Software Testing

Lecture Notes 2 (of 4)

Outline of the Lecture

- **White box testing**
  (Glass box testing, Open box testing, Clear box testing, Structural testing)
  - **Control flow testing**
  - **Data flow testing**
- **Regression testing**

White box testing

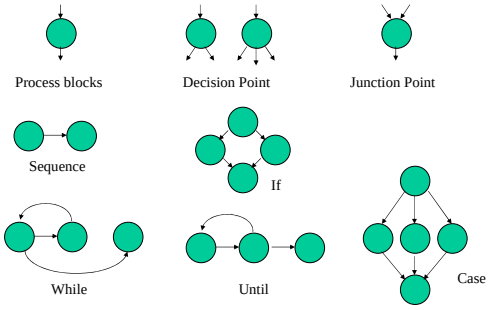
- logical decision
- loops
- internal data structure
- paths
- ...

Coverage!!

White-box Testing Techniques

- Definition: a strategy in which testing is based on the internal paths, structure, and implementation of the software under test (SUT)
- Applicability: all levels of system development (path testing!)
  - Unit
  - Integration
  - System
  - Acceptance
- Disadvantages: 1) number of execution paths may be so large; 2) test cases may not detect data sensitivity; 3) assumes that control flow is correct (nonexistent paths!); 4) tester must have programming skills.
- Advantages: tester can be sure that every path have been identified and tested.

Control Flow Graphs



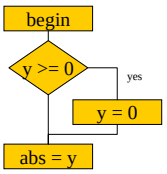
Definition: Given a program written in an imperative programming language, its program graph is a directed graph in which nodes are statement fragments, and edges represent flow of control (a complete statement is a “default” statement fragment).

Levels of Coverage (test coverage metrics)

1. Statement (Line) coverage
2. Decision (Branch) coverage
3. Condition coverage
4. Decision/Condition coverage
5. Multiple Condition coverage
6. Loop coverage
7. Path coverage

Statement Coverage

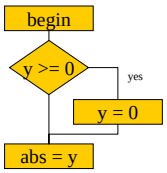
```
Begin
if ( y >=  0)
        then y = 0;
abs = y;
end;
```



test case-1(yes):
input: y = ?
expected result: ?
actual result: ?

Statement Coverage

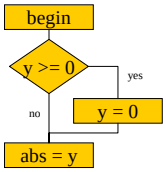
```
Begin
if ( y >=  0)
        then y = 0;
abs = y;
end;
```



test case- (yes):
input: y = 0
expected result: 0
actual result: 0

Branch Coverage

```
Begin
if ( y >=  0)
        then y = 0;
abs = y;
end;
```

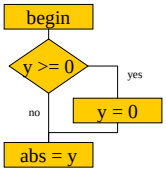


test case-1(yes):
input: y = 0
expected result: 0
actual result: 0

test case-2 (no) :
input: y = ?
expected result: ?
actual result: ?

Branch Coverage

```
Begin
if ( y >=  0)
        then y = 0;
abs = y;
end;
```

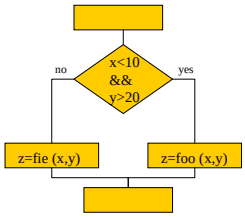


test case-1(yes):
input: y = 0
expected result: 0
actual result: 0

test case-2 (no) :
input: y = -5
expected result: 5
actual result: -5

Condition Coverage

```
Begin
if ( x < 10 && y > 20) {
z = foo (x,y); else z =fie (x,y);
}
end;
```

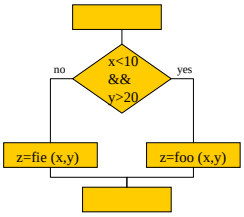


test case-1(T,F):
input: x = ?, y = ?
expected result: ?
actual result: ?

test case-2 (F;T):
input: x = ?, y = ?
expected result: ?
actual result: ?

Condition Coverage

```
Begin
if ( x < 10 && y > 20) {
z = foo (x,y); else z =fie (x,y);
}
end;
```

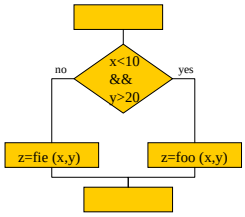


test case- (T,F):
input: x = -4, y = 12
expected result: ?
actual result: ?

test case-2 (F,T):
input: x = 12, y = 30
expected result: ?
actual result: ?

Decision/Condition Coverage

```
Begin
if ( x < 10 && y > 20) {
z = foo (x,y); else z =fie (x,y);
}
end;
```

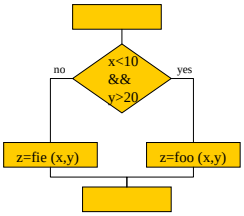


test case-1 (T,T,yes):
input: x = ?, y = ?
expected result: ?
actual result: ?

test case-2 (F,F,no):
input: x = ?, y = ?
expected result: ?
actual result: ?

Decision/Condition Coverage

```
Begin
if ( x < 10 && y > 20) {
z = foo (x,y); else z =fie (x,y);
}
end;
```

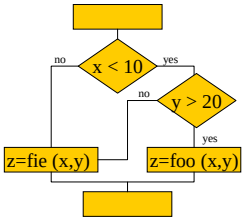


test case-1 (T,T,yes):
input: x = -4, y = 30
expected result: ?
actual result: ?

test case-2 (F,F,no):
input: x = 12, y = 12
expected result: ?
actual result: ?

Multiple Condition Coverage

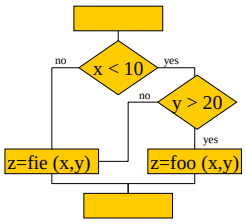
```
Begin
if ( x < 10 && y > 20) {
z = foo (x,y); else z =fie (x,y);
}
end;
```



	x < ?	y > ?
test-case-1:	t	t
test-case:2	t	f
test-case-3:	f	t
test-case-4	f	f

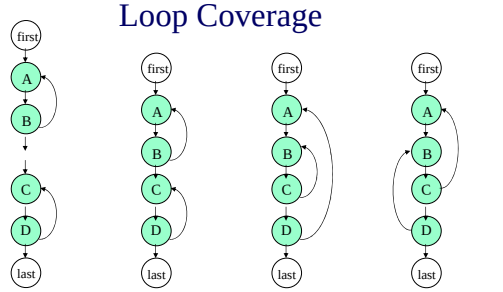
Multiple Condition Coverage

```
Begin
if ( x < 10 && y > 20) {
z = foo (x,y); else z =fie (x,y);
}
end;
```



	x < 10	y > 20
test-case-1:	t	t
test-case:2	t	f
test-case-3:	f	t
test-case-4	f	f

Loop Coverage



Simple | Concatenated | Nested | Knotted (horrible)/ Unstructured

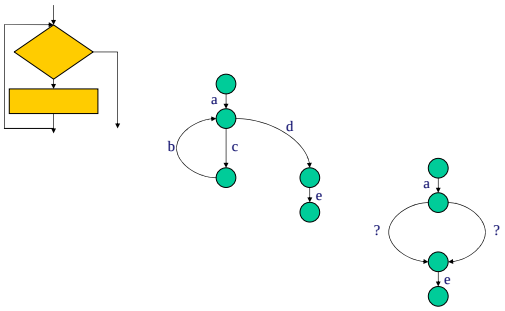
Loop Coverage

- Simple loops :
  - Skip the loop entirely
  - Only one pass through the loop
  - Two passes through the loop
  - *m* passes through the loop when $m < n$
    m: small number representing a typical loop value
    n: maximum number of allowable passes through the loop
  - *n-1*, *n*, *n+1* passes through the loop
- Nested loops:
  - Start at the innermost loop, set all other loops to minimum value.
  - Conduct Simple loop test for innermost loop. Add other tests for out-of-range or excluded values.
  - Continue until all loops have been tested.

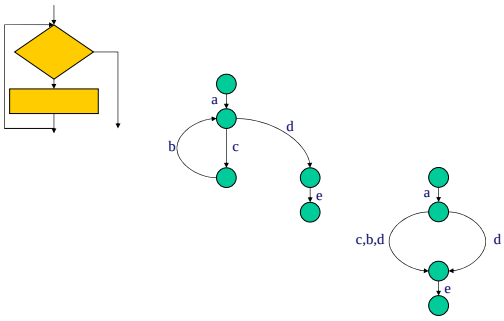
Loop Coverage

- Concatenated loops :
  - If each of the loops is independent of the other, use Simple loops approach
  - If the loops are concatenated and dependent (counter for loop 1 is used as the initial value for loop 2) then use Nested loop approach.
- Knotted (horrible) /unstructured loops:
  - Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

Path with loops



Path with loops



Path Coverage

- All possible execution paths
- Question: How do we know how many paths to look for?
- Answer: The computation of cyclomatic complexity

Cyclomatic Complexity

Cyclomatic Complexity is a **software metric** that provides a quantitative measure of the logical complexity of a program. When used in context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an **upper bound for the number of tests** that must be conducted to ensure that all statements have been executed at least once.

Computation of cyclomatic complexity

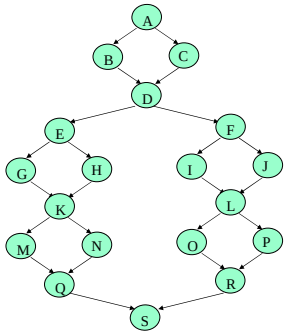
Cyclomatic complexity has a foundation in graph theory and is computed in the following ways:

1. Cyclomatic complexity V(G), for a flow graph, G, is defined as:

   V(G) = E – N + 2
   E: number of edges
   N: number of nodes

2. Cyclomatic complexity V(G), for a flow graph, G, with only binary decisions, is defined as:

   V(G) = P + 1
   P: number of binary decision

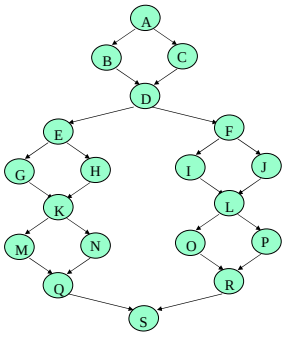


1. V(G) = E – N + 2

E = ?
N = ?
V(G) = ?

2. V(G) = P + 1

P = ?
V(G) = ?



1. V(G) = E – N + 2

E = 24
N = 19
V(G) = 24 – 19 + 2 = 7

2. V(G) = P + 1

P = 6
V(G) = 6 +1 = 7

Independent Paths

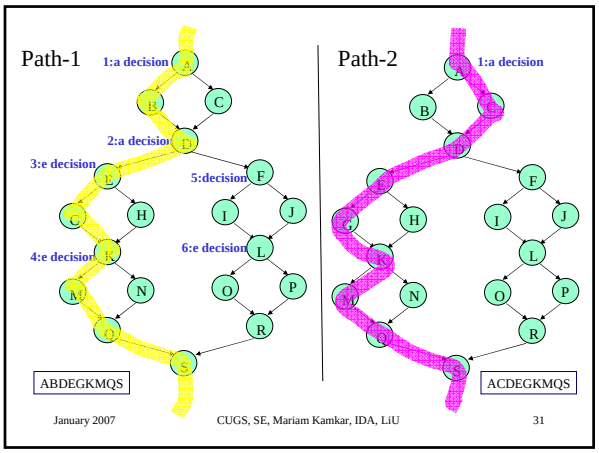
An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined.

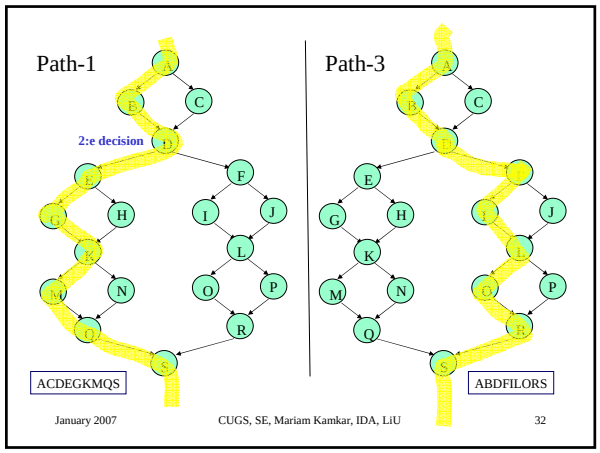
Basis Path Testing

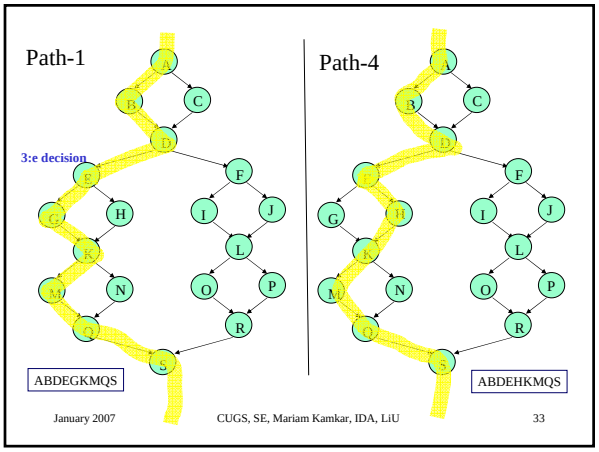
- Derive the control flow graph from the software module.
- Compute the graph's Cyclomatic Complexity of the resultant flow graph.
- Determine a basis set of linearly independent paths.
- Create a test case for each basis path.
- Execute these tests.

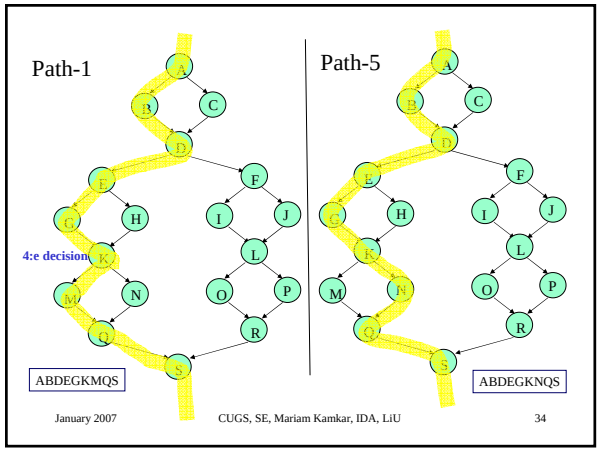
Determine a basis set of linearly independent paths McCabe's baseline method

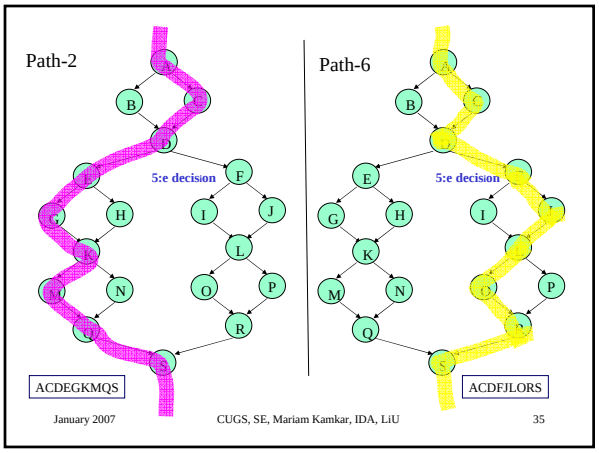
1. Pick a "baseline" path. This path should be a "normal case" program execution. McCabe advises: choose a path with as many decision as possible.
2. To choose the next path, change the outcome of the first decision along the baseline path while keeping the maximum number of other decisions the same as the baseline path.
3. To generate the third path, begin again with the baseline but vary the second decision rather than the first.
4. Repeat the 3 for other paths until all decision along baseline path have been flipped.
5. Now proceed to the second path, flipping its decisions, one by one until the basis path set is completed.

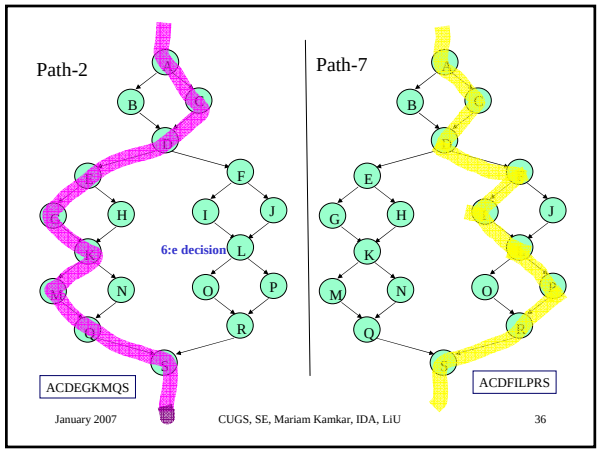

Path-1
1:a decision
2:a decision
3:e decision
5:decision
4:e decision
6:e decision
ABDEGKMQS
Path-2
1:a decision
ACDEGKMQS
January 2007
CUGS, SE, Mariam Kamkar, IDA, LiU
31

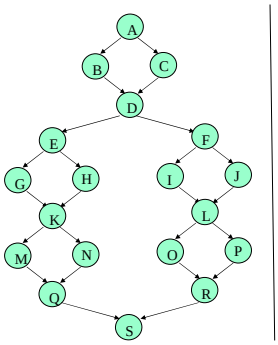

Path-1
2:e decision
ACDEGKMQS
Path-3
ABDFILORS
January 2007
CUGS, SE, Mariam Kamkar, IDA, LiU
32


Path-1
3:e decision
ABDEGKMQS
Path-4
ABDEHKMQS
January 2007
CUGS, SE, Mariam Kamkar, IDA, LiU
33


Path-1
4:e decision
ABDEGKMQS
Path-5
ABDEGKNQS
January 2007
CUGS, SE, Mariam Kamkar, IDA, LiU
34


Path-2
5:e decision
ACDEGKMQS
Path-6
ACDFJLORS
January 2007
CUGS, SE, Mariam Kamkar, IDA, LiU
35


Path-2
6:e decision
ACDEGKMQS
Path-7
ACDFILPRS
January 2007
CUGS, SE, Mariam Kamkar, IDA, LiU
36



Set of basis paths:

1. ABDEGKMQS
2. ACDEGKMQS
3. ABDFILORS
4. ABDEHKMQS
5. ABDEGKNQS
6. ACDFJLORS
7. ACDFILPRS

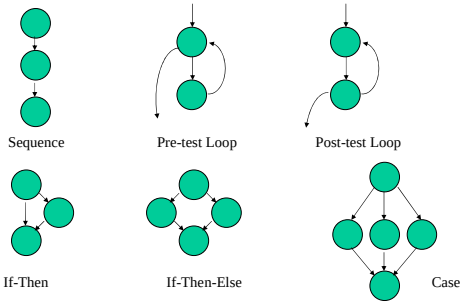
Observation

- Basis path testing calls for the creation of a test case for each of these paths.
- This set of test cases will guarantee both statement and branch coverage.
- Note that multiple sets of basis paths can be created that are not necessarily unique. Each set, however, has the property that a set of test cases based on it will execute every statement and every branch.

Guidelines

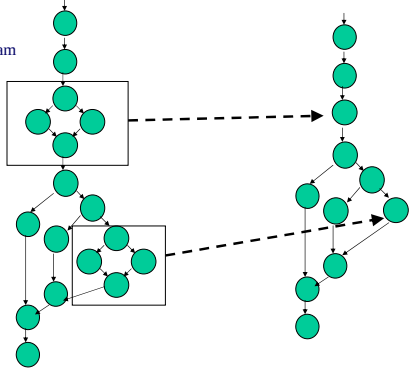
- Program with high cyclomatic complexity require more testing.
- Of the organizations that use the cyclomatic complexity metric, most set some guideline for maximum acceptable complexity; V(G) =10 is a common choice.
- What happens if a unit has higher complexity?
  - Either simplify the unit
  - Or plan to do more testing
- In general, when a program is well structured (i.e., composed solely of the structured programming constructs), it can be reduced to a graph with one path.
- If the unit is well structured, its essential complexity is 1

Structured programming constructs

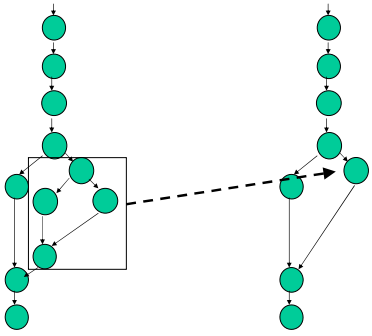


Condensing with respect to the structured programming constructs

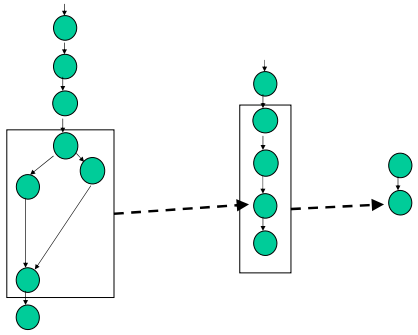
Triangle Program



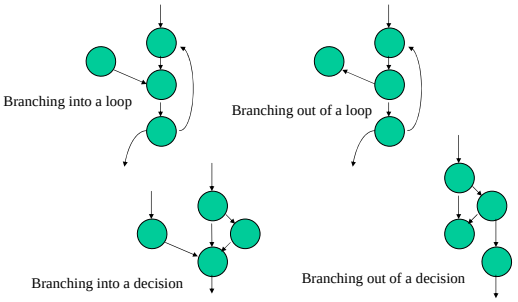
Condensing with respect to the structured programming constructs (cont.)



Condensing with respect to the structured programming constructs (cont.)



Violations of structured programming



Applicability and Limitation

- Control flow testing is the cornerstone of unit testing. It should be used for all modules of code that cannot be tested sufficiently through reviews and inspections.
- Its limitation are that the tester must have sufficient programming skill to understand the code and its control flow.
- Control flow testing can be very time consuming because of all modules and basic paths that comprise a system.

Data Flow Testing

Data flow testing focuses on the points at which variables receive values and the points at which these values are used (or referenced). It detects improper use of data values due to coding errors.

Define/Reference Anomalies

- Early data flow analyses often centered on a set of faults that are known as define/reference anomalies.
  - A variable that is defined but never used (referenced)
  - A variable that is used but never defined
  - A variable that is defined twice before it is used

- **dd**: defined and defined again – not invalid but suspicious
- **du**: defined and used – perfectly correct
- **dk**: defined and then killed – not invalid but probably a programming error
- **ud**: used and defined – acceptable
- **uu**: used and used again – acceptable
- **uk**: used and killed – acceptable
- **kd**: killed and defined – acceptable
- **ku**: killed and used – a serious defect
- **kk**: killed and killed – probably a programming error.

Data flow Graph G(P) = (N, E)

Definition: Given a program (P) written in an imperative programming language, its program graph (G) is a directed graph in which nodes (N) are statement fragments, and edges (E) represent flow of control. In addition it details the **definition**, **use** and **destruction** of each of the module's variable.

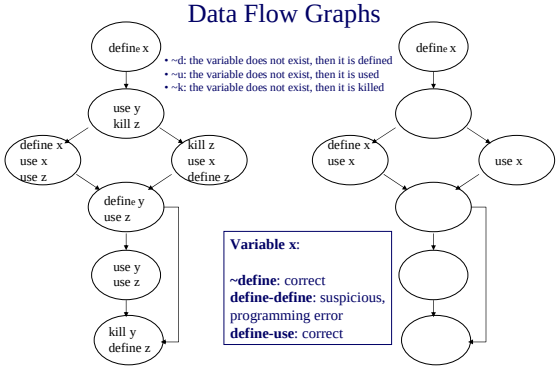
Definitions

- ***DEF(v, n):*** node *n* in *G(P)* is a defining node of variable *v* in *V*, iff the value of variable *v* is defined at the statement fragment corresponding to node *n*.
- ***USE(v, n):*** node *n* in *G(P)* is a usage node of variable *v* in *V*, iff the value of variable *v* is used at the statement fragment corresponding to node *n*.
  - ***P-use, C-use***: a usage node *USE(v, n)* is a predicate use *(P-use)* iff statement *n* is a predicate statement; otherwise, *USE(v, n)* is computation use (*C-use).*

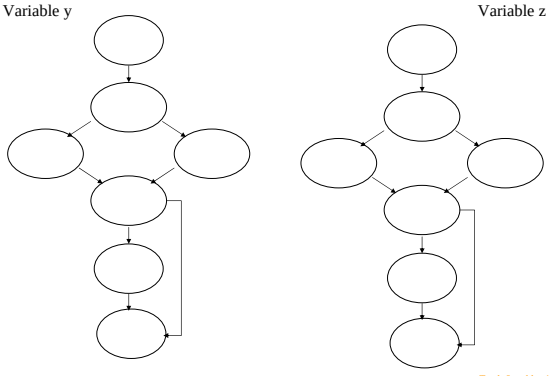
Definitions (cont.)

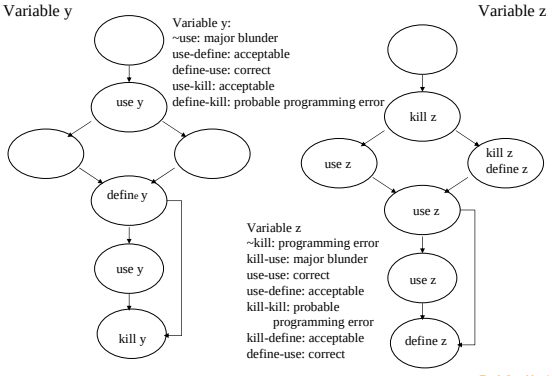
- ***du-path:*** a definition-use path (du-path) with respect to variable *v* is a path in *PATHS(P)* such that, for some *v* in *V*, there are defined and usage nodes *DEF(v, m)* and *USE(v, n)* such that *m* and *n* are initial and final nodes of the path.
- ***dc-path***: a definition-clear path with respect to a variable *v* is a du-path with initial and final node such that no other node in the path is defining node of *v*.

Data Flow Graphs

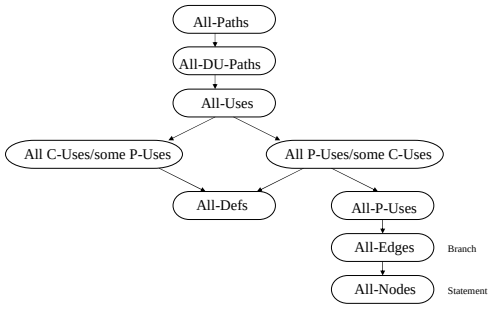


Control flow graph annotated with define-use-kill information for x, y, z





Hierarchy of data flow coverage metrics



Slice-Based Testing

Program slice *S(V, n):* given a program *P* and a set *V* of variables in *P*, a slice on the variable set *V* at statement *n* is the set of all statements in *P* that contribute to the values of variables in *V*.

The idea of slices is to separate a program into components that have some useful (functional) meaning.

Program slice on variable *i*

```
1       s = 0;
2       i = 1;
3       while (i <= n)
        {
4               s + = i;
5               i ++
        }
6       print (s);
7       print (i);
8       print (n);
```

```
2       i = 1;
3       while (i <= n)
        {
5               i ++
        }
7       print (i);
```


- Program slices give us very precise ways to describe parts of a program that we would like to test.
- Obs: good testing practices lead to better programming practices:
  - developing programs in terms of compilable slices,
  - coding and testing one slice
  - coding and testing other slices and merging them into a solid program.

Applicability and Limitations

(data-flow testing)

- It should be used for all modules of code that cannot be tested sufficiently through reviews and inspections.
- Tester must have sufficient programming skill
- Can be very time consuming

Regression testing

- Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.
- Regression testing may be conducted:
  - manually, by re-executing a subset of all test cases
  - using automated capture/playback tools

Regression test

- A regression test is a test applied to a new version or release to verify that it still performs the same functions in the same manner as an old version or release.
- The regression test suite (the subset of tests to be executed) contains three different classes of test cases:
  - A representative sample of tests that will exercise all software functions.
  - Additional tests that focus on software functions that are likely to be affected by the change.
  - Tests that focus on the software components that have been changed.

Obs: it is impractical and inefficient to re-execute every test for every program function once a change has been occurred.