

# Software Testing

## **Structural (White Box) Testing** **Lecture 2(data flow testing)**

5/29/2020

# Basic Idea

- A program unit accepts inputs, performs computations, assigns new values to variables, and returns results.
- One can visualize of “flow” of data values from one statement to another.
- A data value produced in one statement is expected to be used later.
  - Example
    - Obtain a file pointer ..... use it later.
  - If the later use is never verified, we do not know if the earlier assignment is acceptable.
- Two motivations of data flow testing
  - The memory location for a variable is accessed in a “desirable” way.
  - Verify the correctness of data values “defined” (i.e. generated) – observe that all the “uses” of the value produce the desired results.
- Idea: A programmer can perform a number of tests on data values.
  - These tests are collectively known as data flow testing.

# The General Idea

- Data flow testing can be performed at two conceptual levels.
  - Static data flow testing
  - Dynamic data flow testing
- Static data flow testing
  - Identify potential defects, commonly known as **data flow anomaly**.
  - Analyze source code.
  - Do not execute code.
- Dynamic data flow testing
  - Involves actual program execution.
  - Bears similarity with control flow testing.
    - Identify paths to execute them.
    - Paths are identified based on **data flow testing criteria**.

# Data Flow Anomaly

- Anomaly: It is an abnormal way of doing something.
  - Example 1: The second definition of x overrides the first.  
     $x = f1(y);$   
     $x = f2(z);$
- Three types of abnormal situations with using variable.
  - Type 1: Defined and then defined again
  - Type 2: Undefined but referenced
  - Type 3: Defined but not referenced

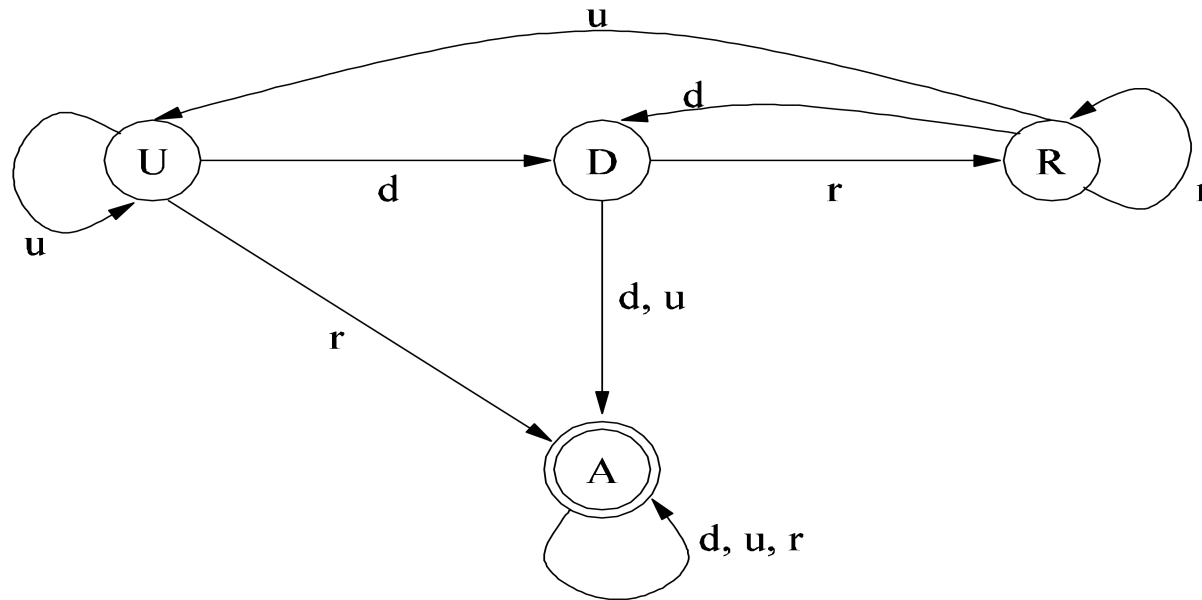
# Data Flow Anomaly

- Type 1: Defined and then defined again (Example 1 above)
  - Four interpretations of Example 1
    - The first statement is redundant.
    - The first statement has a fault -- the intended one might be:  $w = f1(y)$ .
    - The second statement has a fault -- the intended one might be:  $v = f2(z)$ .
    - There is a missing statement in between the two:  $v = f3(x)$ .
  - Note: It is for the programmer to make the desired interpretation.
- Type 2: Undefined but referenced
  - Example:  $x = x - y - w$ ; /\* w has not been defined by the programmer. \*/
  - Two interpretations
    - The programmer made a mistake in using w.
    - The programmer wants to use the compiler assigned value of w.
- Type 3: Defined but not referenced
  - Example: Consider  $x = f(x, y)$ . If x is not used subsequently, we have a Type 3 anomaly.

# Data Flow Anomaly

- The concept of **a state-transition diagram** is used to **model a program variable** to identify data flow anomaly.
- Components of the state-transition diagrams
  - The states
    - U: Undefined
    - D: Defined but not referenced
    - R: Defined and referenced
    - A: Abnormal
  - The actions
    - *d*: define the variable
    - *r*: reference (or, read) the variable
    - *u*: undefine the variable

# Data Flow Anomaly



Legends:

States

U: Undefined

D: Defined but not referenced

R: Defined and referenced

A: Abnormal

Actions

d: Define

r: Reference

u: Undefine

Figure 1: State transition diagram of a program variable [10] (©[1979] IEEE).

# Data Flow Anomaly

- Obvious question: What is the relationship between the **Type 1**, **Type 2**, and **Type 3** anomalies and Figure 1?
- The three types of anomalies (Type 1, Type 2, and Type 3) are found in the diagram in the form of **action sequences**:
  - Type 1: *dd*
  - Type 2: *ur*
  - Type 3: *du*
- Detection of data flow anomaly via program instrumentation
  - Program instrumentation: Insert new code to monitor the states of variables.
  - If the state sequence contains *dd*, *ur*, or *du* sequence, a data flow anomaly is said to occur.
- Bottom line: What to do after detecting a data flow anomaly?
  - Investigate the cause of the anomaly.
  - To fix an anomaly, write new code or modify the existing code.



# Overview of Dynamic Data Flow Testing

- A programmer manipulates/uses variables in several ways.
  - Initialization, assignment, using in a computation, using in a condition
- Motivation for data flow testing?
  - One should not feel confident that a variable has been **assigned the correct value**, if no test causes the execution of a **path** from the point of assignment to a point where the value is **used**.
  - Note
    - Assignment of correct value means whether or not a value has been correctly generated.
    - Use of a variable means
      - If new values of the same variable or other variables are generated.
      - If the variable is used in a conditional statement to alter the flow of control.
- The above motivation indicates that **certain kinds of paths** are executed in data flow testing.

# Overview of Dynamic Data Flow Testing

- Data flow testing is outlined as follows:
  - Draw a data flow graph from a program.
  - Select one or more data flow testing criteria.
  - Identify paths in the data flow graph satisfying the selection criteria.
  - Derive path predicate expressions from the selected paths .
  - Solve the path predicate expressions to derive test inputs.

# Data Flow Graph

- Occurrences of variables
  - Definition: A variable gets a new value.
    - `i = x; /* The variable i gets a new value. */`
  - Undefined or kill: This occurs if the value and the location become unbound.
  - Use: This occurs when the value is fetched from the memory location of the variable. There are **two forms** of uses of a variable.
    - Computation use (c-use)
      - Example: `x = 2*y; /* y has been used to compute a value of x. */`
    - Predicate use (p-use)
      - Example: `if (y > 100) { ...} /* y has been used in a condition. */`

# Data Flow Graph

- A data flow graph is a directed graph constructed as follows.
  - A sequence of **definitions** and **c-uses** is associated with each **node** of the graph.
  - A set of **p-uses** is associated with each **edge** of the graph.
  - The entry node has a definition of each edge parameter and each nonlocal variable used in the program.
  - The exit node has an undefinition of each local variable.

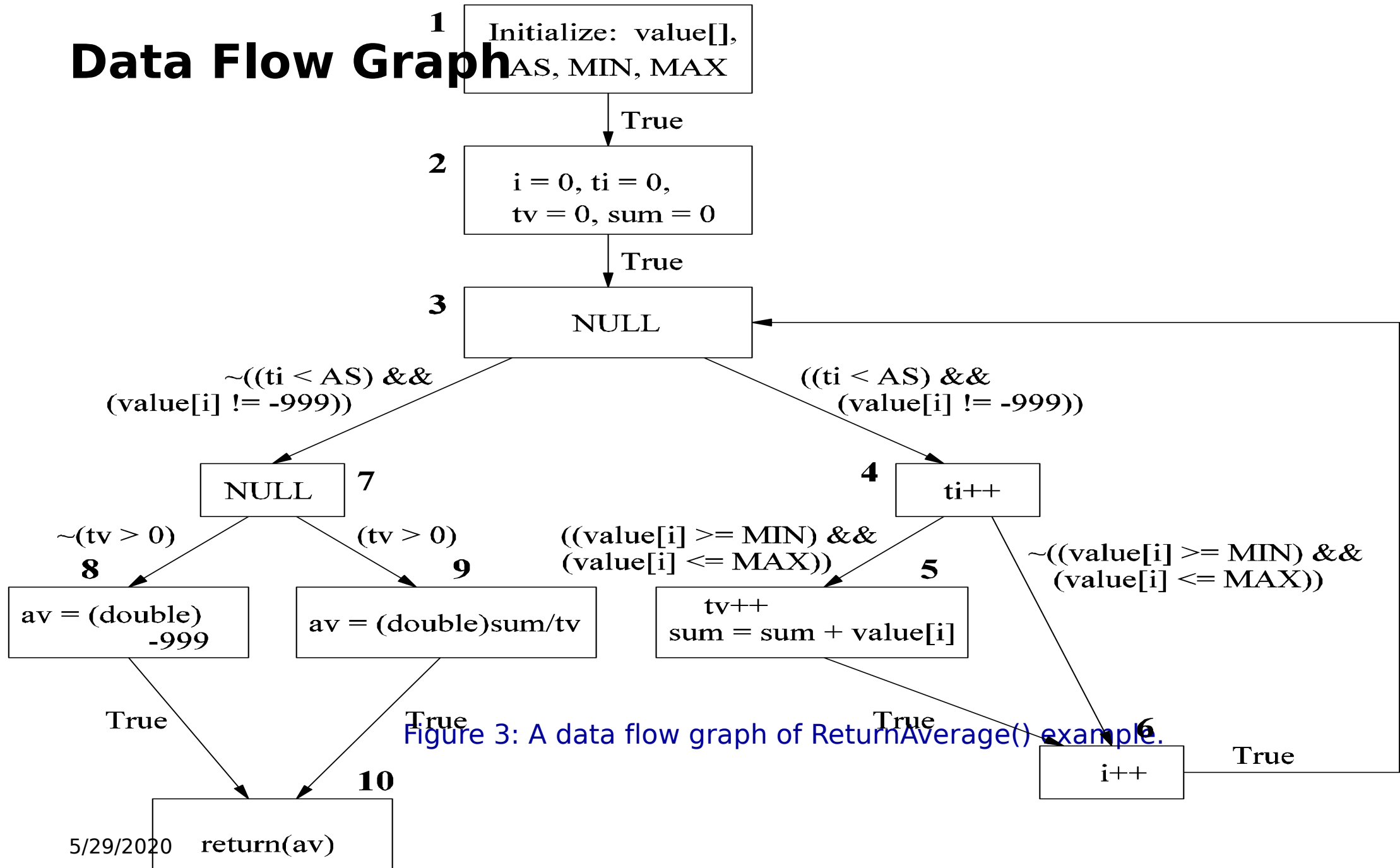
# Data Flow Graph

- **Example code: ReturnAverage() from Chapter 4**

```
public static double ReturnAverage(int value[], int AS, int MIN, int MAX){  
    /* Function: ReturnAverage Computes the average of all those numbers in the input array in  
    the positive range [MIN, MAX]. The maximum size of the array is AS. But, the array size  
    could be smaller than AS in which case the end of input is represented by -999. */  
  
    int i, ti, tv, sum;  
    double av;  
    i = 0; ti = 0; tv = 0; sum = 0;  
    while (ti < AS && value[i] != -999) {  
        ti++;  
        if (value[i] >= MIN && value[i] <= MAX) {  
            tv++;  
            sum = sum + value[i];  
        }  
        i++;  
    }  
    if (tv > 0)  
        av = (double)sum/tv;  
    else  
        av = (double) -999;  
    return (av);  
}
```

Figure 2: A function to compute the average of selected integers in an array.

# Data Flow Graph



# Data Flow Graph

- A data flow graph is a directed graph constructed as follows.
  - A sequence of **definitions** and **c-uses** is associated with each **node** of the graph.
  - A set of **p-uses** is associated with each **edge** of the graph.
  - The entry node has a definition of each edge parameter and each nonlocal variable used in the program.
  - The exit node has an undefinition of each local variable.

# Data Flow Terms

- **Global c-use:** A c-use of a variable  $x$  in node  $i$  is said to be a global c-use if  $x$  has been defined before in a node other than node  $i$ .
  - Example: The c-use of variable  $tv$  in node 9 (Figure 3) is a global c-use.
- **Definition clear path:** A path  $(i - n_1 - \dots - n_m - j)$ ,  $m \geq 0$ , is called a definition clear path (def-clear path) with respect to variable  $x$ 
  - from node  $i$  to node  $j$ , and
  - from node  $i$  to edge  $(n_m, j)$ ,
  - if  $x$  has been neither defined nor undefined in nodes  $n_1 - \dots - n_m$ .
  - Example:  $(2 - 3 - 4 - 6 - 3 - 4 - 6 - 3 - 4 - 5)$  is a def-clear path w.r.t.  $tv$  in Fig. 5.4.
  - Example:  $(2 - 3 - 4 - 5)$  and  $(2 - 3 - 4 - 6)$  are def-clear paths w.r.t. variable  $tv$  from node 2 to 5 and from node 2 to 6, respectively, in Fig. 3.



# Data Flow Terms

- **Global definition:** A node  $i$  has a global definition of variable  $x$  if node  $i$  has a definition of  $x$  and there is a def-clear path w.r.t.  $x$  from node  $i$  to some  
node containing a global c-use, or  
edge containing a p-use of  
variable  $x$ .
- **Simple path:** A simple path is a path in which all nodes, except possibly the first and the last, are distinct.
  - Example: Paths (2 – 3 – 4 – 5) and (3 – 4 – 6 – 3) are simple paths.
- **Loop-free paths:** A loop-free path is a path in which all nodes are distinct.
- **Complete path:** A complete path is a path from the entry node to the exit node.

# Data Flow Terms

- **Du-path:** A path  $(n_1 - n_2 - \dots - n_j - n_k)$  is a du-path path w.r.t. variable  $x$  if node  $n_1$  has a global definition of  $x$  and either
  - node  $n_k$  has a global c-use of  $x$  and  $(n_1 - n_2 - \dots - n_j - n_k)$  is a def-clear simple path w.r.t.  $x$ , or
  - Edge  $(n_j, n_k)$  has a p-use of  $x$  and  $(n_1 - n_2 - \dots - n_j - n_k)$  is a def-clear, loop-free path w.r.t.  $x$ .
- Example: Considering the global definition and global c-use of variable  $tv$  in nodes 2 and 5, respectively,  $(2 - 3 - 4 - 5)$  is a du-path.
- Example: Considering the global definition and p-use of variable  $tv$  in nodes 2 and on edge  $(7, 9)$ , respectively,  $(2 - 3 - 7 - 9)$  is a du-path.

# Data Flow Testing Criteria

Seven data flow testing criteria

All-defs

All-c-uses

All-p-uses

All-p-uses/some-c-uses

All-c-uses/some-p-uses

All-uses

All-du-paths

# Data Flow Testing Criteria

- **All-defs**

- For **each variable**  $x$  and **each node**  $i$ , such that  $x$  has a global definition in node  $i$ , select a complete path which includes a def-clear path from node  $i$  to
  - node  $j$  having a global c-use of  $x$ , or
  - edge  $(j, k)$  having a p-use of  $x$ .
- Example (**partial**): Consider  $tv$  with its global definition in node 2. Variable  $tv$  has a global c-use in node 5, and there is a def-clear path  $(2 - 3 - 4 - 5)$  from node 2 to node 5. Choose a complete path  $(1 - \underline{2 - 3 - 4 - 5} - 6 - 3 - 7 - 9 - 10)$  that includes the def-clear path  $(2 - 3 - 4 - 5)$  to satisfy the all-defs criterion.

# Data Flow Testing Criteria

- **All-c-uses**

- For **each variable**  $x$  and **each node**  $i$ , such that  $x$  has a global definition in node  $i$ , select complete paths which include def-clear paths from node  $i$  to **all** nodes  $j$  such that there is a global c-use of  $x$  in  $j$ .
- Example (**partial**): Consider variable  $t_i$ , which has a global definition in 2 and a global c-use in node 4. From node 2, the def-clear path to 4 is (2 – 3 – 4). One may choose the complete path (1 – 2 – 3 – 4 – 6 – 3 – 7 – 8 – 10). (There three other complete paths.)

- **All-p-uses**

- For **each variable**  $x$  and **each node**  $i$ , such that  $x$  has a global definition in node  $i$ , select complete paths which include def-clear paths from node  $i$  to **all** edges  $(j, k)$  such that there is a p-use of  $x$  on  $(j, k)$ .
- Example (**partial**): Consider variable  $t_v$ , which has a global definition in 2 and p-uses on edges (7, 8) and (7, 9). From node 2, there are def-clear paths to (7, 8) and (7, 9), namely (2 – 3 – 7 – 8) and (2 – 3 – 7 – 9). The two complete paths are: (1 – 2 – 3 – 7 – 8 – 10) and (1 – 2 – 3 – 7 – 9 – 10).

# Data Flow Testing Criteria

- All-p-uses/some-c-uses
  - This criterion is identical to the all-p-uses criterion **except** when a variable  $x$  has no p-use. If  $x$  has no p-use, then this criterion reduces to the some-c-uses criterion.
  - Some-c-uses: For **each variable**  $x$  and **each node**  $i$ , such that  $x$  has a global definition in node  $i$ , select complete paths which include def-clear paths from node  $i$  to **some** nodes  $j$  such that there is a global c-use of  $x$  in  $j$ .
  - Example (**partial**): Consider variable  $i$ , which has a global definition in 2. There is no p-use of  $i$ . Corresponding to the global definition of  $i$  in 2, there is a global c-use of  $i$  in 6. The def-clear path from node 2 to 6 is (2 – 3 – 4 – 5 – 6). A complete path that includes the above def-clear path is (1 – 2 – 3 – 4 – 5 – 6 – 7 – 9 – 10).

# Data Flow Testing Criteria

- **All-c-uses/some-p-uses**
  - This criterion is identical to the all-c-uses criterion **except** when a variable  $x$  has no c-use. If  $x$  has no global c-use, then this criterion reduces to the some-p-uses criterion.
  - Some-p-uses: For **each variable**  $x$  and **each node**  $i$ , such that  $x$  has a global definition in node  $i$ , select complete paths which include def-clear paths from node  $i$  to **some** edges  $(j, k)$  such that there is a p-use of  $x$  on  $(j, k)$ .
- **All-uses**: This criterion produces a set of paths due to the **all-p-uses** criterion **and** the **all-c-uses** criterion.
- **All-du-paths**: For each variable  $x$  and for each node  $i$ , such that  $x$  has a global definition in node  $i$ , select complete paths which include **all du-paths** from node  $i$ 
  - To all nodes  $j$  such that there is a global **c-use** of  $x$  in  $j$ , and
  - To all edges  $(j, k)$  such that there is a **p-use** of  $x$  on  $(j, k)$ .

# Comparison of Data Flow Testing Criteria

- Comparison of two testing criteria  $c_1$  and  $c_2$ 
  - We need a way to compare the two.
- **Includes** relationship: Given two test selection criteria  $c_1$  and  $c_2$ ,  $c_1$  includes  $c_2$  if for every def/use graph, any set of complete paths of the graph that satisfies  $c_1$  also satisfies  $c_2$ .
- **Strictly includes** relationship: Given two test selection criteria  $c_1$  and  $c_2$ ,  $c_1$  strictly includes  $c_2$  provided  $c_1$  includes  $c_2$  **and** for some def/use graph, there is a set of complete paths of the graph that satisfies  $c_2$  but not  $c_1$ .



# Comparison of Data Flow Testing Criteria

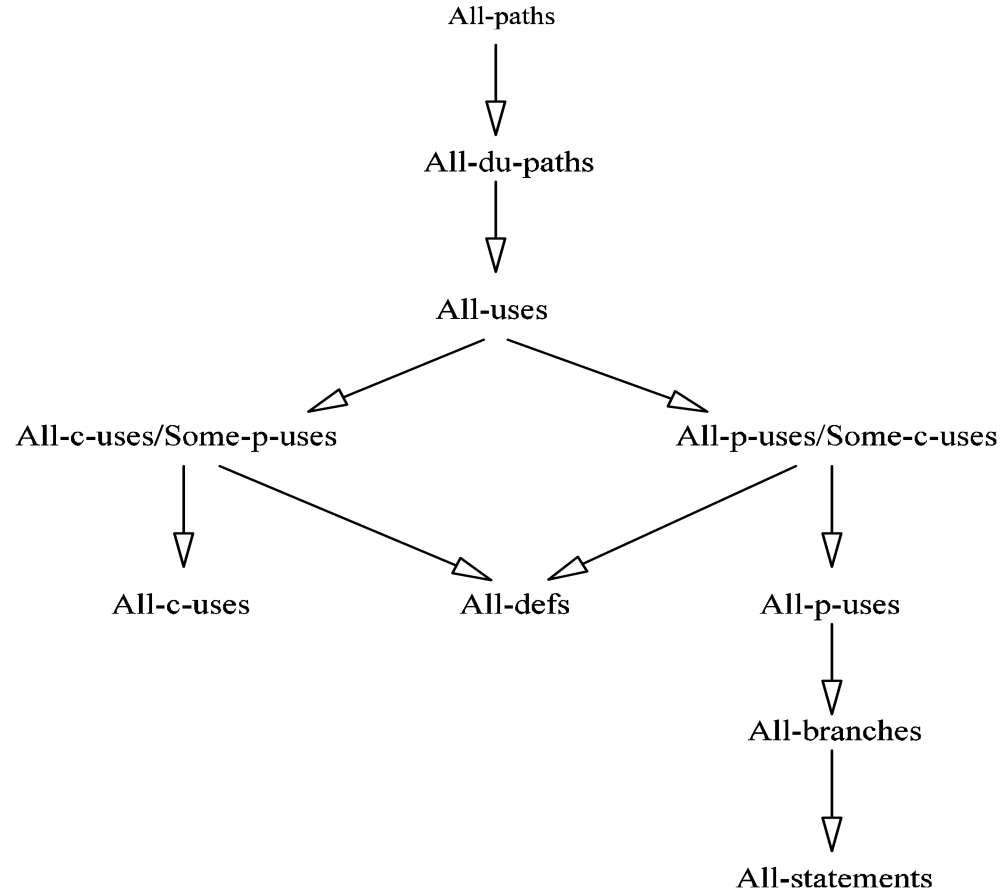


Figure 4: The relationship among DF (data flow) testing criteria [6]  
(©[1988] IEEE).

# Feasible Paths and Test Selection Criteria

- Executable (feasible) path
  - A complete path is executable if there exists an assignment of values to input variables and global variables such that all the path predicates evaluate to true.
- Considering the feasibility of paths, the “includes” relationships of Fig. 4 change to Fig. 5.

# Comparison of Testing Techniques

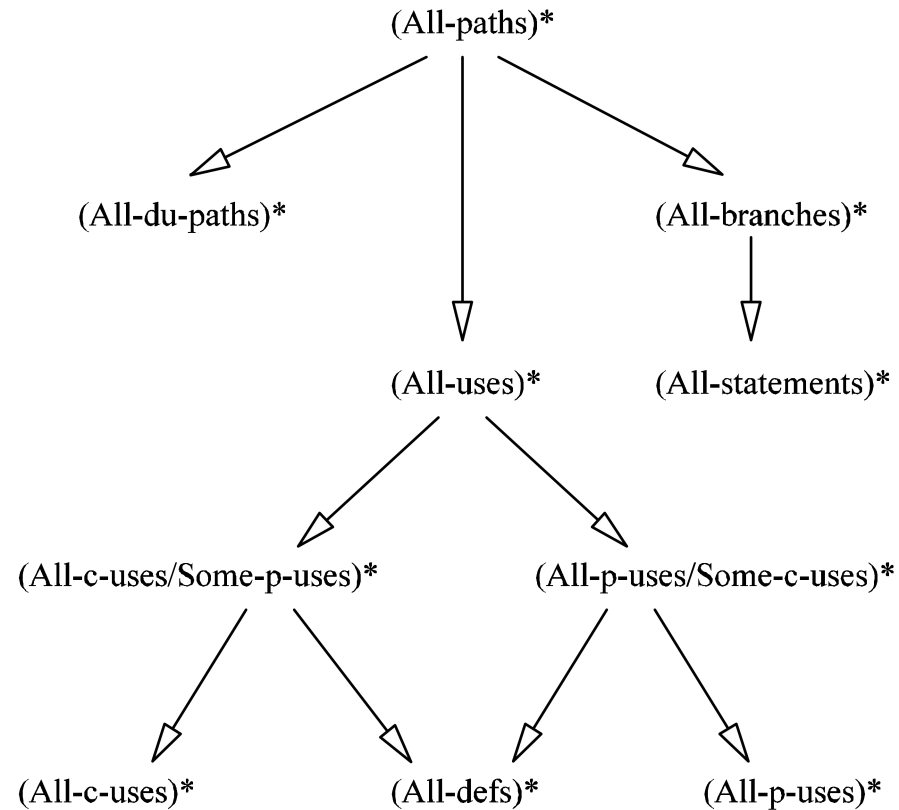


Figure 5: The relationship among the FDF (feasible data flow) testing criteria [6] (©[1988] IEEE).

# Comparison of Testing Techniques

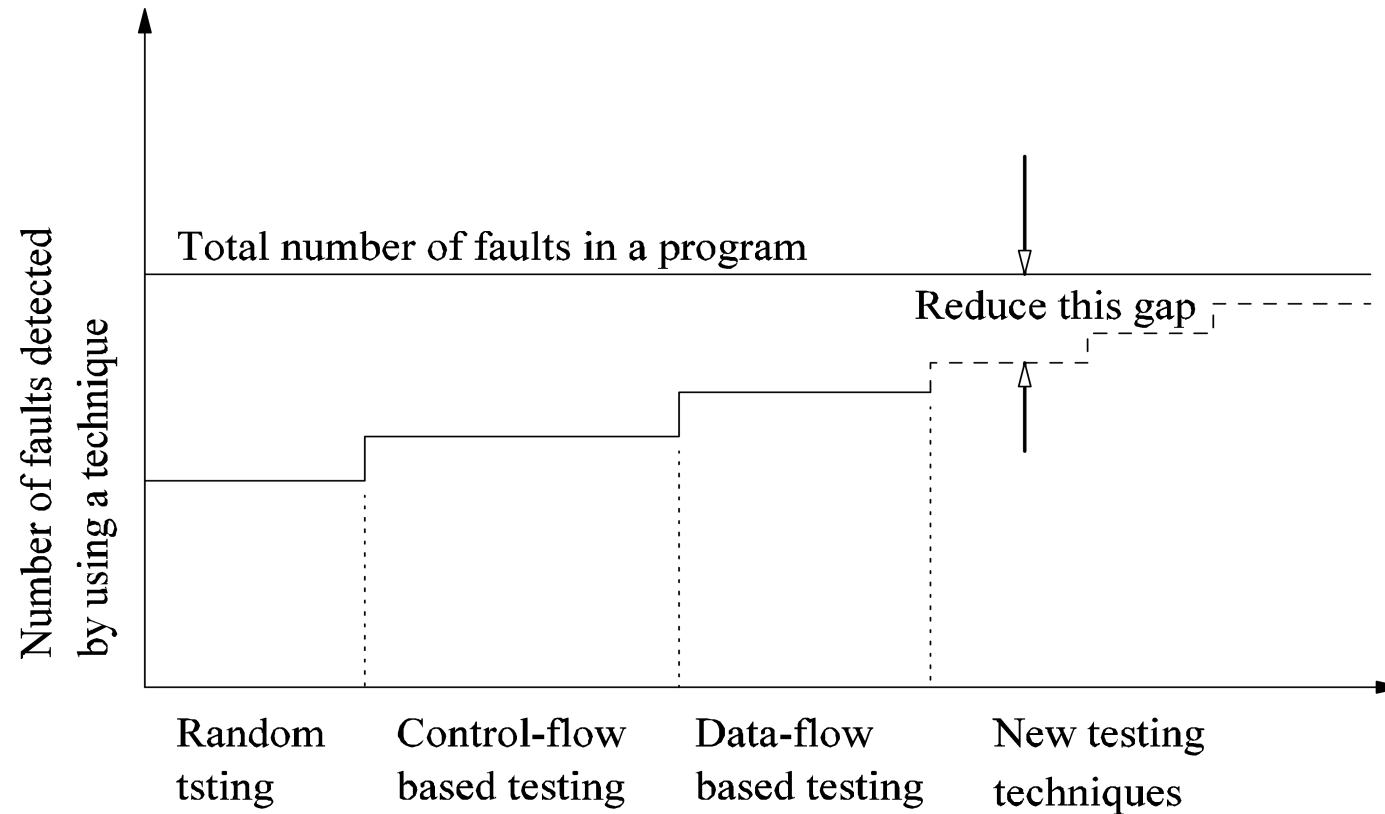


Figure 6: Limitations of different fault detection techniques.

# Summary

- **Data flow** is a readily identifiable concept in a program unit.
- Data flow testing
  - Static
  - Dynamic
- Static data flow analysis
  - Data flow **anomaly** can occur due to programming errors.
  - **Three** types of data flow anomalies
    - (Type 1: *dd*), (Type 2: *ur*), (Type 3, *du*)
    - **Analyze** the code to **identify** and **remove** data flow anomalies.
- Dynamic data flow analysis
  - Obtain a data flow graph from a program unit.
  - Select paths based on DF criteria: *all-defs*, *all-c-uses*, *all-p-uses*, *all-uses*, *all-c-uses/some-p-uses*, *all-p-uses/some-c-uses*, *all-du-paths*.
  - The **includes** relationship is useful in comparing selection criteria.