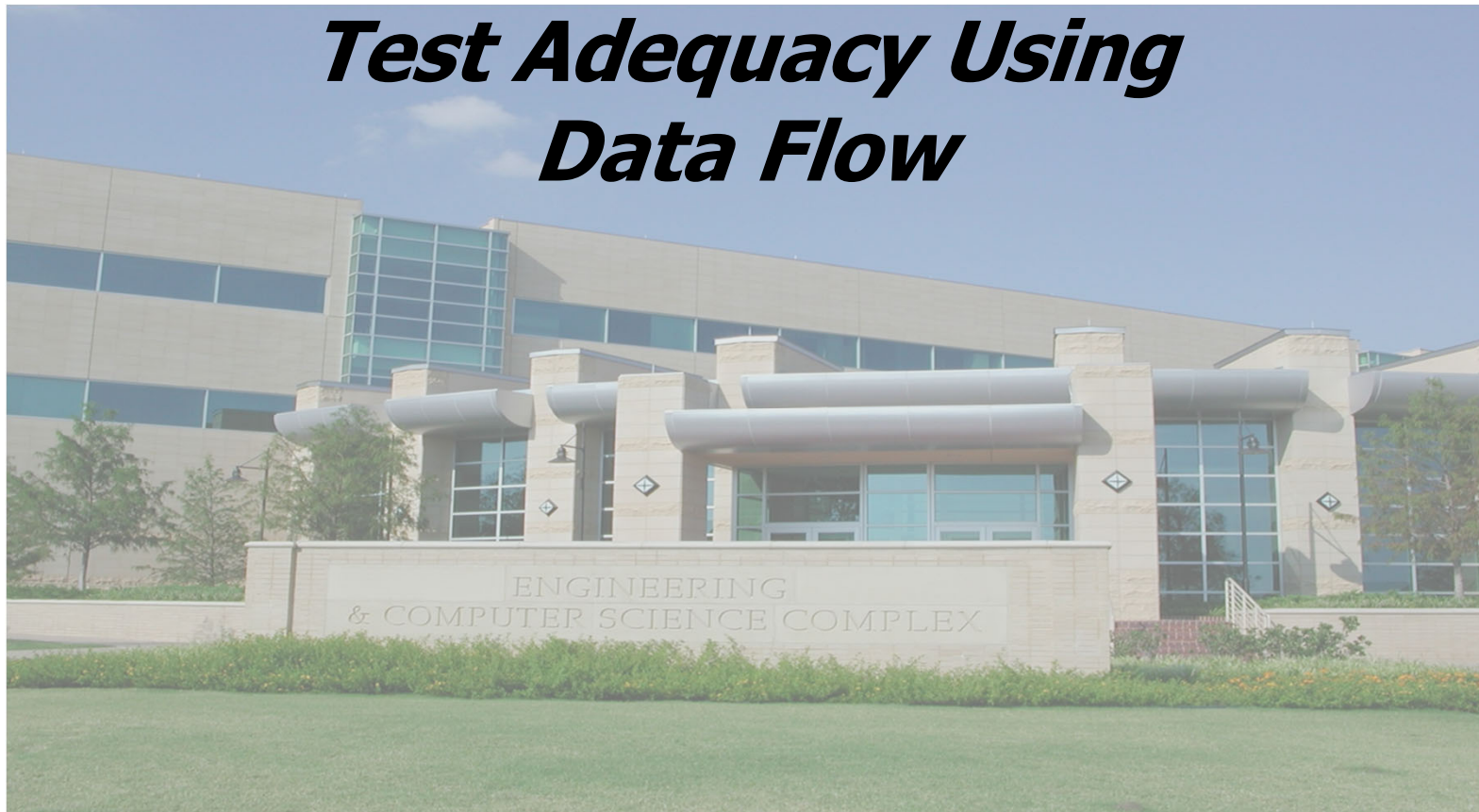


Test Adequacy Using Data Flow




Dr. Mark C. Paulk

SE 4367 – Software Testing, Verification, Validation, and Quality Assurance

Test Adequacy Assessment Topics

Part III. Test Adequacy Assessment and Enhancement

7. Test Adequacy Assessment Using Control Flow and Data Flow

- Basic
- Adequacy criteria based on control flow
-  • Concepts from data flow
- Adequacy criteria based on data flow
- Control flow versus data flow
- The “subsumes” relation
- Structural and functional testing
- Scalability of coverage measurement
- Tools

8. Test Adequacy Assessment Using Program Mutation

Basic Concepts

We will now examine some test adequacy criteria based on the flow of “data” in a program.

- in contrast to criteria based on “flow of control”**

Test adequacy criteria based on the flow of data are useful in improving tests that are adequate with respect to control-flow based criteria.

Mathur, Example 7.28

Program 7.15

```
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
```

The condition at line 8
should be (y != 0 and x != 0).

**This MC/DC adequate test
set does not reveal the error.**

	<u>x</u>	<u>y</u>	<u>z</u>
t ₁	0	0	0.0
t ₂	1	1	1.0

Program 7.15

```
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
```

z is (re)defined at lines 4, 6, 7, 9, and 10.

Neither of the two tests force the use of z as 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.

An MC/DC adequate test does not force the execution of this path and hence the divide-by-zero error is not revealed.

Program 7.15

```
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
```

The error would be revealed by a test of each feasible definition and use pair for z.

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

- in def-use pair (a,b), z is defined in line a and used in line b

	<u>x</u>	<u>y</u>	<u>z</u>	<u>def-use pairs for z</u>
t₁	0	0	0.0	(4,7), (7,10)
t₂	1	1	1.0	(4,6), (6,9)
t₃	0	1	0.0	(4,7), (7,9)
t₄	1	0	1.0	(4,6), (6,10)

Program 7.15

```
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
```

**Would an LCSAJ
adequate test set
reveal the error?**

- **Yes**

**There are examples
where an LCSAJ
adequate test set
would not reveal an
error that a test set
based on data flow
would.**

Definitions and Uses

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.

- assignment statement **$x = y + z$** defines variable **x** and uses variables **y** and **z**
- declaration **$\text{int } x, y, A[10];$** defines three variables
- procedure **$\text{scanf}(\text{"\%d \%d"}, \&x, \&y)$** defines variables **x** and **y**
- procedure **$\text{printf}(\text{"Output: \%d \n"}, x+y)$** uses variables **$x$** and **$y$**

Definitions and Uses Parameters

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, serves 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 of the above statements defines a pointer variable z
- the second defines y and uses z
- the third defines x through the pointer variable z
- the last defines y and uses x accessed through the pointer variable z

Definitions and Uses

Arrays

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 is the requirement for coverage analysis.

C-Use

Uses of a variable that occur

- **within an expression as part of an assignment statement**
- **in an output statement**
- **as a parameter within a function call**
- **in subscript expressions**

are classified as c-use.

The “c” in c-use stands for computational.

C-Use Example

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: 5

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

P-Use

The occurrence of a variable in an expression used as a condition in a branch statement such as an if or a while, is considered as a p-use.

The “p” in p-use stands for predicate.

P-Use Example

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

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

Answer: 3

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

A Confusing Classification of P-Use

Consider the statement:

if ($A[x+1] > 0$) {output (x)};

The use of A is clearly a p-use.

Is use of x in the subscript, a c-use or a p-use?

- **it does not occur directly within the if ...**

C-Uses Within a Basic Block

Consider the basic block

```
p = y + z;  
x = p + 1;  
p = z * z;
```

There are two definitions of p in this block, but only the second definition will propagate to the next block.

- the first definition of p is considered local to the block
- the second definition is global

We are concerned with global definitions, and uses.

Note that y and z are global uses.

- their definitions flow into this block from some other block

Data-Flow Graph

A data-flow graph of a program, also known as def-use graph, captures the flow of definitions (also known as defs) across basic blocks in a program.

It is similar to a control flow graph of a program.

- the nodes, edges, and all paths thorough the control flow graph are preserved in the data-flow graph**

Procedure for Constructing a Data-Flow Graph 1-2

Given a program P , find its basic blocks.

- **CFG = (N, E)**
- **each block becomes a node in the def-use graph**

1. Compute def_i , c-use_i and p-use_i for each basic block i in P .

2. Attach def_i , c-use_i , and p-use_i to each node i in N .

Procedure for Constructing a Data-Flow Graph 3

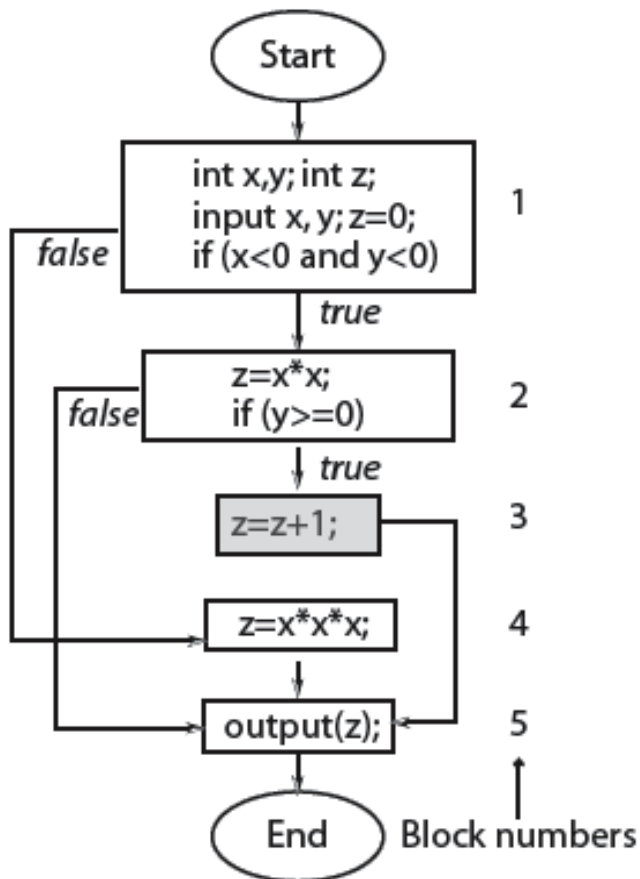
3. For each node i in N that has a non-empty p-use set and ends in condition C
- associate edge (i, j) with C given that edge (i, j) is taken when the condition is true
 - associate edge (i, k) with $!C$ given that edge (i, k) is taken when the condition is false

Label each edge with the condition which when true causes the edge to be taken.

$d_i(x)$ refers to the definition of variable x at node i .

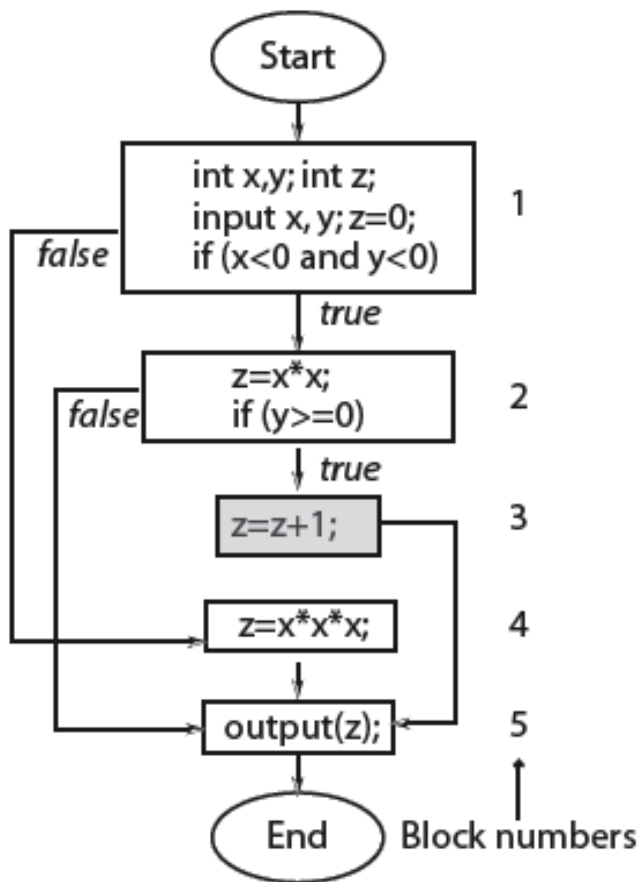
$u_i(x)$ refers to the use of variable x at node i .

Mathur, Example 7.29



Given program P7.4, whose CFG is given in Figure 7.4, what are the def and use sets for each basic block?

Note that the infeasible block 3 does not affect the computation of these sets.

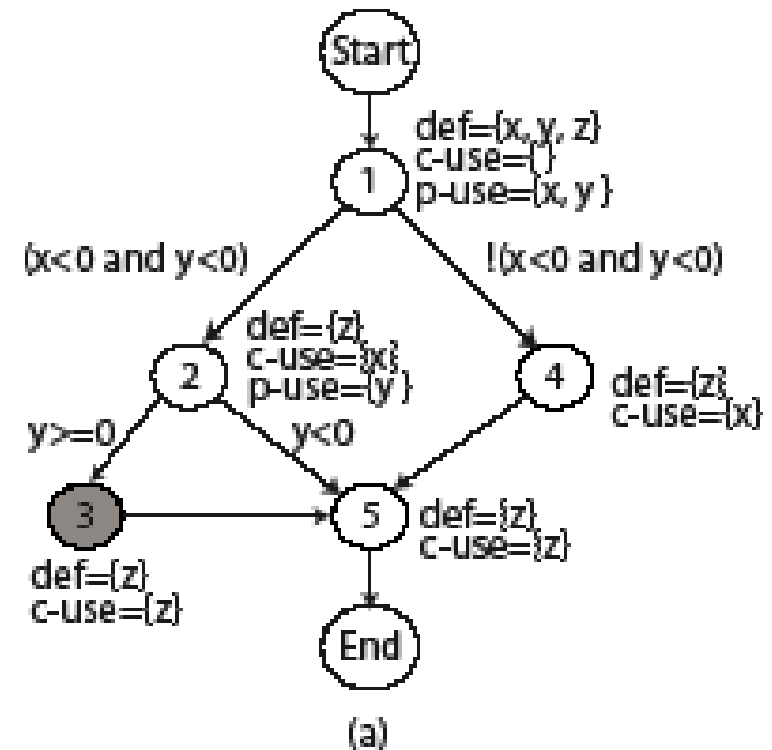
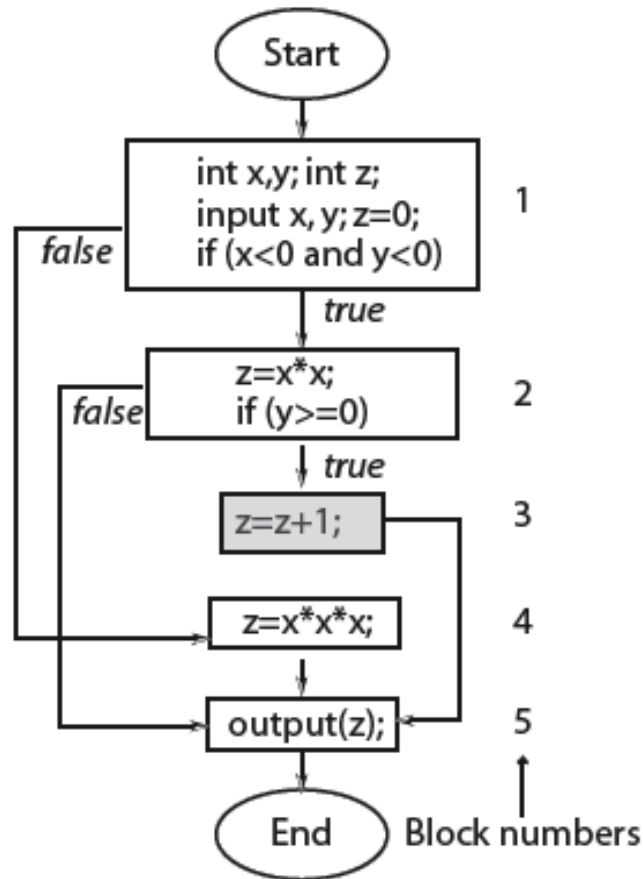


Node	def	c-use	p-use
1	{x, y, z}	{}	{x, y}
2	{z}	{x}	{y}
3	{z}	{z}	{}
4	{z}	{x}	{}
5	{}	{z}	{}

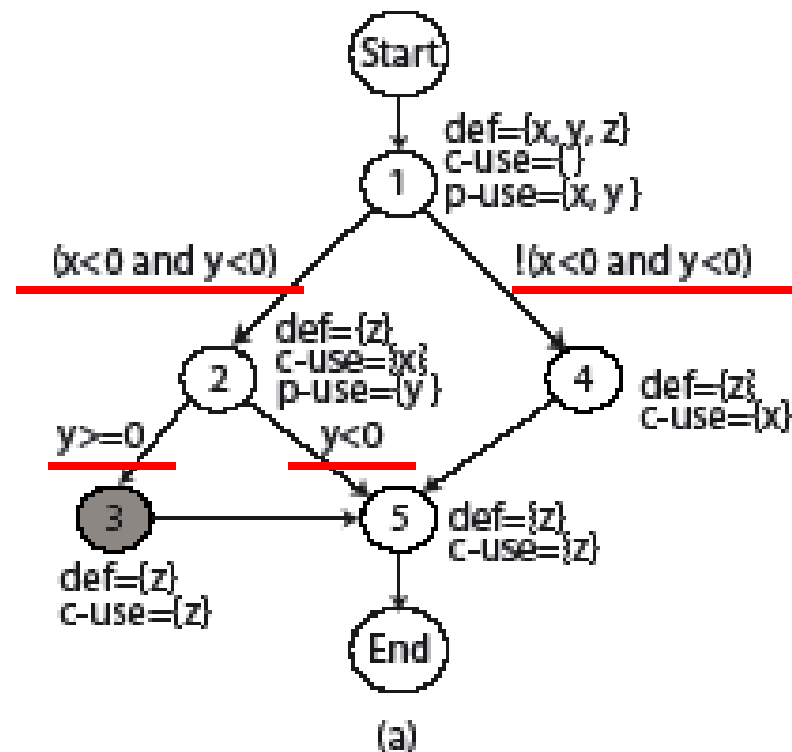
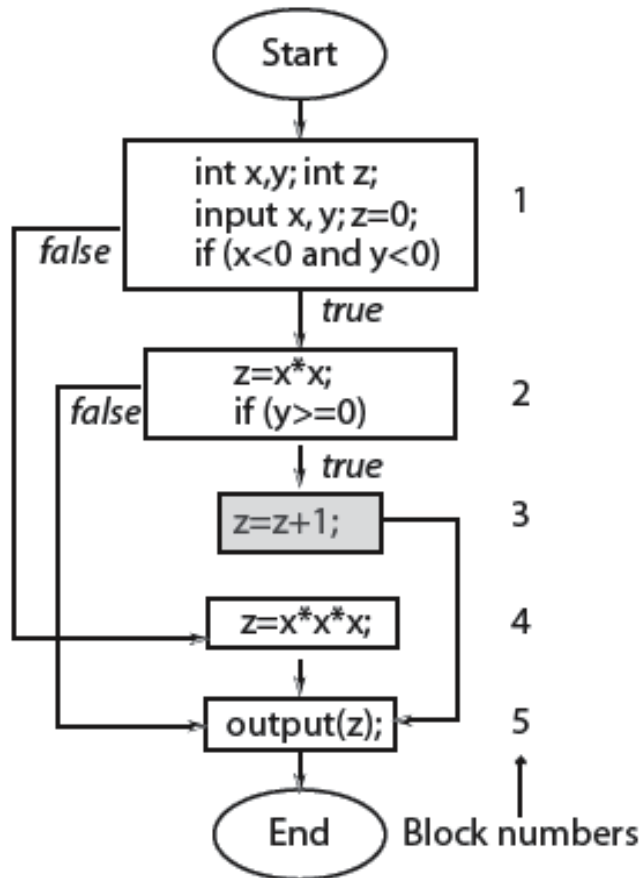
Draw the data-flow graph.

Shade node 3 to emphasize that it is unreachable.

Start and Exit nodes are optional.

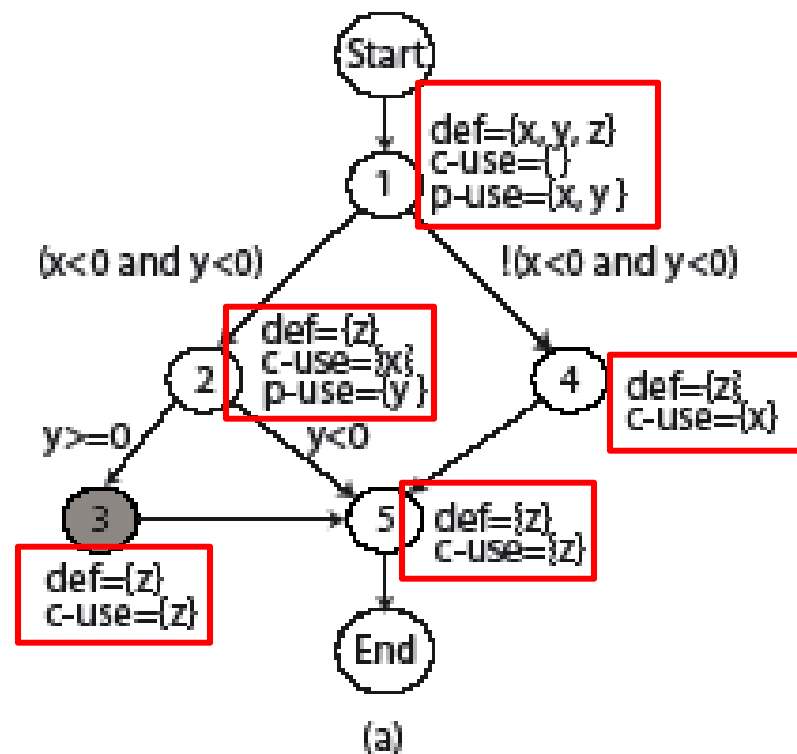
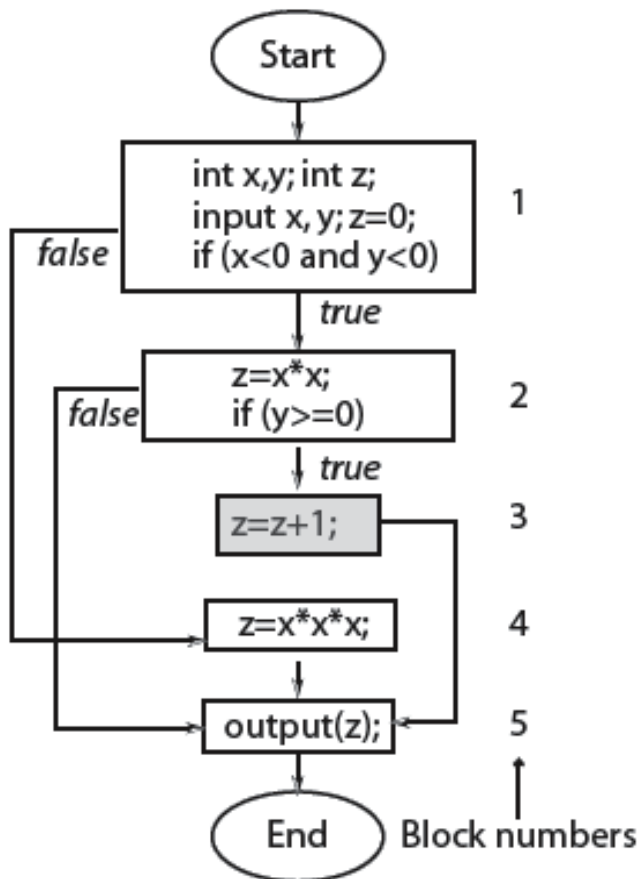


Edges are labeled with conditions in the data-flow graph...



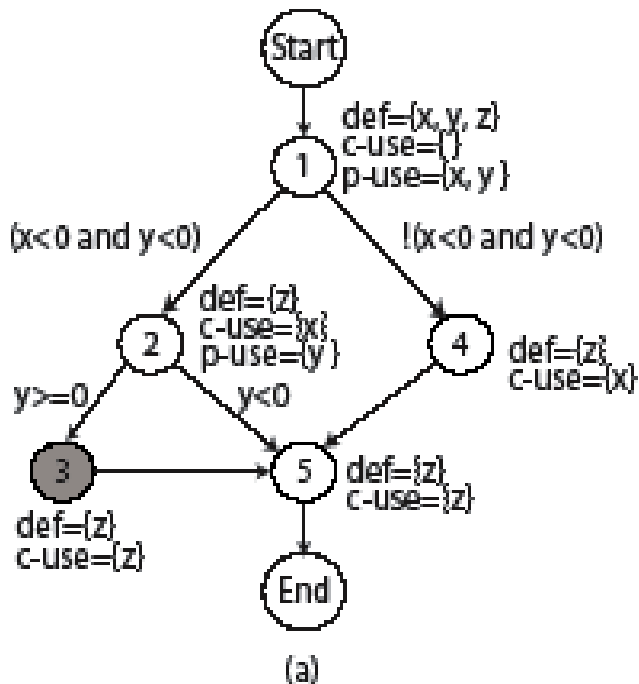
Nodes are labeled with def_i , $c-use_i$, and $p-use_i$ in the data-flow graph...

- p-uses are associated with nodes that end in an if or while



Def-Clear Paths

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 the definition of z at node 2 is live at node 5 while that at node 1 is not live at node 5.

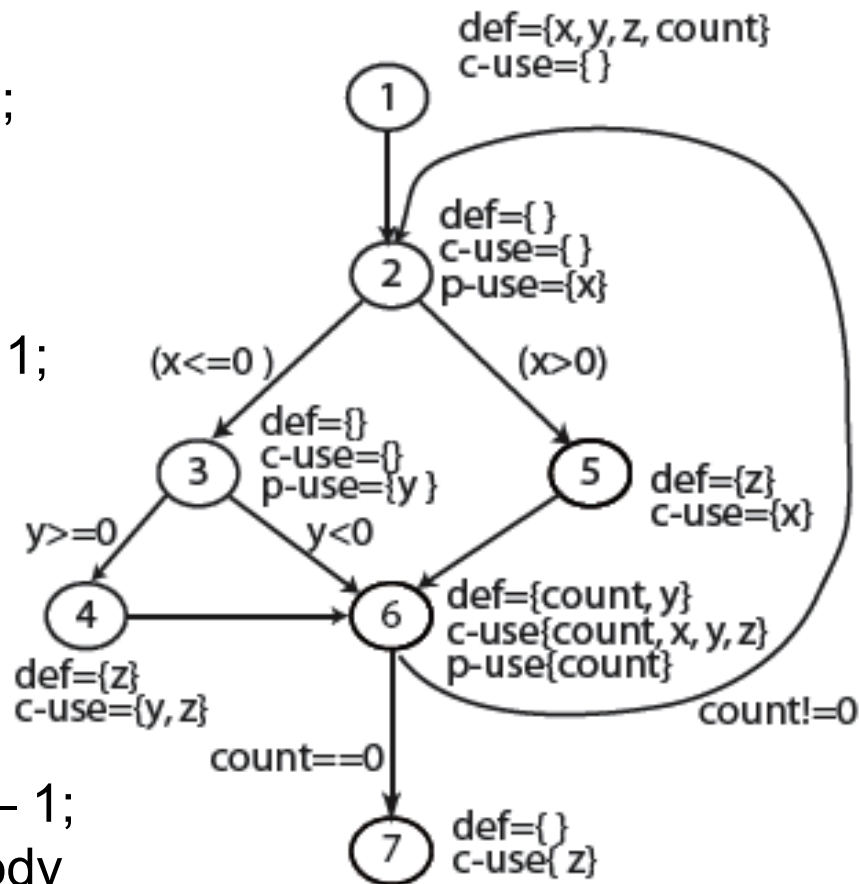
Mathur, Example 7.30

Program P7.16

```

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     } // end of loop body
17     while (count > 0)
18     output (z);
19 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

A “Corrected” Example 7.30

Program P7.16 Revised

```
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              } // if y ≥ 0
10         } // if x ≤ 0
11         else {
12             z = 1 / x;
13         } // else x ≤ 0
14         y = x * y + z;
15         count = count – 1;
16     } while (count > 0) // end loop
17     output (z);
18 end
```

<u>Node</u>	<u>Lines</u>
1	1,2,3,4
2	5,6
3	7
4	8 (9,10,11)
5	12 (13)
6	14,15,16
7	17,18

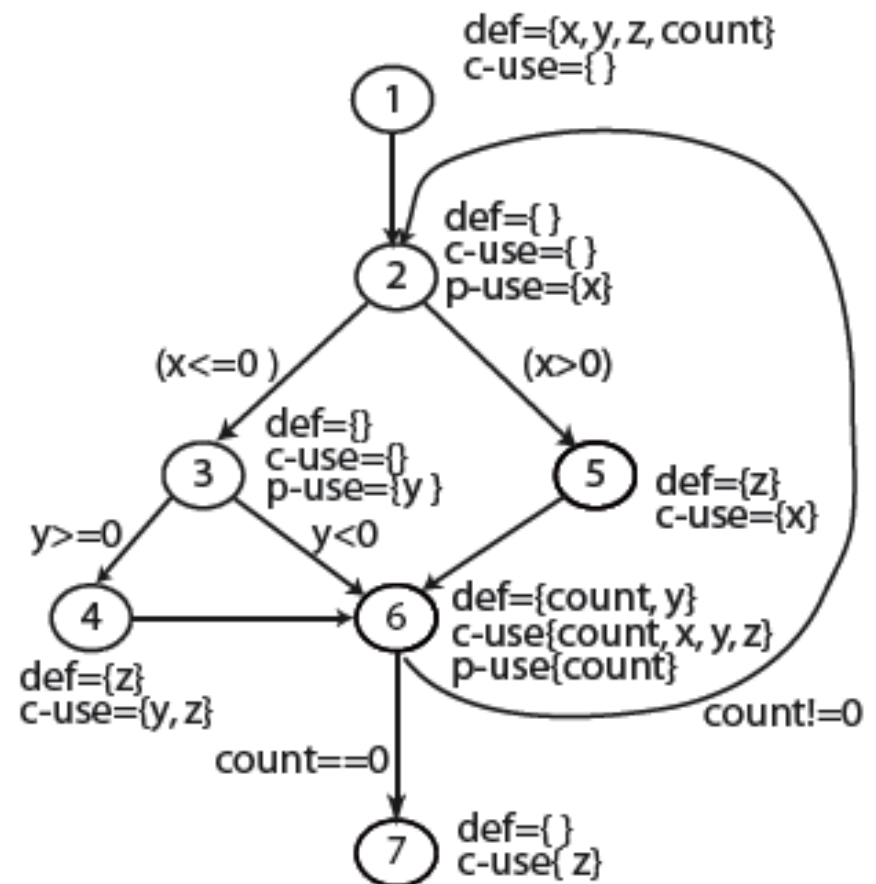
The “do” at line 5 has two entries (from lines 4 and 16) thus has to begin a block but is essentially a syntactical marker (similar to if’s with two exits having to end their block).

What are the def-clear paths for x?

- (1,2)
- (1,2,3,6)
- (1,2,3,4,6)
- (1,2,5)
- (1,2,5,6)

Which definitions are live at node 4?

- $d_1(y)$
- $d_6(y)$
- $d_1(z)$
- $d_5(z)$



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

Two kinds of def-use pairs

- def and c-use (dcu)
- def and p-use (dpu)

dcu

For $d_i(\underline{x})$, $\underline{dcu}(d_i(\underline{x}))$ is the set of all nodes j such that there exists $u_j(\underline{x})$ and there is a def-clear path with respect to \underline{x} from node i to node j .

- denotes the set of all nodes where $d_i(\underline{x})$ is live and used
- $\underline{dcu}(\underline{x}, i)$ is an alternate notation

dpu

When $u_k(\underline{x})$ occurs in a predicate, $\underline{dpu}(d_i(\underline{x}))$ is the set of all edges (k, l) such that there is a def-clear path with respect to \underline{x} from node i to edge (k, l) .

- $\underline{dpu}(\underline{x}, i)$ is an alternate notation
- the number of elements in a \underline{dpu} set will be a multiple of 2

Mathur, Example 7.31

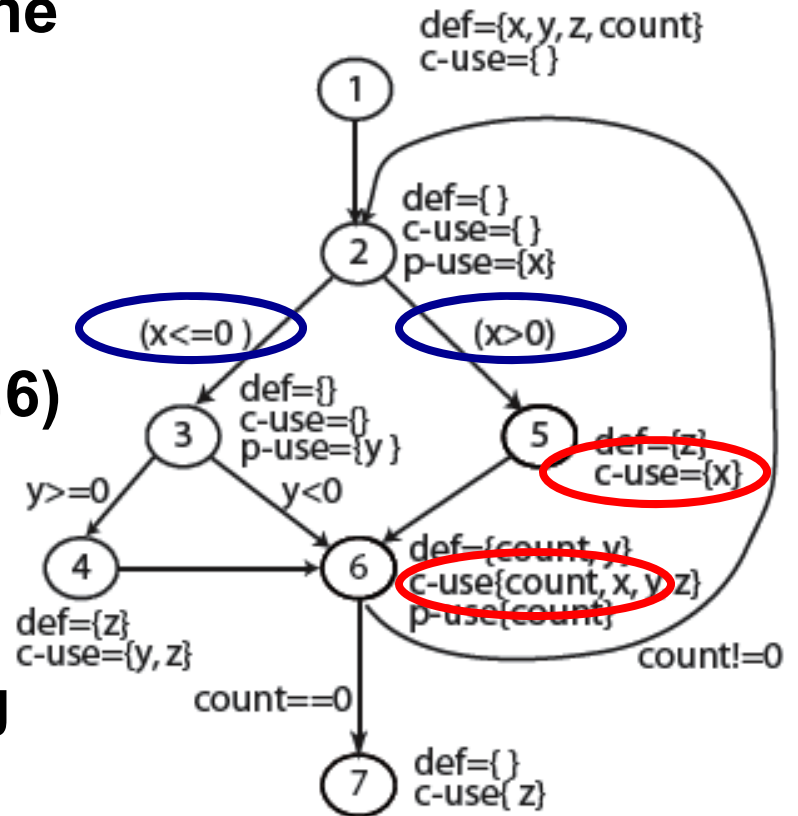
Compute the dcu and dpu sets for the DFG.

dcu (x, 1)

- c-use of x at nodes 5, 6
- def-clear paths for x (1,2,5), (1,2,5,6)
- $\text{dcu}(x, 1) = \{5, 6\}$

dpu (x, 1)

- p-use of x at node 2 with outgoing edges (2,3) and (2,5)
- def-clear paths to both edges
- $\text{dpu}(x, 1) = \{(2,3), (2,5)\}$



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)}

Covering a Def-Use Pair

We say that a def-use pair $(d_i(\underline{x}), u_j(\underline{x}))$ is covered when a def-clear path that includes nodes i to node j is executed.

If $u_j(\underline{x})$ is a p-use, then all edges of the kind (j, k) must also be taken during some executions.

Def-Use Chains (k-dr Interaction)

An alternating sequence of def-use pairs is a def-use chain (or k-dr interaction).

- **d stands for definition**
- **r stands for reference (use)**
- **k denotes the length of the chain**
 - **one more than the number of def-use pairs**

The nodes along the def-use chain are distinct.

Def-use chains can be created with different variables at the use-define node.

Compound predicates are split into two nodes for k-dr analysis.

Mathur, Example 7.32

Given $d_1(\underline{z})$ and $u_4(\underline{z})$.

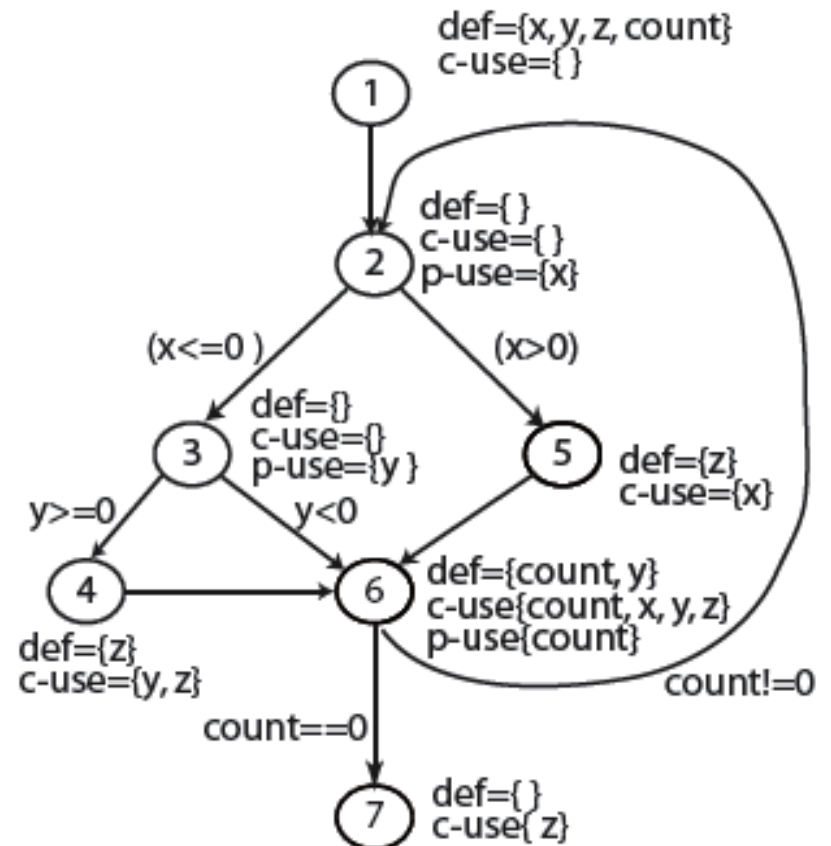
The def-use interaction for \underline{z} at nodes 1 and 4 is a k-dr chain with $k=2$ denoted by (1,4).

Append $d_4(\underline{z})$ and $u_6(\underline{z})$.

3-dr chain is (1,4,6).

There is no $k>3$ chain.

- node 4 repeats



Minimal Set of Def-Use Pairs

Def-use pairs are items to be covered during testing.

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.

Optimizing the Def-Use Test Set

$\text{dcu}(\underline{y}, 1) = \{4, 6\}$

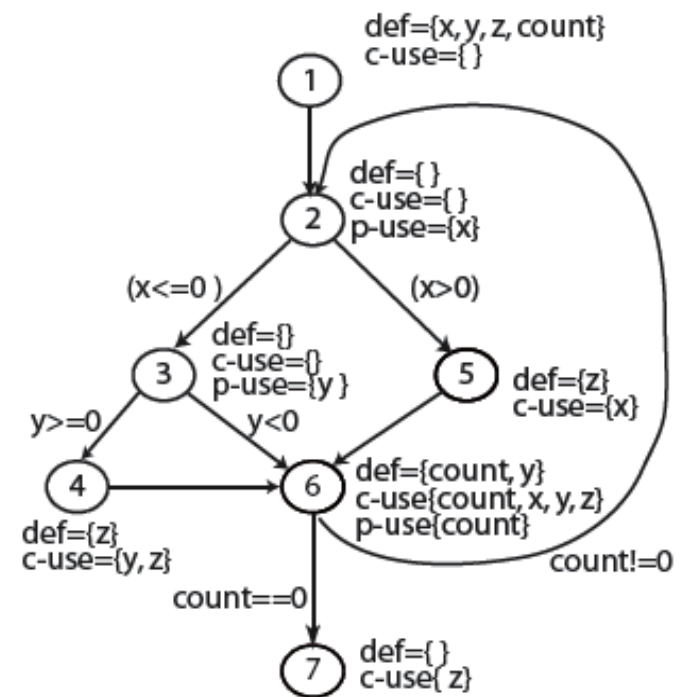
$\text{dcu}(\underline{z}, 1) = \{4, 6, 7\}$

Will the coverage of $\text{dcu}(\underline{z}, 1)$ imply the coverage of $\text{dcu}(\underline{y}, 1)$?

Path (1,2,3,6) must be traversed to cover c-use of \underline{z} at node 6 as defined at node 1.

This implies coverage of \underline{y} at node 6 as defined at node 1.

$\text{dcu}(\underline{y}, 1)$ is covered automatically if we cover $\text{dcu}(\underline{z}, 1)$.



Similarly, $\text{dcu}(x,1)$ is covered when $\text{dcu}(z,5)$ is covered.

$\text{dcu}(\text{count},1)$ is covered when $\text{dcu}(z,5)$ is covered.

Corresponding dpu's are also covered.

Reduces to a minimal set of uses to cover...

- c-uses from 17 to 12**
- p-uses from 10 to 4**

A Minimal Set of Def-Uses

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

Data Environment

Let n be a node in a DFG.

Each variable used at n is an input variable of n .

Each variable defined at n is an output variable of n .

The set of all live definitions of all input variables of n is the data environment of n denoted $DE(n)$.

Data Context

Given $X(n) = \{x_1, x_2, \dots, x_k\}$ is the set of input variables for n in DFG F .

Let x_j^{ij} denote the i_j^{th} definition of x_j in F .

An elementary data context $EDC(n)$ is the set of definitions $\{x_1^{i_1}, x_2^{i_2}, \dots, x_k^{i_k}\}$, $i_k \geq 1$ of all variables in $X(n)$ such that definition is live when control arrives at node n .

The data context of node n is the set of all its elementary data contexts denoted $DC(n)$.

$d_k(x_j)$ is the i_j^{th} definition of x_j at node k .

Ordered Data Context


The ordered elementary data context of node n $OEDC(n)$ is a set of ordered sequence of definitions of the input variables of n .

An ordered data context for node n is the set of all ordered elementary data contexts of n $ODC(n)$.

Test Adequacy Assessment Topics

Part III. Test Adequacy Assessment and Enhancement

7. Test Adequacy Assessment Using Control Flow and Data Flow

- Basic
- Adequacy criteria based on control flow
- Concepts from data flow
-  • Adequacy criteria based on data flow
- Control flow versus data flow
- The “subsumes” relation
- Structural and functional testing
- Scalability of coverage measurement
- Tools

8. Test Adequacy Assessment Using Program Mutation

Data Flow Based Adequacy

Given a total of n variables $v_1, v_2 \dots v_n$ each defined at d_i nodes.

CU: total number of c-uses in a program

PU: total number of p-uses

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

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

C-Use Coverage

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

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

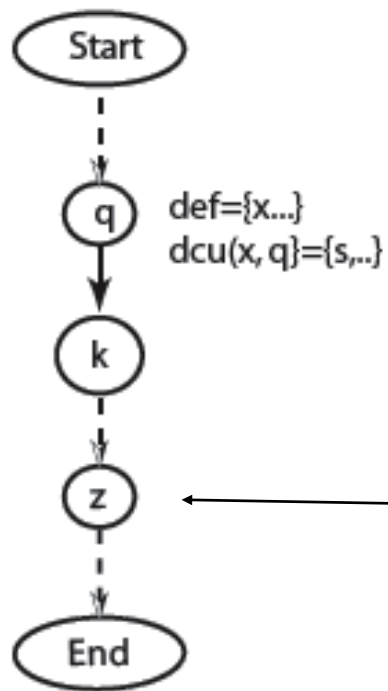
- CU_c is the number of c-uses covered
- CU_f is the number of infeasible c-uses

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,..., z,..., End)
covers the c-use at node z of **x**
defined at node q

- given that (k, ..., z) is def-clear with respect to **x**



P-Use Coverage

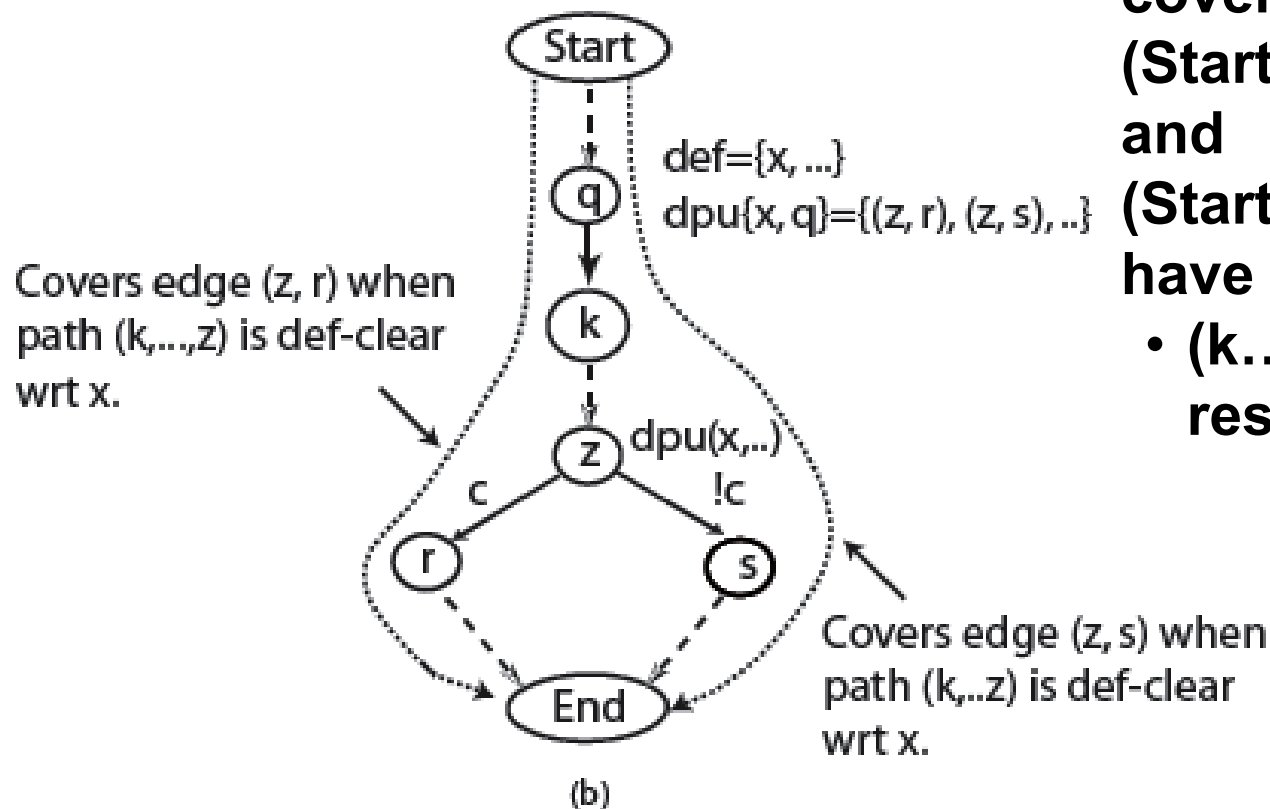
The p-use coverage of T with respect to (P,R) is computed as

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

- PU_c is the number of p-uses covered
- 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



P-use of **x** at node z is covered when paths (Start,..., q, k,..., z,r,..., End) and (Start,...,q, k,...,z, s,..., End) have been traversed

- (k...z) are def-clear with respect to **x**

All-Uses Coverage

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

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

- CU is the total c-uses, PU is the total p-uses
- CU_c is the number of c-uses covered
- PU_c is the number of p-uses covered
- CU_f is the number of infeasible c-uses
- PU_f is the number of infeasible p-uses

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

k-dr Chain Coverage

For a given $k \geq 2$, the $k\text{-dr}(k)$ coverage of T with respect to (P,R) is computed as

$$\frac{C_c^k}{(C^k - C_f^k)}$$

- C_c^k is the number of $k\text{-dr}$ interactions covered
- C^k is the number of elements in $k\text{-dr}(k)$
- C_f^k is the number of infeasible interactions in $k\text{-dr}(k)$

T is considered adequate with respect to the $k\text{-dr}$ coverage criterion if its $k\text{-dr}(k)$ coverage is 1.

Infeasible P-Uses and C-Uses

Coverage of a c-use or a p-use requires a path to be traversed through the program.

If this path is infeasible, then some c-uses and p-uses that require this path to be traversed might also be infeasible.

Infeasible uses are often difficult to determine without some hint from a test tool.

Context Coverage

EDC(k) is considered covered by t if the following two conditions are satisfied during the execution of P:

- Node k is along the path traversed during the execution of P against t.**
- When control arrives at k along this path, all definitions in EDC(k) are live.**

A data context DC(k) for node k is considered covered when all elementary data contexts in DC(k) are covered.

Ordered Elementary Data Context Coverage

An ordered elementary data context coverage $OEDC(k)$ is considered covered by t if the following conditions are satisfied during the execution of P :

- Node k is along the path p traversed during the execution of P against t .**
- All definitions in $OEDC(k)$ are live when control arrives at k along p .**
- The sequence in which variables in $X[k]$ are defined is the same as that in $OEDC(k)$.**

Elementary Data Context Coverage

Given a program P subject to requirements R, and having a DFG containing n nodes, the data context coverage of T with respect to (P,R) is computed as:

$$\frac{\text{EDC}_c}{(\text{EDC} - \text{EDC}_i)}$$

- **EDC** is the number of elementary data contexts in P
- **EDC_c** is the number of elementary data contexts covered
- **EDC_i** is the number of infeasible elementary data contexts

T is considered adequate with respect to the elementary data context coverage criterion if the data context coverage is 1.

Ordered Elementary Data Context Coverage

Given a program P subject to requirements R , and having a DFG containing n nodes, the ordered elementary data context coverage of T with respect to (P,R) is computed as:

$$\frac{\text{OEDC}_c}{(\text{OEDC} - \text{OEDC}_i)}$$

- OEDC is the number of ordered elementary data contexts in P
- OEDC_c is the number of ordered elementary data contexts covered
- OEDC_i is the number of infeasible ordered elementary data contexts

T is considered adequate with respect to the ordered elementary data context coverage criterion if the data context coverage is 1.

Data-Flow Based Criteria

There exist several adequacy criteria based on data flows.

- **some of these are more powerful in their error-detection effectiveness than the c-use, p-use, and all-uses criteria**

Examples:

- **def-use chain or k-dr chain coverage**
 - **alternating sequences of def-use for one or more variables**
- **data context and ordered data context coverage**

Control Flow vs Data Flow

Adequacy criteria based on control flow aims at testing only a few of the many paths through a program.

Sometimes data-flow based criteria turn out to be more powerful than control-flow criteria, including the MC/DC-based criteria.


The test set $T = \{t_1, t_2\}$ for program P7.16 is adequate with respect to block coverage, condition coverage, multiple condition coverage, and MC/DC.

- c-use coverage is 58%**
- p-use coverage is 75%**

Test Adequacy Assessment Topics

Part III. Test Adequacy Assessment and Enhancement

7. Test Adequacy Assessment Using Control Flow and Data Flow

- Basic
- Adequacy criteria based on control flow
- Concepts from data flow
- Adequacy criteria based on data flow
- Control flow versus data flow
-  • The “subsumes” relation
- Structural and functional testing
- Scalability of coverage measurement
- Tools

8. Test Adequacy Assessment Using Program Mutation

Subsumes Relation

Subsumes

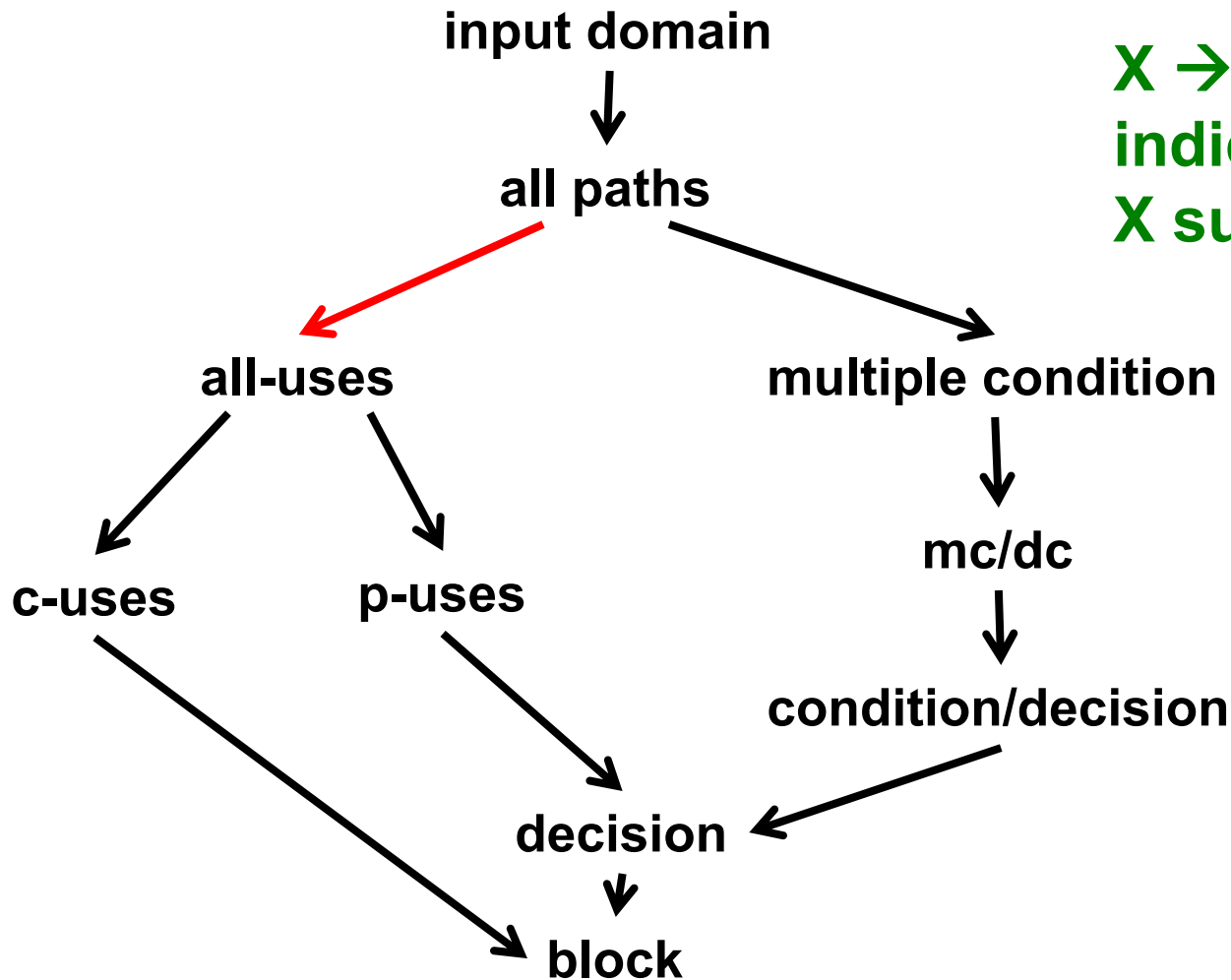
- **Given a test set T that is adequate with respect to criterion C_1 , what can we conclude about the adequacy of T with respect to another criterion C_2 ?**

Effectiveness

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

Subsumes Relationship (Fig 7.11)

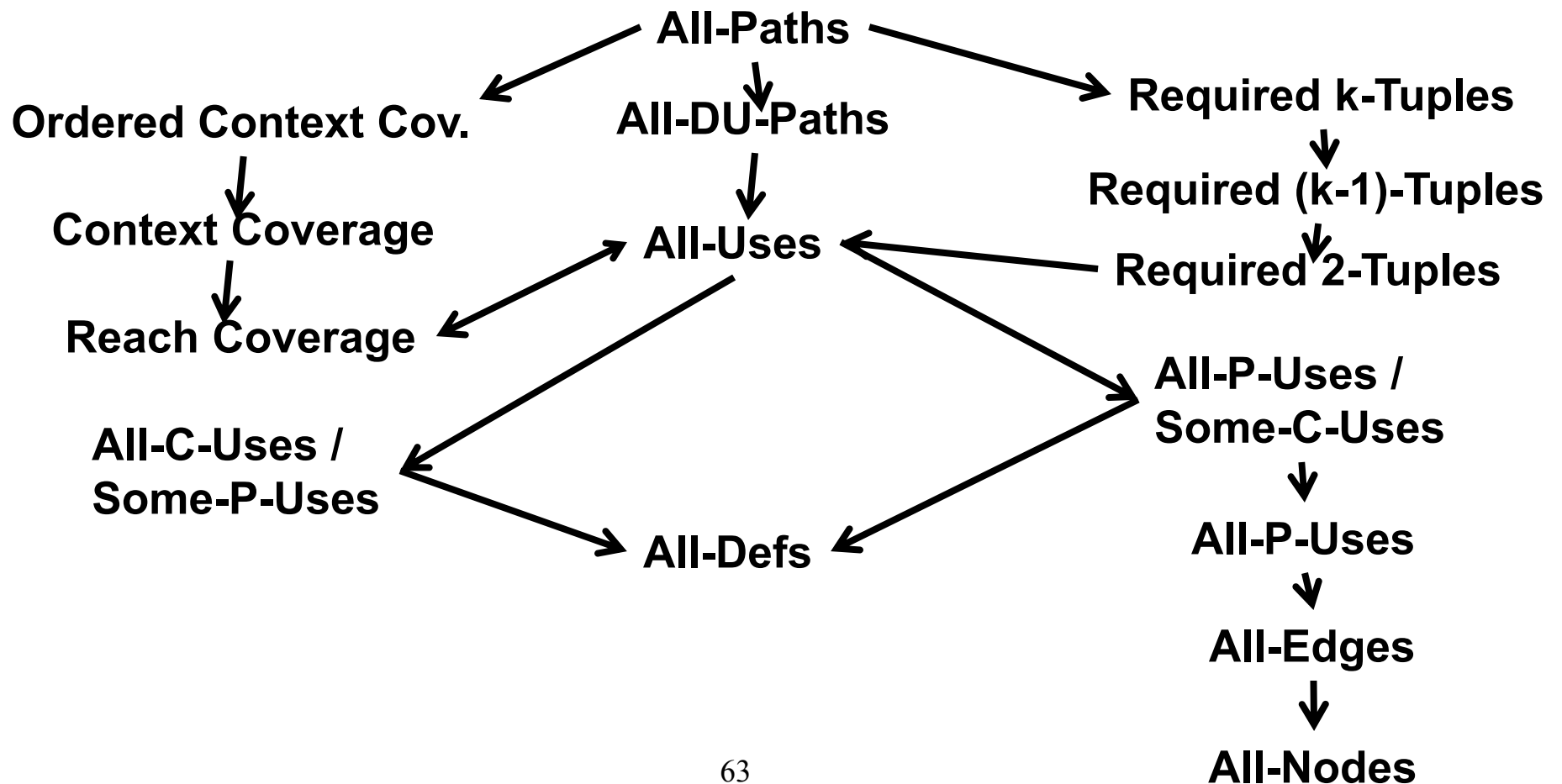
Among Control-Flow and Data-Flow Adequacy Criteria



$X \rightarrow Y$
indicates that
X subsumes Y

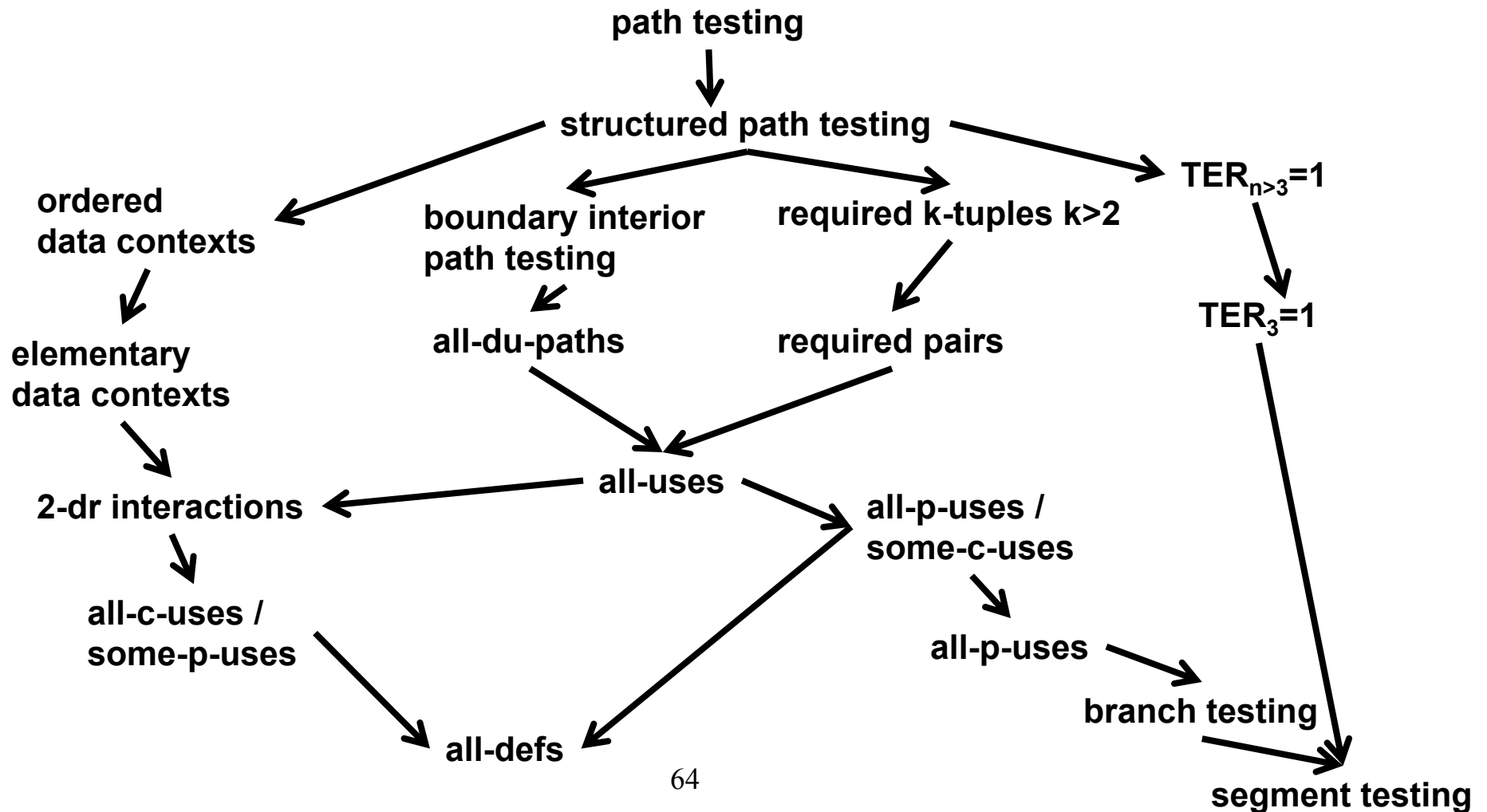
Subsumes Relationships (Clarke 1989)

L.A. Clarke, A. Podgurski, D.J. Richardson, and S.J. Zeil, "A Formal Evaluation of Data Flow Path Selection Criteria," IEEE Transactions on Software Engineering, November 1989.



Subsumes Relationships (Ntafos 1988)

**S.C. Ntafos, "A Comparison of Some Structural Testing Strategies,"
IEEE Transactions on Software Engineering, June 1988.**



Definitions of Testing Strategies from (Ntafos 1988)

Segment testing

- each statement in the program is executed by at least one test case

Branch testing

- each transfer of control (branch) is exercised by at least one test case

Path testing

- all execution paths in a program are tested

Boundary-interior path testing

- **boundary tests: enter the loop but do not iterate it**
- **interior tests: iterate the loop at least once**

Structured path testing

- **representative paths that do not iterate a loop more than k times**

TER_{n+2}

- **Test Effectiveness Ratios (TER)**
- **all subpaths containing up to n LCSAJs are tested**
- **TER₁ is segment testing, TER₂ is branch testing**

2-dr interaction

- variable definition plus reference (use)

Required pairs

- for each 2-dr interaction involving a reference in a branch predicate, produce one required pair for each outcome of the predicate

All-uses

- all interactions between a variable definition and a c-use or p-use of that definition are tested

All-defs

- each variable definition and a p-use or c-use of that definition are tested

All-c-uses

- all interactions between a variable definition and a c-use of that definition are tested

All-p-uses

- all interactions between a variable definition and a p-use of that definition are tested

All-du-paths

- all p-uses or c-uses and a definition that reaches it is tested along all cycle-free paths

Elementary data context

- complete set of definitions for the variables referenced in a statement such that the definitions reach the statement

Ordered data contexts

- definitions in each elementary data context are visited in all possible orders


Required k-tuples

- all sequences of $k-1$ related 2-dr interactions

Test Adequacy Assessment Topics

Part III. Test Adequacy Assessment and Enhancement

7. Test Adequacy Assessment Using Control Flow and Data Flow

- Basic
- Adequacy criteria based on control flow
- Concepts from data flow
- Adequacy criteria based on data flow
- Control flow versus data flow
- The “subsumes” relation
-  • Structural and functional testing
- Scalability of coverage measurement
- Tools

8. Test Adequacy Assessment Using Program Mutation

Structural vs Functional Testing

Structural testing is (arguably) functional testing with the addition of code-based adequacy measurement.

Structural testing is supplementary to functional testing.

Measuring code coverage enhances the test generated using functional testing.

Scalability of Coverage

One might argue that measuring code coverage (and enhancing tests based on code coverage) is practical only during unit testing.

Incremental coverage measurement

- **create a hierarchy of code elements**
- **reduce the number of modules for which coverage measurements are taken**
- **constrain the coverage criteria**

Memory constraints can be a challenge for testing embedded systems.

Summary – Things to Remember

Definitions and usages

Computational uses (c-uses), predicate uses (p-uses)

Data flow graphs

Def clear paths

Def-use pairs (dcu, dpu)

All-uses coverage

Subsumes relationship

Questions and Answers

