Lecture 2

White-box Testing and
Structural Coverage
(see Amman and Offut, Chapter 2)

White-box Testing
(aka. Glass-box or structural testing)

- An error may exist at one (or more) location(s)
  - Line numbers
  - Boolean tests
  - Expressions etc.
- *If tests don't exercise that (those) location(s) then error can never be observed*
- So identify and exercise locations
- No loops – finitely many locations – good!
- Loops – infinitely many locations – bad! cannot write infinite test suite
- Loops + branches – exponential growth in locations with loop depth – very bad !!!!

Structural Testing

Structural testing is the process of exercising software with test scenarios written from the source code, not from the requirements.

Usually structural testing has the goal to exercise a <u>minimum collection</u> of (combinations of) locations.

How many locations and combinations are enough?

Coverage

- *The size of a test suite is an unreliable indicator of the work achieved by testing.*
- Coverage refers to the extent to which a given testing activity has satisfied its objectives.
- *"Enough" testing is defined in terms of coverage rather than test suite size*.

glassbox testing/whitebox testing

- A major advantage of structural testing is that coverage can be easily and accurately defined.
- Structural coverage measures

Structural Testing – Problems!

good for unit testing but not good for scalability

- What about sins of omission? when you forgot to code stuffs
- Missing code = no path to go down! Unimplemented requirements!

when you have a location, but cannot access it

- What about dead code – is a path possible?
- How to avoid redundant testing?
- What about testing the user requirements?

functional statement cannot be tested by testing the line, since the testing is based on source code

- How to handle combinatorial explosion of locations and their combinations?

Problems with requirements-based testing

- A test set that meets requirements coverage is not necessarily a thorough test set
- Requirements may not contain a complete and accurate specification of all code behaviour
- Requirements may be too coarse to assure that all implemented behaviours are tested
- Requirements-based testing alone cannot confirm that code doesn't include unintended functionality. Need structural testing too!
- Three main types of glass box testing based on 3 types of coverage criteria.

Coverage Criteria 1: Control flow

- Measure the flow of control between statements and sequences of statements
- Examples: *node coverage, edge coverage*.
- Mainly measured in terms of statement invocations (line numbers).
- Exercise main flows of control.
- Oldest and most common method

Coverage Criteria 2: Logic

- Analyse the influence of all Boolean variables
- Examples: *predicate coverage, clause coverage, MCDC* (FAA DO178B)
- Exercise Boolean variables at control points.
- Modern method, and increasingly common

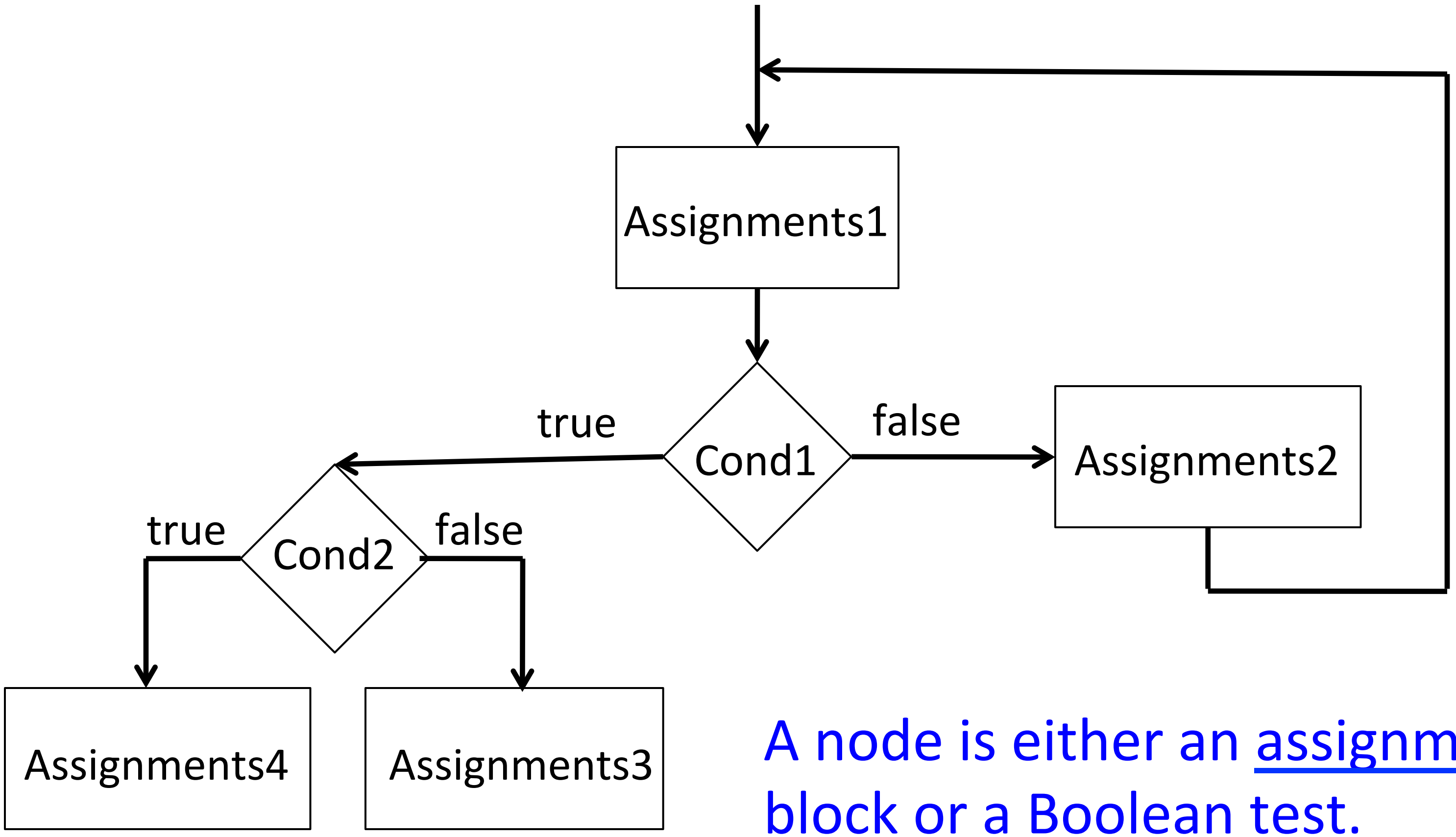
Coverage Criteria 3: Data Flow

- Data flow criteria measure the flow of data between variable assignments (*writes*) and variable references (*reads*).
- Examples: *all-definitions, all-uses*
- Exercise paths between definition of a variable and its subsequent use.
- Still rather rare in industry

Glass-box Test Requirements

- These are requirements on input values to satisfy a property: either
  - Graph-theoretic property (control & data flow)
  - data constraint (logic)
- Easy to define using graph theory
- Possible to automate generation by constraint solving
- Easy to measure coverage!
- Oracle is usually just crash (fail)/no crash(pass)
- Therefore they ignore functionality!

Starting Point: a Condensation Graph aka. Flowchart



A node is either an <u>assignment block</u> or a <u>Boolean test</u>.
Every program statement is associated with exactly one node.

Building condensation graphs

- Boolean tests can be from:
  - If-then-else statements if (bexp) then .. else ..
    boolean expression
  - If statements if (bexp) …
  - Loops (of any kind) while (bexp) …
- An assignment block contains consecutive
  - assignments x = exp
  - return statements return exp
  - procedure/method calls myFunc(…)
  - expressions e.g. i++

Type 1 and Type 3 Coverage

- A path is a sequence of nodes $n_0 ,\ldots, n_k$ in a (condensation) graph G, such that each adjacent node pair, (n_i, n_{i+1}) forms an edge in G.
- For type 1 and type 3 testing, a test requirement $tr(.)$ is a path

Covering a Graph Criterion C

Definition: Let TR be a set of test requirements demanded by a graph criterion C. A test suite TC satisfies C on graph G if, and only if for every test requirement $p = n_0, \ldots, n_k \in TR$, there is at least one test case t in TC such t takes path p.

- This definition implies 100% coverage.
- We can also have < 100%

Why so Formal?

Answers:

1. Sometimes coverage properties become very technical to define for reasons of accuracy.
2. Precise definitions can be automated to make test case generation tools.

Type 1 : Graph Coverage

2.1. **Node Coverage (NC)** Each reachable node in G is contained in some path $p \in TR$.

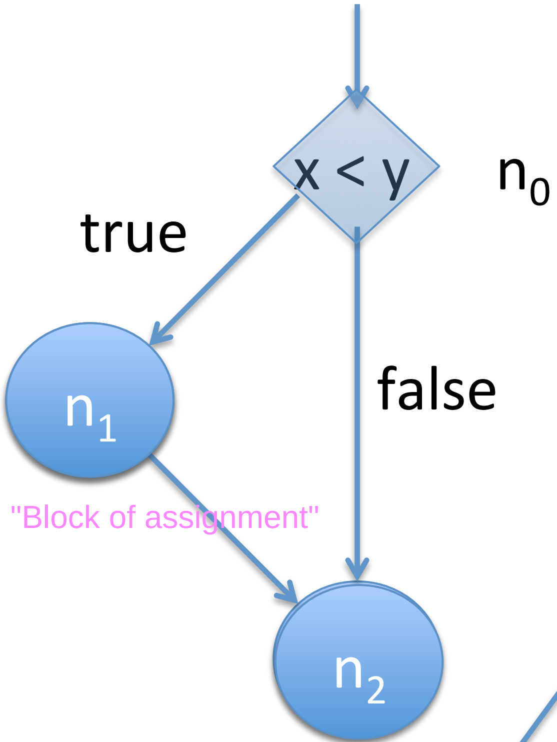
"The modern line coverage"

Myers : "NC is so weak that it is generally considered useless"

"Taking a particular branch. Hitting an edge with a certain path"

2.2. **Edge Coverage (EC)** Each reachable path of length ≤ 1 in G is contained in some path $p \in TR$.

Node vs. Edge Coverage



$p_1 = n_0, n_1, n_2$

$p_2 = n_0, n_2$

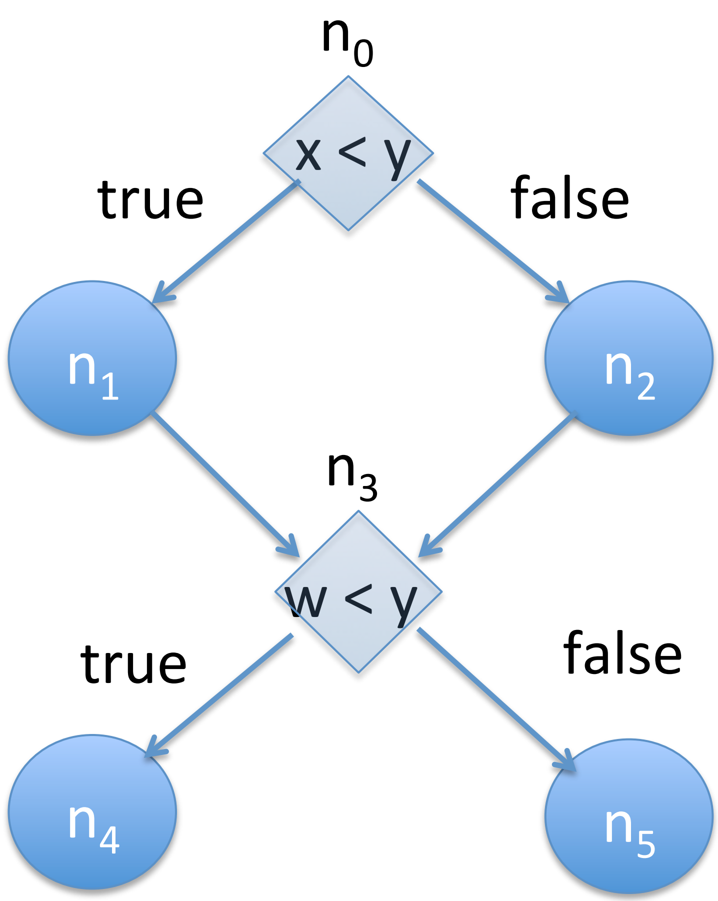
$TS_1 = \{p_1\}$ satisfies
node coverage

$TS_2 = \{p_1, p_2\}$ satisfies
edge coverage

Abstract test cases

2.3 **Edge-Pair Coverage** **(EPC = EC^2)** Each reachable path of length ≤ 2 in G is contained in some path $p \in TR$.

- Clearly we can continue this beyond 1,2 to **EC^n**
- Combinatorial explosion in TR size!
- **EC^n** doesn't deal with loops, which have unbounded length.



$p_1 = n_0, n_1, n_3, n_4$

$p_2 = n_0, n_2, n_3, n_5$

$p_3 = n_0, n_2, n_3, n_4$

$p_4 = n_0, n_1, n_3, n_5$

$TS_1 = \{p_1, p_2\}$ satisfies edge coverage

$TS_2 = \{p_1, p_2, p_3, p_4\}$ satisfies edge-pair coverage

Simple Paths

- How to deal with code loops?
- A path p is simple if it has no repetitions of nodes other than (possibly) the first and last node.
- *So a simple path p has no internal loops, but may itself be a loop*
- **Problem**: there are too many simple paths, since many are just sub-paths of longer simple paths.

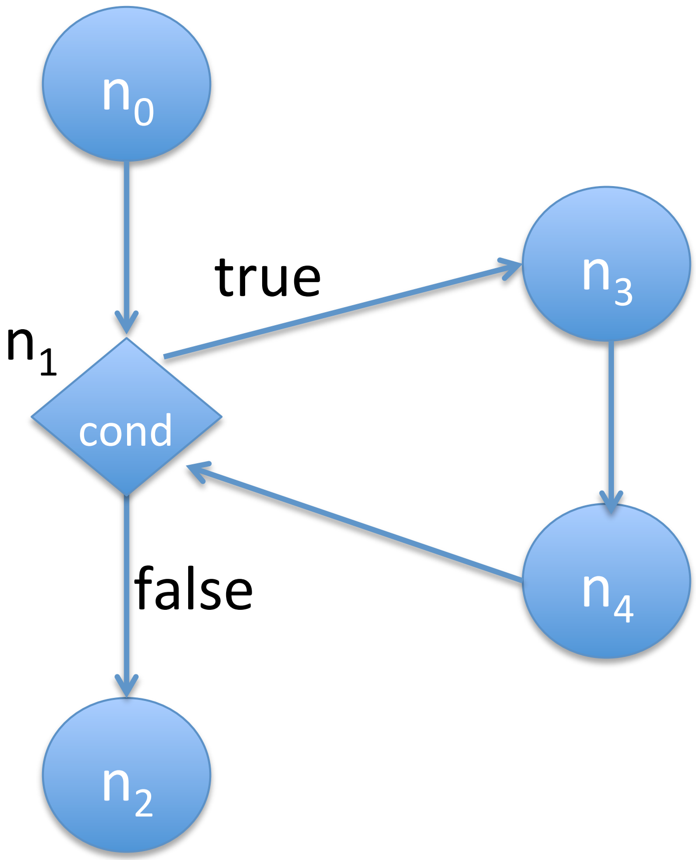
Prime Paths

if add anything more, will not be simple anymore

- A path p is prime iff p is a maximal simple path i.e. p cannot be extended without losing simplicity.
- This cuts down the number of cases to consider

2.4. **Prime Path Coverage (PPC)** Each reachable prime path in G is contained in some path

$p \in TR$.



Prime Paths =

Maximal simple paths =

(n_0, n_1, n_2),
(n_0,n_1,n_3,n_4),
(n_1,n_3,n_4,n_1),
(n_3,n_4,n_1,n_3),
(n_4,n_1,n_3,n_4),
(n_3,n_4,n_1,n_2)

$P_1 = (n_0, n_1, n_2)$

$P_2 = (n_0, n_1, n_3, n_4, n_1, n_2)$

$P_3 = (n_0, n_1, n_3, n_4, n_1, n_3, n_4, n_1, n_2)$

every prime paths can be found in either P1 or P3

$TS_1 = \{p_1, p_3\}$ satisfies **prime path coverage**

$TS_2 = \{p_1, p_2\}$ doesn't satisfy **prime path coverage! (why?)**

Computing Prime Paths

- One advantage is that the set of all prime paths can be computed by a simple dynamic programming algorithm
- See Amman and Offut Chpt. 2 for details
- Then test cases can be derived manually (heuristic: start from longest paths?) or automatically.

2.7. Complete Path Coverage (CPC) Every reachable path in G is contained in some path $p \in TR$.

Infeasible if G has infinitely many paths

2.8. Specified Path Coverage (SPC) Every reachable path in a set S of test paths is contained in some path $p \in TR$.
Here S is supplied as a parameter.

Example heuristic. S contains paths that traverse every loop free path p in G and every loop in G both 0 and 1 times.

Type 2 : Logic Coverage

- Graph and data flow coverage force execution of certain paths (branches) through code.
- They don't necessarily exercise different ways of taking the same branch.
- We can partition the number of ways to be finite and coverable.
- Since a Boolean condition may contain many Boolean and data (int, float, object) variables, we consider exercising the condition in different ways.

Clauses and Predicates

- A clause is a Boolean valued expression with no Boolean valued sub-expression (i.e. indivisible atomic)
- boolean variable binary predicate such as == , <=, >
  Examples: p, myGuard, x==y, x<=y, x>y
- A predicate is a Boolean combination of clauses using Boolean operators e.g. &, |, !, eager operator
- Let P be a set of predicates
- For $p \in P$, let C_p be the set of all clauses in p.

For each predicate in P

e.g P: {A|B|C, A&B&C, A|B&C}
p: A|B|C
Cp: {A,B, C}

eager operataor vs lazy operator
e.g. &, | vs e.g. &&,||

Type 2 Coverage

- For logic coverage, a test requirement tr is a logical constraint on input data values

Covering a Logic Criterion C

Definition: Let TR be a set of test requirements demanded by a logic criterion C. A test suite TC satisfies C if, and only if for every test requirement $\Phi \in$ TR, there is at least one test case t in TC such t satisfies Φ .

- Again this definition implies 100% coverage.
- We can also have < 100%

Logic Coverage Measures

- Look at some well known coverage models

so that can really dig in to the structure of the code

- Increasingly sophisticated and subtle,
- Powerful for exactly these reasons!
- Should produce better testing results?

Does better theory led to better practices?

- Since a test requirement is a constraint, it may not be solvable i.e. dead code

Predicate Coverage

- **3.12 Predicate Coverage (PC)** For each predicate $p \in P$, the set TR contains: (1) a requirement that implies p is reached and evaluates to true, and (2) a requirement that implies p is reached and evaluates to false.
- Example: p = a I b
- TR1 P = true, TC1 = (a = T, b = F)
- TR2 P = false, TC2 = (a = F, b = F)
- Notice here we never test for b = T

It is already 100% predicate coverage.

Clause Coverage

- **3.13 Clause Coverage (CC)** For each predicate $p \in P$, and each clause $c \in C_p$ the set TR contains: (1) a requirement that implies c is reached and evaluates to true, and (2) a requirement that implies c is reached and evaluates to false.
- Example: p = a l b p is a predicate, a and b makes 2 clauses
- TR1 a = true, TC1 = (a = T, b = F)
- TR2 a = false, TC2 = (a = F, b = T)
- TR3 b = true, TC3 = TC2
- TR4 b = false, TC4 = TC1 100% clause coverage
- Notice p is always true, so we satisfy CC but not PC
- So PC and CC are independent coverage criteria.

Distributive vs. non-distributive

- Note: these definitions of PC and CC are non-distributive, i.e. We don't take all combinations of all predicate or clause values. (Can be unsolvable combinations even without dead code!)
- (Non-distributive) PC – linear growth.
- (Non-distributive) PC implies EC, but not EC^n for $n >= 2$.
- Distributive PC - exponential growth.

Brute Force Approach to PC & CC

- **3.14 Combinatorial Coverage (CoC)** For each predicate $p \in P$, and every possible truth assignment α to the clauses C_p of p the set TR contains a requirement which implies p is reached and the clauses evaluate to α.

A | B | C | p1 | p2 | p3 ...| pn

F | F | F
F | F | T
F | T | F
F | T | T
T | F | F
T | F | T
T | T | F
T | T |T

- CoC implies both PC and CC.
- CoC is also called multiple condition coverage (MCC).
- Too strong? Too many test cases? Use less?

Active Clause Coverage

Example: p = a I b
TC1 = (a = T, b = T), TC2 = (a = F, b = F)
(TC1, TC2) satisfies both PC and CC

- Effect of a on its own and b on its own are never considered.
- Notice b = T masks the effect of a (and vice versa)
- But b = F completely exposes the effect of a (vice versa)
- We say that a determines p in this latter case
  not enough
- Can we find something *more expressive than PC or CC* but *less expensive than CoC* which handles this?
  too much
- Use notion of active clause which determines the overall predicate value

Determination

Definition: Given a clause c in a predicate p we say that c determines p under assignment α iff changing the value of c under α (and only this value) changes the truth value of p.

values chosen for other clauses must be the same

The idea is that in some contexts (assignments) c "has complete control" of p, and we should test this context.

Notice determination is a <u>local property</u> depending only on a truth table.

Below, when G1(C) = G1(D) = G1(E) = T then D determines P

When R1(C) = R1(D) = R1(E) = F then D again determines P.

C	D	E	P	
T	T	T	T	G1
F	T	T	T	
T	F	T	F	G2
F	F	T	T	
T	T	F	T	
F	T	F	F	R1
T	F	F	T	
F	F	F	T	R2

Refined Clause Coverage Models

3.43 Active Clause Coverage (ACC) For each predicate $p \in P$ and each clause $c \in C_p$ which determines p (under some α), the set TR contains two requirements for c: c is reached and evaluates to true, and c is reached and evaluates to false.

Example: For ACC of clause D above there are 4 possible pairs of test cases

(G1,G2), (R1,R2), (G1,R2), (G2,R1)

Ambiguity

- ACC can be seen as ambiguous.
- Do the other clauses get the same assignment when c is true and c is false, or can they have different assignments?
- We may not be able to isolate individual clauses
- Problems of masking, logical overlap and side-effects (e.g. variable synonyms) between clauses
- Consider e.g. p = (x>10) -> (x>0)
- If the first clause is set to true the second can never be false.

Different Values

- **3.15 General Active Clause Coverage (GACC)** For each predicate $p \in P$ and each clause $c \in C_p$ which determines p, the set TR contains two requirements for c: c is reached and evaluates to true, and c is reached and evaluates to false. The

referring to the truth value in C, D, E in slide 36

values chosen for the other clauses $d \in C_p$, $d \neq c$, need not be the same in both cases.

Example: For GACC of clause D above there are 4 possible pairs of test cases

(G1,G2), (R1,R2), (G1,R2), (G2,R1)

GACC Problem: Clause Correlation

- One problem is that GACC does not imply PC.

Consider the predicate $p = a \leftrightarrow b$

For some α, a determines p (so does b) so let:

TC1: $a = T$, $b = T$ so $p = true$

TC2: $a = F$, $b = F$ so $p = true$

For this test suite p never becomes false so PC is not satisfied although GACC is!

- Here the correlation between a and b is explicit in the condition, but it may be implicit in the code.

Combining GACC and PC

3.16 Correlated Active Clause Coverage (CACC)

For each predicate $p \in P$ and each clause $c \in C_p$ which determines p, the set TR contains two requirements for c: c is reached and evaluates to true, and c is reached and evaluates to false. The values chosen for the other clauses $d \in C_p$, $d \neq c$, must cause p to be true in one case and false in the other.

Example: For CACC of clause D above there are 2 possible test suites (G1,G2), (R1,R2).

Why not solve using Determination?

is about keeping the colour constant

3.15 Restricted Active Clause Coverage (RACC)
For each predicate $p \in P$ and each clause $c \in C_p$ which determines p, the set TR contains two requirements for c: c is reached and evaluates to true, and c is reached and evaluates to false. The values chosen for the other clauses $d \in C_p$, $d \neq c$, must be the same in both cases.

refer to slide 35

Example: For RACC of clause D above there are 2 possible test suites (G1,G2), (R1,R2).

Determination is really a global property!

Consider the code snippet

```
x := y;
...
if (x>0 or y>0 ) then …
```

For predicate p = (x>0 I y>0) it looks like PC using RACC is possible from a determinacy analysis of (a I b) where a = x>0 and b = y>0 .

But this is misleading, since here x and y are effectively synonyms and RACC is not satisfiable at all.

Amman and Offut give a different kind of example, based (essentially) on logical overlap of clauses.

Safety Critical Testing: MCDC

- In DO-178B, the Federal Aviation Authority (FAA) has mandated a minimum level of logic coverage for level A (highest safety) avionic software.

- "*Modified Condition Decision Coverage*" (MCDC)
- Has been some confusion about this definition
- "*Unique Cause MCDC*" (original definition) is RACC
- "*Masking MCDC*" (new definition) is CACC

Type 3: Data Flow Coverage

- A **definition** of a variable v is any statement that writes to v in memory
- A **use** of v is any statement that reads v from memory.
- A path $p = (n_1, .., n_k)$ from a node n_1 to a node n_k is **def-clear** for v if for each $1 < j < k$ node n_j has no statements which write to v

If at least one node nj has statement which write to v, it is not def-clear

Definition/Use Paths (du-paths)

no repetition apart from start and end

- A du-path w.r.t. v is a simple path

  $p = (n_1, .., n_k)$

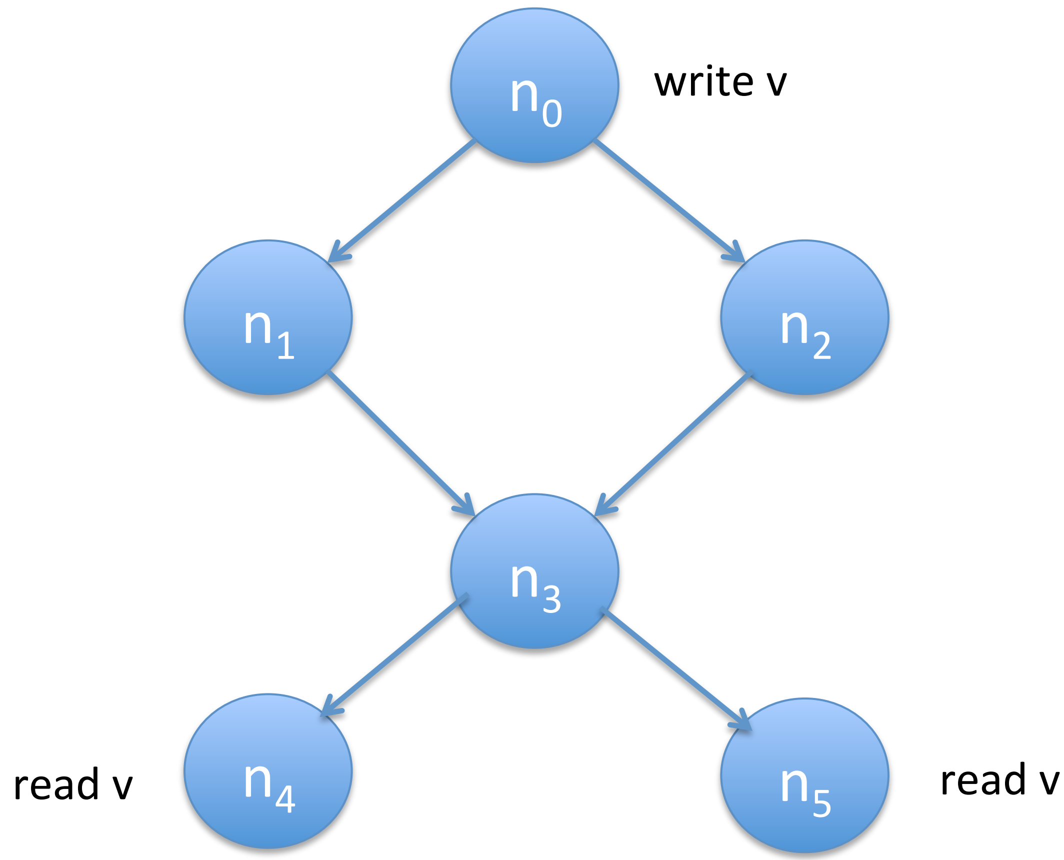
- Such that:
  1. A statement in n_1 writes to v
  2. Path p is def-clear for v
  3. A statement in n_k reads v

write to v

- $du(n,v)$ = set of all du-paths wrt v starting at n
- $du(m,n,v)$ = set of all du-paths wrt v starting at m and ending at n

Data flow coverage models

- **2.9. All-defs Coverage (ADC)** For each def-path set S = du(n, v) the set TR contains at least one path d in S.
- **2.10 All-uses Coverage (AUC)** For each def-pair set S = du(m,n,v) the set TR contains at least one path d in S. AUC contains ADC
- **2.11 All-du-paths Coverage (ADUPC)** For each def-pair set S = du(m,n,v) the set TR contains every path d in S.



All-defs = $\{(n_0, n_1, n_3, n_4)\}$

All-uses = $\{(n_0, n_1, n_3, n_4)$
$(n_0, n_1, n_3, n_5)\}$

All-du-paths = $\{(n_0, n_1, n_3, n_4)$
$(n_0, n_1, n_3, n_5),$
$(n_0, n_2, n_3, n_4),$
$(n_0, n_2, n_3, n_5)\ \}$