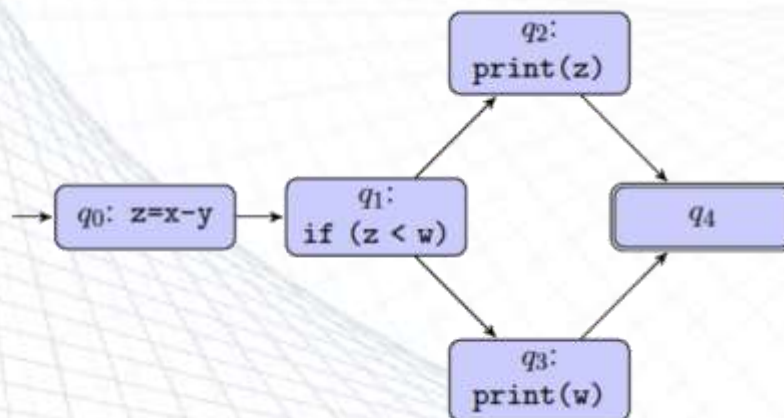
Def and Use

- **Goal:** Try to ensure that values of the declared variables are assigned and used correctly
- **def:** a location where a value for a variable is stored
- **use:** a location where a variable's value is accessed (read – evaluate – compute)
- **def(n):** The set of variable defined by node n
- **use(n):** the set of variable used by node n
 - We usually differentiate between computational (c-use) and predicate (p-use) type



Def and Use Path

- A **definition-clear path (def-clear)** p with respect to x is a sub-path where x is not defined at any of the nodes in p
- A **du-path** is a simple path where the initial node of the path is the only defining node of x in the path
- **Reach**: if there is a def-clear path from the nodes m to p with respect to x , then the def of x at m reaches the use at p



Example for z :

- du-path: q_0, q_1, q_2
- def-clear: q_1, q_2, q_4
- q_0 reaches q_2



Def and Use Coverage

- **All-Defs-Coverage (ADC):**
 - Some def-clear sub-path from **each** definition to **some** use reached by that definition



$x = \dots$ is a *definition* of x
 $= x \dots$ is a *use* of x

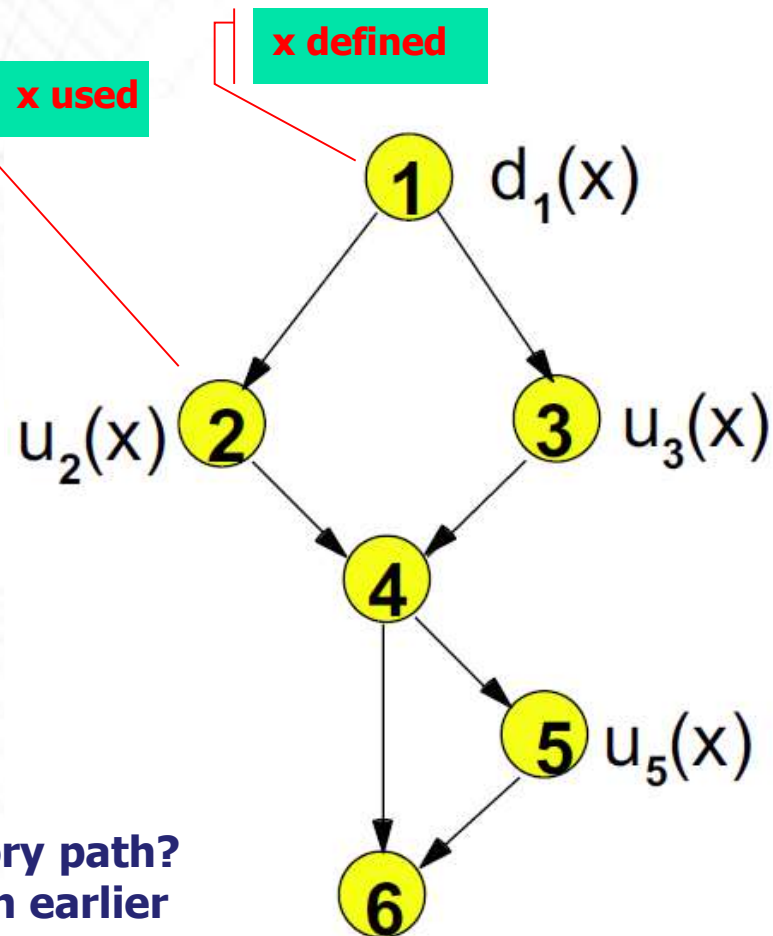


All-Defs Coverage Example

- Requires:
 $d_1(x)$ to a use

- Satisfactory Path:
[1, 2, 4, 6]
[1, 3, 4, 6]

Q. Why 5 is not part of ADC satisfactory path?
A. Because value of x is overwritten in earlier nodes 2 or 3

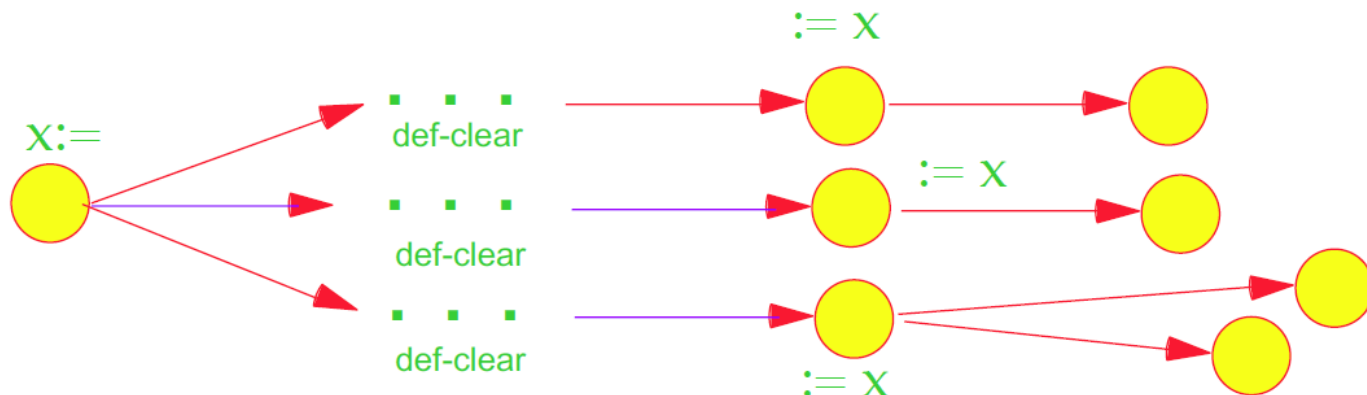




Def and Use Coverage

■ All-Uses Coverage (AUC)

- **Some** definition-clear subpath from **each** definition to **each** use reached by that definition and each successor node of the use





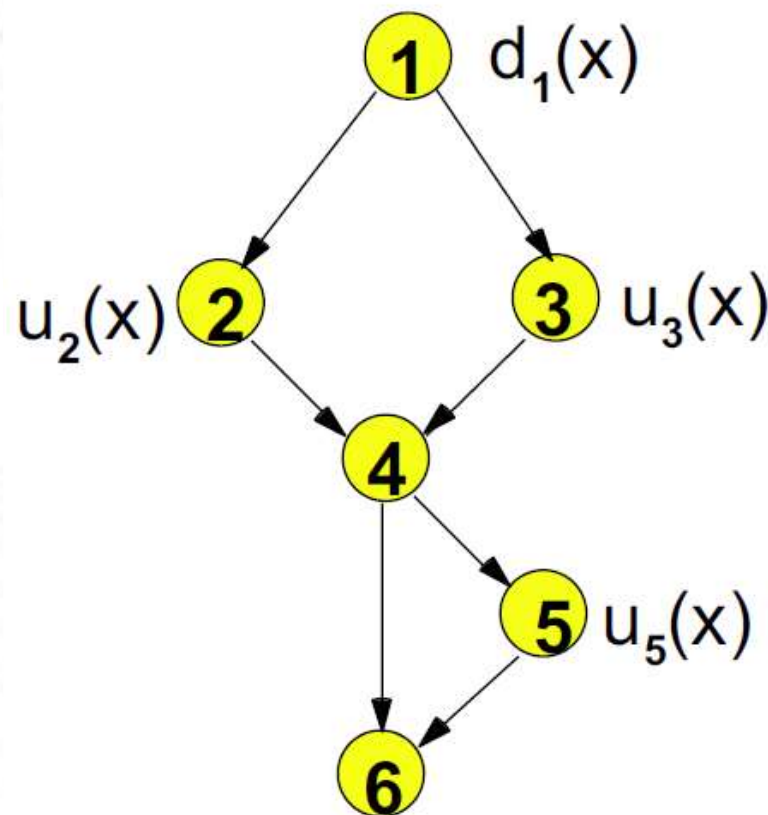
All-Uses Coverage Example

- Requires:

- $d_1(x)$ to $u_2(x)$
- $d_1(x)$ to $u_3(x)$
- $d_1(x)$ to $u_5(x)$

- Satisfactory Path:

- $[1, 2, 4, 5, 6]$
- $[1, 3, 4, 6]$

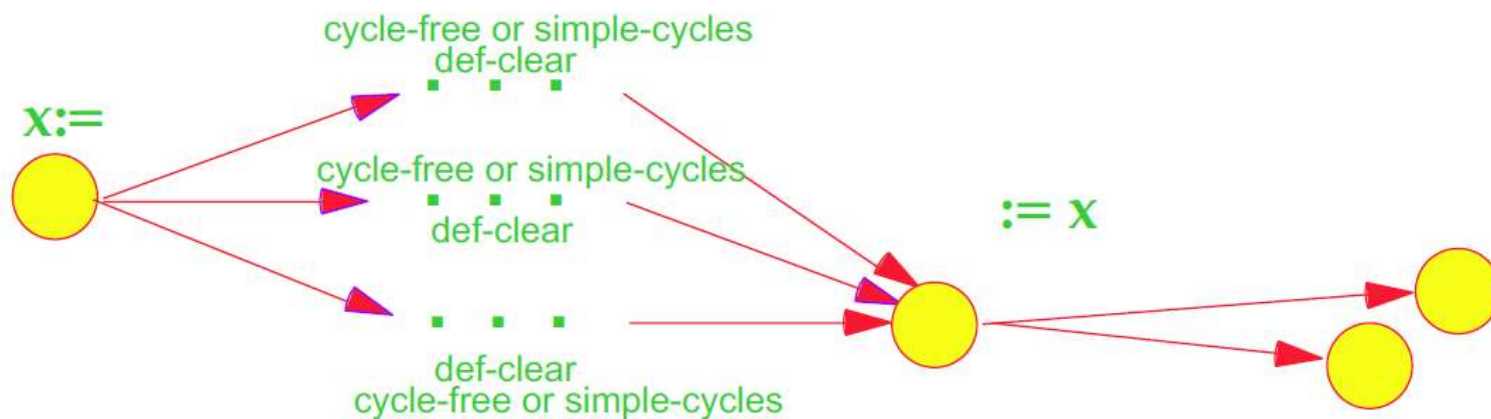




Def and Use Coverage

■ All-Du-Paths Coverage (ADUPC):

- All def-clear sub-paths that are cycle-free or simple-cycles from **each definition** to **each use** reached by that definition and each successor node of the use





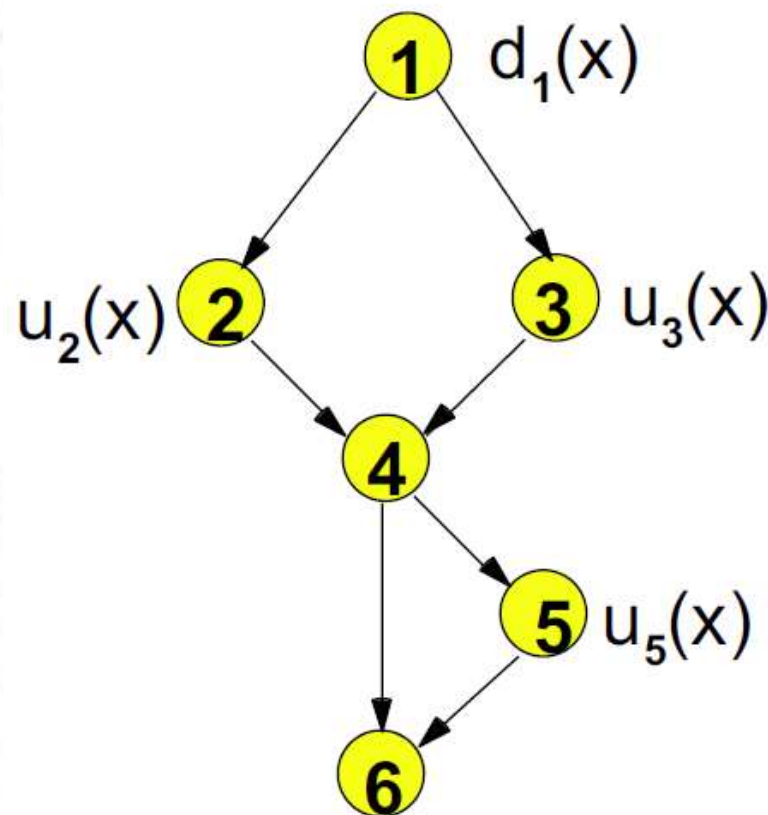
All-Du-Path Coverage Example

- Requires:

- All $d_1(x)$ to $u_2(x)$: $[1,2]$
- All $d_1(x)$ to $u_3(x)$: $[1,3]$
- All $d_1(x)$ to $u_5(x)$:
 $[1,2,4,5]$, $[1,3,4,5]$

- Satisfactory Path:

- $[1, 2, 4, 5, 6]$
- $[1, 3, 4, 5, 6]$

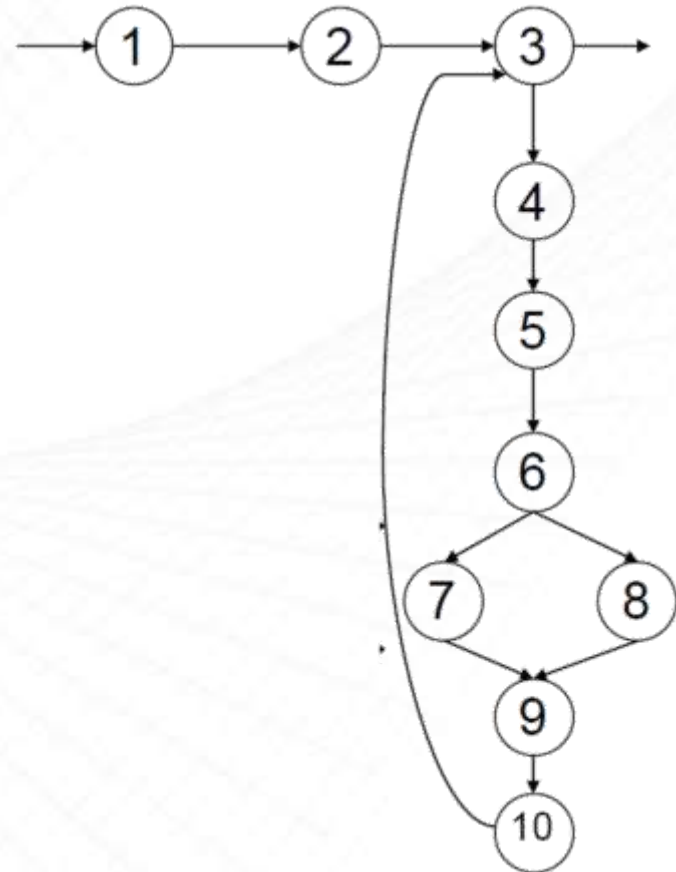




Exercise: Def and Use

- Assume **y** is already initialized

```
1:  s := 0;  
2:  x := 0;  
3:  while (x < y) {  
4:    x := x + 3;  
5:    y := y + 2;  
6:    if (x + y < 10)  
7:      s := s + x + y;  
8:      else  
9:        s := s + x - y;  
10:   endif  
10: }
```





Exercise: Reach

- A definition of variable s at node n_1 reaches node n_2 if and only if there is a path between n_1 and n_2 that does not contain a definition of s

Reaches
nodes
2,3,4,5,6,7,8,
but not 9 and
10.

DEF(1) := {s}, USE(1) := \emptyset
DEF(2) := {x}, USE(2) := \emptyset
DEF(3) := \emptyset , USE(3) := {x,y}
DEF(4) := {x}, USE(4) := {x}
DEF(5) := {y}, USE(5) := {y}
DEF(6) := \emptyset , USE(6) := {x,y}
DEF(7) := {s}, USE(7) := {s,x,y}

DEF(8) := {s}, USE(8) := {s,x,y}
DEF(9) := \emptyset , USE(9) := \emptyset
DEF(10) := \emptyset , USE(10) := \emptyset

```
1:  s := 0;  
2:  x := 0;  
3:  while (x < y) {  
4:    x := x + 3;  
5:    y := y + 2;  
6:      if (x + y < 10)  
7:        s := s + x + y;  
      else  
8:        s := s + x - y;  
9:      endif  
10: }
```

This is called a Def-Use table



Exercise: Def and Use Pairs

Reaches nodes 2, 3, 4, 5, 6, 7, 8, but not 9,10

For this definition, two DU pairs: 1-7, 1-8

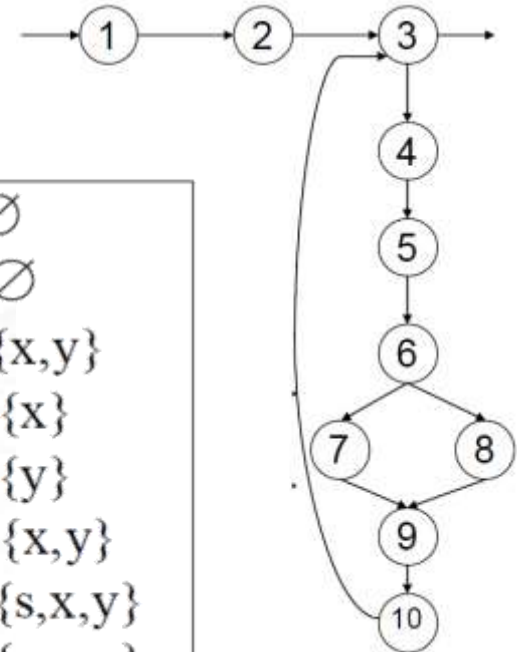
DEF(1) := {s}, USE(1) := \emptyset
DEF(2) := {x}, USE(2) := \emptyset
DEF(3) := \emptyset , USE(3) := {x,y}
DEF(4) := {x}, USE(4) := {x}
DEF(5) := {y}, USE(5) := {y}
DEF(6) := \emptyset , USE(6) := {x,y}
DEF(7) := {s}, USE(7) := {s,x,y}
DEF(8) := {s}, USE(8) := {s,x,y}

DU pair tests:

TC1: x=0, y=3 (for L7)

TC2: x=0, y=6 (for L8)

This is called a CFG



This is called a Def-Use table



Data Flow Based Testing

Adequacy Criteria

- **All DU pairs:** Each DU pair is exercised by at least one test case
- **All DU paths:** Each *simple* (non looping) DU path is exercised by at least one test case
 - Remember that for each DU pair there can be several DU paths
- **All definitions:** For each definition, there is at least one test case which exercises a DU pair containing it
 - (Every defined value is used somewhere)
- Corresponding coverage fractions can also be defined

All DU paths > All DU pairs > All definitions



Data Flow Based Testing

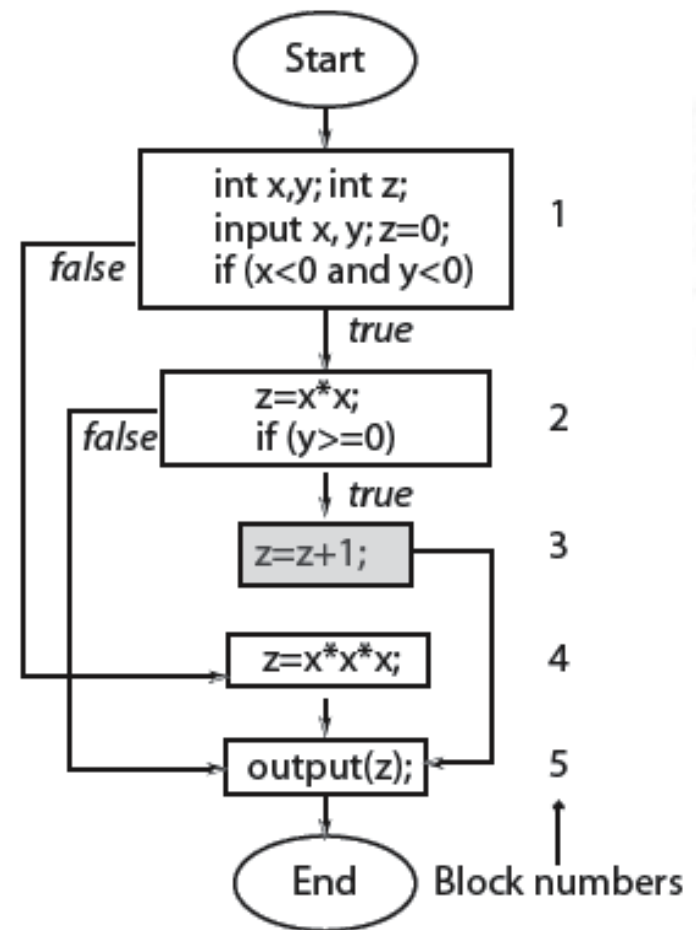
- How to enhance test suite using Data Flow based testing?
 - If a variable is defined at node-m and used at node-n, then the path between node m and n should be tested
 - If a variable is modified (c-use) within the path between m and n, say in node-q, then we should have separate tests for
 - m to q (called def-coverage)
 - q to n and m to n (called use-coverage)
 - The def+use coverage is combining both

You can use the program itself, DFD and Def-Use table for this purpose



Data Flow Graph (DFG): Example

- Let's derive the DFG for this given flowchart
- First the def-use table
- Then, the data flow graph (DFG)

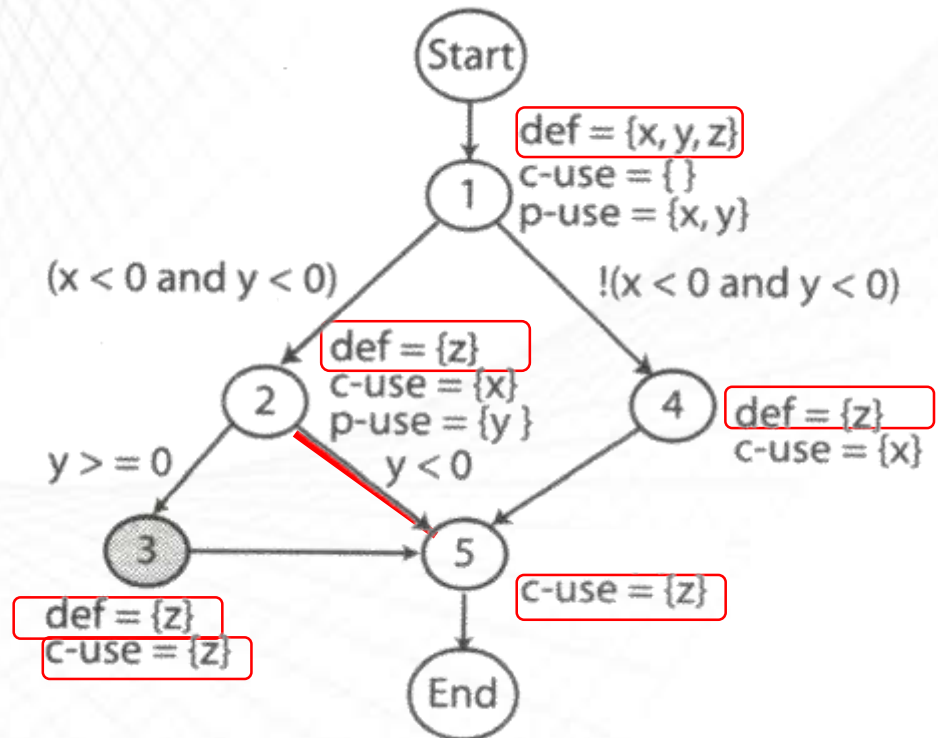




Definition-clear (def-clear) Path

- Any path starting from a node at which a variable is defined and ending at a node at which that variable is used, without redefining the variable anywhere else along the path, is a def-clear path for that variable

This is called a DFG



Therefore, any values set for z at L1 is redefined at L2, L3, L4 and L5.

But anything defined at L2 or L4, is still alive at L5

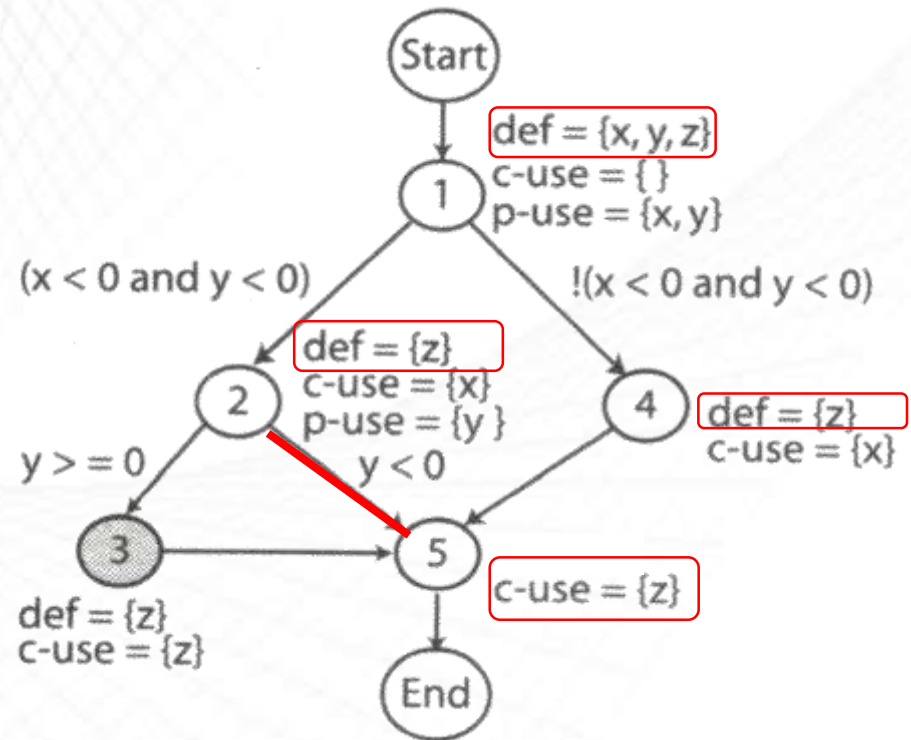
c-use: computational use
p-use: predicate use



Definition-clear (def-clear) Path

Example:

- 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 (since it is redefined at node 2)
- The definition of z at node 2 is live at node 5,
- ... while the definition of z at node 1 is not live at node 5



c-use: computational use
p-use: predicate use



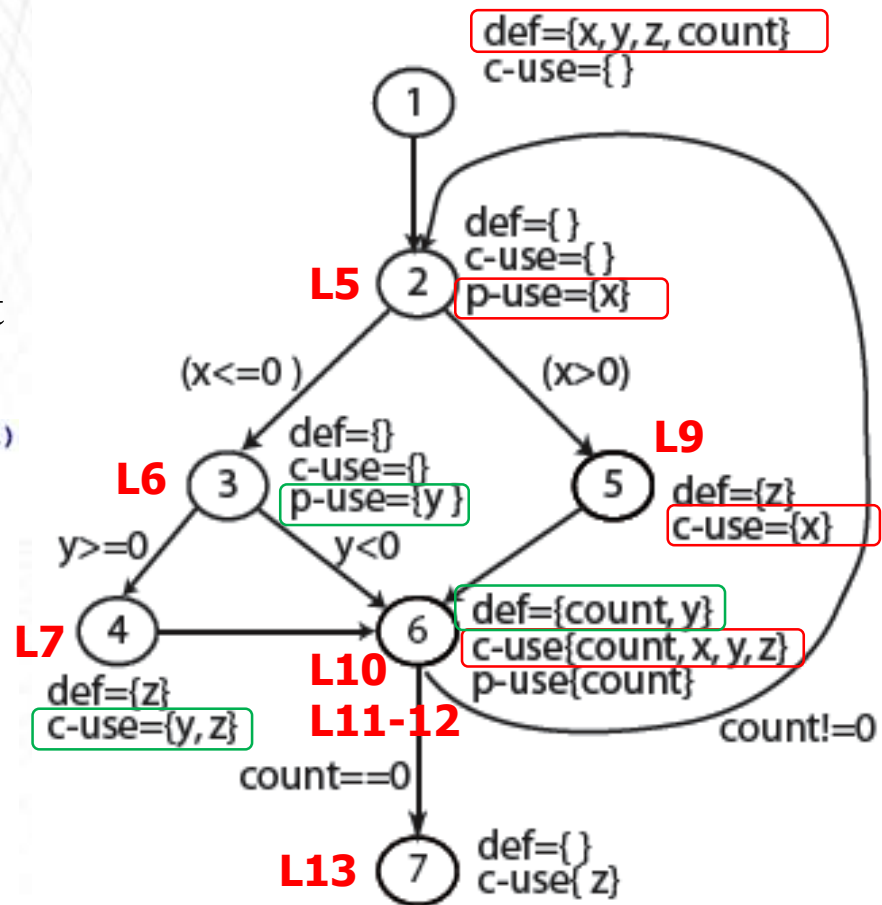
Example 2: Def-clear Path

- Draw DFG for the program given below
- Find the def-clear paths for x and y and z
- Which definitions are live at node 4?

```

1  double calculate(int x, int y, int count)
2  {
3      double z=0.0;
4      do {
5          if (x<=0)
6              if (y>=0)
7                  z=y*z+1;
8          else
9              z=1/x;
10         y=(int) (x*y+z);
11         count--;
12     } while (count > 0);
13     return z;
14 }

```





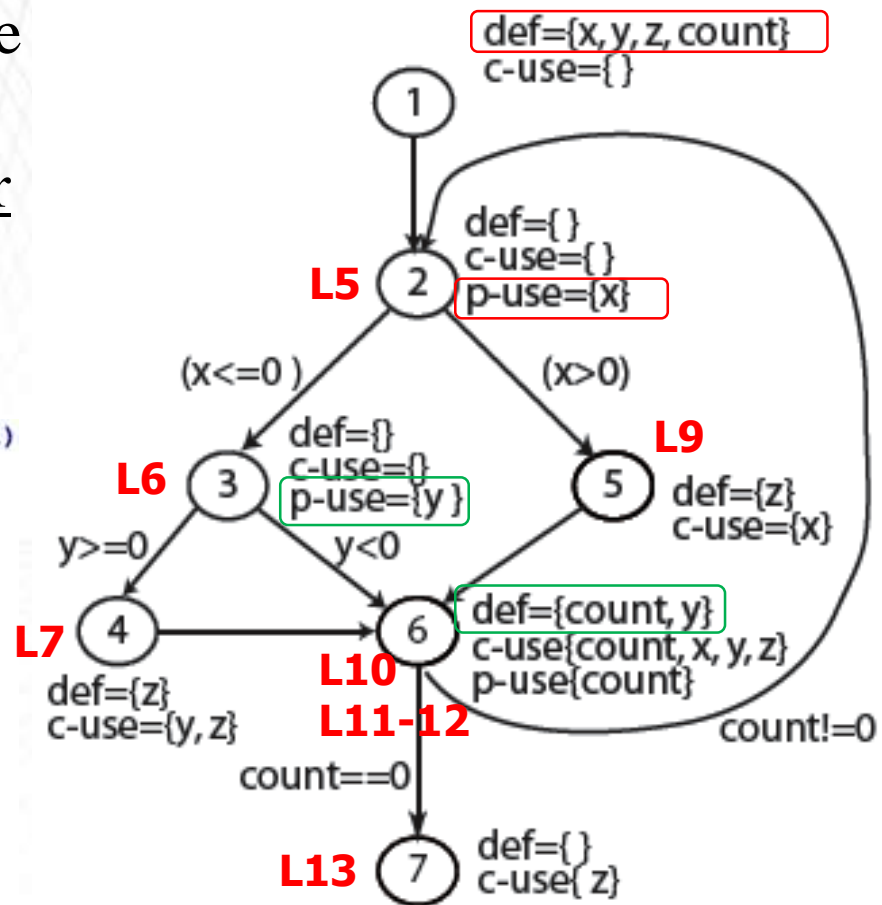
Example 2: Def-use Pairs

- Definitions of a variable L_{11} and its use at L_{12} constitute a def-use pair
- (L_{11} and L_{12} can be the same, e.g., $y=++x$;)

```

1  double calculate(int x, int y, int count)
2  {
3      double z=0.0;
4      do {
5          if (x<=0)
6              if (y>=0)
7                  z=y*z+1;
8          else
9              z=1/x;
10         y=(int) (x*y+z);
11         count--;
12     } while (count > 0);
13     return z;
14 }

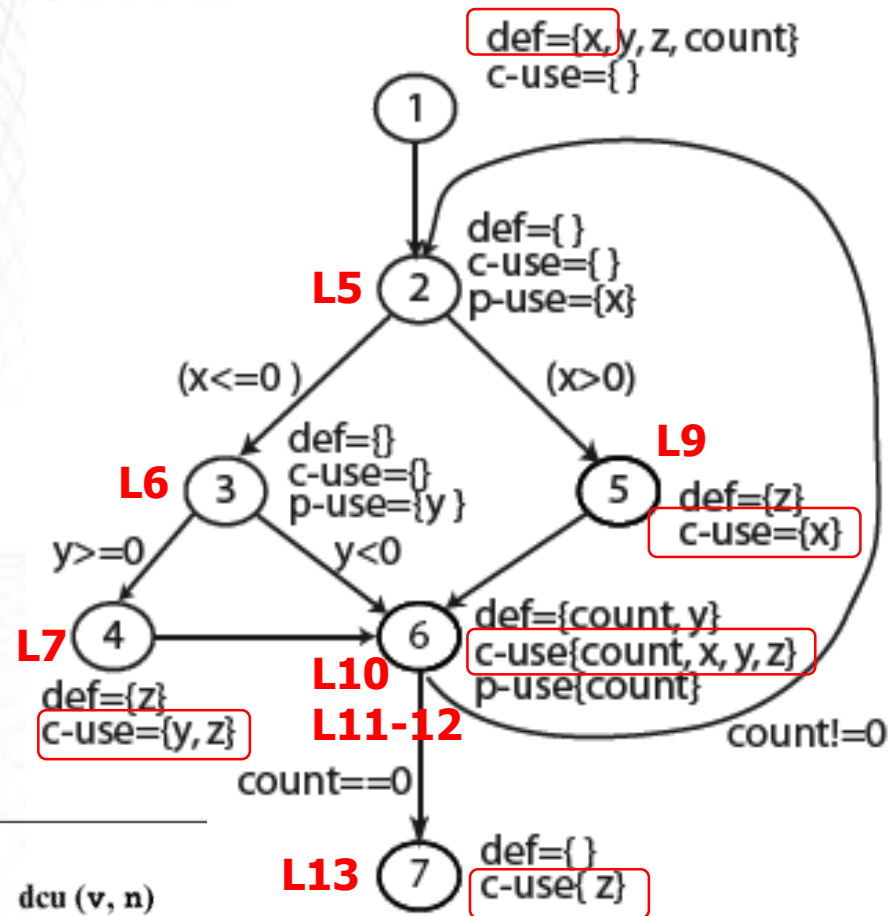
```





Example 2: Def-use Measure

- Definition c-use pairs $dcu(di(x))$:**
denotes the set of all nodes using variable x from the live definitions of the variable at a given previous node i , $di(x)$



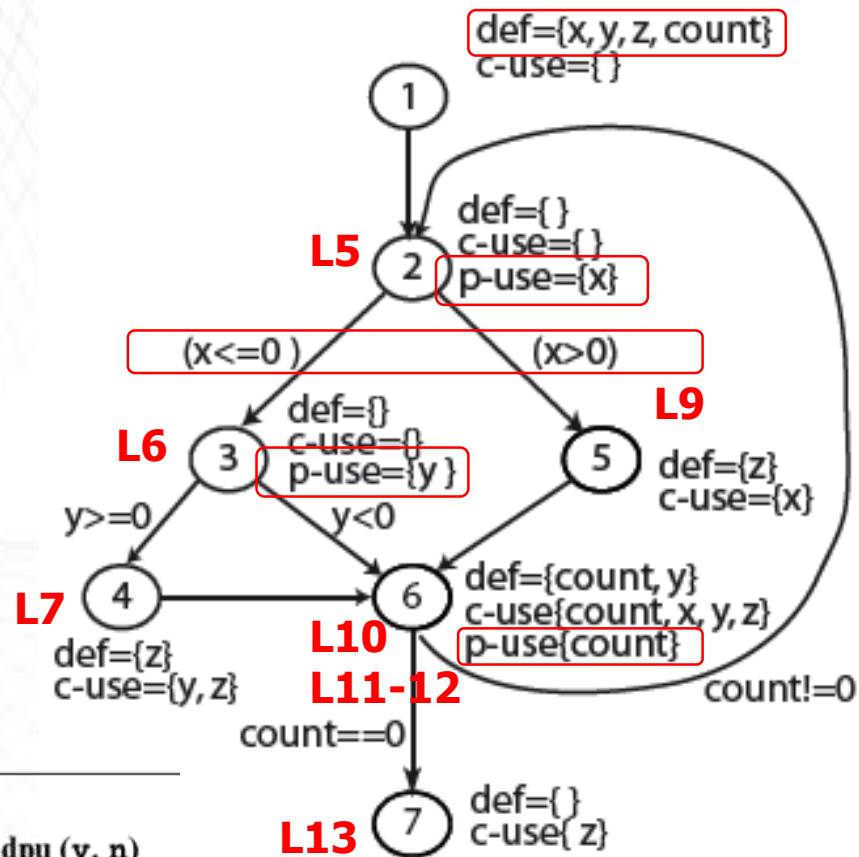
Variable (v)	Defined at node (n)	$dcu(v, n)$
x	1	{5, 6}

Therefore, tests for (1-5), (1-6) and (5-6) should be added



Example 2: Def-use Measure

- Definition p-use pairs $dpu(di(x))$:**
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 used at node k



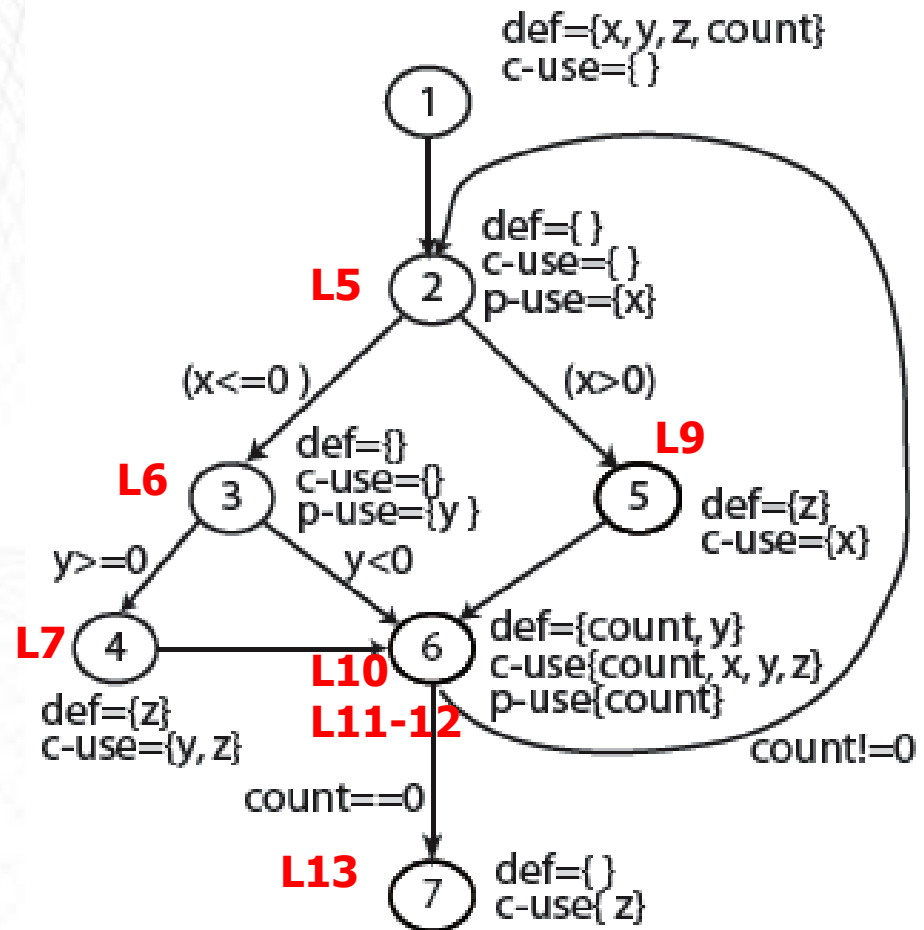
Variable (v)	Defined at node (n)	dcu (v, n)	dpu (v, n)
x	1	{5, 6}	{(2, 3), (2, 5)}

Therefore, tests for (2-3) and (2-5) should be added



Example 2: Def-use Measure

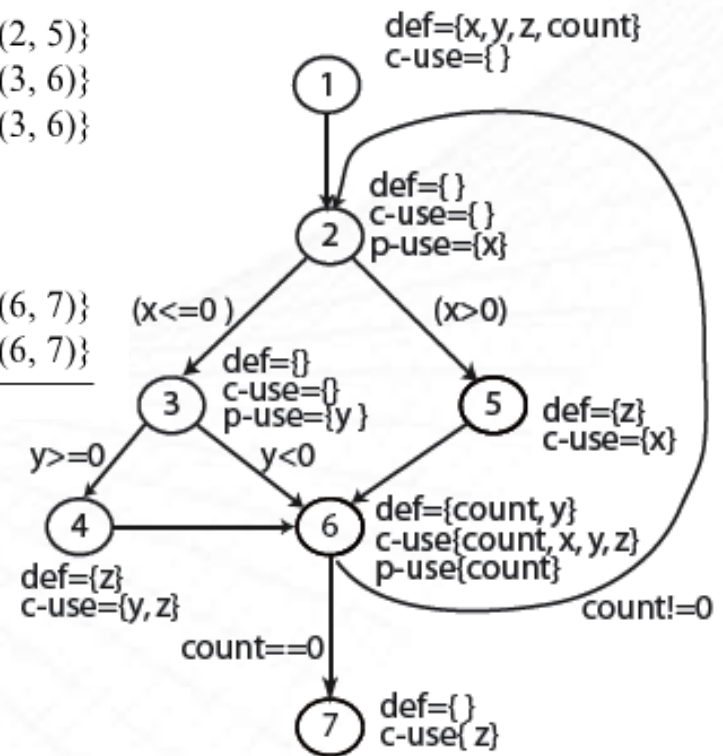
- Let's derive:
 - $dn(v)$
 - $dcu(v, n)$
 - $dpu(v, n)$





Example 2: Answer

Variable (v)	Defined at node (n)	dcu (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)}





Data-flow Coverage Measure

- To calculate data-flow test coverage, we should measure how many c-uses and p-uses we have in total

$$CU = \sum_{i=1}^n \sum_{\forall n} |dcu(v_i, n)|$$

$$PU = \sum_{i=1}^n \sum_{\forall n} |dpu(v_i, n)|$$

Loop for variables

Loop for all nodes defining a variable

- CU: total number of c-uses in a program
- PU: total number of p-uses
- Given a total of n variables $v_1, v_2 \dots v_n$



Data-flow Coverage: Example

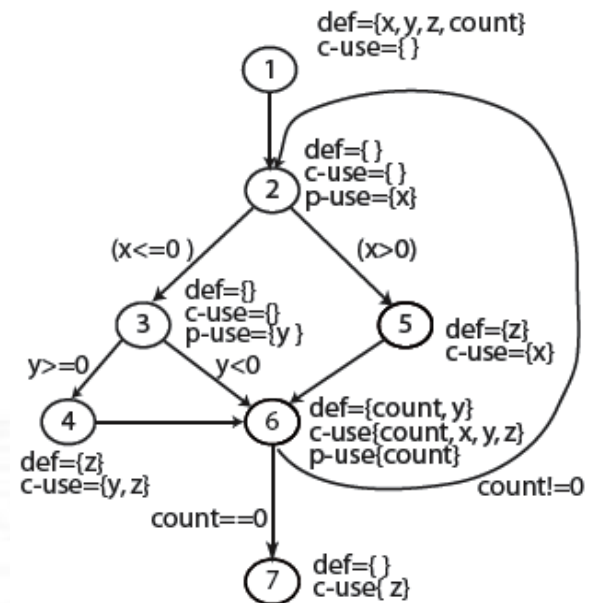
- Calculate CU and PU

$$CU = \sum_{i=1}^n \sum_{\forall n} |dcu(v_i, n)|$$

$$PU = \sum_{i=1}^n \sum_{\forall n} |dpu(v_i, n)|$$

Variable (v)	Defined in node (n)	dcu (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}	
count	6	{6}	{(6, 2), (6, 7)}

Total: 17 8





All-uses Coverage Measure

- All-uses coverage is computed as

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

- Where CU is the total c-uses, CU_c is the number of c-uses covered, PU_c is the number of p-uses covered, CU_f the number of infeasible c-uses and PU_f the number of infeasible p-uses
- It is considered adequate with respect to the all-uses coverage criterion if the c-use coverage is % 100



All-uses Coverage – Example

Exercise: For the following DFG, show by analysis whether $T=\{TC1, TC2\}$ is adequate w.r.t. to all-uses coverage. Calculate all-uses coverage ratio. If coverage ratio is not 100%, what def-use pairs need to be covered, and what test cases should be added to cover them?

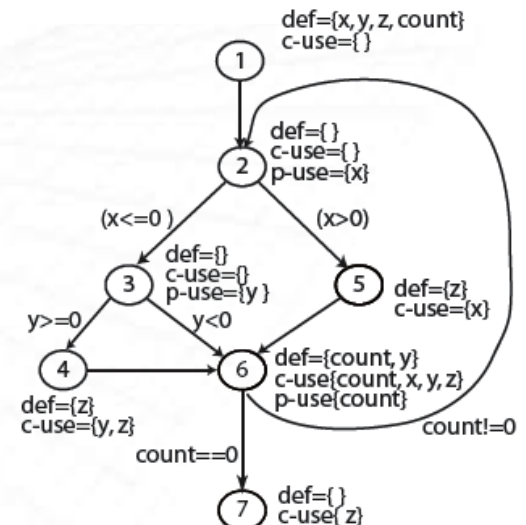
- TC1: $\langle x=5, y=-1, count=1 \rangle$
- TC2: $\langle x=-2, y=-1, count=3 \rangle$

Variable (v)	Defined in node (n)	dcu (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}	{}
count	6	{6}	{{(6, 2), (6, 7)}

```

1 double calculate(int x, int y, int count)
2 {
3     double z=0.0;
4     do {
5         if (x<=0)
6             if (y>=0)
7                 z=y*z+1;
8         else
9             z=1/x;
10        y=(int) (x*y+z);
11        count--;
12    } while (count > 0);
13    return z;
14 }

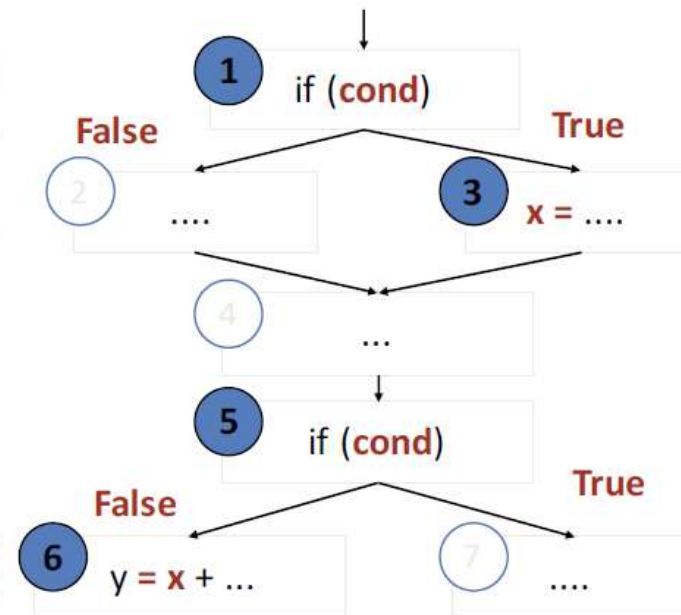
```





Infeasibility

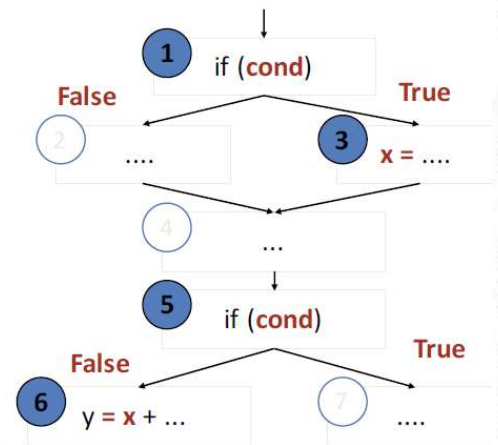
- Suppose *cond* has not changed between 1 and 5
 - Or the conditions could be different, but the first implies the second
- Then (3,6) is not a (feasible) DU pair
 - But it is difficult or impossible to determine which pairs are infeasible
- Infeasible test obligations are a problem
 - No test case can cover them





Infeasibility

- The path-oriented nature of data flow analysis makes the infeasibility problem especially relevant
 - Combinations of elements matter!
 - Impossible to (precisely) distinguish feasible from infeasible paths
 - More paths = more work to check manually.
- In practice, reasonable coverage is often, not always achievable
 - Number of paths is linear (often) or exponential (worst case)
- All DU *paths* is more often impractical





Summary - Data Flow

- Data flow testing attempts to distinguish “important” paths: Interactions between statements
 - Intermediate between simple statement and branch coverage and more expensive path-based structural testing
- Cover Def-Use (DU) pairs: From computation of value to its use
 - Intuition: Bad computed value is revealed only when it is used
 - Levels: All DU pairs, all DU paths, all defs (some use)
- Limits: Aliases, infeasible paths
 - Worst case is bad (undecidable properties, exponential blow up of paths), so pragmatic compromises are required



Coverage Tools





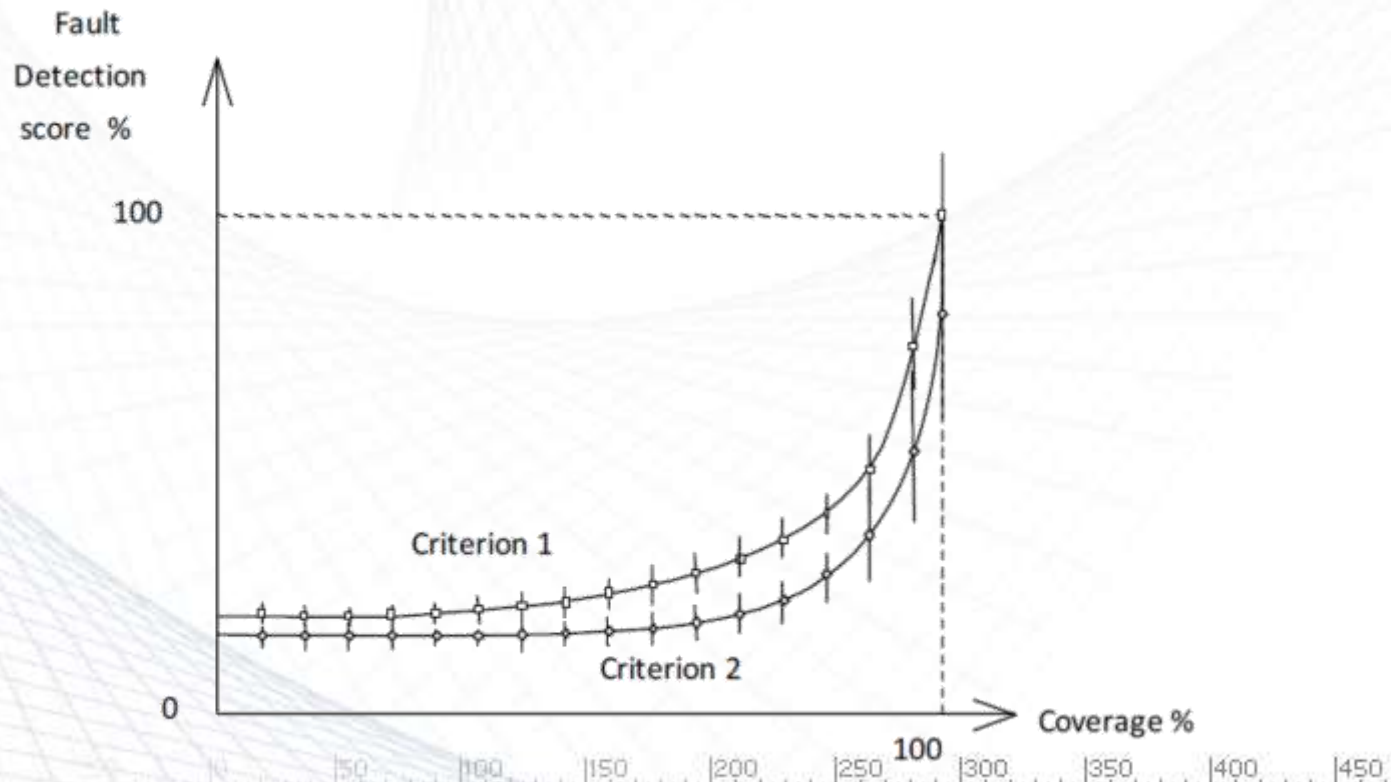
Measuring Test Coverage

- One advantage of coverage criteria is that it can be measured *automatically*
 - To control testing progress
 - To assess testing completeness in terms of remaining faults and reliability
- High coverage is not a guarantee of fault-free software, just an element of information to increase our confidence → statistical models



Analysis of Coverage Data

- Regardless of criterion used for WB testing, only higher coverage rates contribute to finding more faults ← e.g. try at least 60-80% coverage to find reasonable number of faults





Test Coverage: Benefits

- If you haven't exercised part of the code, it could contain a bug
- Any coverage criteria performs better than random test selection – especially DU-coverage
- Helps us in measuring how much unit testing has been done and how much is left
- Significant improvements occurred as coverage increased from 90% to 100%
- Helps us in test case design (deriving test cases)
- We can prioritize the most critical modules (units) of the system and target higher coverage for those critical modules, would mean more rigorous testing for them



Test Coverage: Caveats

- Coverage tests that the code has been exercised (mostly in unit testing level), and that each unit has been verified
- But it does not tell us that we have built what the customer wanted
- If the logical structure of the code contains errors such as a missed case in a switch statement, the absence of code may not be detected
- 100% coverage alone is not a reliable indicator of the effectiveness of a test set – especially statement coverage
- Yet, 100% coverage is often unrealistic, especially for fine-grained coverage measures (e.g., condition coverage)
- Effort to achieve high coverage may be unjustified in large scale projects, in terms of time spent and producing unreliable test cases
- 60%-80% coverage is a “rule of thumb” (according to several surveys)



Test Coverage Measurement

- Two usages:
- Will help us derive test cases
- And to also know the progress of test activities

```
public class absTestSuite {  
  
    private absClass a;  
  
    @Before  
    public void setUp() throws Exception  
    {  
        a = new absClass();  
    }  
  
    @Test  
    public void testAbsPositive() {  
        assertTrue(a.abs(5)==5);  
    }  
}
```

Tests

```
public class absClass {  
  
    public static int abs(int x){  
        int a=1; // dummy code  
        if (x>0)  
            return x;  
        else  
            return -x;  
    }  
}
```

Manual Tests



Problems Javadoc Declaration		
Show methods with Statement Coverage		
Name	Statement	Branch
abs	66.7 %	50.0 %

2/3



Test Coverage Tools

- Most established languages have solid test coverage tools available for them, but the depth of functionality differs significantly from one to another
- Python has `sys.settrace` to tell you directly which lines are executing
- Emma (for Java) has a `ClassLoader` which re-writes byte-code on the fly

```
public class absClass {  
    public static int abs(int x){  
        int a=1; // dummy code  
        if (x>0)  
            return x;  
        else  
            return -x;  
    }  
}
```

Problems @ Javadoc Declaration

☐ Show methods with Statement Coverage

Name	Statement	Branch
abs	66.7 %	50.0 %



Code Coverage Tools

Java - CursorableLinkedList.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run Window Help

JUnit
Finished after 34,896 seconds
Runs: 13009/13009 Errors: 0 Failures: 0

Failures Hierarchy

TestAllPackages (31.10.2006 15:04:14)

Element	Coverage	Covered Lines	Total Lines
java - commons-collections	79,5 %	10927	13738
org.apache.commons.collections	74,1 %	3642	5183
ArrayStack.java	86,5 %	32	37
BagUtils.java	86,7 %	13	15
BeanMap.java	72,4 %	155	214
BinaryHeap.java	87,6 %	127	145
BoundedFifoBuffer.java	93,2 %	82	88
BufferOverflowException.java	55,6 %	5	9
BufferUnderflowException.java	88,9 %	8	9
BufferUtils.java	30,8 %	4	13
ClosureUtils.java	93,9 %	31	33
CollectionUtils.java	92,4 %	293	317
ComparatorUtils.java	8,6 %	3	35
CursorableLinkedList.java	85,4 %	444	520

```
public boolean addAll(int index, Collection c) {  
    if(c.isEmpty()) {  
        return false;  
    } else if( size == index || size == 0) {  
        return addAll(c);  
    } else {  
        Listable succ = getListableAt(index);  
        Listable pred = (null == succ) ? null : succ.prev();  
        Iterator it = c.iterator();  
        while(it.hasNext()) {  
            pred = insertListable(pred, succ, it.next());  
        }  
        return true;  
    }  
}
```

Emma (command line) and EcEmma



Emma

- Open-source tool
- Supports class, method, block, and line coverage
- Eclipse plug-in EclEmma also available
 - <http://www.eclemma.org>
- Uses byte-code instrumentation
 - Classes can be instrumented in advance, or during class loading
- Tool keeps a metadata file to associate byte-code with source code



CodeCover

- CodeCover is an open-source code coverage tool for Java under Eclipse
- It can be used inside Eclipse
- There are many other code coverage tool out there, but this is one of the lightest and powerful ones
- See the list @ <http://java-source.net/open-source/codecoverage>

<http://www.codecover.org>





Code Coverage Tools - CoverLipse

CoverLipse (supports data-flow based testing)

Java - Computation.java - Eclipse Platform

File Edit Source Refactor Navigate Search Project Run Window Help

Coverlip... 3

All-Uses Coverage

Clear results

simpletests

Computation

Computation.java

```
public class Computation {  
    public int add(int arg1, int arg2) {  
        int result = arg1 + arg2; int meinInt = 0;  
    }  
}
```

Show all definitions
Show all definitions
Show uses of the variable meinInt
Show uses of the variable result

Problem description: Definition of the variable result

Message	Line	covered uses	uncovered uses	Resource
Definition of the variable arg1	12	13 15		Computation.java
Definition of the variable arg2	12	13		Computation.java
Definition of the variable meinInt	13	19		Computation.java
Definition of the variable result	13	14 19	16	Computation.java
Definition of the variable result2	14	18		Computation.java
Definition of the variable result3	18			Computation.java

Writable Smart Insert 13 : 1



Code Coverage Tools - Clover

Clover Coverage Report

Dashboard
Coverage Reports
Coverage (Aggregate)
Test Code (Aggregate)
Test Results

Application Packages

- com.catalog.wiki (0%)
- com.catalog.wiki.block (0%)
- org.weceem.blog (70.6%)
- org.weceem.content (87.6%)
- org.weceem.controllers (71.4%)
- org.weceem.css (71.4%)
- org.weceem.event (-)
- org.weceem.export (68.5%)
- org.weceem.files (25.5%)
- org.weceem.html (87%)
- org.weceem.jobs (8.3%)
- org.weceem.js (68.2%)
- org.weceem.security (88.3%)

Classes	Tests	Results
AccessDeniedException		(0%)
AdminTagLib		(20%)
Blog		(66.7%)
BlogEntry		(72.7%)
CacheService		(100%)
CacheTagLib		(11.6%)
Comment		(80%)
ConfluenceSpaceImporter		(79.3%)
Content		(94%)
ContentController		(7.1%)
ContentDirectory		(37.9%)
ContentFile		(24.8%)
ContentRepositoryService		(48.3%)
ContentVersion		(100%)
ContentVersionService		(0%)
DefaultSpaceExporter		(0%)
DefaultSpaceImporter		(69.3%)
DiffUtils		(0%)
DomainConverter		(0%)
EditorController		(9.7%)
EditorFieldTagLib		(27.4%)
EditorService		(89.8%)
EventBuilder		(100%)
EventService		(100%)
ExternalLink		(54.5%)
ExternalLinkWeb		(18.3%)
Folder		(100%)
HTMLContent		(87%)
ImportException		(0%)
ImportExportConverter		(52.9%)
ImportExportService		(61.9%)
JavaScript		(89.2%)
ParagraphBlock		(0%)
PortletController		(54.5%)
RelatedContent		(25%)
RepositoryController		(12.6%)
SAASConfluenceParser		(97.8%)
SecurityPermissionsBuilder		(93.3%)
SecurityPolicyBuilder		(100%)
SimpleSpaceImporter		(100%)

Clover Coverage Report -
Coverage timestamp: Mon Feb 15 2010 15:29:05 EST

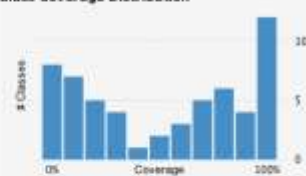
Overview Package: File
FRAMES NO FRAMES SHOW HELP

Statistics for project Clover database Mon Feb 15 2010 15:29:05 EST:

Stats: 2,700 LOC: 7,139 Total cmp: 1,874 Stmt/Method: 5.29
Branches: 1,344 NLOC: 74 Cmp density: 8.68 Methods/Class: 8.36
Methods: 510 Files: 60 Avg method cmp: 3.67 Classes/Package: 3.59
Classes: 81 Packages: 17

Coverage 61 classes, 1,919 / 4,554 elements
82.1%

Class Coverage Distribution



Class Complexity



Most Complex Packages

- 32.1% org.weceem.services (368)
- 11.4% org.weceem.controllers (424)
- 25.7% org.weceem.tags (342)
- 68.5% org.weceem.export (195)
- 87.6% org.weceem.content (101)

Most Complex Classes

- 48.3% ContentRepositoryService (444)
- 29.5% WeceemTagLib (253)
- 13.8% RepositoryController (228)
- 9.94% Content (58)
- 79.7% SimpleSpaceImporter (54)

Test Results 59 / 10 tests 8.88 secs
100%

Top 20 Project Risks

DiffUtils WidgetTagLib ParagraphBlock ContentController
SynchronizationController RepositoryController DefaultSpaceExporter VersionController CacheTagLib
ContentRepositoryService ContentDirectory WikiItemRenderController WeceemTagLib
EditorController DefaultSpaceImporter SpaceController ContentFile DomainConverter
CacheService WeceemEditor

Coverage Tree Map



Least Tested Methods

- 0% ContentController.field show() (42)
- 0% WeceemTagLib.field init() (49)
- 0% ContentRepositoryService.createContentFile(def) : def (31)
- 0% RepositoryController.field searchRequest() (31)
- 0% WidgetTagLib.field widget() (29)
- 0% DefaultSpaceImporter.executeSpace() : File (7)
- 0% ContentRepositoryService.updateNode(Content,def) : def (25)
- 0% RepositoryController.field beforeIntercept() (19)
- 0% DiffUtils.artSpring(def,def) : def (18)
- 0% SynchronizationController.field newContentFile() (13)
- 0% ContentDirectory.deleteContent() : Boolean (17)
- 0% SynchronizationController.field deleteContent() (13)
- 0% DiffUtils.def(def) : def (11)
- 0% ContentRepositoryService.synchronizeSpace(def) : def (13)
- 0% WeceemTagLib.field addGlobal() (17)
- 0% RepositoryController.field uploadFile() (15)
- 0% CacheTagLib.field cache() (13)
- 0% CacheService.getDiff(def,Boolean,def,def) : def (13)
- 0% WeceemTagLib.field humanDate() (11)
- 0% WeceemTagLib.field timestamp() (11)



Comparison of Coverage Tools

*2011 International Conference on Telecommunication Technology and Applications
Proc .of CSIT vol.5 (2011) © (2011) IACSIT Press, Singapore*

An Evaluation of Test Coverage Tools in Software Testing

Muhammad Shahid¹, Suhaimi Ibrahim

Advanced Informatics School (AIS), Universiti Teknologi Malaysia

International Campus, Jalan Semarak, Kuala Lumpur, Malaysia

smuhammad4@live.utm.my, suhaimiibrahim@utm.my



Comparison of Coverage Tools

Table 1 Tools With Supported Language

Tools	Java	C/C++	Other
<i>JavaCodeCoverage</i>	x		
<i>JFeature</i>	x		
JCover	x		
Cobertura	x		
Emma	x		
Clover	x		.Net
Quilt	x		
Code Cover	x		COBOL
Jester	x		
<i>GroboCodeCoverage</i>	x		
<i>Hansel</i>	x		
<i>Gretel</i>	x		
<i>BullseyeCoverage</i>		x	
<i>NCover</i>			.Net
<i>Testwell CTC++</i>		x	
<i>TestCocoon</i>		x	C#
<i>eXVantage</i>	x	x	
<i>OCCF</i>	x	x	x
<i>JAZZ</i>	x		

Table 3 Coverage Measurement Level

Tools	Statement/ Block	Branch / Decision	Method/ Function	Class	Requirement
<i>JavaCodeCoverage</i>	x	x	x		
<i>JFeature</i>			x		x
JCover	x	x	x	x	
Cobertura	x	x			
Emma	x		x	x	
Clover	x	x	x	x	
Quilt	x	x			
Code Cover	x	x			
Jester					
<i>GroboCodeCoverage</i>				x	
<i>Hansel</i>		x			
<i>Gretel</i>	x				
<i>BullseyeCoverage</i>		x	x		
<i>NCover</i>			x	x	
<i>Testwell CTC++</i>		x			
<i>TestCocoon</i>					
<i>eXVantage</i>	x	x	x		
<i>OCCF</i>	x	x			
<i>JAZZ</i>		x			



Code Coverage and Test Tools

www.opensourcetesting.org

- Code Coverage of Eclipse Code-base

<http://relengofthenerds.blogspot.com/2011/03/sdk-code-coverage-with-jacoco.html>

- Demo of test coverage for Python code

www.youtube.com/watch?v=jGJa_2UyHrY (video)

Etc.





Conclusion

- Coverage is a measure of WB testing effort to detect potential faults
- 100% statement coverage means that you tested for every bug that can be revealed by simple execution of a line of code
- 100% branch coverage means you will find every fault that can be revealed by testing each branch
- 100% coverage means that you tested for every possible fault, which is obviously impossible for larger projects