Lecture 15: Testing (Cont.)

Test Design Techniques

- Specification-based Testing
  - Black-box Testing
- Structure-based Testing
  - White-box Testing
- Experience-based Testing

Path Coverage

- Design test cases such that:
  - all linearly independent paths in the program are executed at least once.
  - Combination of branches

Linearly independent paths

- Defined in terms of
  - control flow graph (CFG) of a program.

Control flow graph (CFG)

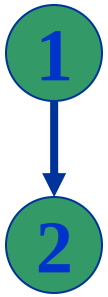
- A control flow graph (CFG) describes:
  - the sequence in which different instructions of a program get executed.
  - the way control flows through the program.

How to draw Control flow graph?

- Number all the statements of a program.
- Numbered statements:
  - represent nodes of the control flow graph.

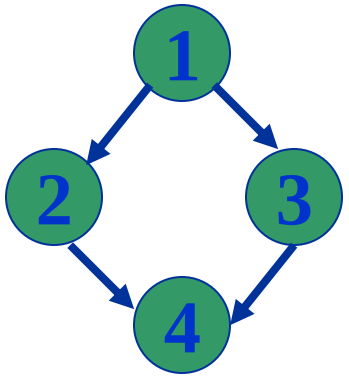
How to draw Control flow graph?

- Sequence:
  - 1 a=5;
  - 2 b=a*b-1;



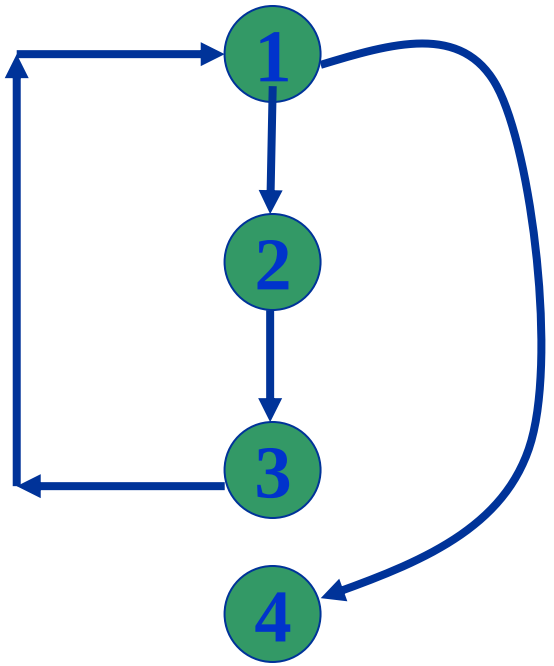
How to draw Control flow graph?

- Selection:
  - 1 if(a>b) then
  - 2 c=3;
  - 3 else c=5;
  - 4 c=c*c;



How to draw Control flow graph?

- Iteration:
  - 1 while(a>b){
  - 2 b=b*a;
  - 3 b=b-1;}
  - 4 c=b+d;



Path

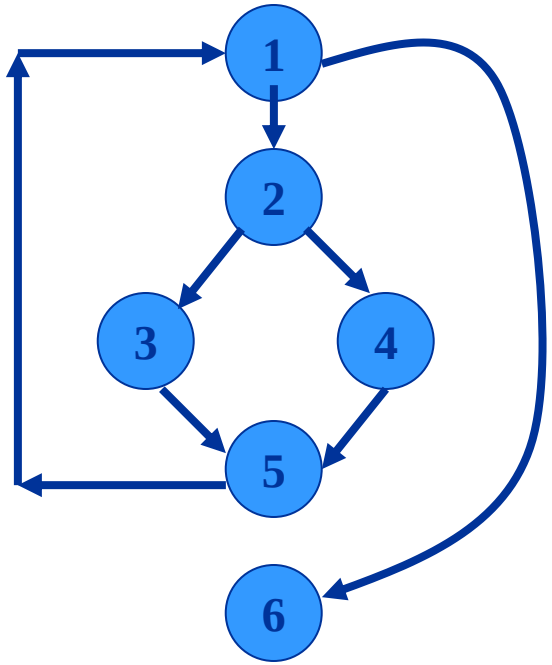
- A path through a program:
  - a node and edge sequence from the starting node to a terminal node of the control flow graph.
  - There may be several terminal nodes for program.

Derivation of Test Cases

- Draw control flow graph.
- Determine V(G).
- Determine the set of linearly independent paths.
- Prepare test cases:
  - to force execution along each path.

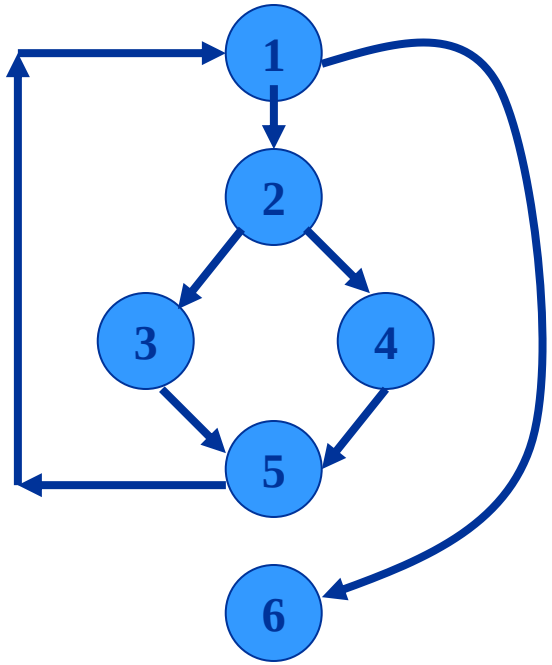
Example

- int f1(int x,int y){
- 1 while (x != y){
- 2 if (x>y) then
- 3 x=x-y;
- 4 else y=y-x;
- 5 }
- 6 return x; }



Derivation of Test Cases

- Number of independent paths: 3
  - 1,6 test case (x=1, y=1)
  - 1,2,3,5,1,6 test case(x=1, y=2)
  - 1,2,4,5,1,6 test case(x=2, y=1)



Dynamic Data Flow Testing

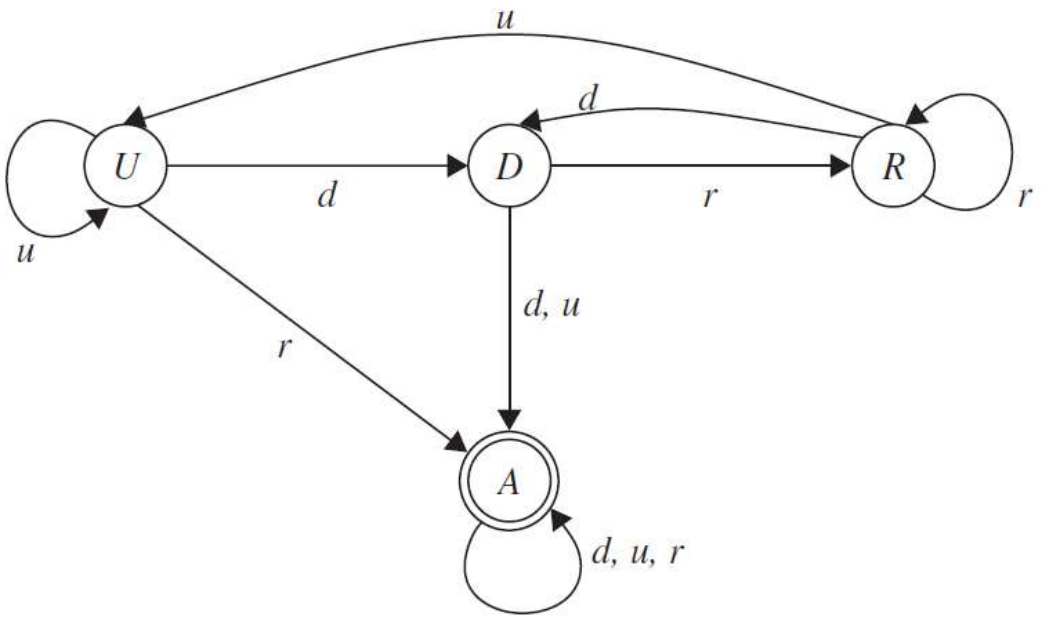
- Motivation
  - How do you know that a variable is assigned the correct value?
  - From: when the value is assigned
  - To: when the value is used later
- Process
  - Draw a data flow graph from a program.
  - Select one or more data flow testing criteria.
  - Identify paths in the data flow graph satisfying the selection criteria.
  - Derive path predicate expressions from the selected paths and solve those expressions to derive test input.

Identify data flow anomalies

- Type 1: Defined and Then Defined Again
- Type 2: Undefined but Referenced
- Type 3: Defined but Not Referenced
- These anomalies may not be bugs, but should be clarified for the readers.

Identify data flow anomalies (cont.)

- Each variable has a state machine
- Check whether certain state machine can reach abnormal state



Legend:

States

U: Undefined
D: Defined but not referenced
R: Defined and referenced
A: Abnormal

Actions

d: Define
r: Reference
u: Undefine

Terminologies

- *Definition*: When a value is moved into the memory location of the variable.
- *Undefinition or Kill* : When the value and the location become unbound.
- *Use*: When the value is fetched from the memory location of the variable
  - Computation use (c-use): directly affects the computation being performed
  - Predicate use (p-use): use of the variable in a predicate controlling the flow of execution

Example

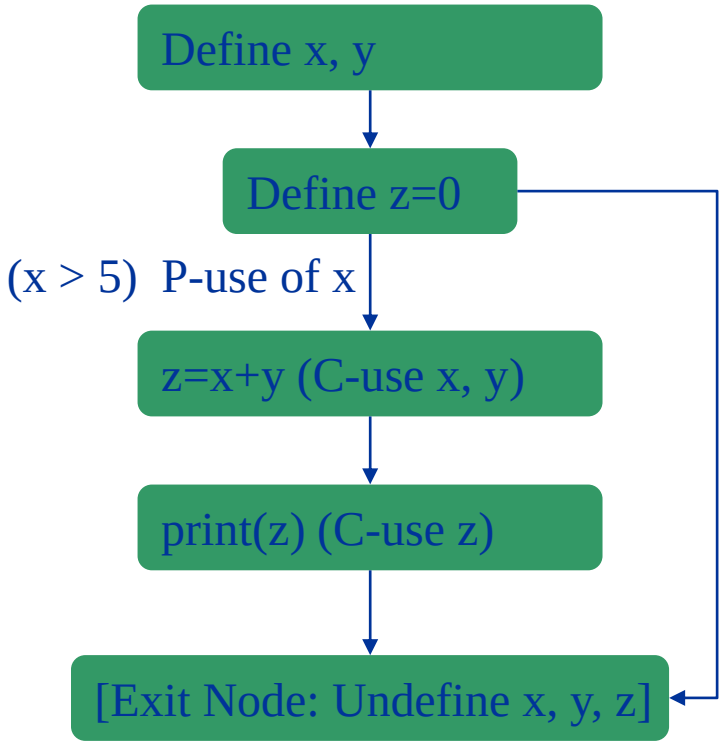
```
1: int x = 10;          // Definition of x
2: int y = 20;          // Definition of y
3: int z = 0;           // Definition of z
4: if (x > 5)           // P-use of x
5:     z = x + y;       // C-use of x and y, definition of z
6: print(z);
```

Data flow diagram construction

- 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 parameter and each nonlocal variable which occurs in the subprogram.
- The exit node has an *undefinition* of each local variable.

Example

```
1: int x;          // Definition of x
2: int y;          // Definition of y
3: int z = 0;          // Definition of z
4: if (x > 5)          // P-use of x
5:     z = x + y;       // C-use of x and y, definition of z
6: print(z);
```



Data flow testing criteria (cont.)

- **All-Def-Criterion:** Every variable definition must be reached from some test path.
- **All-C-Use Criterion:** Every computational use of a variable is executed at least once.
- **All-P-Use Criterion:** Every predicate use of a variable is executed at least once.
- **All-C-Use/P-Use Criterion:** Every possible combination of computational and predicate uses covered by paths.

Example

```
1: int x;          // Definition of x
2: int y;          // Definition of y
3: int z = 0;          // Definition of z
4: if (x > 5)          // P-use of x
5:     z = x + y;       // C-use of x and y, definition of z
6: print(z);
```

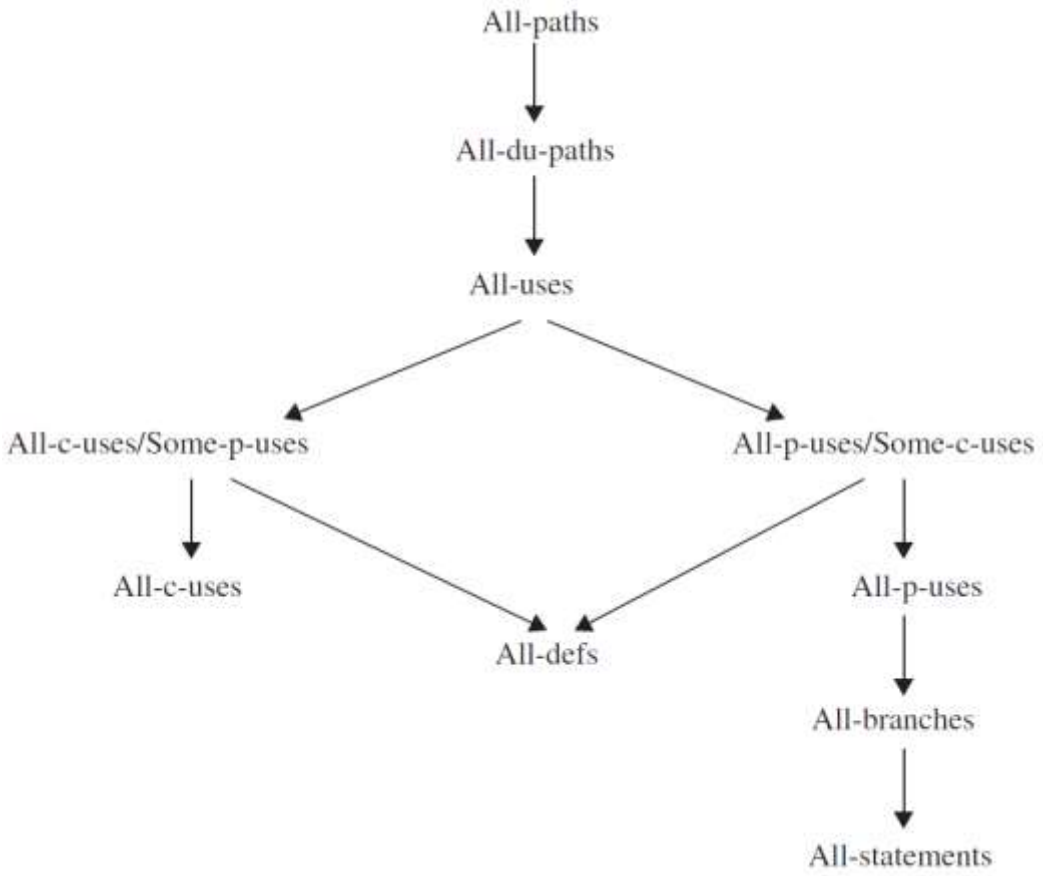
1.Test Case 1: **x = 10, y = 20**.
2.Test Case 2: **x = 4, y = 20**

All-Def-Criterion: Testcase 1
All-c-Use: Testcase 1
All-p-Use: Testcase 1,2

Data flow testing criteria (cont.)

- All-c-uses/Some-p-uses:
  - When x does not have c-use
- All-p-uses/Some-c-uses:
- *All-uses*: conjunction of the all-p-uses criterion and the all-c-uses criterion
- *Du-path*: A path ($n1 - n2 - \cdots - nj - nk$) is a definition-use path (du-path) with respect to variable x if node $n1$ has a global definition of x and *either*
  - node nk has a global c-use of x and ($n1 - n2 - \cdots - nj - nk$) is a def-clear simple path w.r.t. x
  - edge (nj ,nk) has a p-use of x and ($n1 - n2 - \cdots - nj$) is a def-clear, loop-free path w.r.t. x.
- *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

Criteria Comparison



Testing with Use cases

- Use cases
  - Business use case
- Use cases represented by sequence diagram or activity diagram
- Usually during acceptance testing
- Pros
  - Comprehensible

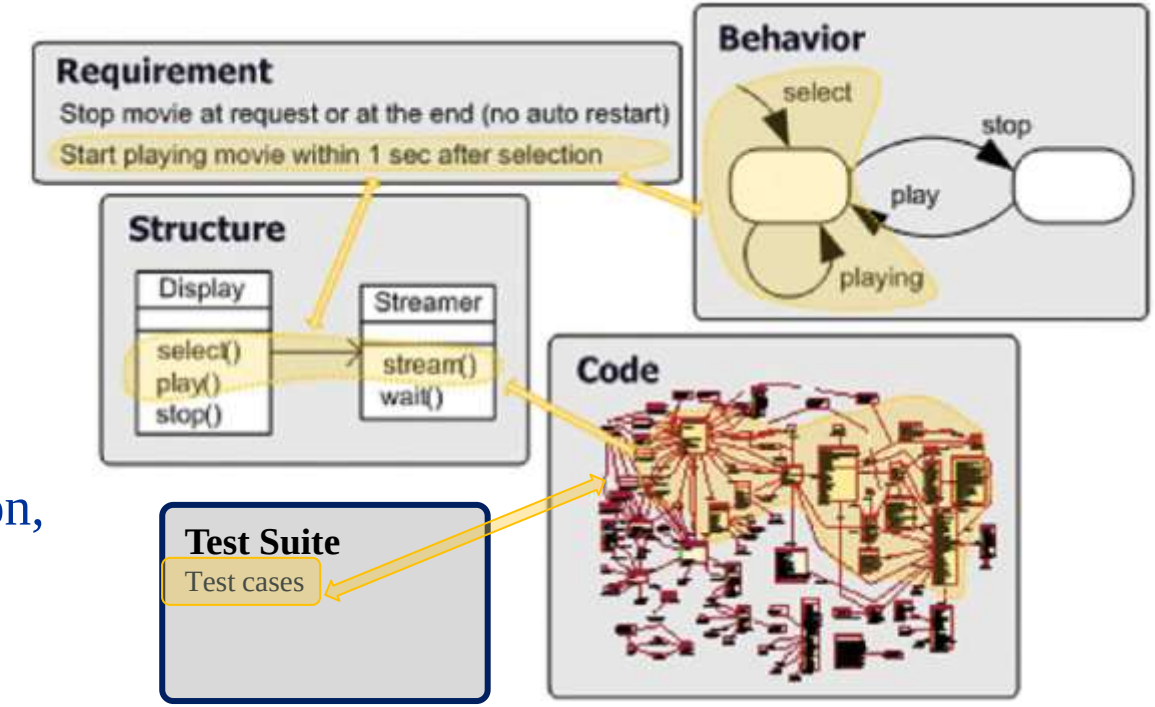
Reference

- Fundamentals of software testing by Bernard Homès
- Available from the library website

Traceability

What is traceability?

- We would like to make sure that
  - All requirements are implemented
  - All implementations are necessary
- Trace artifacts
  - Requirements, models, code, etc.
- Trace link
  - Association between two trace artifacts
  - Type: Refinement, Abstraction, Implementation, etc.
- Trace granularity: component level, statement level, etc.
- Trace quality: completeness, correctness, etc.

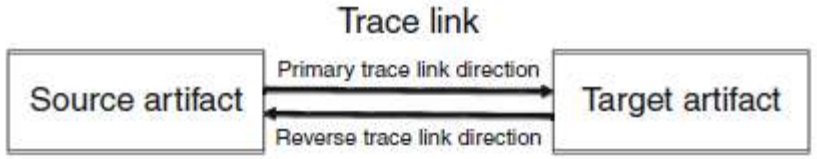


Objectives of Traceability

- Software lifecycle involves more than one person
- Within the team
  - Make sure the requirements are faithfully translated to code
- For the customers and regulation agencies
  - Part of validation evidence

Traceability Activities

- Trace Creation
  - Establish *trace link* between a *source artifact* and a *target artifact*
  - Traceability document



- Trace Validation
  - Between requirements and model: Model checking
  - Between concept model and implementation model: Model translation
  - Between model and code: Conformance testing
- Trace Maintenance
  - Update trace when modification happened