Software Testing and Reliability

Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
计算机学院E301

Lecture 8

White-box Testing
(Data Flow Coverage)

Data flow coverage

- Generate test cases according to the pattern of data usage inside the software
- Three types of data usage

    -- Define

        -- The variable is given a value, for example, **X**= 5;

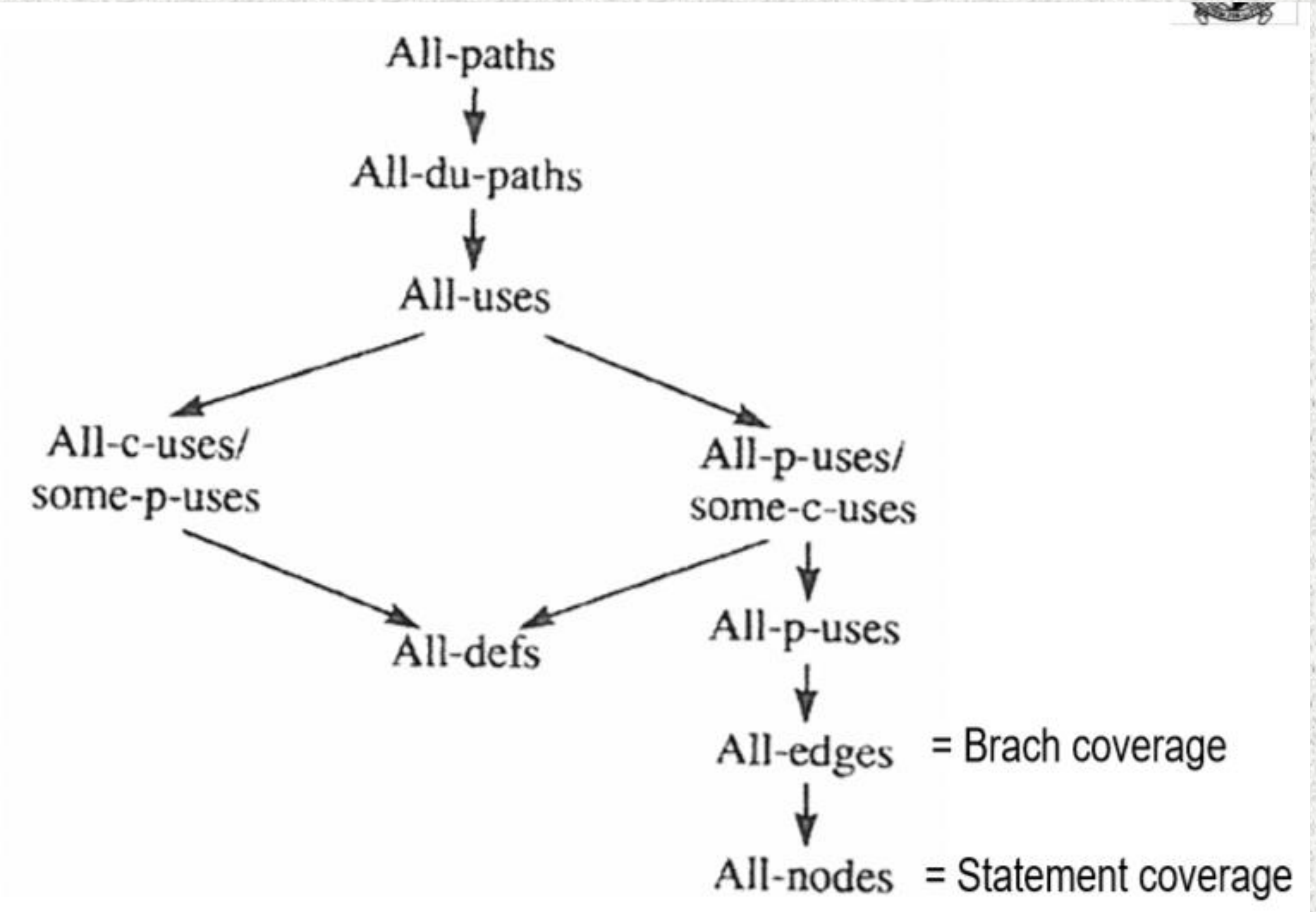
    -- Predicate use

        -- The variable's value is used to decide the TRUE or FALSE outcome of a predicate, for example, if(**X**>3)

    -- Computational use

        -- The variable's value is used for computation, for example, Y = **X**+ 1

Partial ordering of coverage criteria



Example for Def-Use pair coverage

Example:

S1: X = ……

S2: WHILE ….. DO { …………..

S3: X= X* I}

S4: Y = X

What are the def-use pairs for variable X, I and Y?

Example for Def-Use pair coverage (continued)

- Regarding to variable X, “def” occurs in S1 and S3, while “use” occurs in S3 and S4.
- For variable X, we have 4 pairs to cover (S1, S3), (S1, S4), (S3, S3) and (S3, S4)

Example for Def-Use pair coverage (continued)

Without entering the loop – cover (S1, S4)

S1: X = ……

S2: WHILE ….. DO { …………

S3: X = X * I}

S4: Y = X

Example for Def-Use pair coverage (continued)

One loop – cover (S1, S3) and (S3->S4)

```
S1:   X = ......
S2:   WHILE ..... DO { ...... .....
S3:                         X = X * I}
S4:   Y = X
```

Example for Def-Use pair coverage (continued)

More than one loops – cover (S1, S3), (S3, S3) and

(S3 ->S4)

S1: X = ...

S2: WHILE ..... DO { ....

S3: X = X * I}

S4: Y = X

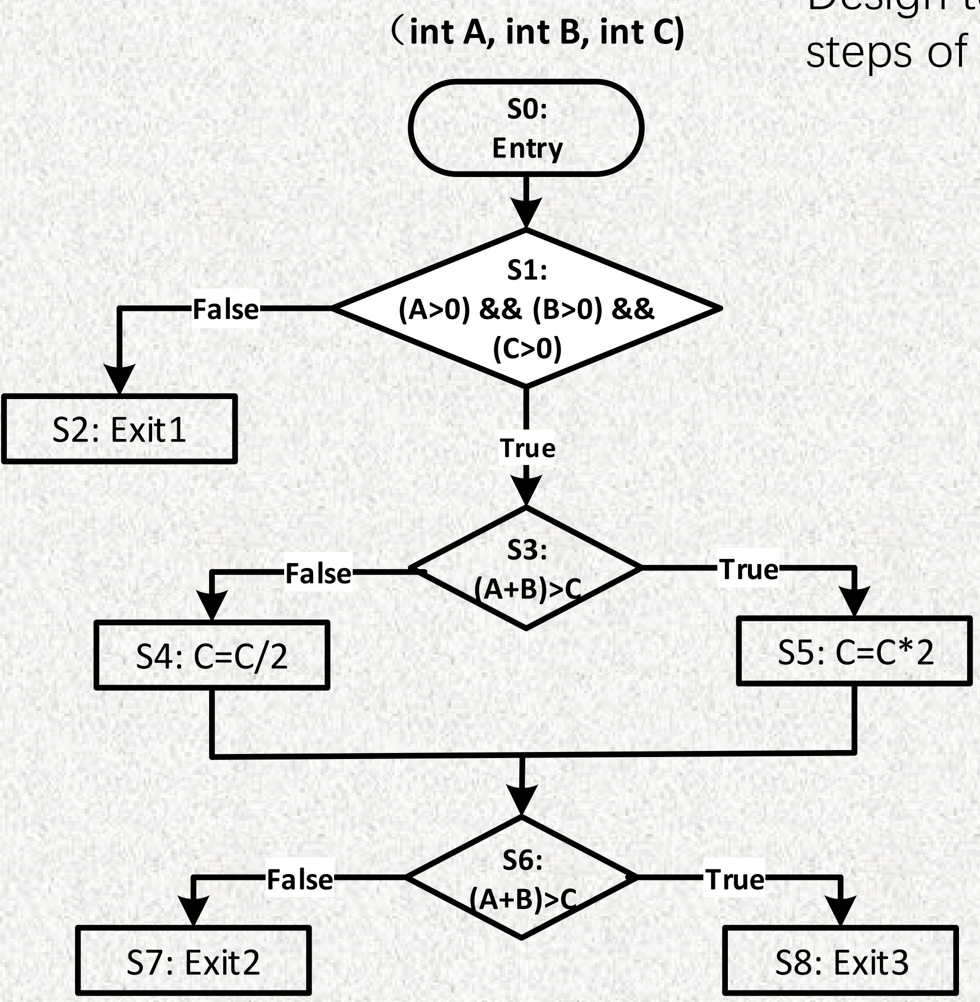
Example for Def-Use pair coverage

- Without entering the loop – cover (S1, S4) √
- ~~One loop – cover (S1, S3) and (S3->S4)~~
- More than one loops – cover (S1, S3), (S3, S3) and (S3->S4) √

Two paths are sufficient to cover all def-use pairs of variable x

About the quiz

Control flow coverage and data flow coverage



Design test cases for path coverage, and write down the steps of your calculation

Input：A=?, B=?, C=?	Covered path
A = -2, B = 3, C=-6	<S0, S1, S2>
A = 5, B = 6, C = 60	<S0, S1, S3, S4, S6, S7>
A = 5, B = 6, C = 14	<S0, S1, S3, S4, S6, S8>
A = 5, B = 6, C = 3	<S0, S1, S3, S5, S6, S8>
A = 5, B = 7, C = 11	<S0, S1, S3, S5, S6, S7>

- Path: <S0, S1, S2>, Path condition: any integers A, B and C that satisfy (A<=0) || (B<=0) || (C<=0)
- Path: <S0, S1, S3, S4, S6, S7>. Path condition: any integers A, B and C that satisfy (A>0) && (B>0) && (C>0) && (A+B)<=C && (A+B)<=(C/2)
- Path <S0, S1, S3, S4, S6, S8>, Path condition: any integers A, B and C that satisfy (A>0) && (B>0) && (C>0) && (A+B)<=C && (A+B)>(C/2)
- Path: <S0, S1, S3, S5, S6, S8>, Path condition: any integers A, B and C that satisfy (A>0) && (B>0) && (C>0) && (A+B)>C && (A+B)>2C
- Path: <S0, S1, S3, S5, S6, S7>, Path condition: any integers A, B and C that satisfy (A>0) && (B>0) && (C>0) && (A+B)>C && (A+B)<=2C

Similar to other coverage criteria

Data Flow Analysis

Data Flow Analysis

- NOT the same as the data flow coverage testing
- A testing method for detecting improper use of variables in programs
  - -- "Improper use of a variable" = "improper sequence of actions on a variable"

Data Flow Analysis (continued)

- Three possible actions

    -- Define - d

        -- The variable is assigned a value

    -- Reference - r

        -- The variable's value is referred

    -- Undefine - u

        -- The variable is declared but not yet assigned any value

        -- The variable's value is destroyed

        -- The variable's value goes out of the scope

Data Flow Analysis (continued)

- **Data Flow Anomalies**

    -- **undefine-reference (called ur anomaly)**

        -- A variable has not been assigned any value, but its value is referred in the program

    -- **define-define (called dd anomaly)**

        -- A variable's value has been defined but not used before another value is assigned to it

    -- **define-undefine (called du anomaly)**

        -- A variable's value has been defined but not used before the value is destroyed

Data Flow Anomalies

- Suppose a program accepts marks and numOfMarks parameters, and one of its functions is to calculate the total marks.

```
{
  int m = 0;
  double total = 0;
  do {
    total = total + marks[m];
    m = m + 1;
  }while(m < numOfMarks);
  return total;
}
```

- **Faulty version:**

```
{
  int m = 0;
  do {
    double total = 0;
    total = total + marks[m];
    m = m + 1;
  }while(m < numOfMarks);
  return total;
}
```

- total is only declared inside the do-while loop. After exiting the loop, total is undefined. But the program tries to return total at the end of its function.
- There is an ur anomaly for the total variable.

Data Flow Anomalies (continued)

- Suppose a program accepts marks and numOfMarks parameters, and one of its functions is to calculate the total marks.

```
{
    int m = 0;
    double total = 0;
    do {
      total = total + marks[m];
      m = m + 1;
    }while(m < numOfMarks);
    return total;
}
```

- **Faulty version:**

```
{
    int m = 0;
    double total = 0;
    do {
      total = marks[m];
      m = m + 1;
    }while(m < numOfMarks);
    return total;
}
```

- total has been defined consecutively without being referenced.
- There is a dd anomaly for the total variable.

Data Flow Anomalies (continued)

- Suppose a program accepts marks and numOfMarks parameters, and one of its functions is to calculate the total marks

```
{
   int m = 0;
   double total = 0;
   do {
     total = total + marks[m];
     m = m + 1;
   }while(m < numOfMarks);
   return total;
}
```

- **Faulty version:**

```
{
   int m = 0;
   double total = 0;
   do {
     double total = marks[m];
     m = m + 1;
   }while(m < numOfMarks);
   return total;
}
```

- The second total has been declared and assigned values inside the do-while loop repeatedly, but never been used before it is destroyed.
- There is a du anomaly for the total variable.

Data Flow Anomalies (continued)

- Suppose the correct statements should be:

```
.
.
input M
input N
I = 2*N
.
.
```

(Mistyping of Names)

```
input N  /* mistype M as N */
input N
I = 2*N
```

What anomaly can you detect?

Data Flow Anomalies (continued)

- Suppose the correct statements should be:

```
.
.
input M
input N
I = 2*N
.
.
```

(Omission of Statements)

```
input M
/*omit the statement "input N"*/
I = 2*N
```

What anomaly can you detect?

Data Flow Anomalies (continued)

- Improper uses of data

  -- questionable coding

  -- bad programming practices

  -- not necessarily program errors

Programming Errors Lead to Anomalies

- Misspelling of identifiers
- Uninitialized variables
- Misplacement of statements
- Omission of statements
- Passing of incorrect parameters
- Incorrect pointers

Approaches

- Static analysis

  -- performs the analysis without executing the program

- Dynamic analysis

  -- analysis is performed by executing an instrumented version of the program

Static Data Flow Analysis

- Automation of the analysis process

  -- **Scan through** the source code of software under test

  -- **Analyze** the variable use inside the program

  -- "Automation" $\neq$ "Execution of the program under analysis"

- Two supporting tools

  -- DAVE - a static data flow analysis system for FORTRAN programs

  -- Lint – a C program checker, which provides elementary static data flow analysis

Static Data Flow Analysis (continued)

- Useful when

  -- The program size is small but the execution time is long

- Limitations.

  -- Problem to detect anomalies when the software contains dynamic attributes, such as the array and pointer

Static Data Flow Analysis (continued)

- Let us analyse the following program

```
int temp;
int v[ ] = new int[100];
....
temp = v[k];
v[k] = v[h];
v[h] = temp;
```

If $h \neq k$, the process here do not contain any data anomaly

[Source: Detection of Data Flow Anomaly Through Program Instrumentation]

Static Data Flow Analysis (continued)

- Since the program under analysis is not executed, the tool will not know the value of h and k. **How to conduct analysis?**
- Possible solution - ignore the array index and just consider all array elements are identical

  -- The tool will consider the program as the following form, and then report detection of the dd anomaly.

```
temp = v;
v = v;
v = temp;
```

False alarm !

[Source: Detection of Data Flow Anomaly Through Program Instrumentation]

Static Data Flow Analysis (continued)

- False alarm

    -- Annoying

    -- Waste programmers' time and effort to re-examine the code

[Source: Detection of Data Flow Anomaly Through Program Instrumentation]

Static Data Flow Analysis (continued)

- Let us analyse another program

```
int h
int v[ ] = new int[100];
....
h = 0;
while h < 100 do begin
   v[k] = v[k+1];
   h = h + 1
end while
```

If a fault happens to be in this statement (that is, v[1] = v[k+1] instead of v[k] = v[k+1]), the program would contain dd anomaly

[Source: Detection of Data Flow Anomaly Through Program Instrumentation]

Static Data Flow Analysis (continued)

- For the previously mentioned fault, if the tool was implemented to ignore the array index and just consider all array elements identical, the tool will consider the program as the following form, and then would not report detection of a **dd** anomaly.

```
while h < 100 do begin
    v = v;
    h = h + 1
end while
```

False positive !

[Source: Detection of Data Flow Anomaly Through Program Instrumentation]