Lecture 14: Testing (Cont.)

Test Design Techniques

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

Specification-based Testing

- Equivalence Partitioning
- State Transition Testing
- Scenario Testing

Test Coverage Measurement

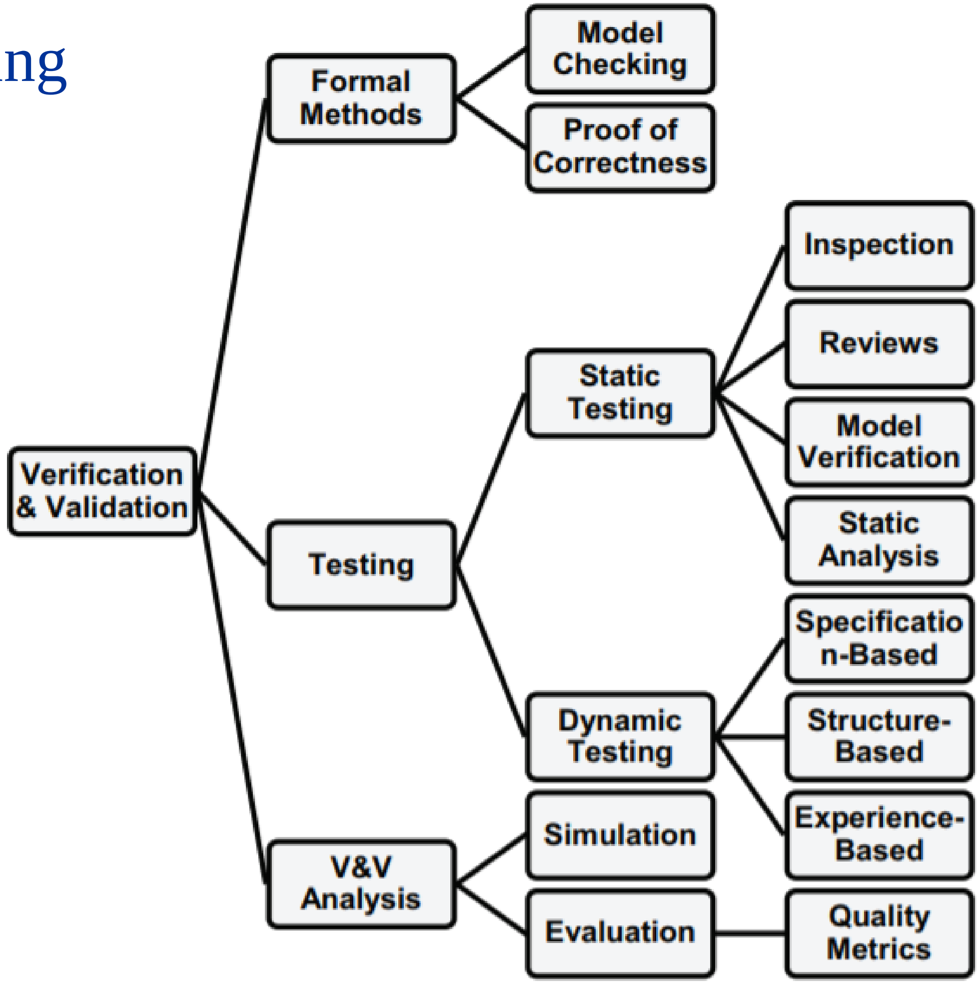
- $Coverage = \left(\frac{N}{T} \times 100\right)\%$
  - N is the number of test coverage items covered by test cases
  - T is the number of identified test coverage items

- Coverage is only measured for a particular criteria

- Coverage criteria has strength

Testing in V&V

- Both verification and validation involves testing
- Verification
  - Whether the code conform with software specification
  - Conformance testing
- Validation
  - Functional testing
  - Scenario testing
  - Risk based testing



Test Design Techniques

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

White-Box Testing/Structure-based Testing

- There exist several popular white-box testing methodologies:
  - Statement coverage
  - branch coverage
  - condition coverage
  - path coverage
    - Control path
    - Data path

Statement Coverage

- Statement coverage methodology:
  - Design test cases so that
    - Every statement in a program is executed at least once.

Statement Coverage

- The principal idea:
  - Unless a statement is executed,
  - We have no way of knowing if an error exists in that statement.

Branch Coverage

- Test cases are designed such that:
  - different branch conditions
    - given true and false values in turn.

Condition Coverage

- Test cases are designed such that:
  - Each component of a composite conditional expression
    - Given both true and false values.

Example

- Consider the conditional expression
  - ((c1.and.c2).or.c3):

- Branch coverage
  - ((c1.and.c2).or.c3)==true
  - ((c1.and.c2).or.c3)==false

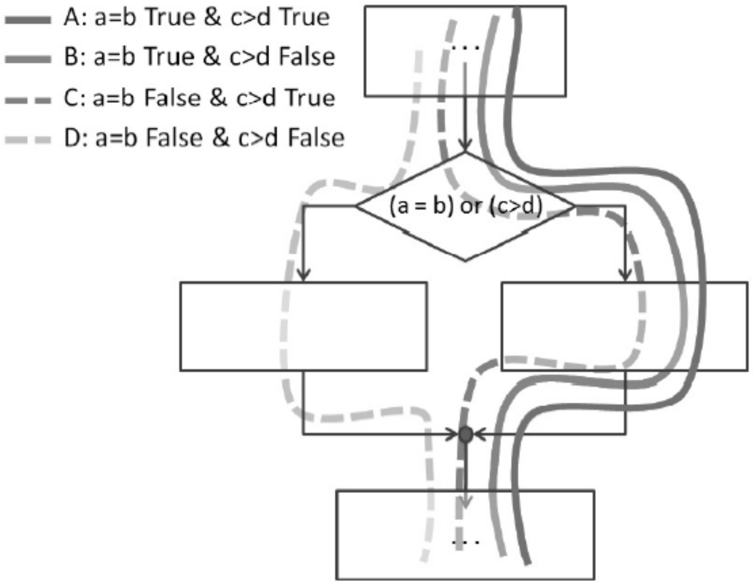
- Condition coverage
  - Each of c1, c2, and c3 is evaluated to true and false

Comparison

- Condition testing
  - stronger testing than branch testing:
- Branch testing
  - stronger testing than statement coverage testing.

MC/DC Coverage

- Modified Condition and Decision Coverage
  - Each entry point and exit point is covered (Decision coverage)
  - Each condition within a decision demonstrates its impact on the result of the decision
- Condition coverage: A,B,C,D
- MC/DC: A,D || B,C,D
- Widely used test coverage criteria in aviation industry



Limitations of Structure-based Testing

- MC/DC is very heavy
  - Only for critical software components
- May not cover major scenarios that a software may encounter
  - i.e. Statement y=sqrt(1/x) should be tested for x=0, x>0, x<0

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)

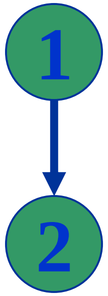
- A control flow graph (CFG) describes:
  - the sequence in which different instructions of a program get executed.
  - the way control flows through the program.

How to draw Control flow graph?

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

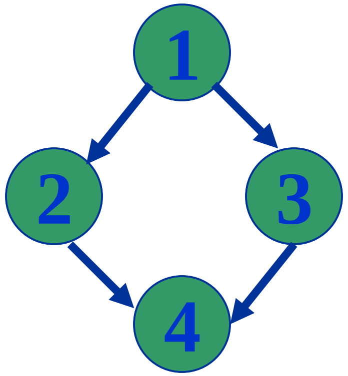
How to draw Control flow graph?

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



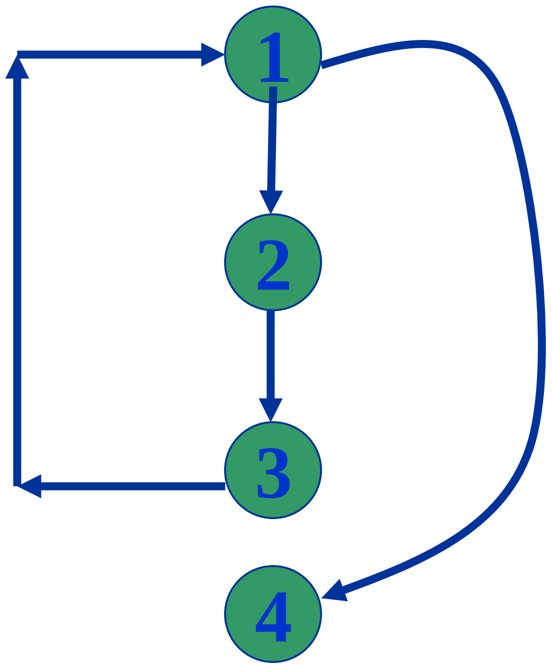
How to draw Control flow graph?

- 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

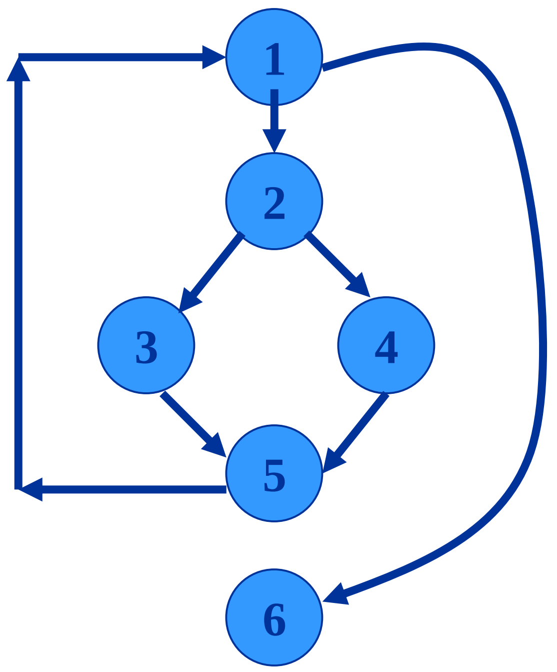
- A path through a program:
  - a node and edge sequence from the starting node to a terminal node of the control flow graph.
  - There may be several terminal nodes for program.

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.

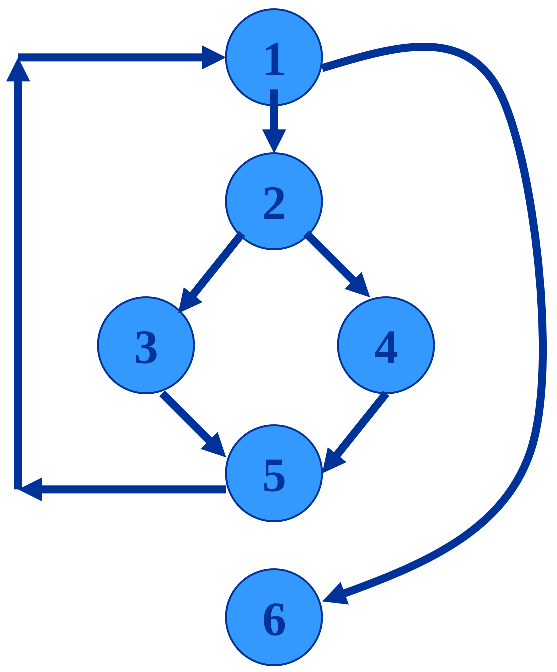
Example

- 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

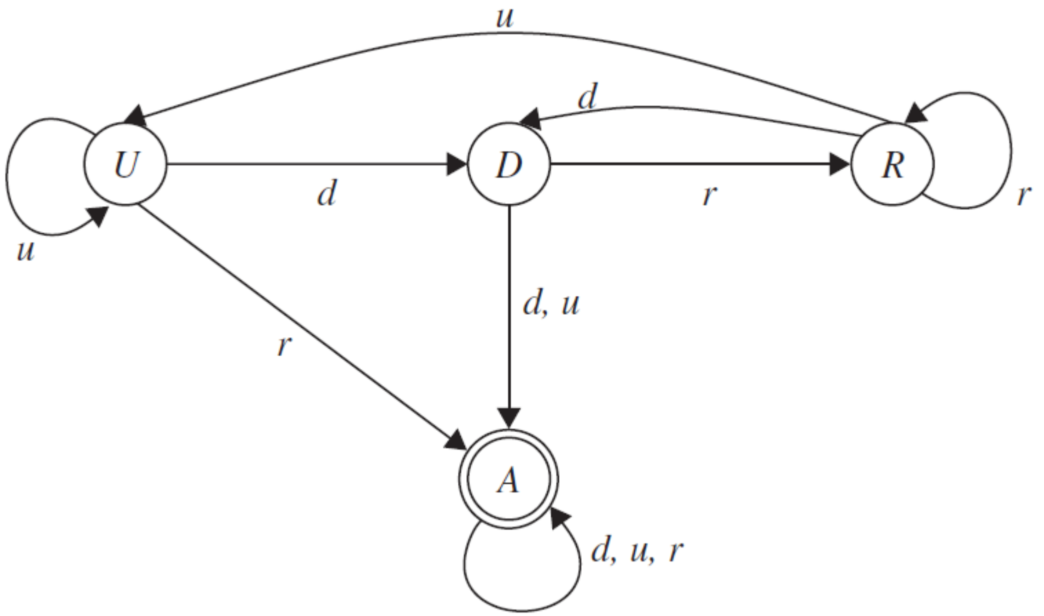
- Motivation
  - How do you know that a variable is assigned the correct value?
  - From: when the value is assigned
  - To: when the value is used later
- Process
  - Draw a data flow graph from a program.
  - Select one or more data flow testing criteria.
  - Identify paths in the data flow graph satisfying 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

```
int VarTypes(int x, int y){
      int i;
      int *iptr;
      i = x;
      iptr = malloc(sizeof(int));
      *iptr = i + x;
if (*iptr > y)
            return (x);
else {
            iptr = malloc(sizeof(int));
            *iptr = x + y;
            return(*iptr);
}
}
```

Definition

Definition

C-use

P-use

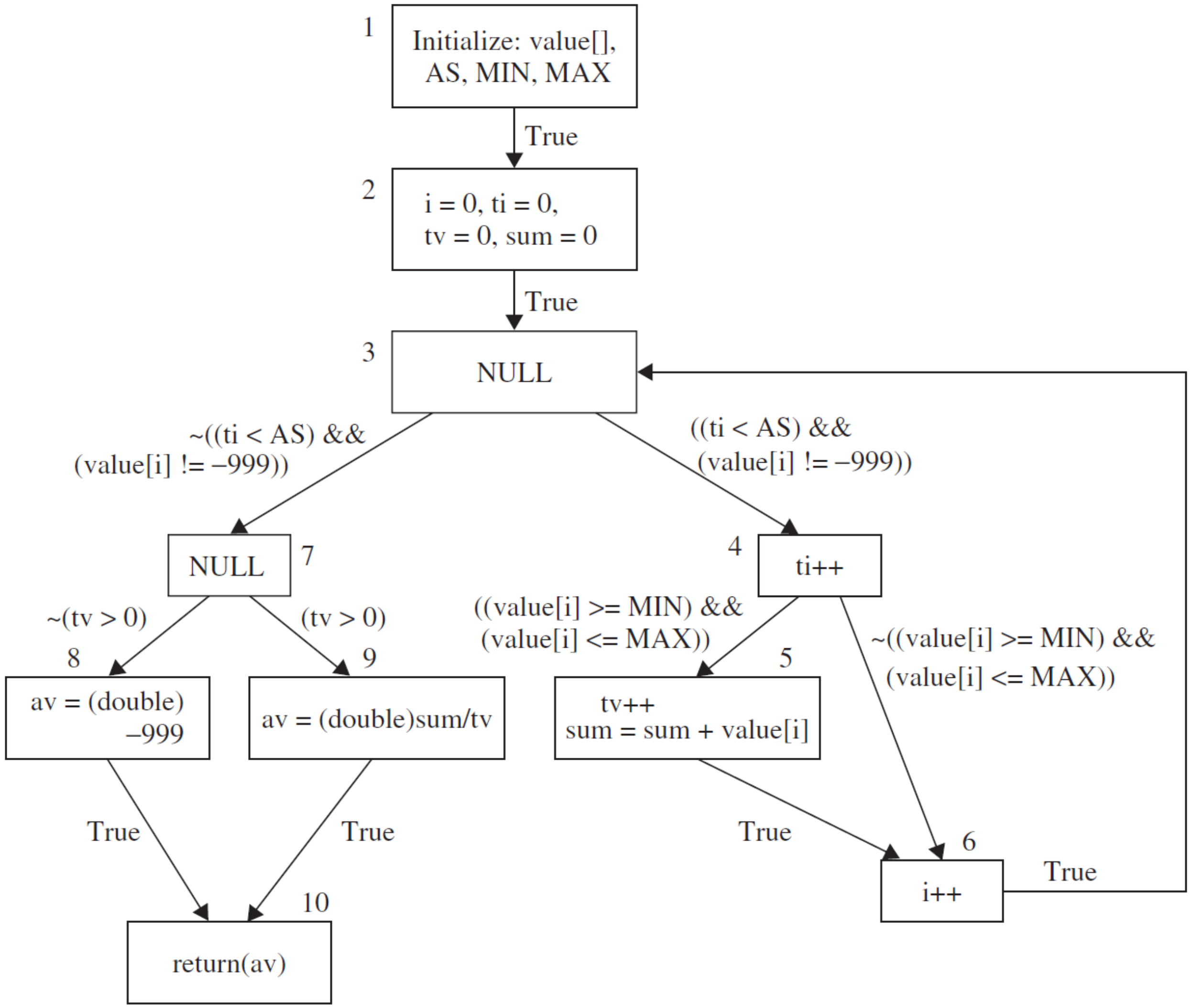
Undefinition

Data flow diagram construction

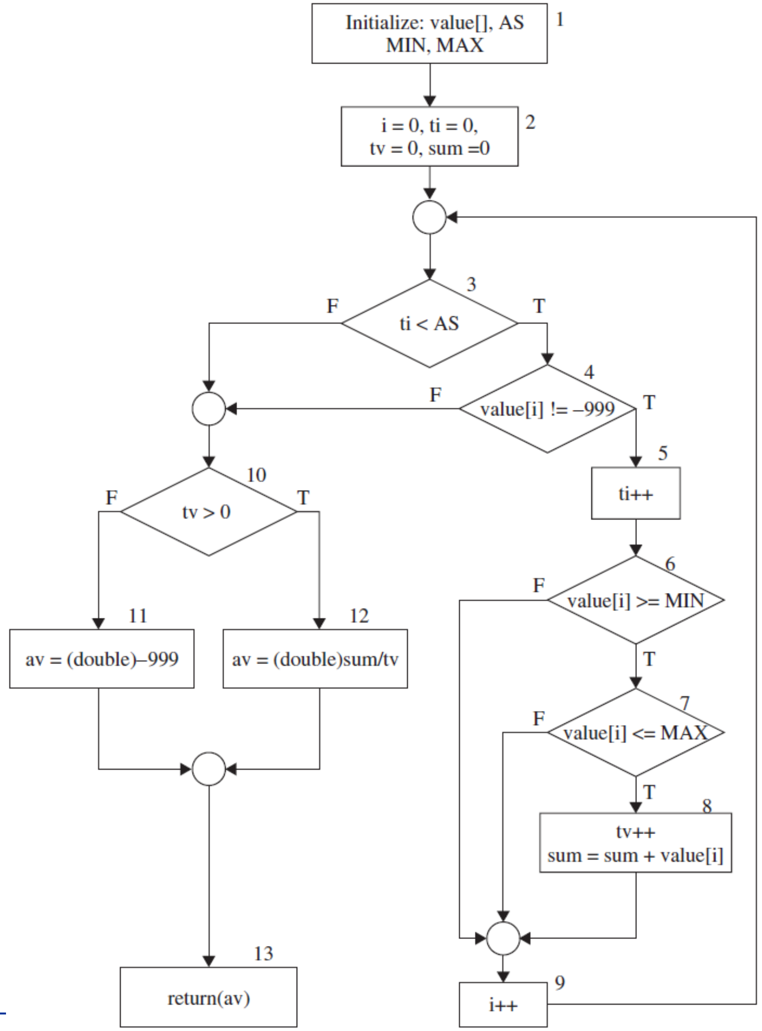
- A sequence of definitions and c-uses is associated with each node of the graph.
- A set of p-uses is associated with each edge of the graph.
- The entry node has a definition of each parameter and each nonlocal variable which occurs in the subprogram.
- The exit node has an *undefinition* of each local variable.

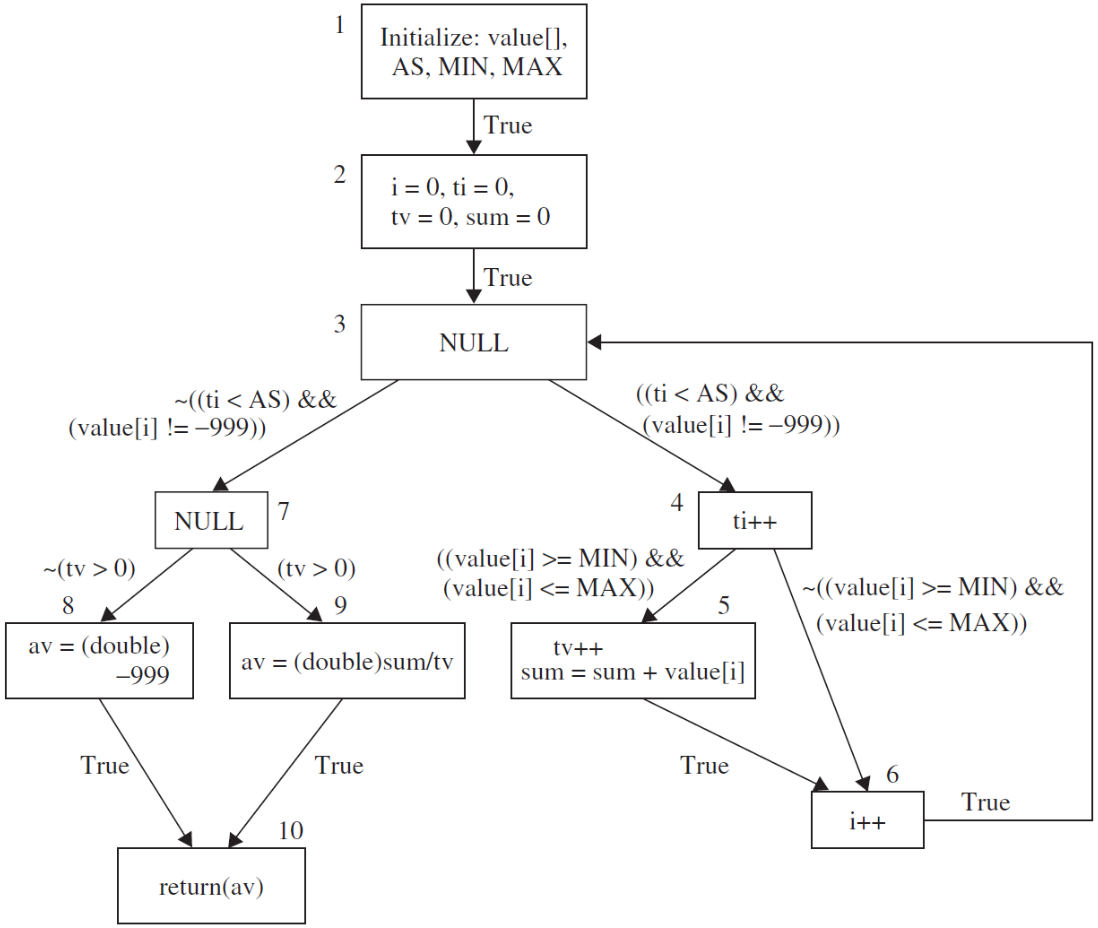
Example

```
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);
}
```



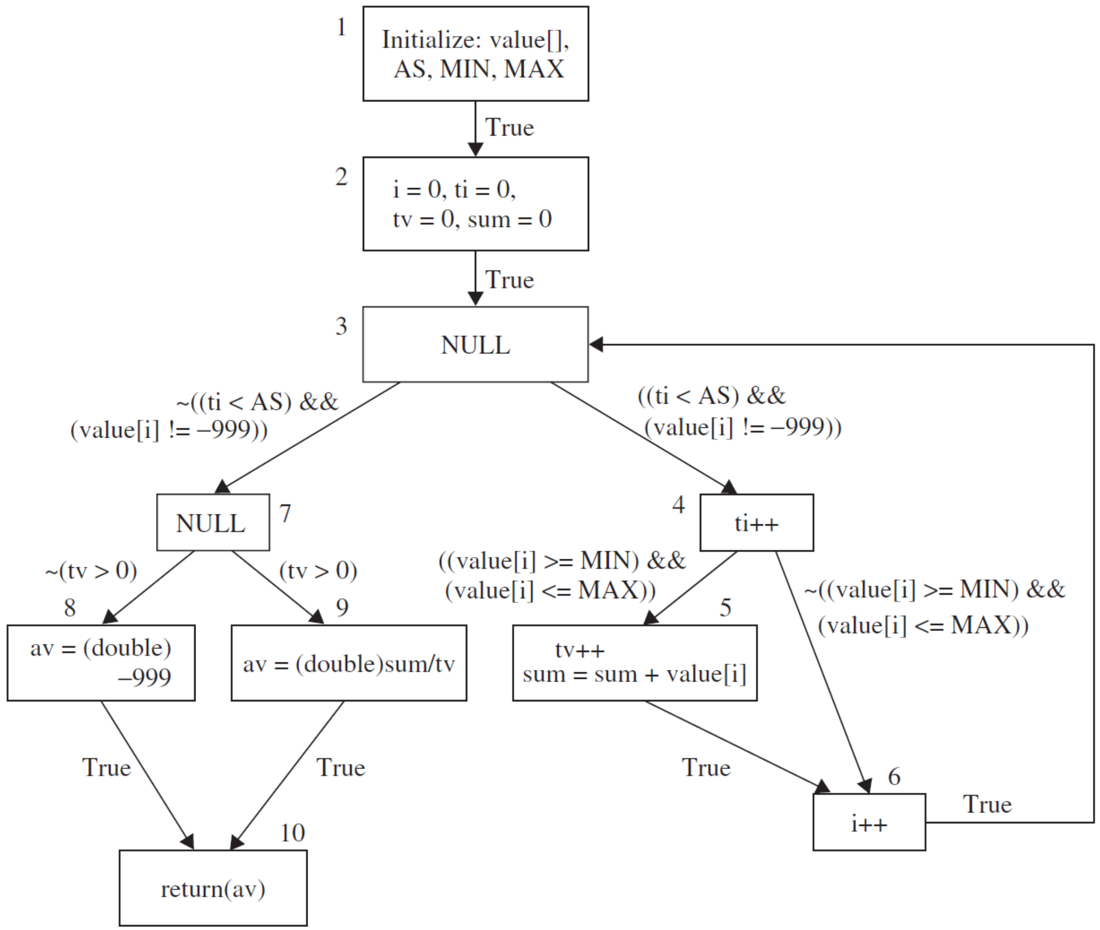
Control flow graph vs. Data flow graph





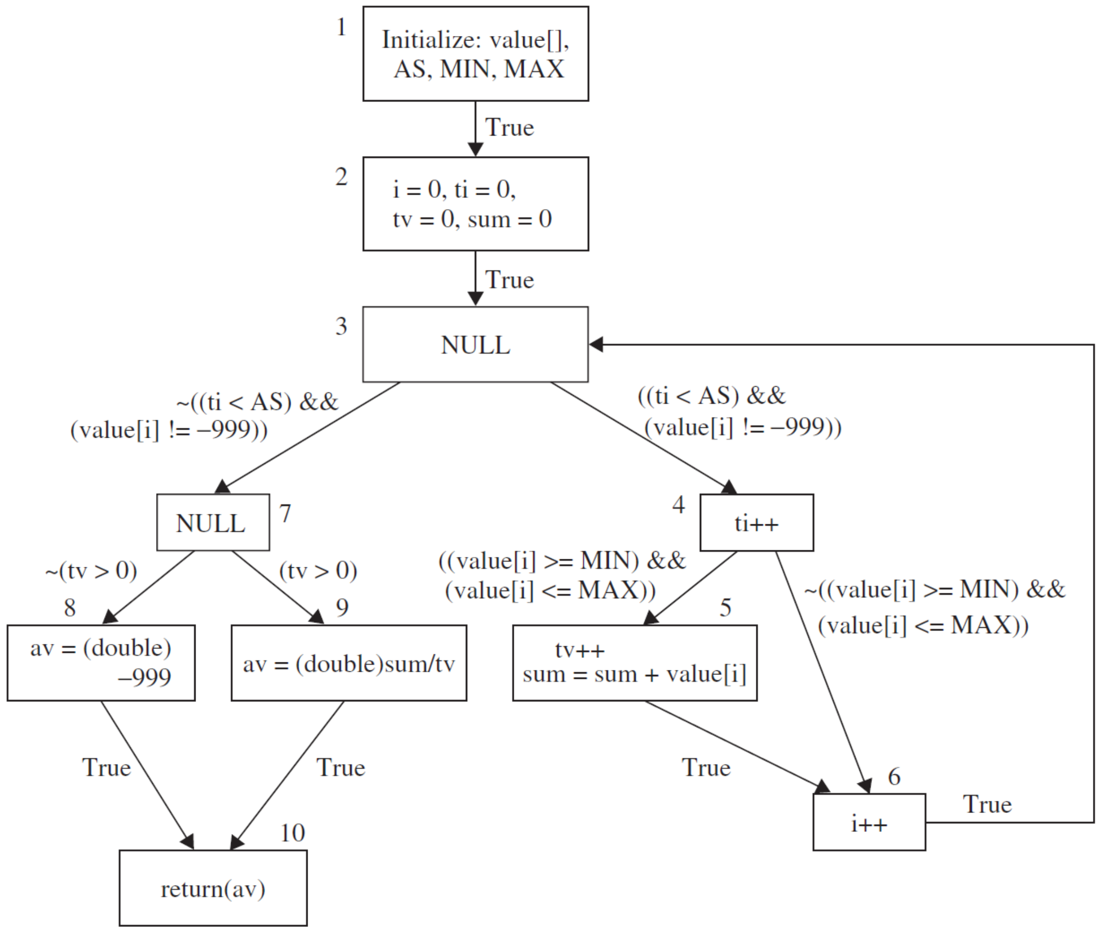
Path selection criteria

- Global c use
  - *x* has been defined before in a node other than node *Initialize*
  - *tv* in node 9 is global c use (2,5)
- Definition clear path for x
  - $(i - n1 - \cdots - nm - j)$
  - If x has been neither *defined* nor *undefined* in nodes $n1, \ldots, nm$
  - *2,3,4,5 and 2,3,4,6 for tv*
- Global definition
  - node i has a definition of x and there is a def-clear path with respect to x from node i to some global c use or p use of x
  - 8,9 for global definition of av
- Complete path
  - A path from entry to exit node



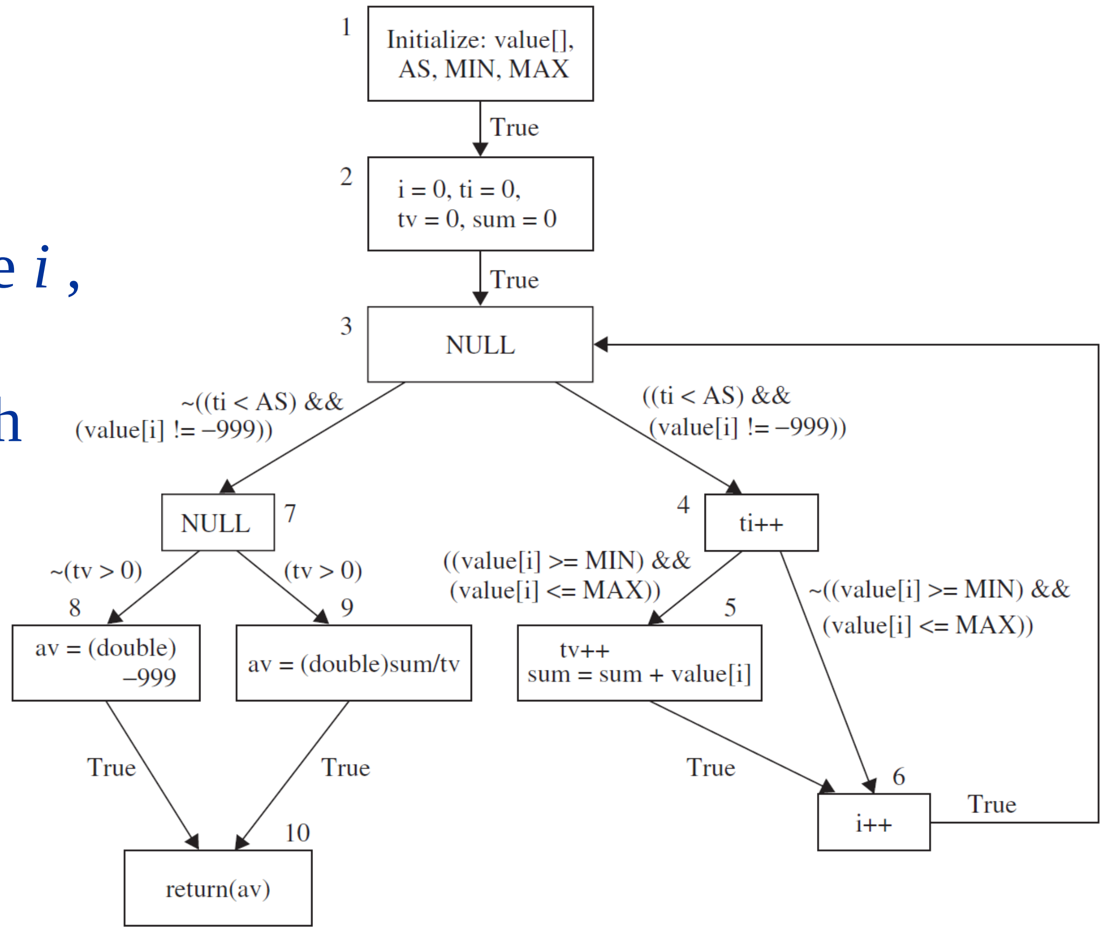
Data flow testing criteria

- All defs
  - For each variable *x* and for 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*.
  - i.e. 2,3,4,5 is a def-clear path tv
  - 1,<u>2,3,4,5</u>,6,3,7,9,10 is a all def path
  - 2,3,7,8 is also a def-clear path for tv
  - 1,<u>2,3,7,8</u>,10 is a all def path



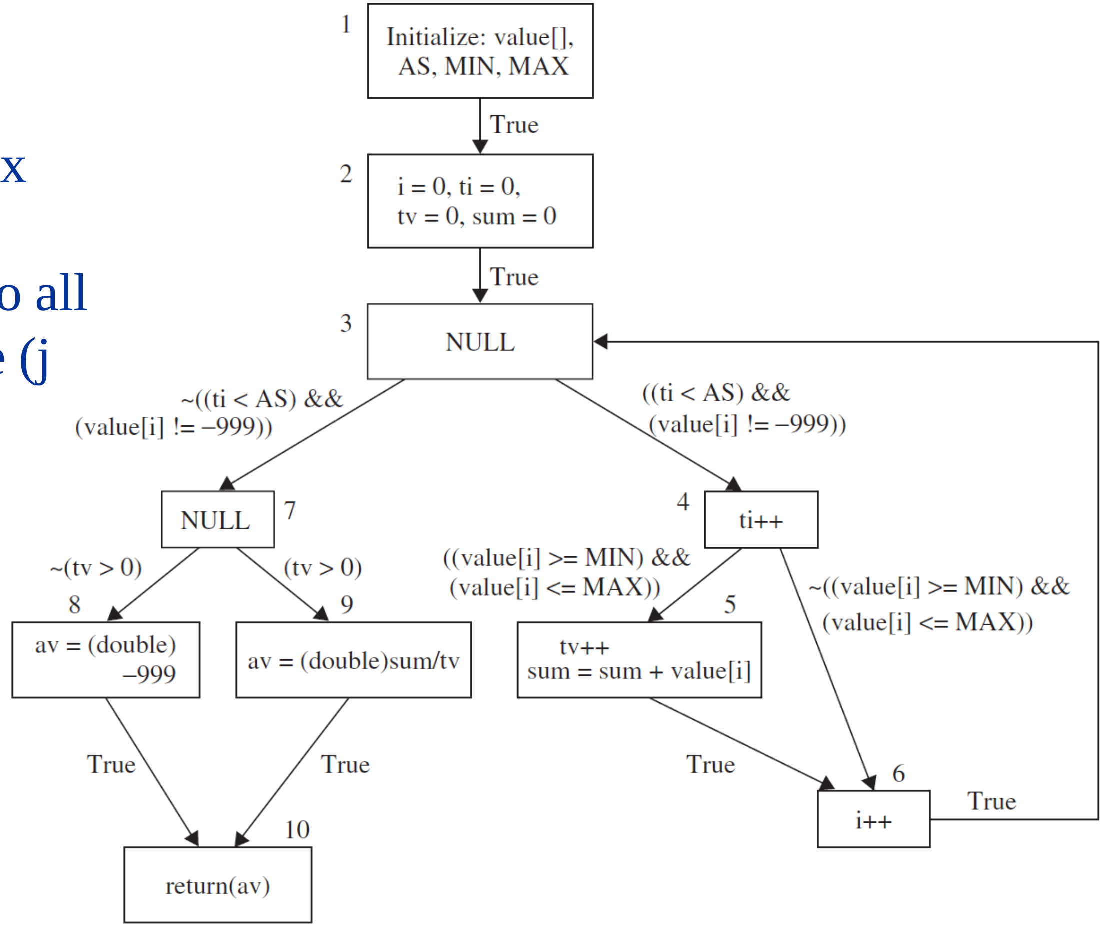
Data flow testing criteria (cont.)

- All-c-uses:
  - For each variable *x* and for 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* .
  - i.e. 2,3,4 is a def-clear path for ti
    - 1-2-3-4-5-6-3-7-8-10,
    - 1-2-3-4-5-6-3-7-9-10,
    - 1-2-3-4-6-3-7-8-10, and
    - 1-2-3-4-6-3-7-9-10.



Data flow testing criteria (cont.)

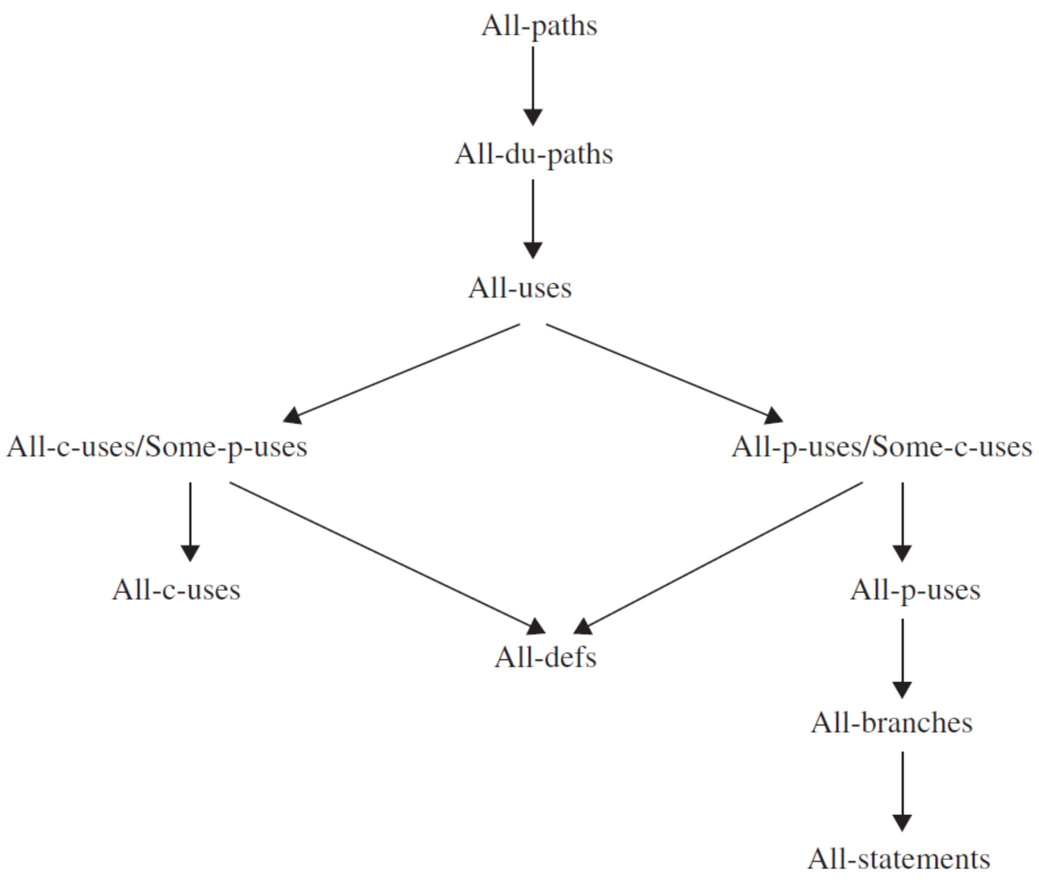
- All-p-uses:
  - For each variable x and for each node i such that x has a global definition in node i , select complete paths which include def-clear paths from node i to all edges (j ,k) such that there is a p-use of x on edge (j ,k).
  - i.e. 2,3,7,8; 2,3,7,9; 5,6,3,7,8; 5,6,3,7,9 for tv
    - 1-<u>2-3-7-8</u>-10,
    - 1-<u>2-3-7-9</u>-10,
    - 1-2-3-4-<u>5-6-3-7-8</u>-10, and
    - 1-2-3-4-<u>5-6-3-7-9</u>-10.



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

Class-based Testing in Matlab

- testCase.verifyEqual

```
%% Test Class Definition
classdef MyComponentTest < matlab.unittest.TestCase

    %% Test Method Block
    methods (Test)

        %% Test Function
        function testASolution(testCase)
            %% Exercise function under test
            % act = the value from the function under test

            %% Verify using test qualification
            % exp = your expected value
            % testCase.<qualification method>(act,exp);
        end
    end
end
```

Testing APP

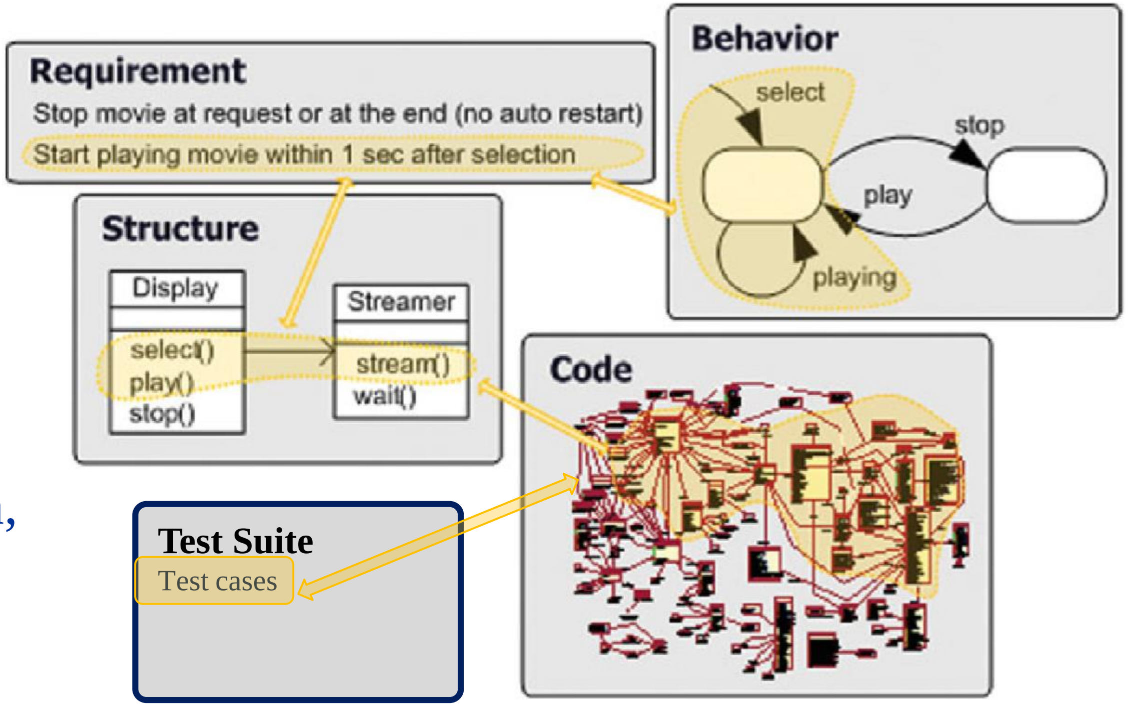
```
classdef TestPatientsDisplay < matlab.uitest.TestCase
    properties
        App
    end

    methods (TestMethodSetup)
        function launchApp(testCase)
            testCase.App = PatientsDisplay;
            testCase.addTeardown(@delete,testCase.App);
        end
    end

    methods (Test)
        function test_plottingOptions(testCase)
            % Press the histogram radio button
            testCase.press(testCase.App.HistogramButton)

            % Verify xlabel updated from 'Weight' to 'Systolic'
            testCase.verifyEqual(testCase.App.UIAxes.XLabel.String,'Systolic')

            % Change the Bin Width to 9
            testCase.choose(testCase.App.BinWidthSlider,9)

            % Verify the number of bins is now 4
            testCase.verifyEqual(testCase.App.UIAxes.Children.NumBins,4)
        end

        function test_tab(testCase) ...

    end
end
```

Component	`matlab.uitest.TestCase` Gesture Method				
	`press`	`choose`	`drag`	`type`	`hover`
Button	✓				
State button	✓	✓			
Check box	✓	✓			
Switch	✓	✓			
Discrete knob		✓			
Knob		✓	✓		
Drop-down		✓		✓	
Edit field				✓	
Text area				✓	
Spinner	✓			✓	
Slider		✓	✓		
List box		✓			
Button group		✓			
Tab group		✓			
Tab		✓			
Tree node		✓			
Menu	✓				
Date Picker				✓	
Axes	✓				✓
UI Axes	✓				✓
UI Figure	✓				✓

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

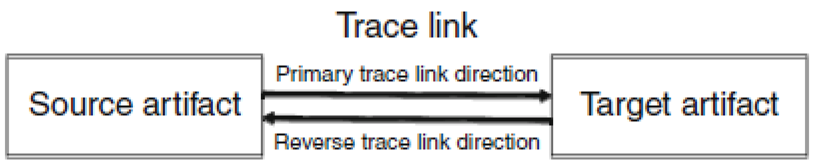
- Software lifecycle involves more than one person

- 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