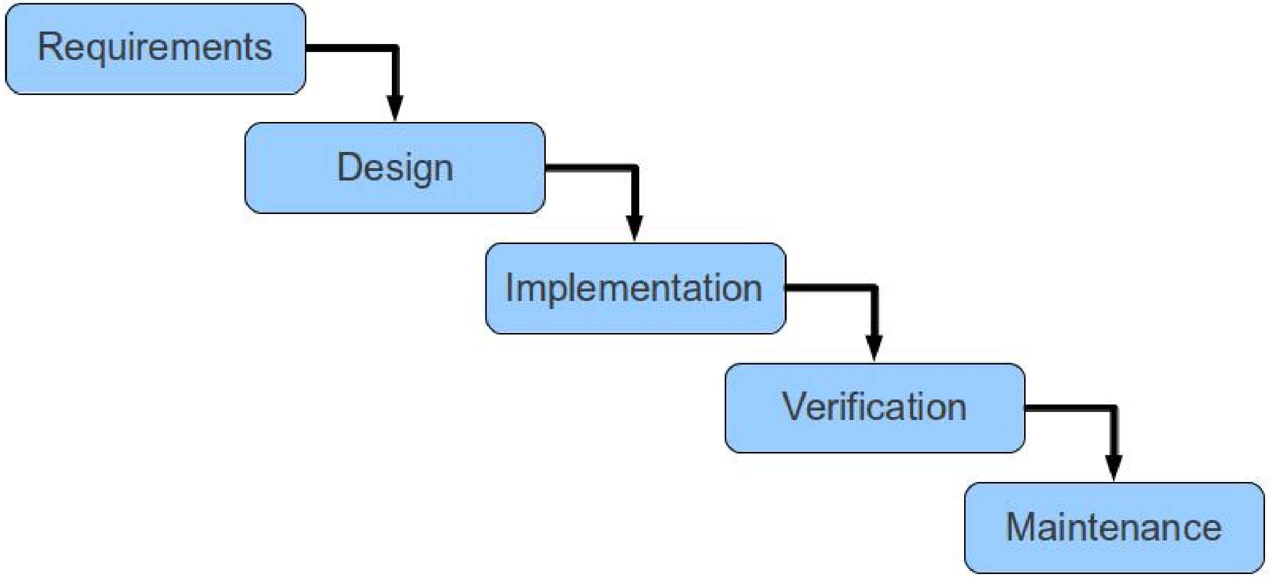
Lecture 14: Testing (Cont.)

Model-independent testing principles

- Each design activity, and each deliverable, must have a corresponding test activity that will search for defects introduced by this activity or in this deliverable;
- Each test level has its own specific objectives, associated with that test level, so as to avoid testing the same characteristic twice;
- Analysis and design of tests for a given level start at the same time as the design activity for that level, thus saving as much time as possible;
- Testers are involved in document review as soon as drafts are available, whichever the development model selected.

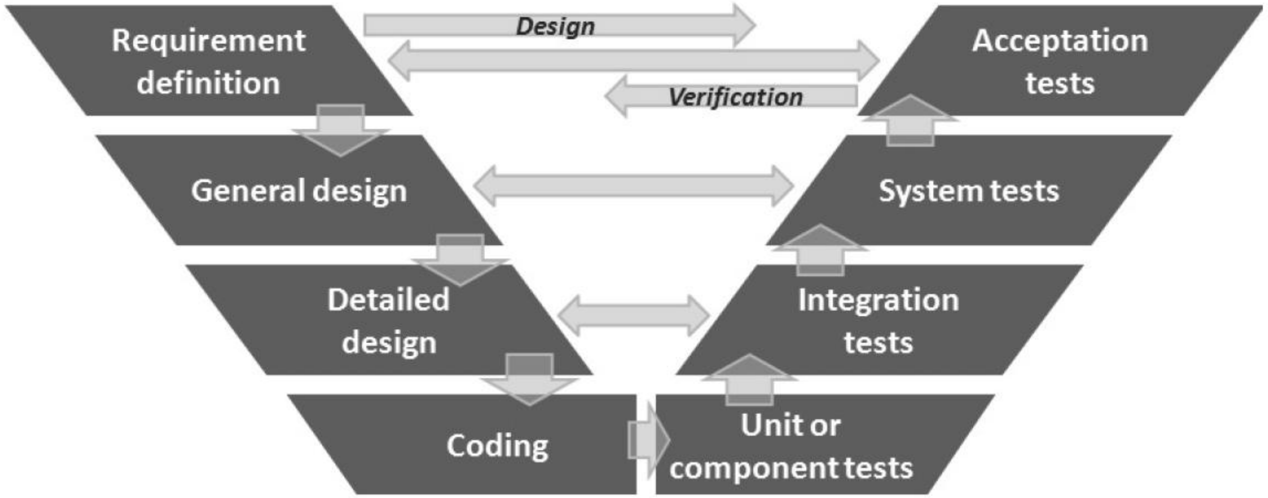
Waterfall software development model

- Applicable to applications with confirmed requirements at early stage
- Cons: Cost is huge once validation fails at later development phase



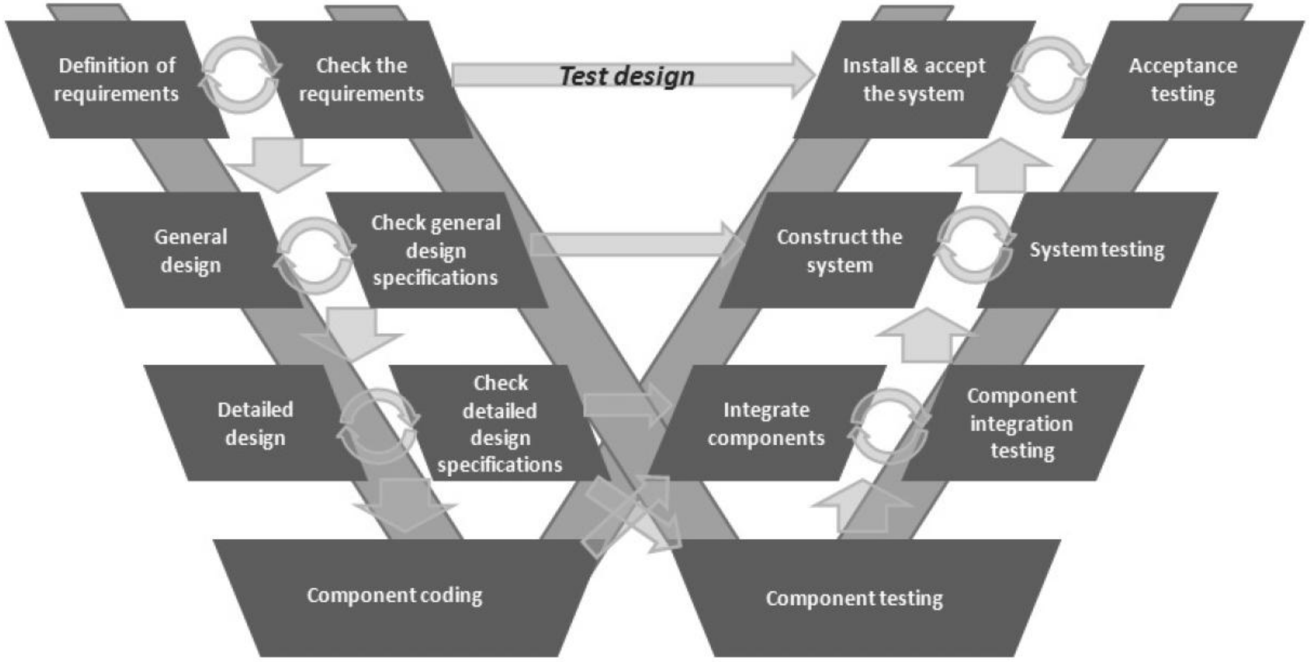
The V model

- Testing for each sequential stages
- Test design at the same time with the development stage



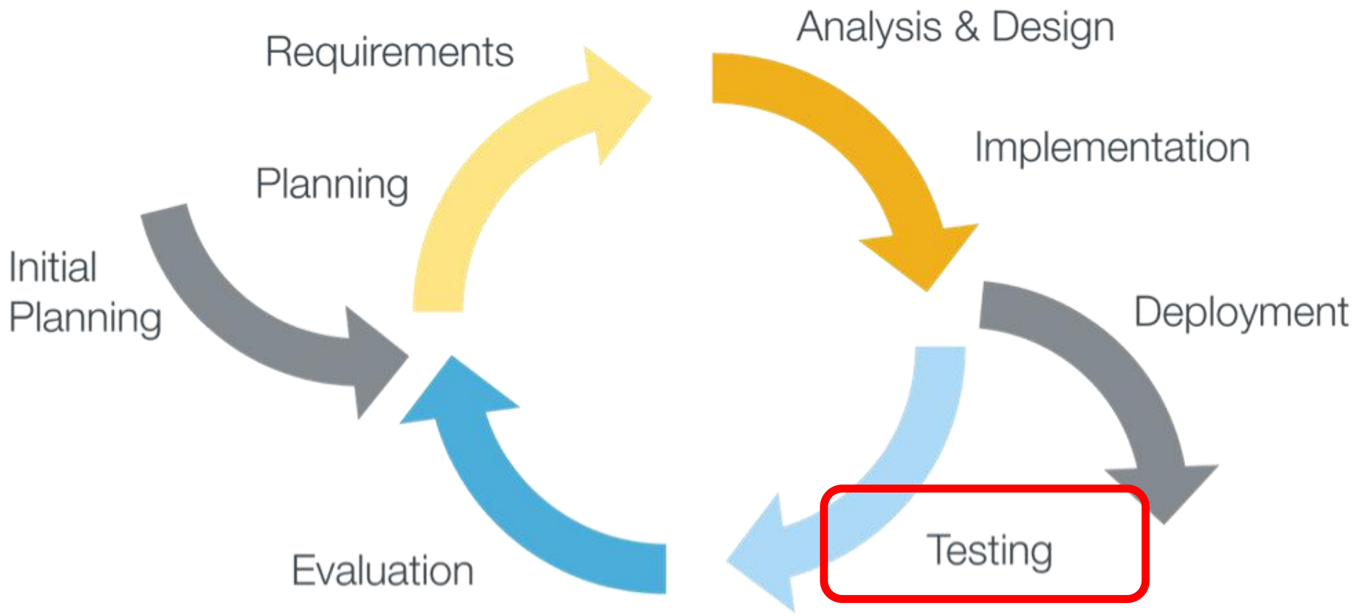
The W/VVV model

- Incorporating static testing techniques
- Early V&V



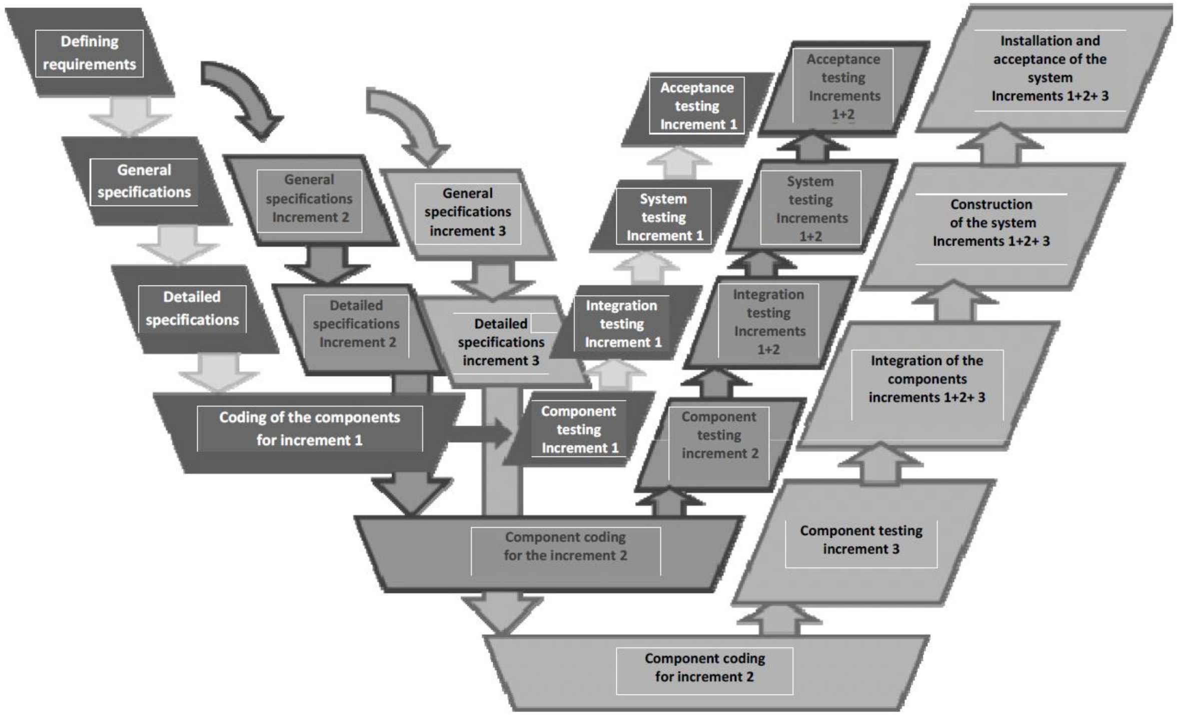
Iterative software development model

- Develop core functionalities first
- Improve/refine software in later iterations
- Problem: later iterations may damage previous iterations



Incremental model

- Increments expected before design
- Increments defined during design
- Cons: spend too much time developing an increment if system is not divided properly

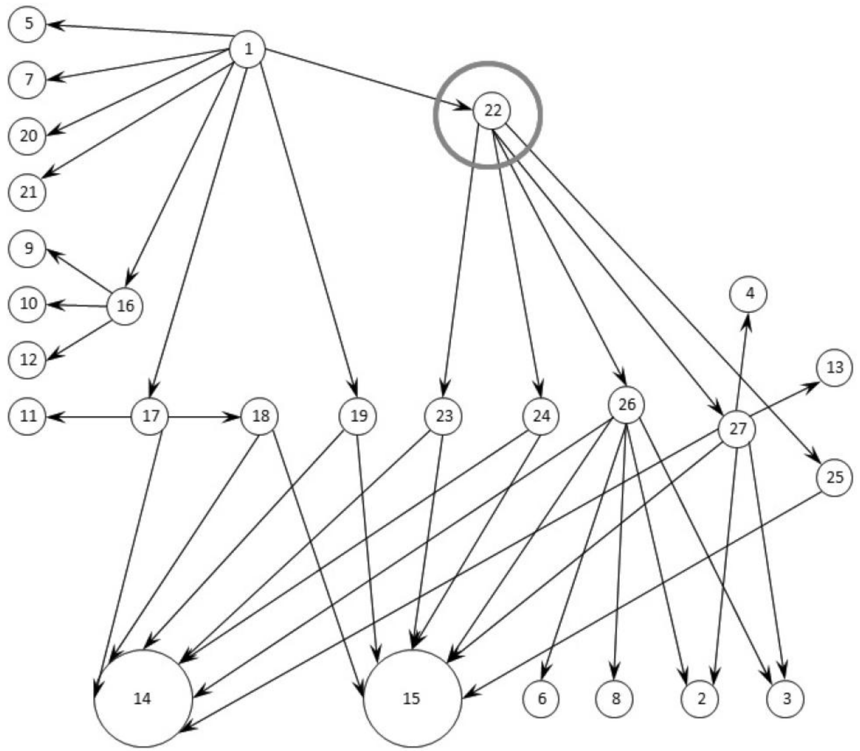


After a defect has been corrected

- Confirmation tests or retests
  - Verifying that the defect has been corrected and the software operates as expected
- Regression tests
  - Make sure that the correction did not introduce any side effects (regression) on the rest of the software

Regression test example

- Changes in node 22
- Direct impact: 1, 23, 24, 25, 26, and 27
- Indirect impact: 14, 15, 6, 8, 2, 3, 4, and 13



International Standard

- ISO/IEC/IEEE 29119 Software and Systems Engineering – Software Testing
  1. Concepts and definitions
  2. Test process
  3. Test documentation
  4. Test techniques

- An informative standard, not conformance standard

- Available on Blackboard

Test cases and Test suite

- Test a software using a set of carefully designed test cases:
  - The set of all test cases is called the test suite

Test cases and Test suite

- A test case is a triplet [I,S,O]:
  - I is the data to be input to the system,
  - S is the state of the system at which the data is input,
  - O is the expected output from the system.

Design of Test Cases

- Exhaustive testing of any non-trivial system is impractical:
  – input data domain is extremely large.

- Design an optimal test suite:
  – of reasonable size
  – to uncover as many errors as possible.

Design of Test Cases

- If test cases are selected randomly:
  - Many test cases do not contribute to the significance of the test suite,
  - Do not detect errors not already detected by other test cases in the suite.

- The number of test cases in a randomly selected test suite:
  - Not an indication of the effectiveness of the testing.

Design of Test Cases

- Testing a system using a large number of randomly selected test cases:
  - does not mean that many errors in the system will be uncovered.

- Consider an example:
  - finding the maximum of two integers x and y.

Design of Test Cases

- If (x>y) max = x;
  else max = x;
- The code has a simple error:
- Test suite {(x=3,y=2);(x=2,y=3)} can detect the error,
- A larger test suite {(x=3,y=2);(x=4,y=3); (x=5,y=1)} does not detect the error.

Test Design (TD) Process

- TD1: Identify feature set
- TD2: Derive test conditions
- TD3: Derive test coverage items
- TD4: Derive test cases
- TD5: Assemble test sets
- TD6: Derive test procedures

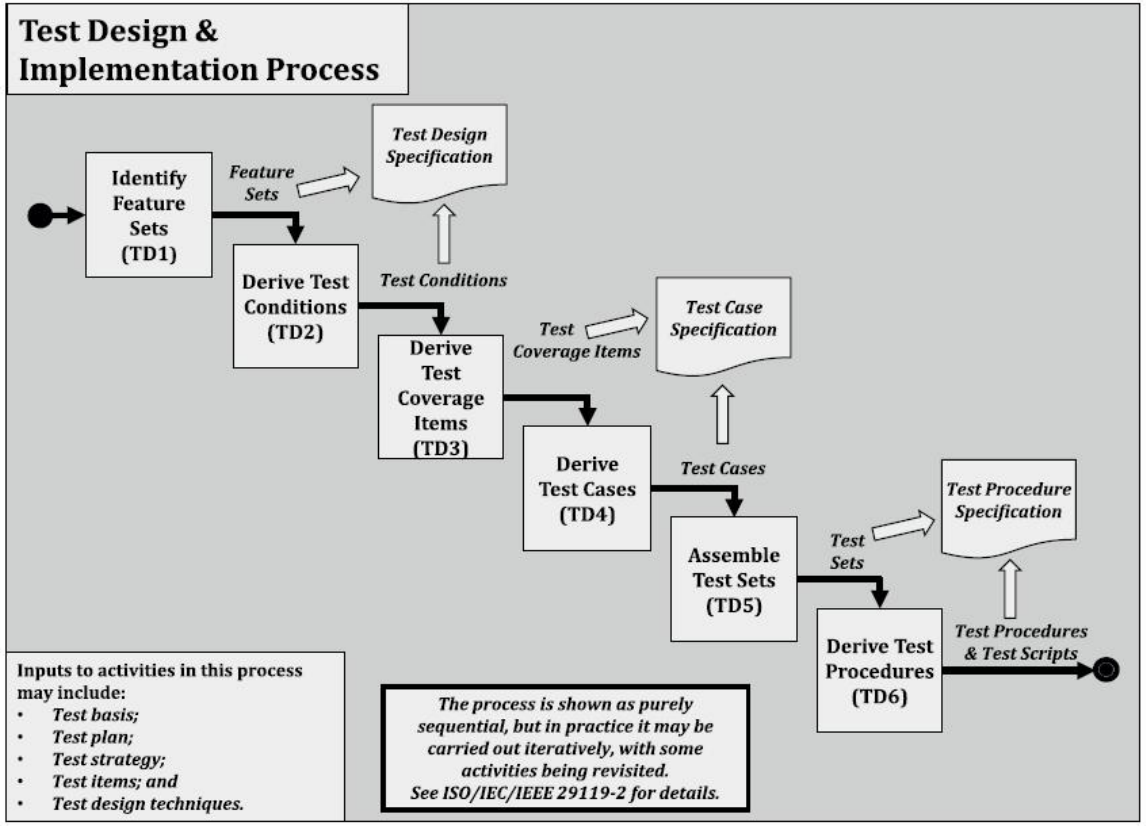


Figure 1 — ISO/IEC/IEEE 29119-2 Test Design and Implementation Process

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

EQUIVALENCE PARTITIONING

Example: Equivalence Partitioning

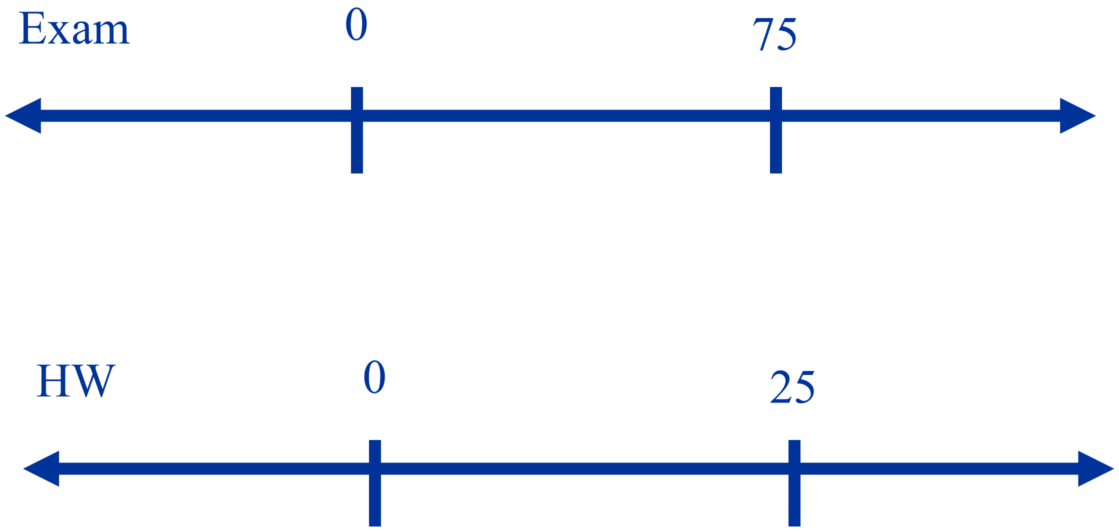
- Homework (HW) 25pt
- Exam 75pt
- Specification: A function *generate_grading*
  - HW+Exam>=70 -> ‘A’
  - 50<= HW+Exam<70 -> ‘B’
  - 30<= HW+Exam<50 -> ‘C’
  - HW+Exam<30 -> ’D’
  - Invalid inputs -> ‘FM’

Step 1: Identify Feature Sets (TD1)

- FS1: generate_grading function

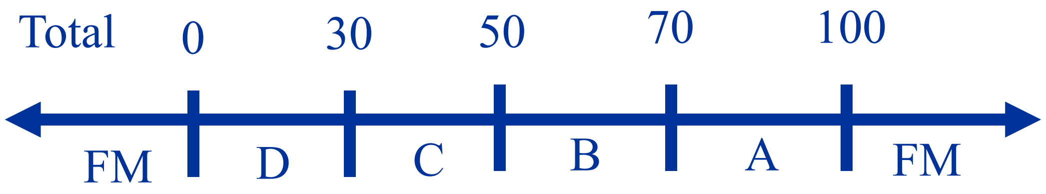
Step 2: Derive Test Conditions (TD2)

- Input Partitions
  - Valid
    - TCOND1: 0<=Exam<=75
    - TCOND2: 0<=HW<=25
  - Invalid
    - TCOND3: Exam<0
    - TCOND4: Exam>75
    - TCOND5: HW<0
    - TCOND6: HW>25
    - TCOND7: non-integer Exam input
    - TCOND8: non-integer HW input



Step 2: Derive Test Conditions (TD2) (CONT)

- Output Partitions
  - TCOND9: ‘A’ induced by 70<=Total<=100
  - TCOND10: ‘B’ induced by 50<=Total<70
  - TCOND11: ‘C’ induced by 30<=Total<50
  - TCOND12: ‘D’ induced by 0<=Total<30
  - TCOND13: ‘Fault Message’ (FM) induced by Total>100
  - TCOND14: ‘Fault Message’ (FM) induced by Total<0
  - TCOND15: ‘Fault Message’ (FM) induced by non-integer inputs



Step 3: Derive Test Coverage Items (TD3)

- Specify a test coverage item (TCOVER) for each test condition (TCOND)
  - TCOVER1: 0<=Exam<=75 (for TCOND1)
  - TCOVER2: 0<=HW<=25 (for TCOND2)
  - …

Step 4: Derive Test Cases (TD4)

- Attempt to “hit” Test Coverage Items
  - One-to-One: One test case for EACH Test Coverage item
    - More test cases but easy to automate
  - Minimized: Each test case may exercise more than one Test Coverage Items
    - Less test cases

Step 4: Derive Test Cases (TD4)
One-to-one

- Test cases for input Exam
- Test cases for input HW
- Test cases for non-integer inputs
- Test cases for valid output

Step 4: Derive Test Cases (TD4)
One-to-one

Test Case	1	2	3
Input (Exam)	60	-10	93
Input (HW)	15	15	15
Total	75	5	108
Test Coverage Item	TCOVER1	TCOVER3	TCOVER4
Partition Tested	0<=Exam<=75	Exam<0	Exam>75
Expected Output	‘A’	‘FM’	‘FM’

Step 4: Derive Test Cases (TD4) Minimized

Test Case	1	2	3	4
Input (Exam)	60	50	35	19
Input (HW)	20	16	10	8
Total	80	66	45	27
Test Coverage Item	TCOVER1 TCOVER2 TCOVER9	TCOVER1 TCOVER2 TCOVER10	TCOVER1 TCOVER2 TCOVER11	TCOVER1 TCOVER2 TCOVER12
Partition Exam	0<=Exam<=75	0<=Exam<=75	0<=Exam<=75	0<=Exam<=75
Partition HW	0<=HW<=25	0<=HW<=25	0<=HW<=25	0<=HW<=25
Partition Total	70<=Total<=100	50<=Total<70	30<=Total<50	0<=Total<30
Expected Output	‘A’	‘B’	‘C’	‘D’

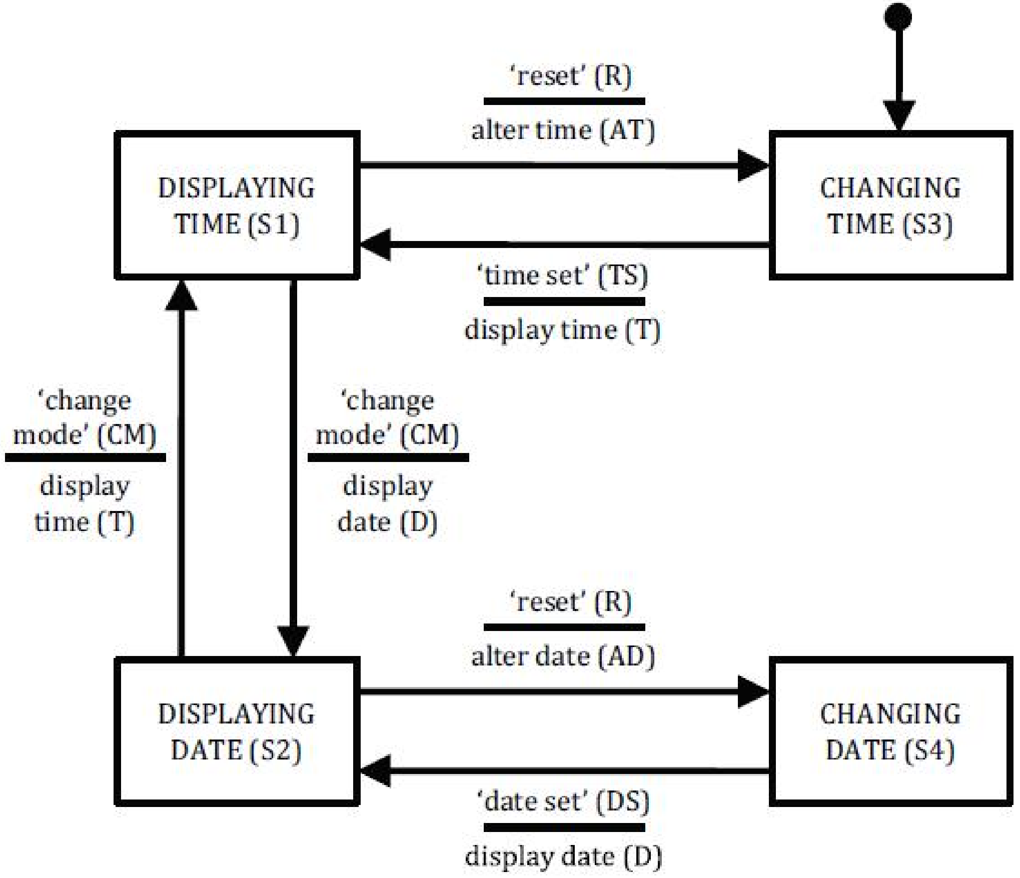
Test Coverage Measurement

- $Coverage = \left(\frac{N}{T} \times 100\right)\%$
  - 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

STATE TRANSITION TESTING

Example: Manage Display

- A function: manage_display_changes
- 4 inputs
  - Change Mode (CM)
  - Reset (R)
  - Time Set (TS)
  - Date Set (DS)

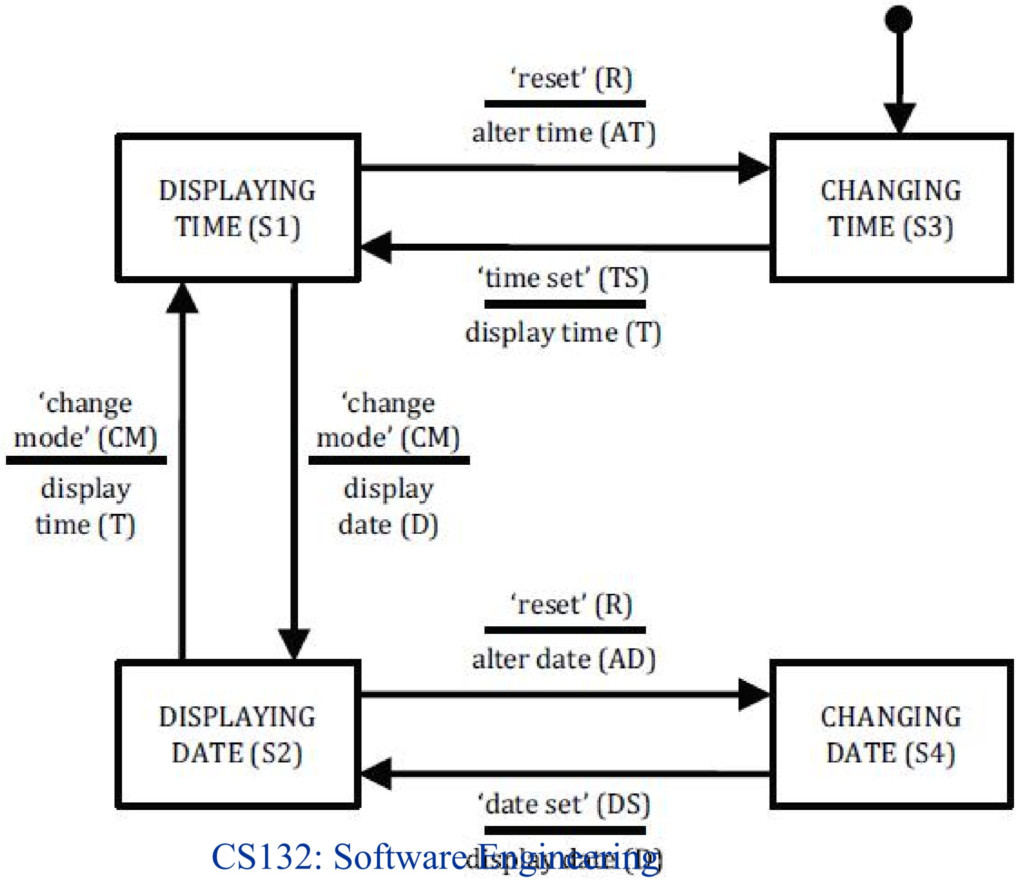


Step 1: Identify Feature Sets (TD1)

- FS1: manage_display_changes

Step 2: Derive Test Conditions (TD2)

- The state model is the test condition



Step 3: Derive Test Coverage Items

- All states
  - Test cases should visit all states in the model
- Single transition (0-switch coverage)
  - Only valid transitions
- All transitions
  - Both valid and invalid transitions
- Multiple transitions (N-switch coverage)
  - Valid sequences of N+1 transitions in the state model

Table B.31 — State table for manage_display_changes

	CM	R	TS	DS
S1	S2/D	S3/AT	S1/–	S1/–
S2	S1/T	S4/AD	S2/–	S2/–
S3	S3/–	S3/–	S1/T	S3/–
S4	S4/–	S4/–	S4/–	S2/D

Step 3: Derive Test Coverage Items (TD3)

- TCOVER1: S1 to S2 with input CM (valid)
- TCOVER2: S1 to S3 with input R (valid)
- …

Table B.31 — State table for manage_display_changes

	CM	R	TS	DS
S1	S2/D	S3/AT	S1/–	S1/–
S2	S1/T	S4/AD	S2/–	S2/–
S3	S3/–	S3/–	S1/T	S3/–
S4	S4/–	S4/–	S4/–	S2/D

Step 4: Derive Valid Test Cases (TD4) 0-switch test cases

- 0-switch test cases
- Invalid test cases should not cause state changes

Table B.31 — State table for manage_display_changes

	CM	R	TS	DS
S1	S2/D	S3/AT	S1/–	S1/–
S2	S1/T	S4/AD	S2/–	S2/–
S3	S3/–	S3/–	S1/T	S3/–
S4	S4/–	S4/–	S4/–	S2/D

Table B.33 — 0-switch test cases for manage_display_changes

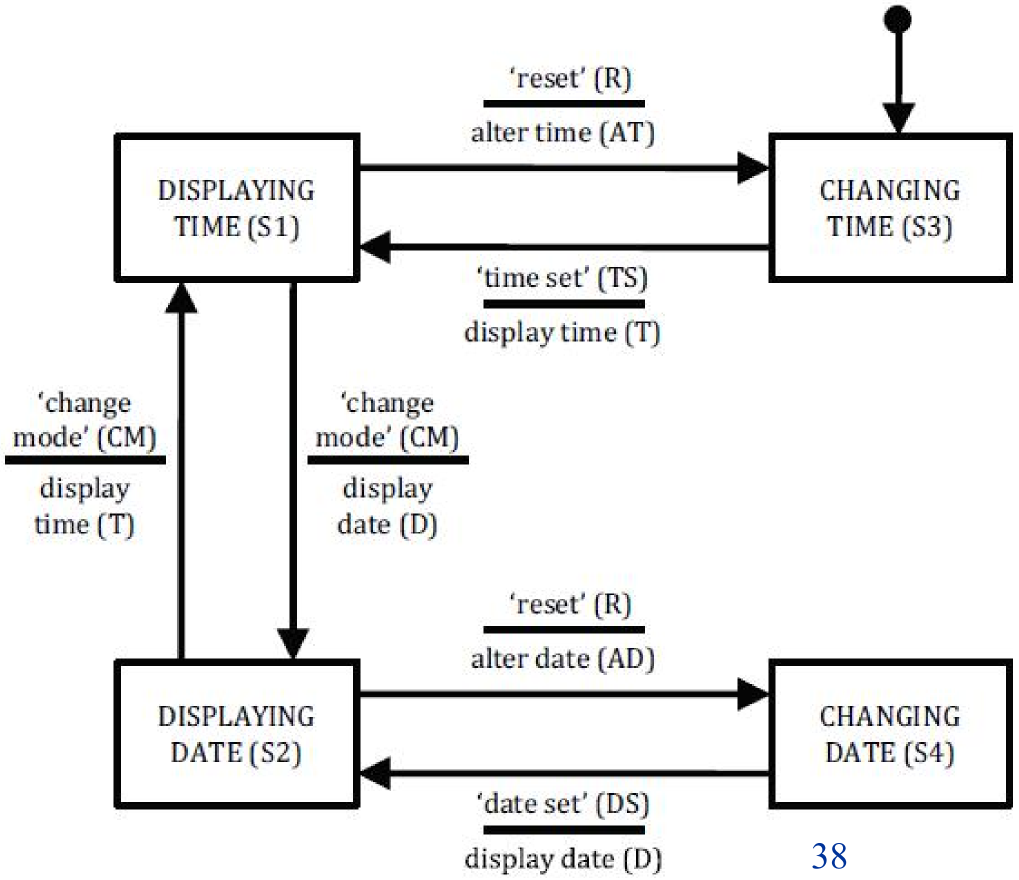
Test Case	1	2	3	4	5	6
Start State	S1	S1	S2	S2	S3	S4
Input	CM	R	CM	R	TS	DS
Expected Output	D	AT	T	AD	T	D
Finish State	S2	S3	S1	S4	S1	S2
Test Coverage Item	1	2	5	6	11	16

Step 4: Derive Valid Test Cases (TD4)
1-switch test cases

- TCOVER 17: S1 to S2 to S1 with inputs CM and CM
- …

Table B.35 — 1-switch test cases for manage_display_changes

Test Case	17	18	19	20	21	22	23	24	25	26
Start State	S1	S1	S1	S3	S3	S2	S2	S2	S4	S4
Input	CM	CM	R	TS	TS	CM	CM	R	DS	DS
Expected Output	D	D	AT	T	T	T	T	AD	D	D
Next State	S2	S2	S3	S1	S1	S1	S1	S4	S2	S2
Input	CM	R	TS	CM	R	CM	R	DS	CM	R
Expected Output	T	AD	T	D	AT	D	AT	D	T	AD
Finish State	S1	S4	S1	S2	S3	S2	S3	S2	S1	S4
Test Coverage Item	17	18	19	20	21	22	23	24	25	26



Step 5: Assemble Test sets

- TS1: 0 switch test cases - Test cases 1,2,3,4,5,6
- More efficient if rearranged to 5,1,4,6,3,2
  - The finish state of test case n is the start state of test case n+1

Table B.33 — 0-switch test cases for manage_display_changes

Test Case	1	2	3	4	5	6
Start State	S1	S1	S2	S2	S3	S4
Input	CM	R	CM	R	TS	DS
Expected Output	D	AT	T	AD	T	D
Finish State	S2	S3	S1	S4	S1	S2
Test Coverage Item	1	2	5	6	11	16

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

- Branch testing
  - stronger testing than statement coverage testing.

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)

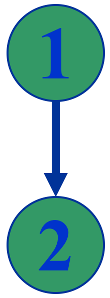
- A control flow graph (CFG) describes:
  - the sequence in which different instructions of a program get executed.
  - the way control flows through the program.

How to draw Control flow graph?

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

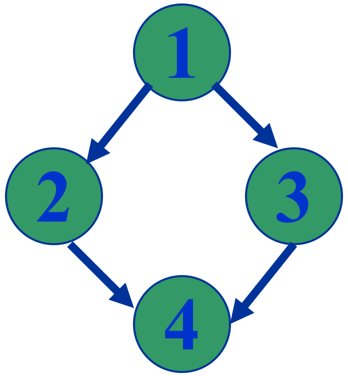
How to draw Control flow graph?

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



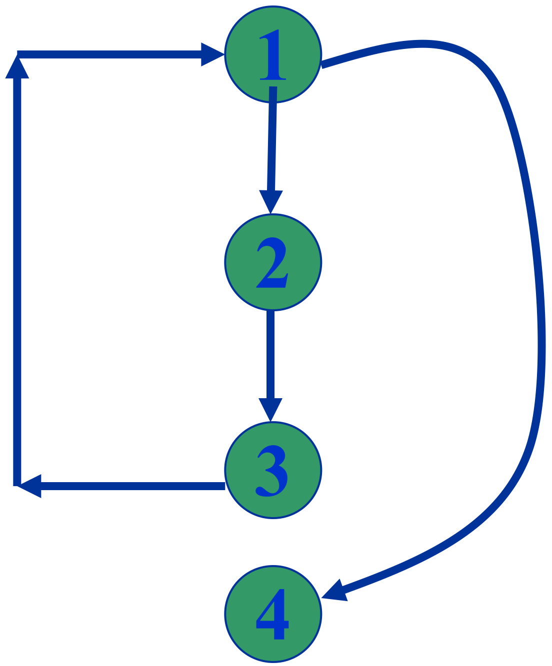
How to draw Control flow graph?

- 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

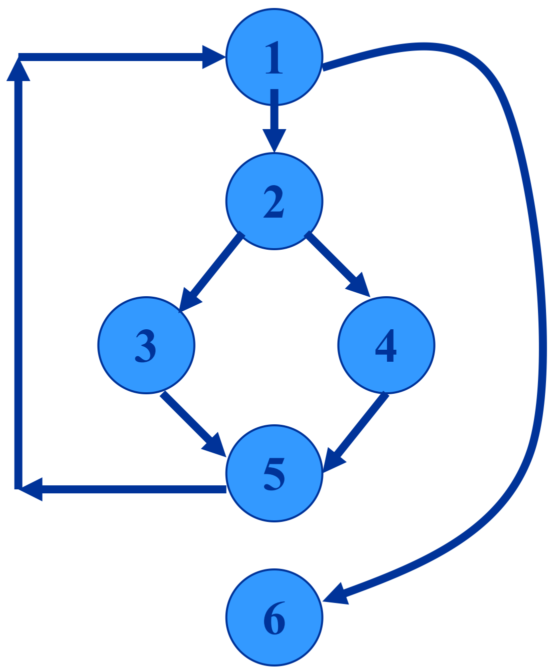
- A path through a program:
  - a node and edge sequence from the starting node to a terminal node of the control flow graph.
  - There may be several terminal nodes for program.

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.

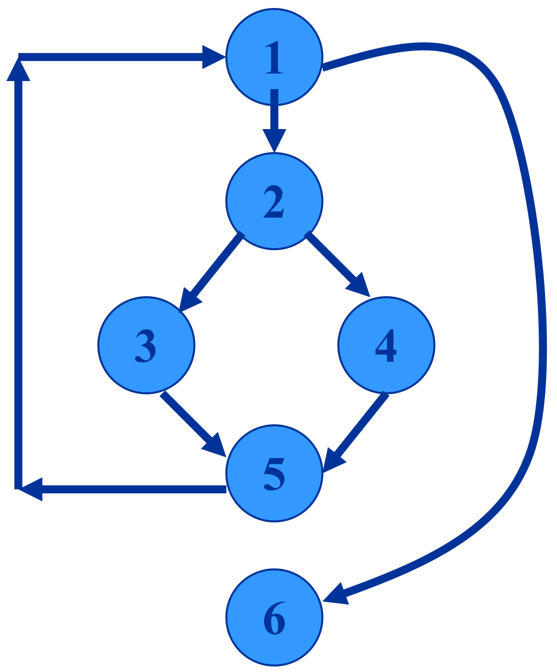
Example

- 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

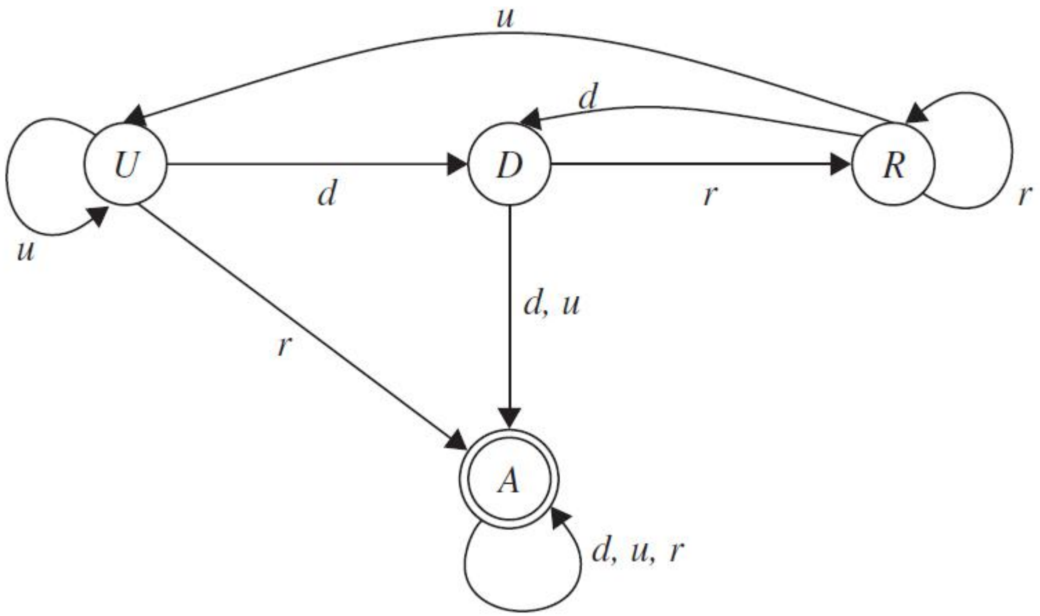
- Motivation
  - How do you know that a variable is assigned the correct value?
  - From: when the value is assigned
  - To: when the value is used later
- Process
  - Draw a data flow graph from a program.
  - Select one or more data flow testing criteria.
  - Identify paths in the data flow graph satisfying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

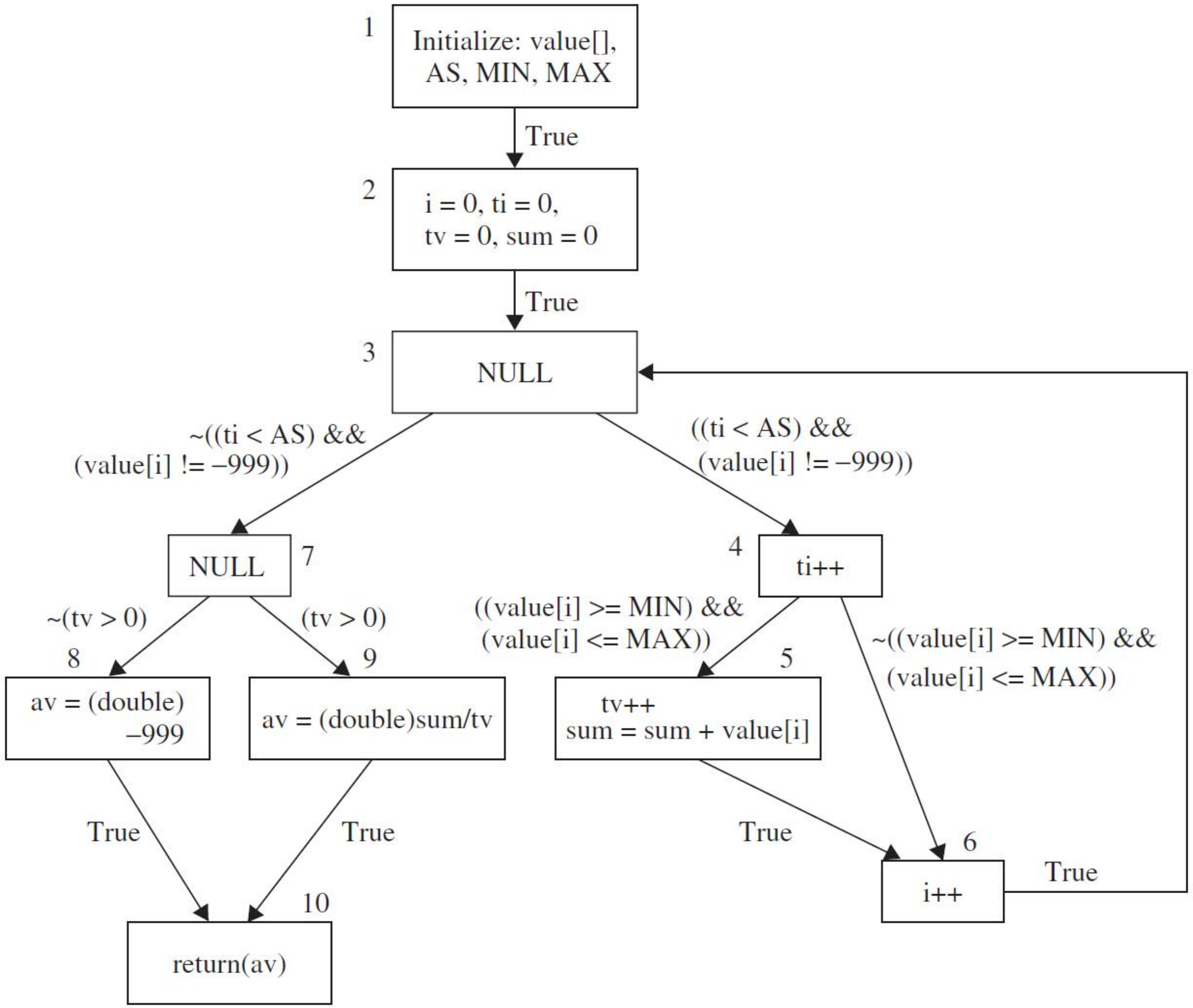
Undefinition

Data flow diagram construction

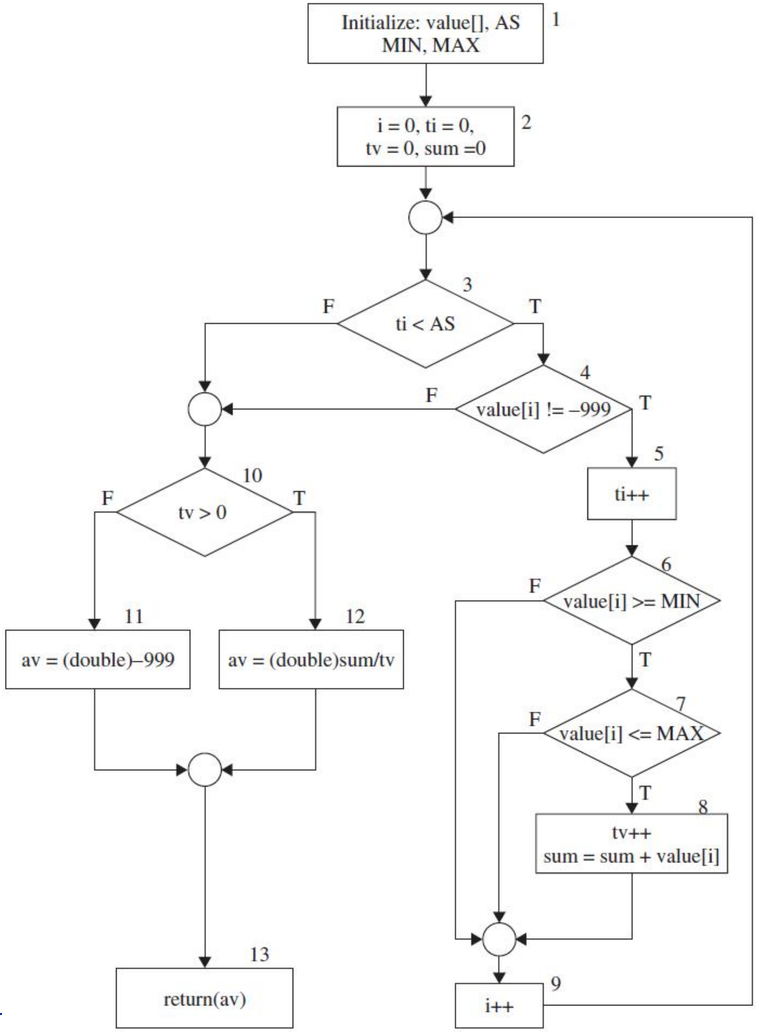
- A sequence of definitions and c-uses is associated with each node of the graph.
- A set of p-uses is associated with each edge of the graph.
- The entry node has a definition of each parameter and each nonlocal variable which occurs in the subprogram.
- The exit node has an *undefinition* of each local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

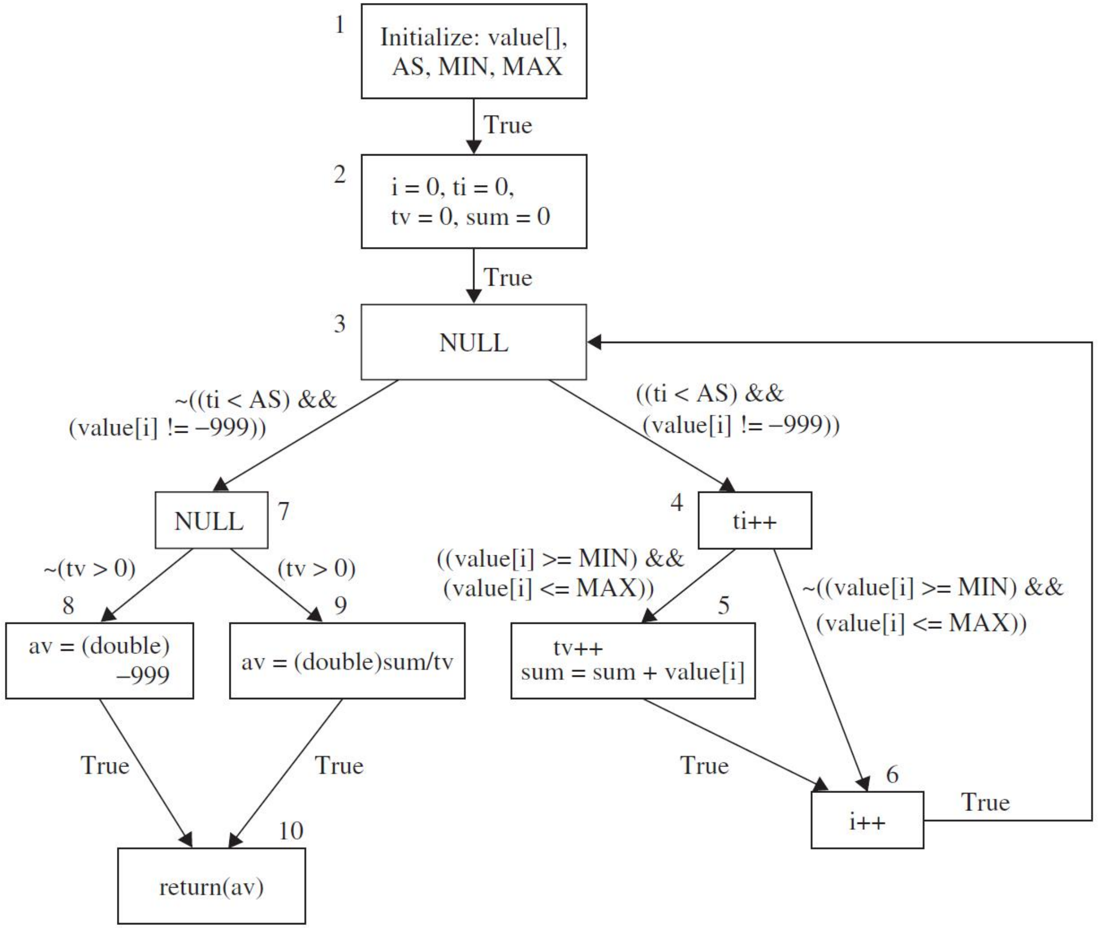
Example

```
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);
}
```



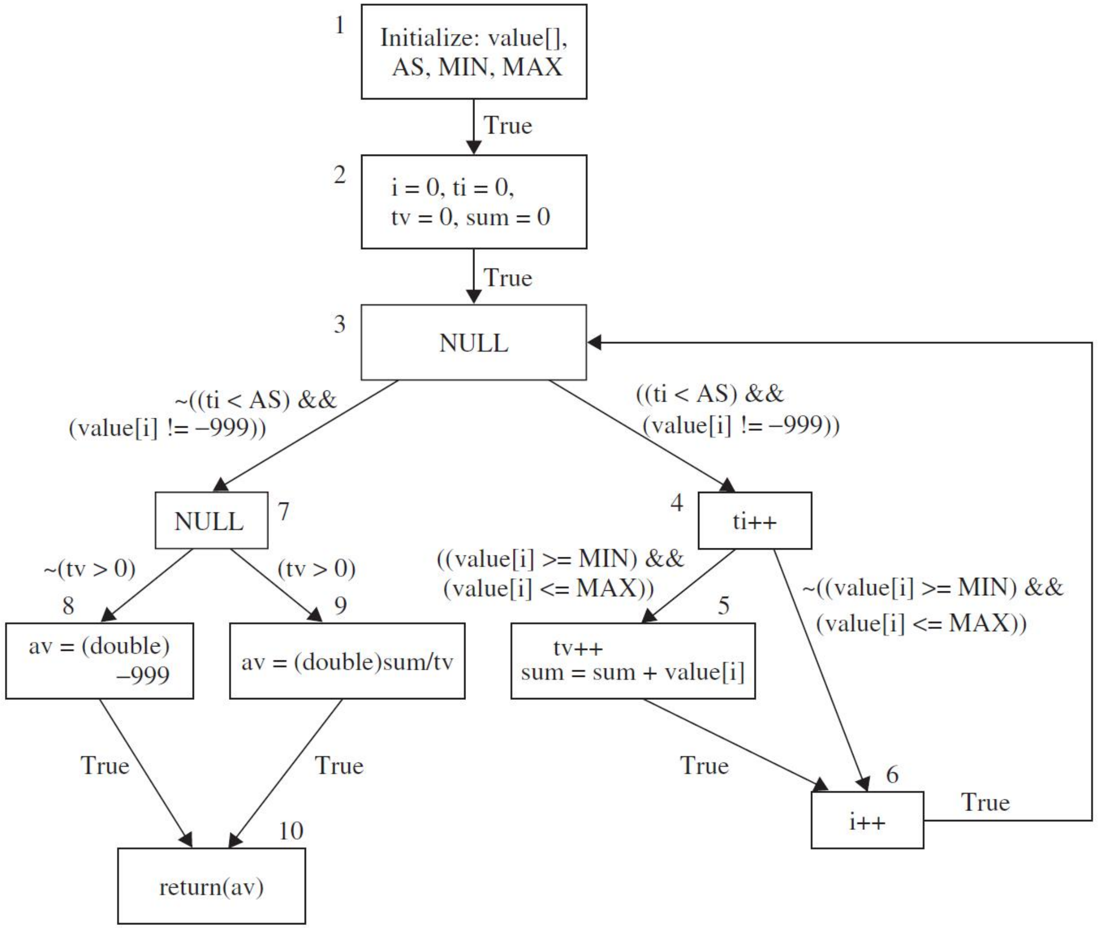
Control flow graph vs. Data flow graph





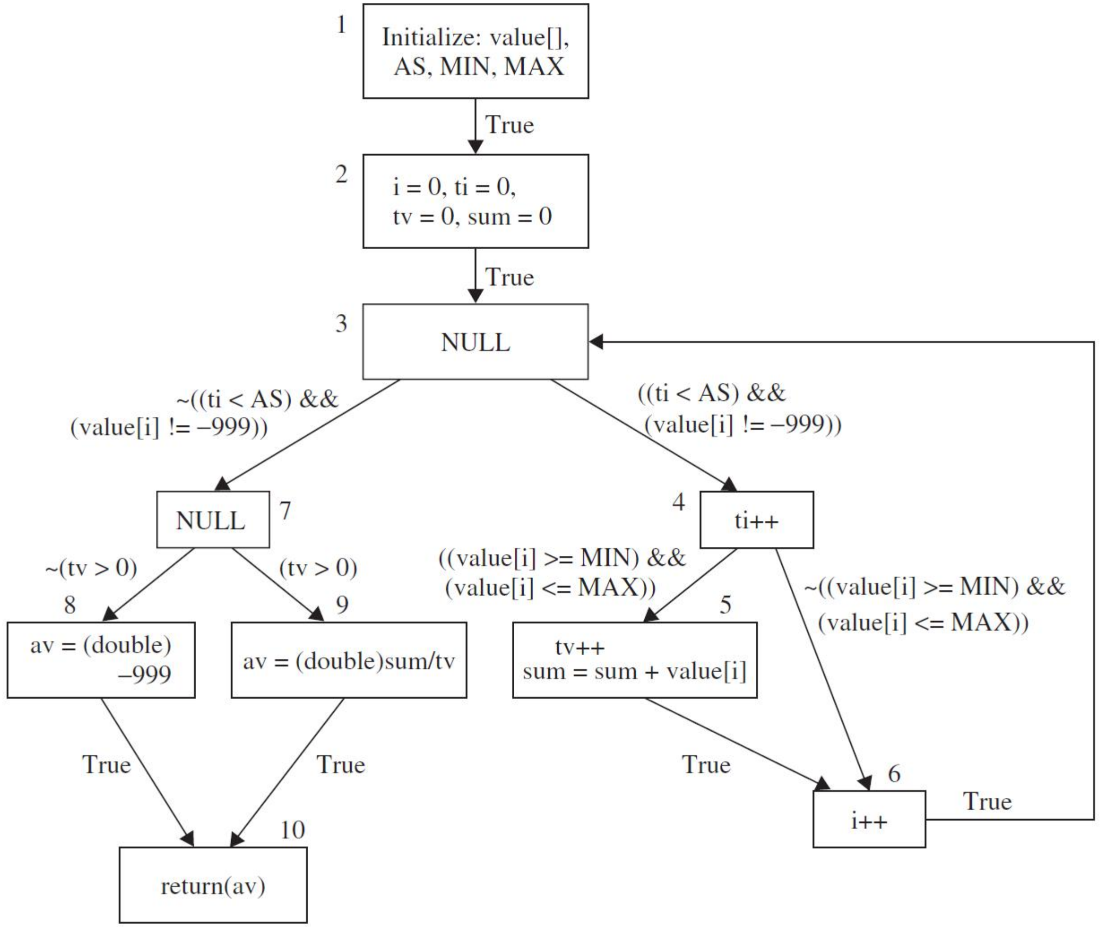
Path selection criteria

- 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



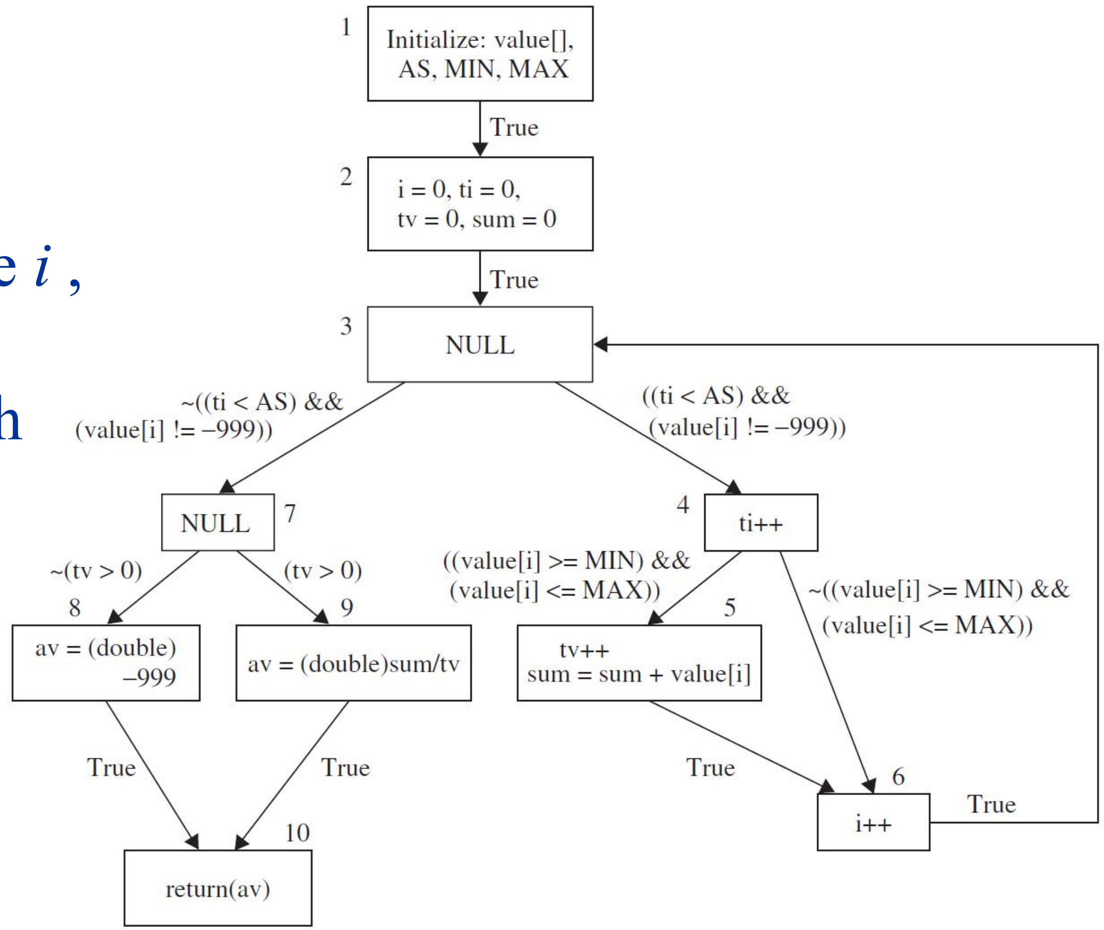
Data flow testing criteria

- 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
  - 1,2,3,4,5,6,3,7,9,10 is a all def path
  - 2,3,7,8 is also a def-clear path for tv
  - 1,2,3,7,8,10 is a all def path



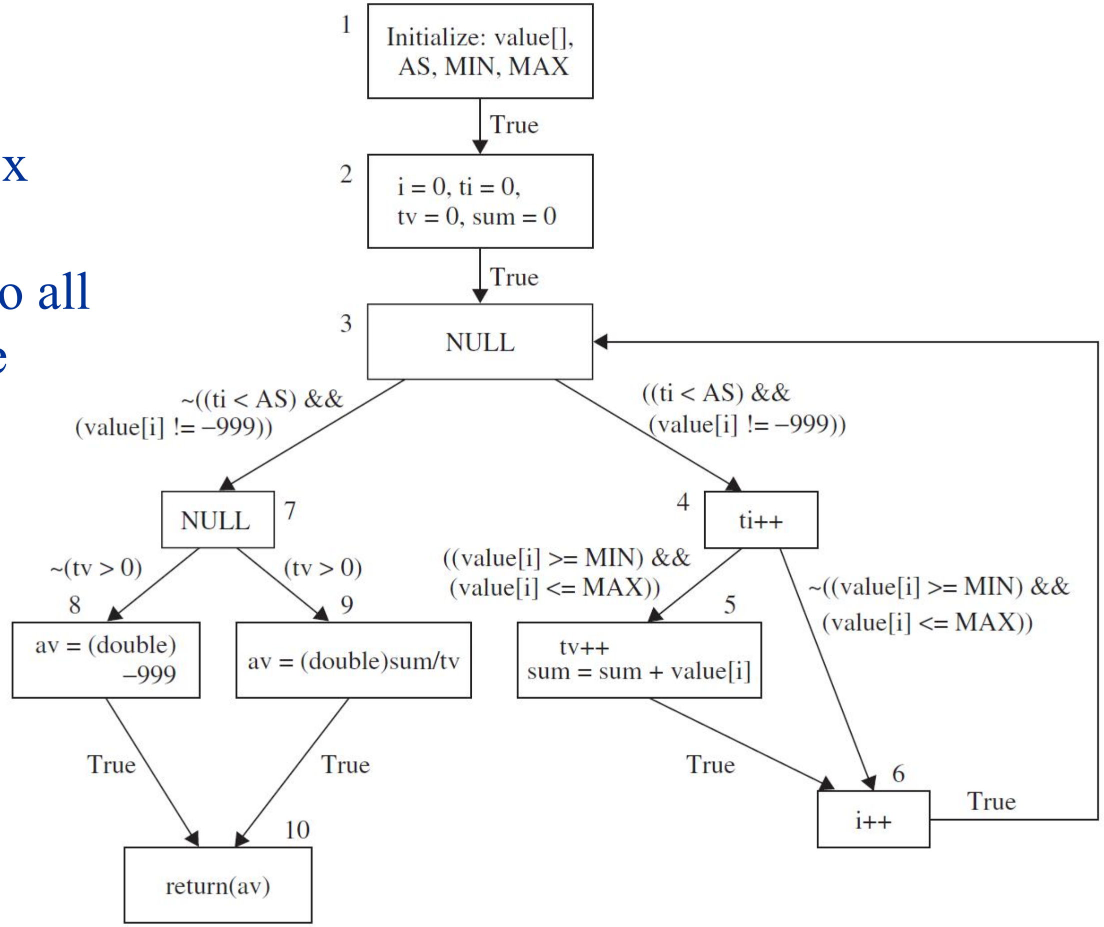
Data flow testing criteria (cont.)

- 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.)

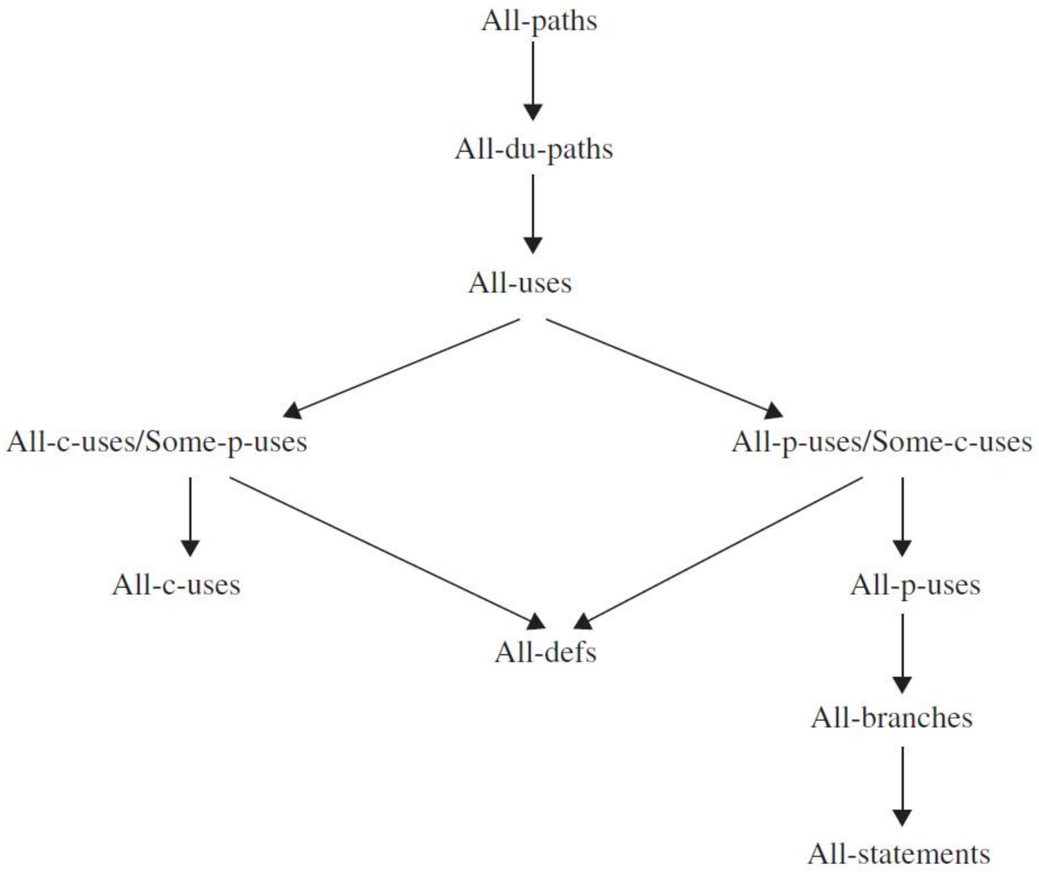
- All-p-uses:
  - For each variable x and for each node i such that x has a global definition in node i , select complete paths which include def-clear paths from node i to all edges (j ,k) such that there is a p-use of x on edge (j ,k).
  - i.e. 2,3,7,8; 2,3,7,9; 5,6,3,7,8; 5,6,3,7,9 for tv
    - 1-2-3-7-8-10,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



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

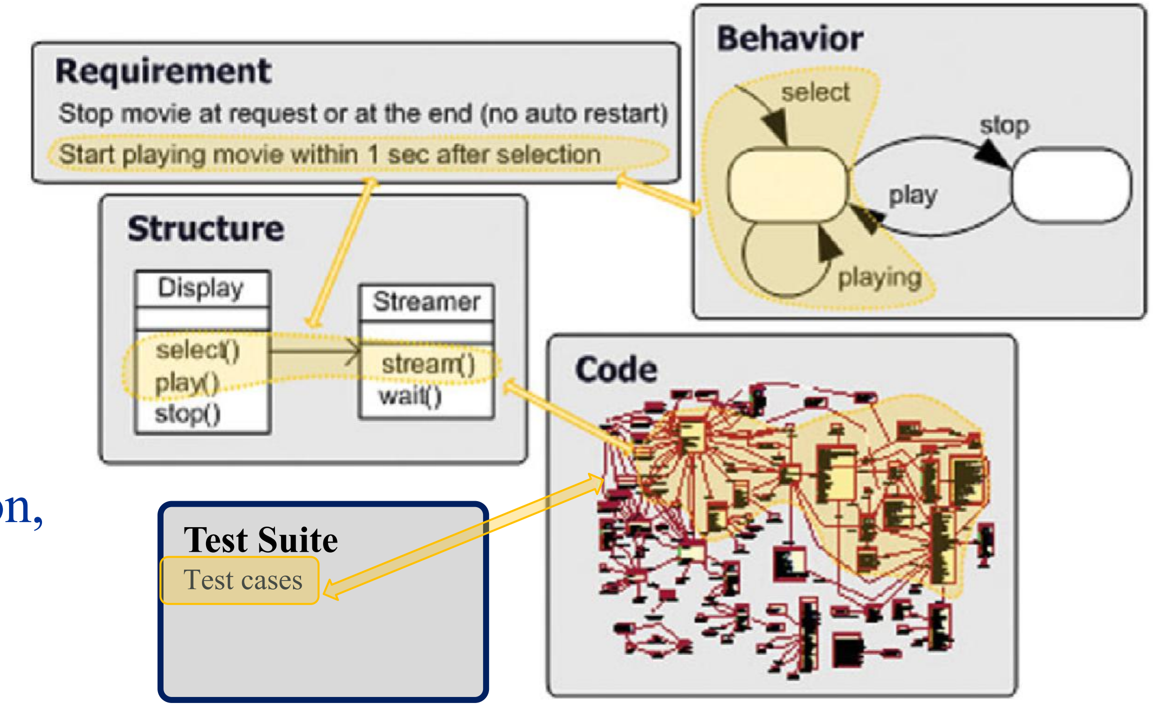
Reference

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

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.

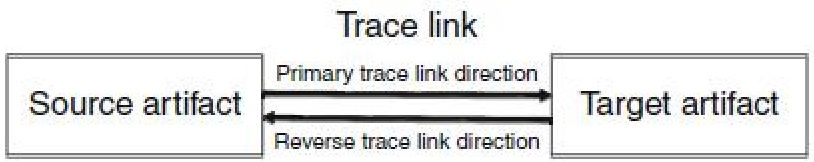


Objectives of Traceability

- 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