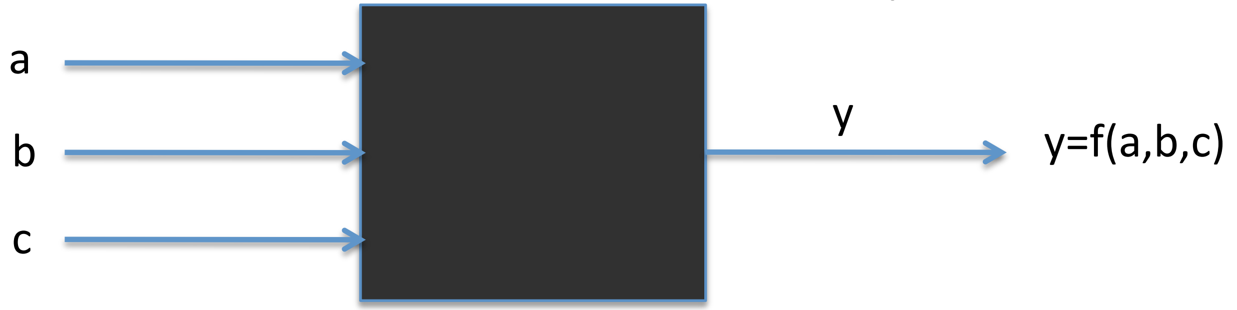
Verification and Validation (PA2516)

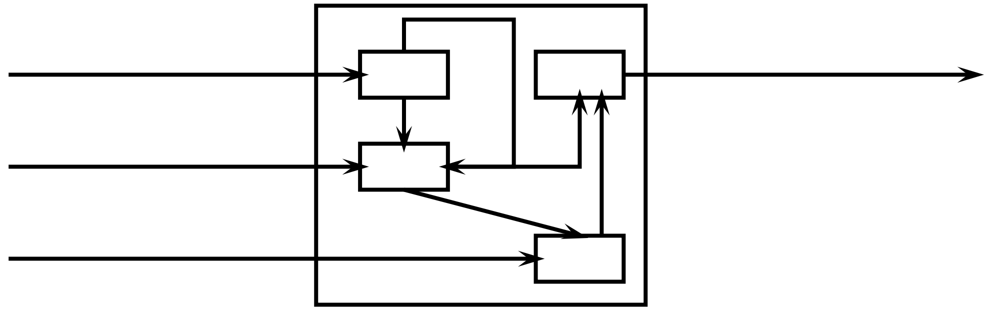
Lecture 3: Dynamic V&V (Structure Based)

A simple classification of software testing

- Specification based testing (black-box testing)
  - Test-case selection based on requirements specification
  - Structure of the program is not considered
  - Desired behavior is determined based on the specification



- Structural testing (White-box testing)
  - Test case selection based on code/system structure
  - Desired behavior determined based on the specification



Structural testing

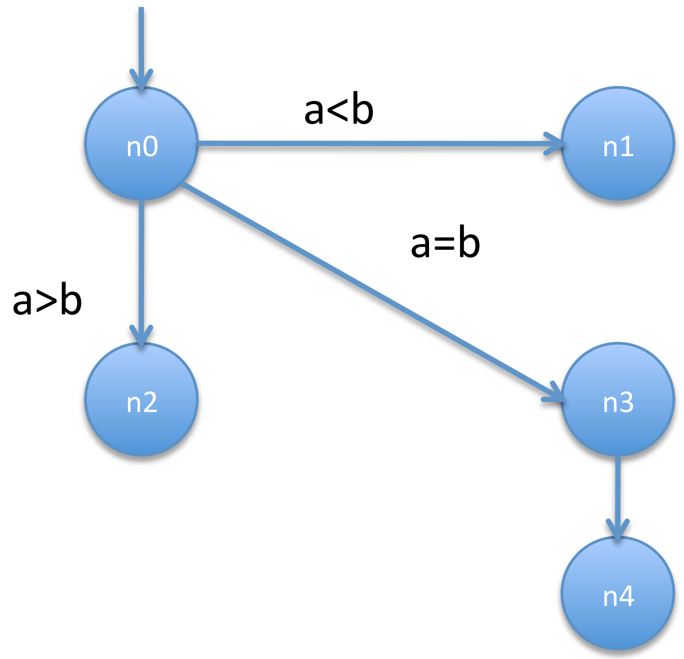
- Test cases are selected so that the program is executed systematically.
- Test reference is the source code
- The completeness of the test is based on the structure of the code/system
  - e.g. statement coverage = (exercised statements/total statements) * 100
- To know whether the test results derived from the system structure are correct, the spec is used as a reference (but not for test case derivation!!)

Structural testing

- In comparison to specification based testing:
- The advantages are:
  - Formal test criteria can be easily and efficiently defined (tool support!)
  - Techniques can also be used in specification based testing
    - Then it is applied on models built in the requirements engineering phase
      - Behavioral models, e.g. finite state machines, or
      - activity diagrams (see activity diagram derived from use case specification in previous lecture)
- The disadvantage is:
  - Missing functionalities described in the specification might not be recognized

In order to test code, the code is translated into a model/graph

- Graph Example



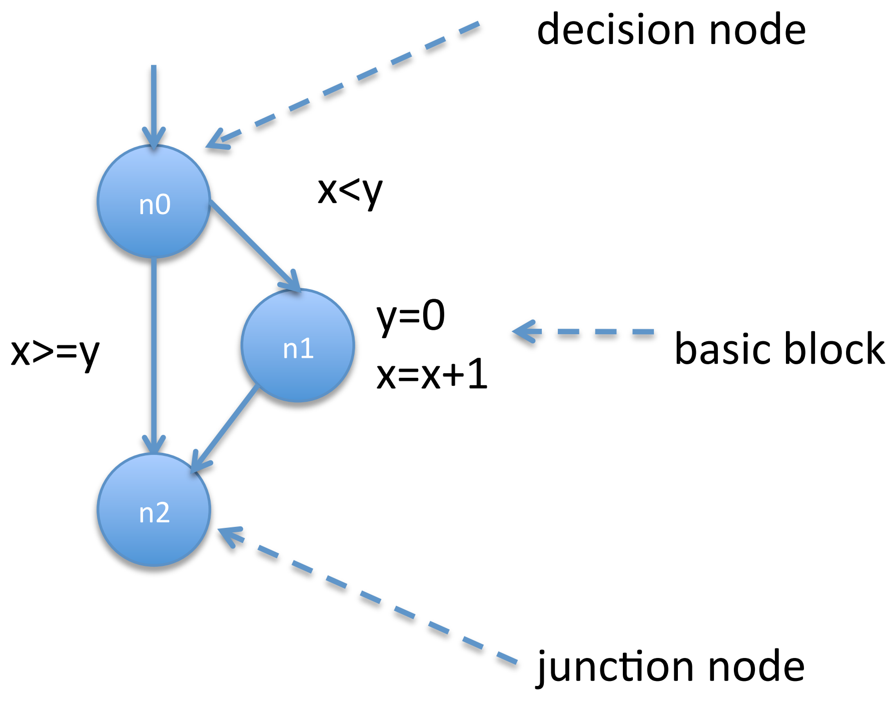
TC1: a=0, b=1
Test Path: n0, n1

TC2: a=0, b=0
Test Path: n0, n3, n4

In order to test code, the code is translated into a model/graph

- In order to analyze a graph for different coverage criteria, we need to derive it from the source code.

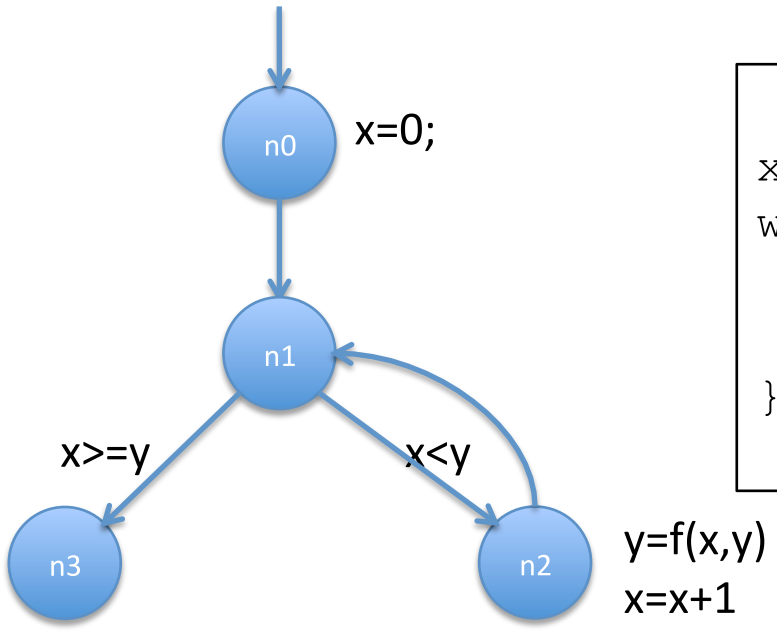
If structure (without an else)



```
if (x<y) {
    x=0;
    x=x+1;
}
```

In order to test code, the code is translated into a model/graph

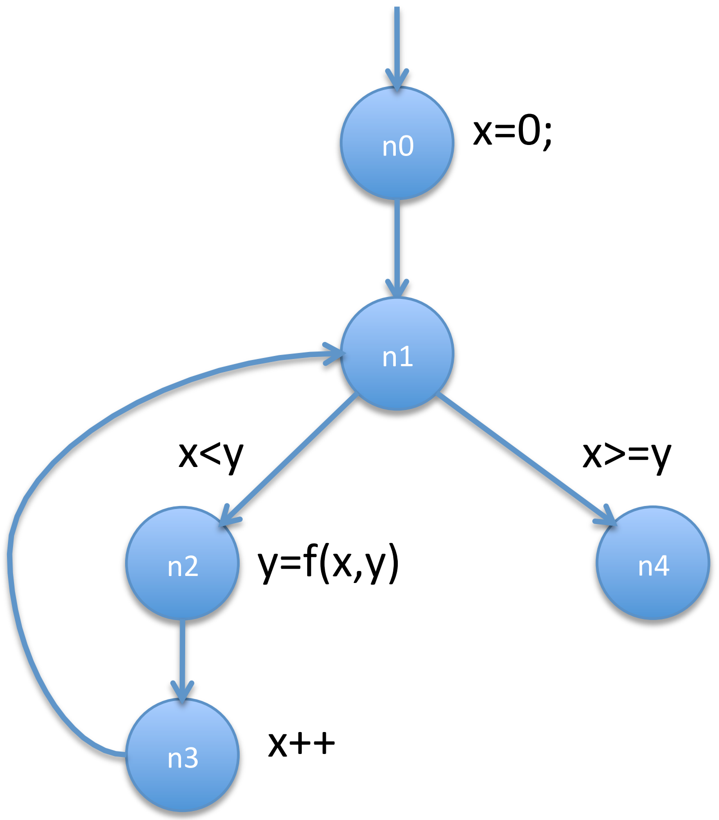
While loop structure



```
x=0
while (x<y){
    y=f(x,y);
    x=x+1;
}
```

In order to test code, the code is translated into a model/graph

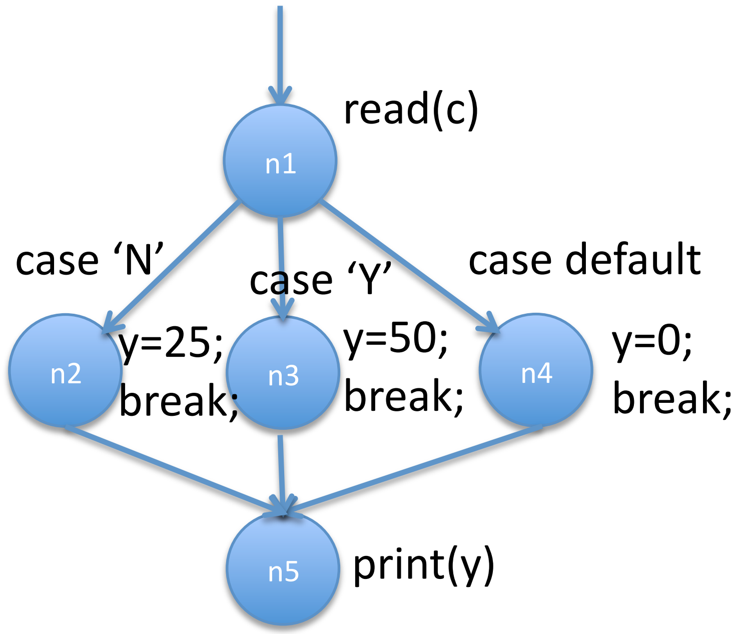
For-Loop



```
for(x=0; x<y; x++){
    y=f(x,y);
}
```

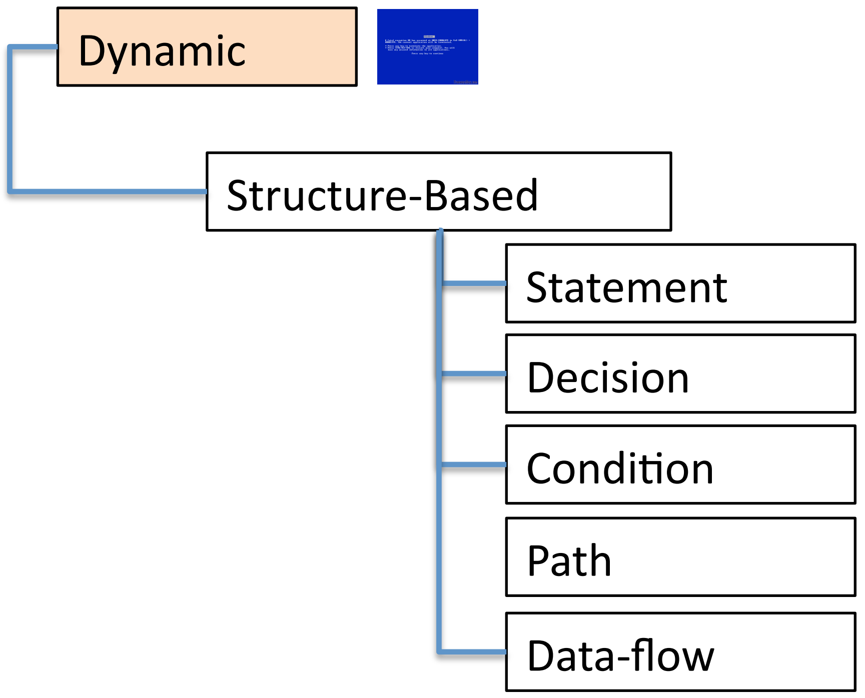
In order to test code, the code is translated into a model/graph

Switch-Statement

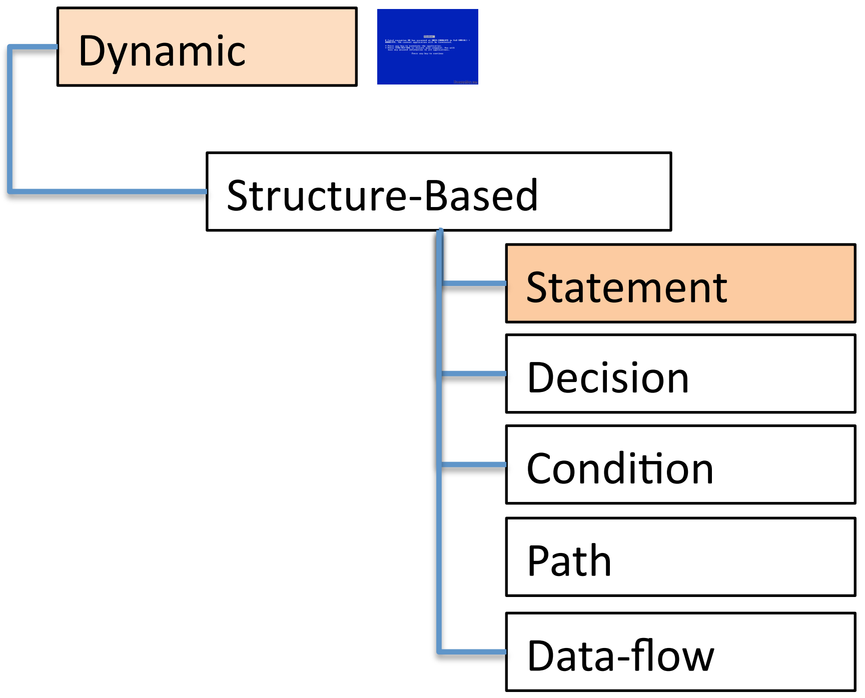


```
read(c);
switch(c)
{
case 'N':
    y=25;
    break;
case 'Y':
    y=50;
    break;
default:
    y=0;
    break;
}
print(y)
```

So … how do you select test cases based on the code/system structure?

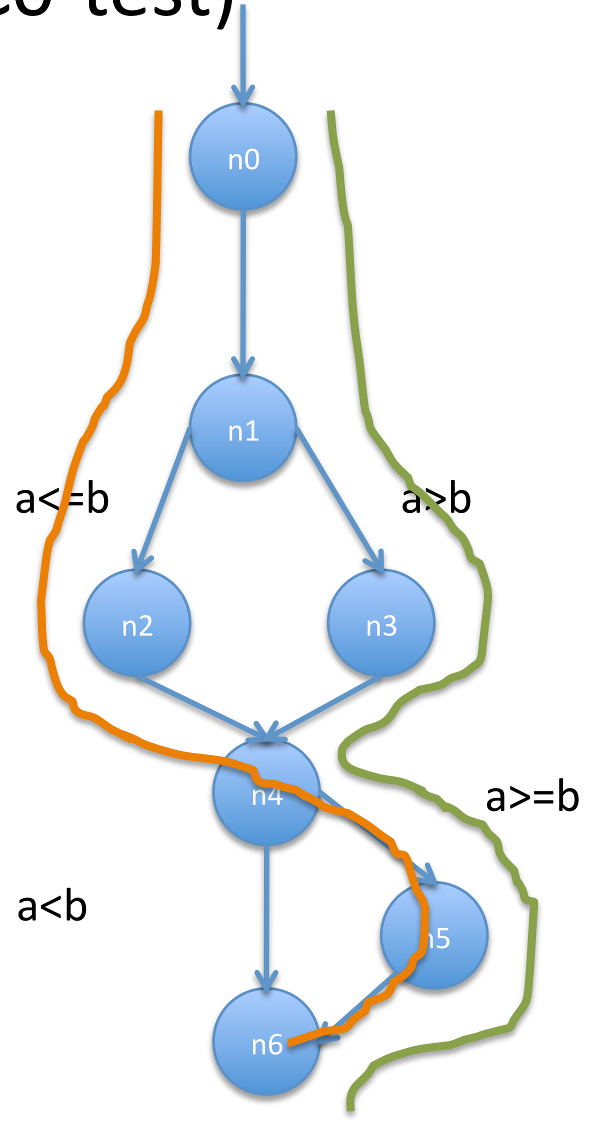


So … how do you select test cases based on the code/system structure?



Statement coverage (C0-test)

- Has every statement been executed?
- A set of test cases fulfills the test requirement for statement coverage (C0) when for each statement A in program P there exists a test case executing A.
- Coverage measurement:
  - C0 Cov.=number of executed statements/ total number of statements

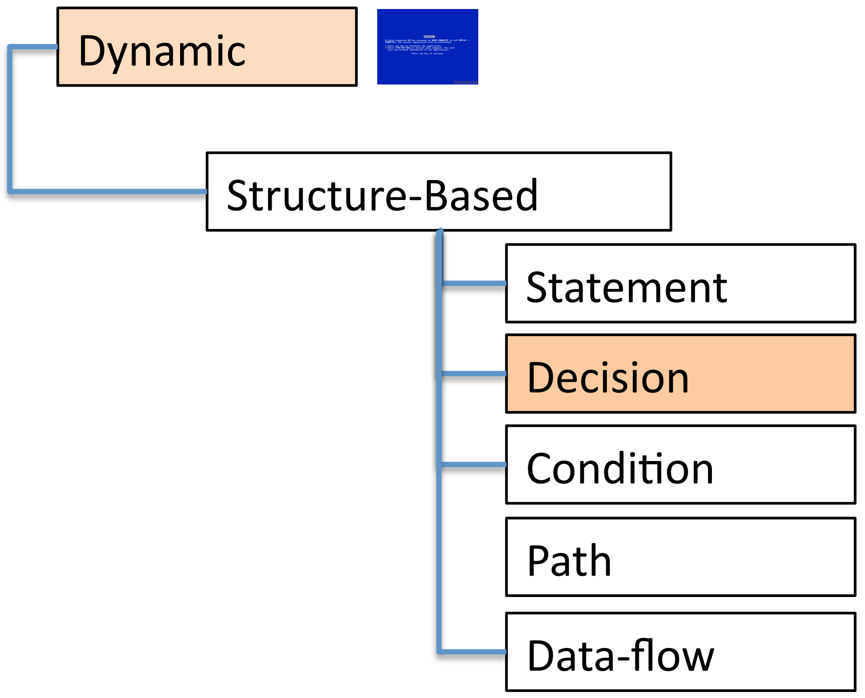


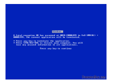
TC1: a=2, b=1; Path: n0, n1, n3, n4, n5, n6
TC2: a=2, b=2; Path: n0, n1, n2, n4, n5, n6

Statement Coverage

- Low defect detection capability
- Reasons:
  - No data dependencies are considered
  - Every statement is of equal importance
  - Loops only executed once to get to the statement
  - No mistakes detected in connection to testing combinations of different decisions
- Helps to find dead code

So … how do you select test cases based on the code/system structure?

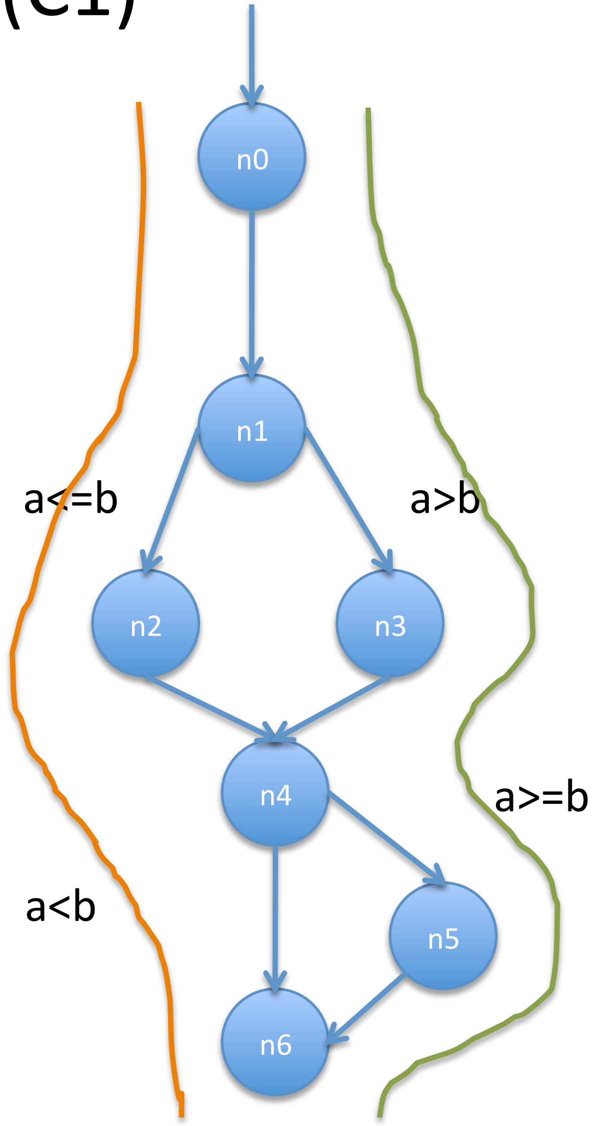




Decision Coverage (C1)

- Has every edge in the program been executed?
- A set of test cases T fulfills decision coverage (C1) if for each edge e in the graph P there is a path in the test paths to which e belongs, i.e. every decision has to be true or false once
- Coverage measurement:
  - C1 Cov = executed edges/total number of edges

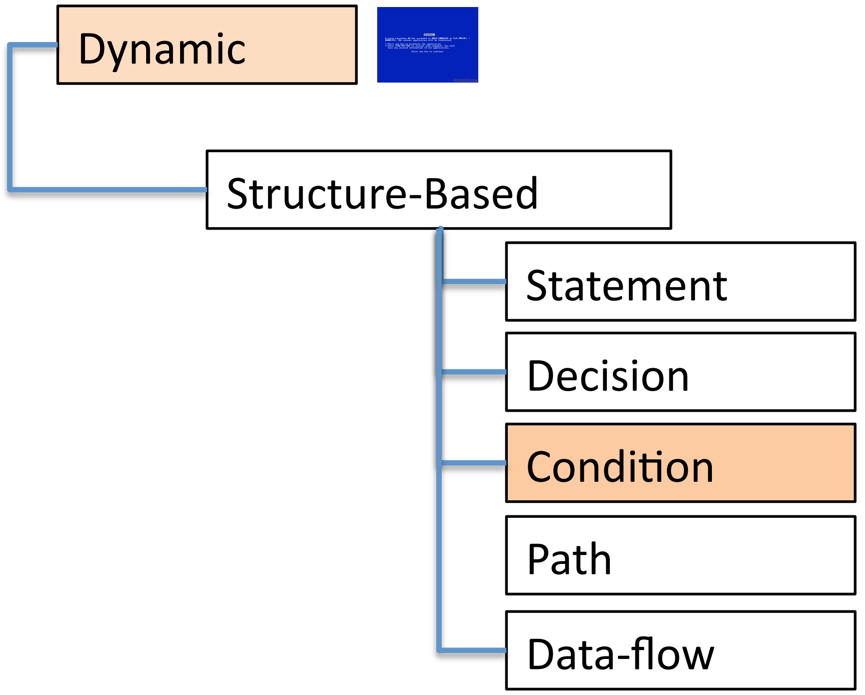
TC1: a=3, b=4; Path: n0, n1, n2, n4, n6
TC2: a=4, b=3; Path: n0, n1, n3, n4, n5, n6



Decision Coverage

- Problematic relationship between number of test cases and coverage
  - Considered the minimum test criterion
  - Not well suited to assess loops properly, and also not suited to test combinations of branches
  - Combined/nested decisions are not properly tested
    - e.g. (((A AND B) OR (C AND D))) AND (F OR G)

So … how do you select test cases based on the code/system structure?



Condition coverage

- Example of a condition:
  - (x > 1) && ((u == 0) || (v == 0))
  - The decision contains contains a hierarchy of conditions evaluating to true or false
  - Conditions are connected through logic operators
  - At the lowest level: e.g. (x > 1)
- C1-test: Each decision has to evaluate to true / false
- Condition Coverage (C2 and C3) – atomic conditions also have to evaluate to true / false

Condition coverage

- Simple condition coverage
  - Test of all atomic conditions against true/false
  - does not fulfill C1-test
- Min. multiple condition coverage (Condition/Decision)
  - In addition to the atomic conditions all partial decisions in the decision hierarchy (also including the root) have to be evaluated against true/false
- Modified condition coverage (MC/DC)
  - Check whether atomic terms/clauses have an effect on the root-decision in the decision hierarchy independently of other atomic decisions
- Multiple decision coverage
  - test all combinations of true/false on atomic conditions
  - includes the C1 test
  - includes all other condition coverage criteria
  - high effort: 2^n tests (n number of decisions in tree)

Simple condition coverage

- Simple condition coverage
  - Test of all atomic decisions against true/false in their clauses
  - does not fulfill C1-test
- Example: (a && b)
  - a/b have to be true/false at least once
  - TC1: a=T, b=F
  - TC2: a=F, b=T
- Hence, it does not include C1 coverage

for above example: fulfilled with TC6 and TC11

((A || B) && (C || D))

	A	B	C	D	(A || B)	(C || D)	((A || B) && (C || D))
1	f	f	f	f			
2	f	f	f	t			
3	f	f	t	f			
4	f	f	t	t			
5	f	t	f	f			
6	f	t	f	t	t	t	t
7	f	t	t	f			
8	f	t	t	t			
9	t	f	f	f			
10	t	f	f	t			
11	t	f	t	f	t	t	t
12	t	f	t	t			
13	t	t	f	f			
14	t	t	f	t			
15	t	t	t	f			
16	t	t	t	t			

Minimum Multiple Condition Coverage

((A || B) && (C || D))

- In addition to the atomic conditions all partial decisions in the decision hierarchy (also including the root) have to be evaluated against true/ false

for above example: fulfilled with TC1 and TC16

	A	B	C	D	(A || B)	(C || D)	((A || B) && (C || D))
1	f	f	f	f	f	f	f
2	f	f	f	t			
3	f	f	t	f			
4	f	f	t	t			
5	f	t	f	f			
6	f	t	f	t			
7	f	t	t	f			
8	f	t	t	t			
9	t	f	f	f			
10	t	f	f	t			
11	t	f	t	f			
12	t	f	t	t			
13	t	t	f	f			
14	t	t	f	t			
15	t	t	t	f			
16	t	t	t	t	t	t	t

Modified Condition Coverage (MC/DC)

- every condition true/false at least once
- every decision true/false at least once
- every condition in the decision should be shown to effect the outcome
- This is e.g .required by NASA

for above example: fulfilled with TC4, TC6, TC7 and TC8

	A	B	C	(A || B || C)
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	T
5	F	T	T	T
6	F	T	F	T
7	F	F	T	T
8	F	F	F	F

A does not affect the outcome when being changed

In case 4: A(T->F) effects outcome
In case 6: B(T->F) affects outcome
In case 7: C(T->F) affects outcome
In case 8: Needed so that the overall decision turns to F

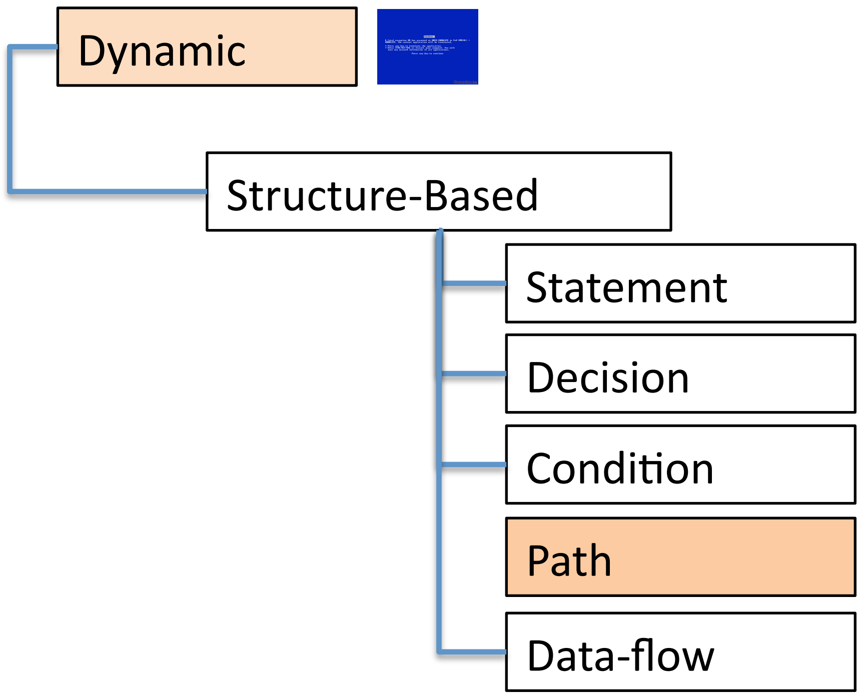
Multiple Decision Coverage

- test all combinations of true/false on atomic conditions

((A || B) && (C || D))

	A	B	C	D	(A || B)	(C || D)	((A || B) && (C || D))
1	f	f	f	f	f	f	f
2	f	f	f	t	f	t	f
3	f	f	t	f	f	t	f
4	f	f	t	t	f	t	f
5	f	t	f	f	t	f	f
6	f	t	f	t	t	t	t
7	f	t	t	f	t	t	t
8	f	t	t	t	t	t	t
9	t	f	f	f	t	f	f
10	t	f	f	t	t	t	t
11	t	f	t	f	t	t	t
12	t	f	t	t	t	t	t
13	t	t	f	f	t	f	f
14	t	t	f	t	t	t	t
15	t	t	t	f	t	t	t
16	t	t	t	t	t	t	t

So … how do you select test cases based on the code/system structure?

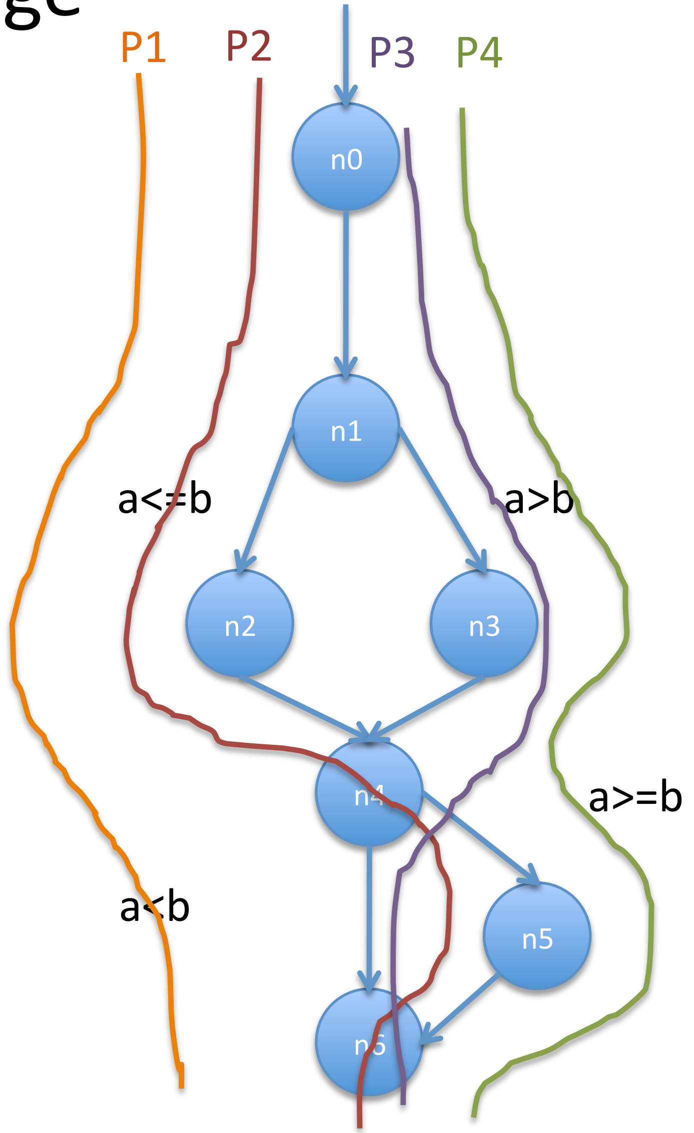


Path Coverage

- C7 test is the most exhaustive control-flow based testing method
- Goal: All different paths should be executed at least once
- Two paths are considered identical if they have exactly the same order of nodes
- Loops should be tested for every possible run-through

Test coverage of 75%

	a	b
P1	1	2
P2	2	2
P3	-	.
P4	2	1



Path Coverage (Problems)

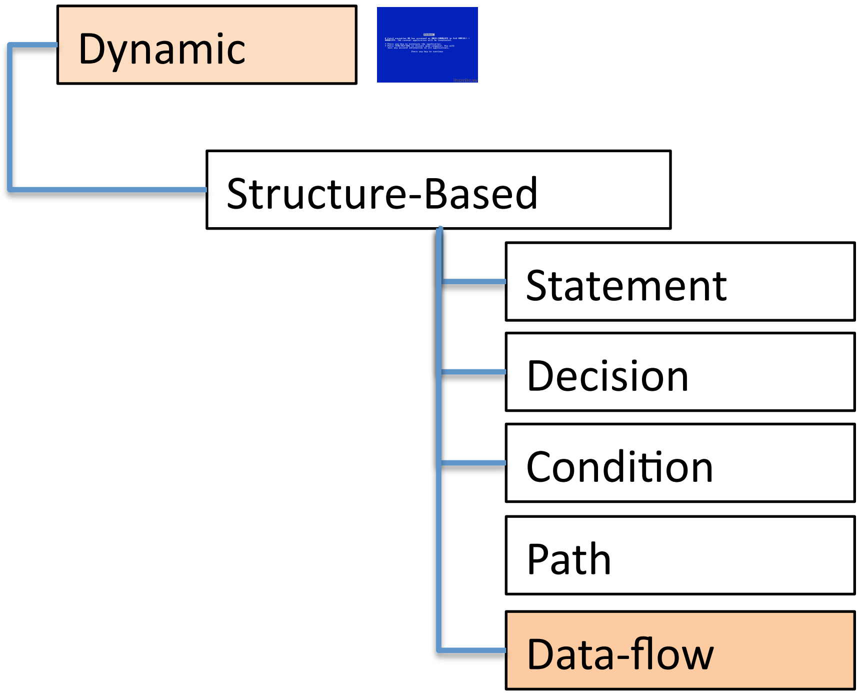
- Limitations for practical usage:
  - Number of combinations is astronomically high or infinite (e.g. infinite loop)
  - It is only visible which paths could be reached, but not under which conditions (e.g. user enters correct password to proceed)
  - Derivation from program code -> for complex systems the evaluation of the result is highly complex

Path Coverage (Solution)

- Partial coverage of loops
  - Not all loop run-throughs are considered
  - Only paths are considered that test new aspects of a loop
- Focus on five aspects:

Aspect	Test focus
0 iterations	Loop not entered
1 iteration	Reveals initialization mistakes
2 iterations	Tests the body of the loop
typical number of iterations	Test actual usage
Last iteration	Test exit criterion

So … how do you select test cases based on the code/system structure?



Data-flow based techniques

- Two types of instructions:
  - Control of flow (If, while, goto)
  - Manipulation of data
- Data-flow based techniques: Access of variables is the focus of this test
- Lifecycle of a variable:
  - Memory is allocated
  - read variable (reference, r, use, u)
  - write variable (definition, def, write)
  - Free memory (free, dispose)

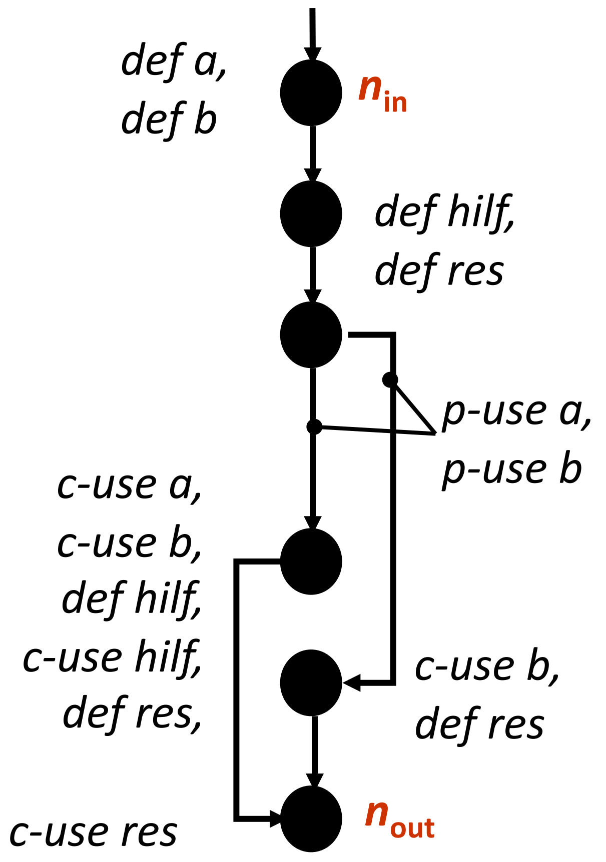
Data-flow based techniques

- Reasons for reading a variable:
  - Computational use (**c-use**)
    - Read variables for a calculation
    - Example: y= a*b, c-use(a,b)
  - Predicate use (**p-use**)
    - Determine whether a decision evaluates to true/false
    - Example: if (a==b), p-use(a,b)
- The control flow is complemented with information about data access (c-use, p-use, def)
- Statements are aggregated to program blocks if they belong to one node in the program.
  - Decision is a block
  - Statement, that is accessible through more than one control flow path belongs to an own block

Data-flow based techniques

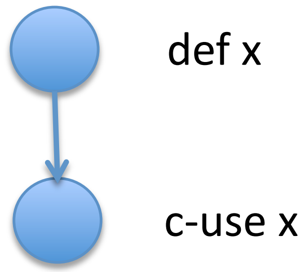
```
int doIt(int a, int b) {
  int hilf, res = 0;
  if  (a > b)
     hilf = a-b;
     res = hilf;
  else
     res = b;
  return res;
}
```

- Def/uses have to be stated in the order in which they are executed, and not in the order in which they appear in the program code!

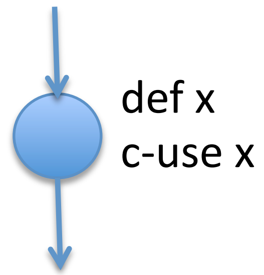


Data-flow based techniques (Definitions)

- Global access (c-use): No def x in the same block preceeds a c-use x



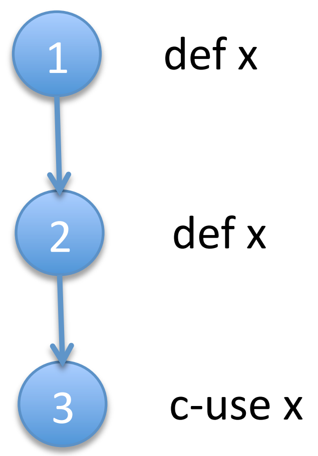
- Local access (c-use): A def x in the same block preceeds a c-use x.



- Definition free: Path p (n_in, ..., n_m) is definition free for x if no def x has been executed on the path.

Data-flow based techniques (Definitions)

- Definition free path def x -> c-use x: Path (ni)p(nj) is definition free, where ni=node in which def x took place, node nj where c-use took place



(n1)p(n3) not definition free, p=n1, n2, n3

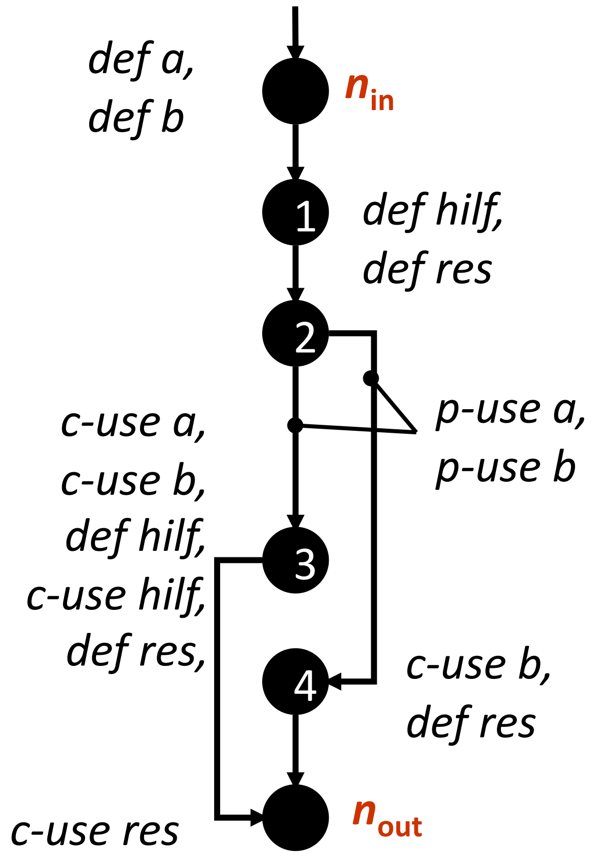
- Definition free path def x -> p-use x: Path (ni)p(nj,nk) is definition free where ni=node in which def x took place, edge (nj,nk) where p-use x took place

Data-flow based techniques (Definitions)

- Set Def(ni): Set of global variables defined in node ni
- Set C-USE(nj): Set of global variables accessed (c-use) in node ni
- Set P-USE(nj,nk): Set of variables with predicative access at node (ni,nj)
- Set DCU(x,ni): Set of all nodes ni with def x in ni reaching c-use x in nj and are element of C-USE(nj)
- Set DPU(x,ni): Set of all nodes (nj,nk) with def x in ni that reach p-use x in nj.

Data-flow based techniques (Application)

- Set Def(ni): Set of global variables defined in node ni
  - Def(n_in) = {a,b}
  - Def(n_1) = {hilf, res}
  - Def(n_3) = {res} (not hilf as it is not global)
  - Def(n_4) = {res}
- Set C-USE(nj): Set of global variables accessed (c-use) in node ni
  - C-USE(n_3) = {a, b}
  - C-USE(n_4) = {b} (not c-use hilf as it is not global access)
  - C-USE(n_out) = {res}
- Set P-USE(nj,nk): Set of variables with predicative access at node (ni,nj)
  - P-USE(n_2,n_4) = {a}
  - P-USE(n_2, n_3) = {b}
- Set DCU(x,ni): Set of all nodes ni with def x in ni reaching c-use x in nj and are element of C-USE(nj)
  - DCU(a,n_in)={n_3}
  - DCU(b,n_in)={n_3}
  - DCU(b,n_in)={n_4}
  - DCU(res,n_4)={n_out}
  - DCU(res_n3)={n_out}
- Set DPU(x,ni): Set of all nodes (nj,nk) with def x in ni that reach p-use x in nj.
  - DPU(a,n_in)={(n_2,n_3), (n_2,n4)}
  - DPU(b,n_in)={(n_2,n_3), (n_2,n4)} -> note you have to evaluate poth variables, no matter which path you choose



Test all-uses criterion

- All-uses:
  - Test cases have to contain all definition-free paths to a predicative (p-use) or access (c-use) for all defined variables in def(ni)
  - More formal:
    - For each node ni and each variable x in def(ni)
    - There has to be a definition free path for x
    - from ni to all elements in DCU(x,nj) and DPU(x,nj)
    - in the set of paths to be tested.

Note: there are weaker froms also, e.g. All-defs (For all defs go to at least one use on a definition free path), as well as All C-Uses, and All P-Uses

Test all-uses criterion

Test all-uses criterion (paths)

- **$n_{in}, n_1, n_2, n_3, n_{out}$**

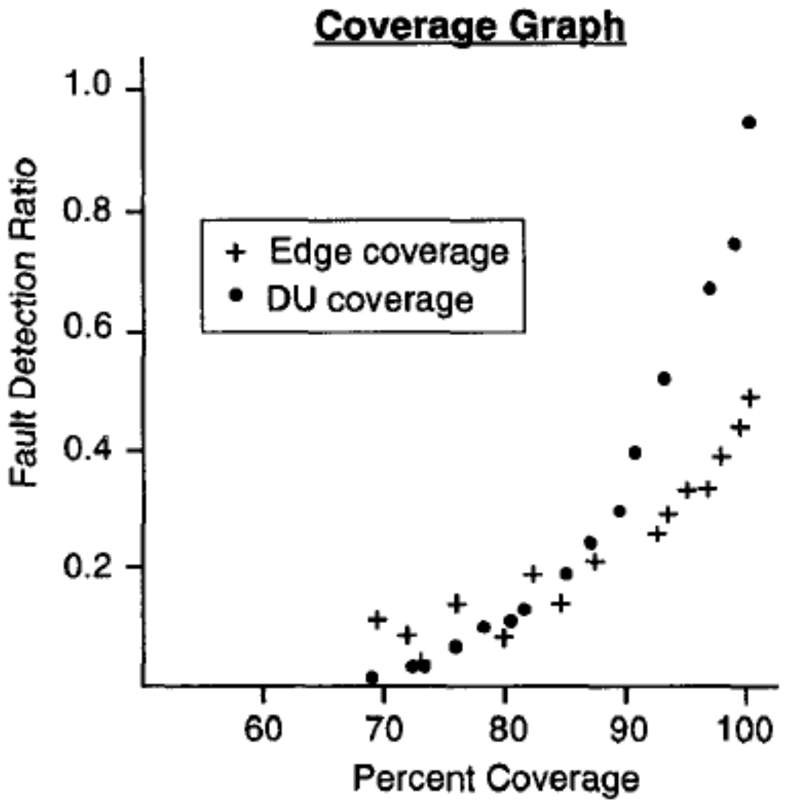
  DCU(a, n_3),
  DCU(b, n_3),
  DCU(res_{n3}, n_{out}),
  DPU(a,(n_2, n_3)),
  DPU(b,(n_2, n_3))

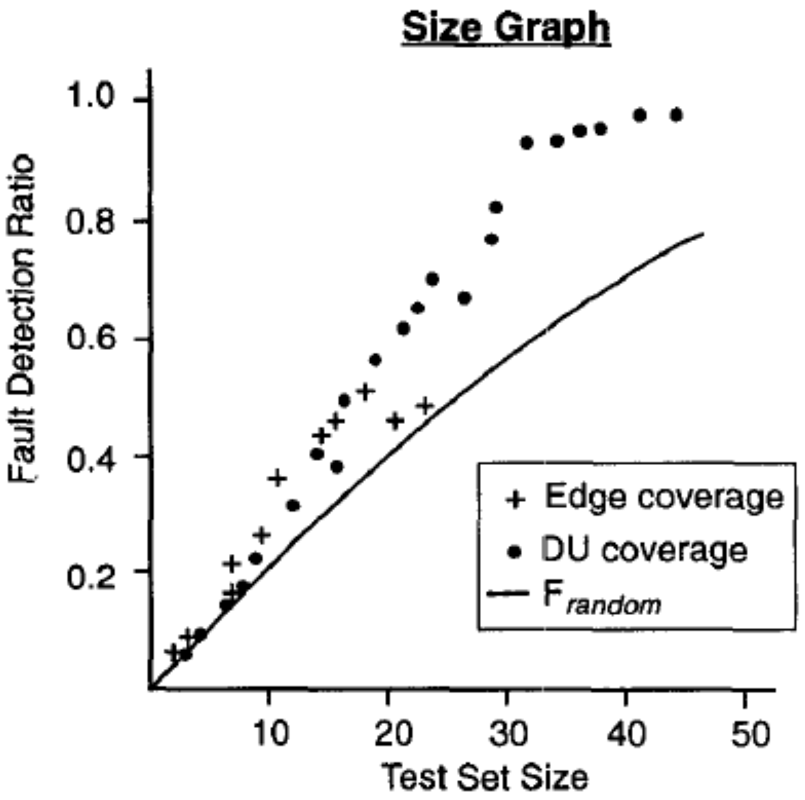
- **$n_{in}, n_1, n_2, n_4, n_{out}$**

  DCU (b, n_4),
  DCU(res_{n4}, n_{out}),
  DPU(a,(n_2, n_4)),
  DPU(b,(n_2, n_4))

Variable	Node $\boldsymbol{n_i}$	DCU(x, $\boldsymbol{n_i}$)	DPU(x, $\boldsymbol{n_i}$)
a	n_{in}	$\{n_3\}$	$\{(n_2, n_3), (n_2, n_4)\}$
b	n_{in}	$\{n_3, n_4\}$	$\{(n_2, n_3), (n_2, n_4)\}$
res	n_3 n_4	$\{n_{out}\}$ $\{n_{out}\}$	{}

From the def nodes (n_in, n_3, n_4) all uses have to be covered by the test paths, which in this case is achieved by two different phats





Hutchins et al., Experiments on the effectiveness of dataflow and control flow based test adequacy criteria, IEEE 1994

Problem Set 3:

Given is the following function:

```
1 boolean ALL_POSITIVE(int[] array) {
2       boolean result;
3       int i,len,tmp;
4       len = array.length;
5       i=0;
6       result=true;
7       while (i<len&&result) {
8               tmp=array[i];
9               if (tmp<=0)
10                      result=false;
11              i++;
12      }
13      return result;
14 }
```

1) For the given code:

a) Determine the test path needed to achieve statement coverage
b) Determine the test path needed to achieve decision coverage
c) Conduct a data-flow analysis and derive the test cases to satisfy the all-uses criterion

2) For the following expression:

A AND (B OR C)

a) Determine simple condition converge (CD)
b) Determine min. mult. condition coverage (also called condition/decision coverage) C/DC
c) Determine MC/DC
d) Determine Multiple decision coverage MDC

Multiple Decision Coverage

	C0	C1	Simp Cond. Cov.	Min. Mult Con.Cov	MC/DC	Mult. Con. Cov
Every statement in the program has been invoked at least once		X	X	X	X	X
Every decision in the program has taken all possible outcomes at least once	X	X		X	X	X
Every condition in a decision in the program has taken all possible outcomes at least once		X		X	X	X
Every condition in a decision has been shown to independently affect the decision's outcome			X	X	X	X
Every combination of condition outcomes with a decision has been invoked at least once					X	X