DD2459 Software Reliability
Lecture 2

Glass-box Testing and
Glass-box Coverage
(see Amman and Offut, Chapter 2,
also my online lecture notes)

Part 1 Control Flow Coverage

Glass-box Testing:
Basic Idea

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

Glass-box Testing:
Definition

Glass box or *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 minimum collection 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 gives a measure of the amount of testing work achieved (c.f. *energy* in physics).
- *"Enough" testing is defined in terms of coverage rather than size.*
- A major advantage of structural testing is that coverage can be easily and accurately defined.
- Structural coverage measures

Problems of Glass-Box Testing

- What about sins of omission?
- Missing code = no path to go down!
  Unimplemented requirements!
- What about dead code – is a path possible?
- How to avoid redundant testing?
- What about testing the user requirements?
- How to handle combinatorial explosion of locations and their combinations?

Problems of 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!

Coverage Model Type 1:
Control flow

- Model 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 Model Type 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 Model Type 3:
Data Flow

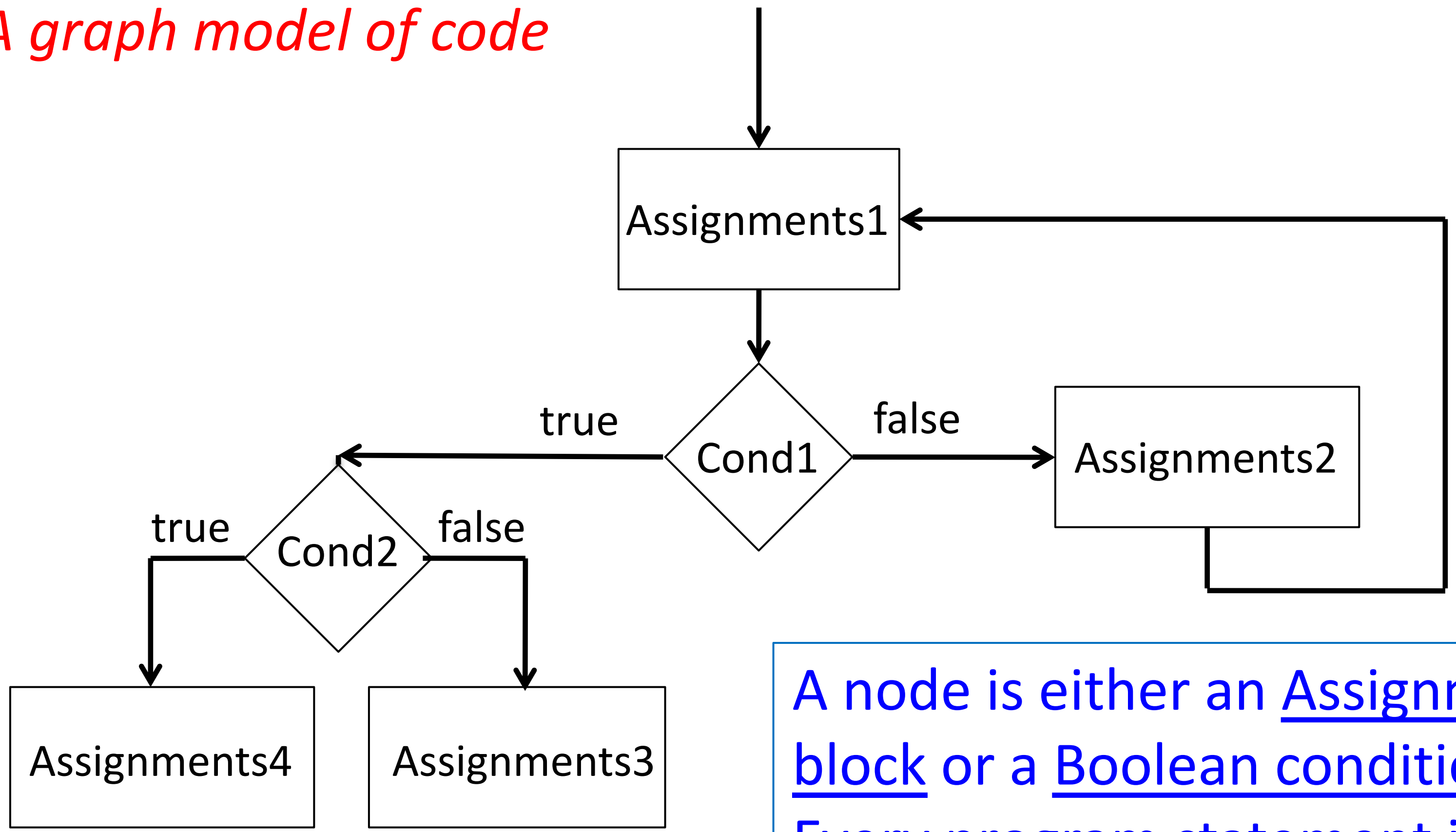
- Data flow criteria measure the flow of data between variable assignments (*writes*) and variable references (*reads*).
- Examples: *all-definitions, all-uses*
- Exercise paths between definition of a variable and its subsequent use.
- Still rather rare in industry

Glass-Box
Test Requirements

- Generated by a coverage model
- 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

A graph model of code



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

Graph Components

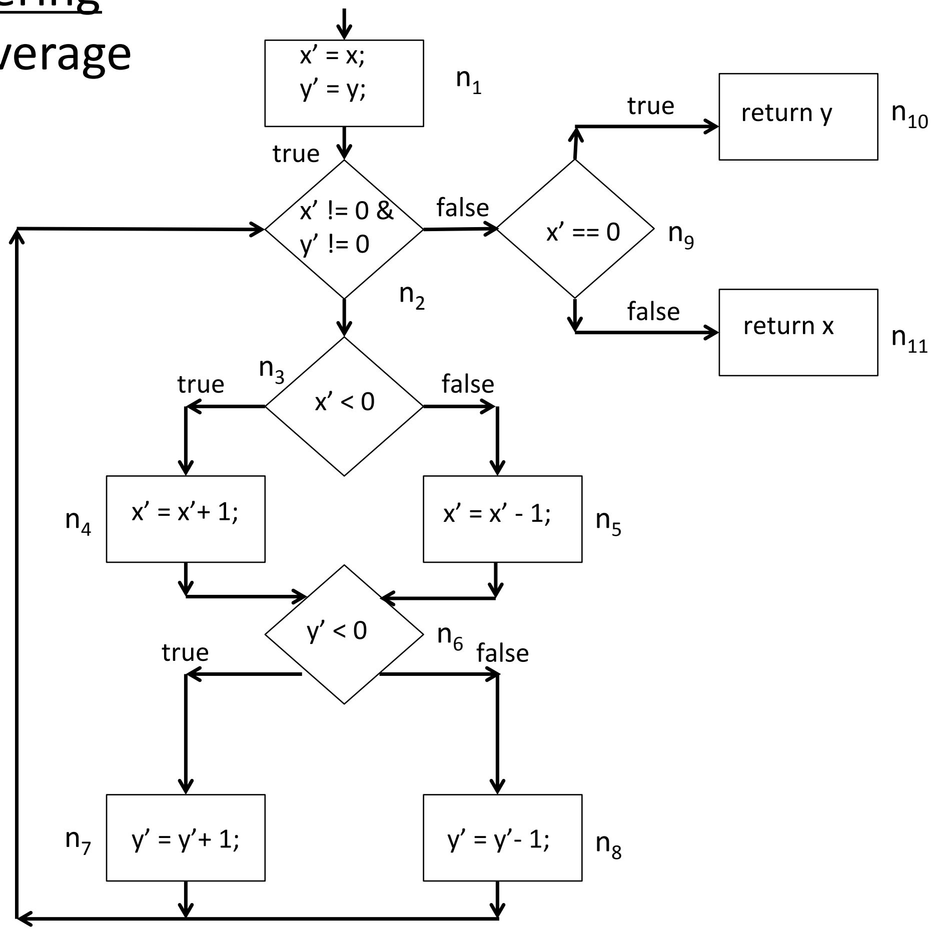
- A Boolean condition models:
  - If-then-else statements if (bexp) then .. else ..
  - If statements if (bexp) …
  - Loops (of any kind) while (bexp) …
- An Assignment block contains *consecutive*
  - assignments x = exp
  - return statements return exp
  - procedure/method calls myFunc(…)
  - expressions e.g. i++

Example Code

```
x′ = x;
y′ = y;
while (x′ != 0 & y′ != 0) do {
    if (x′ < 0) then x′ = x′ + 1 else x′ = x′ - 1;
    if (y′ < 0) then y′ = y′ + 1 else y′ = y′ - 1;
}
if (x′ == 0) then return y else return x;
```

Corresponding Graph

Notice the <u>node numbering</u> which is needed for coverage definition



Types 1 and 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

Satisfying a Coverage Model

Definition: Let TR be a set of test requirements demanded by a coverage model 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.

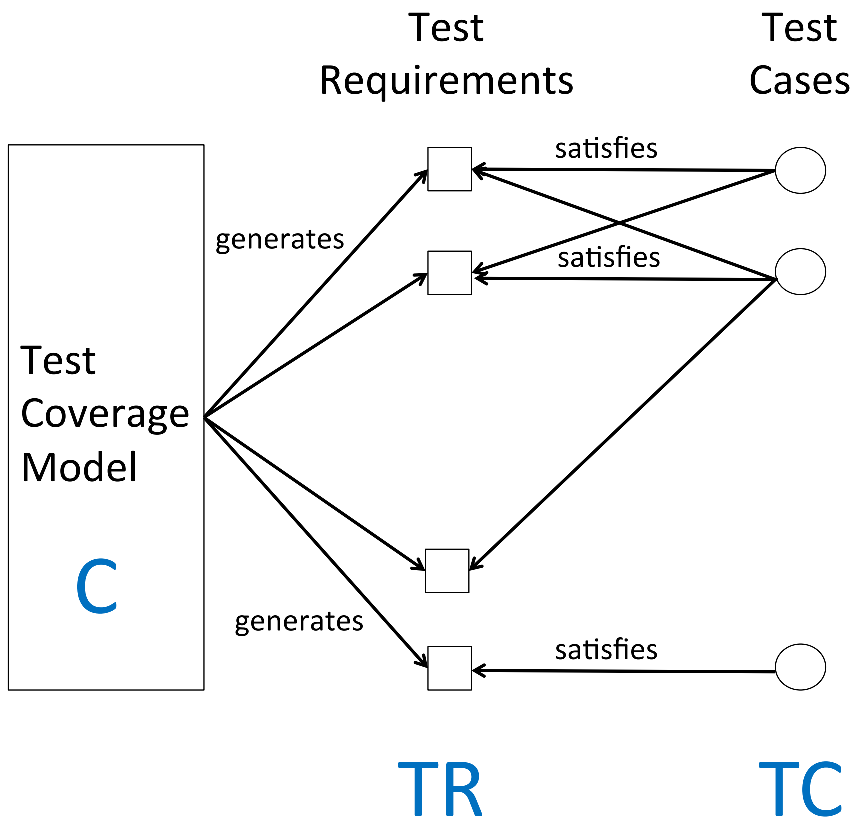
- Implicitly assumes 100% coverage is possible.
- Maybe only < 100% is achievable?

Why so Formal?

Answers:

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

Relationship between Coverage Model, Test Requirements and Test Cases



Type 1 Examples:

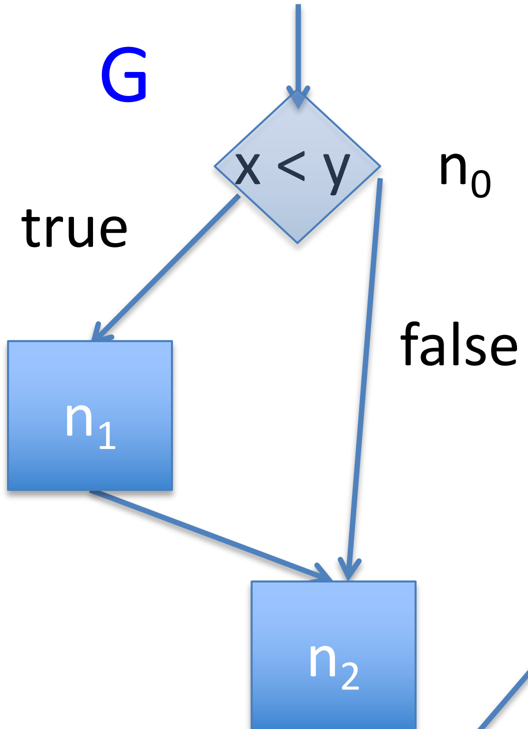
Control Flow Coverage

2.1. **Node Coverage (NC)** Each reachable path p of length 1 in G is a test requirement $p \in TR_{NC}(G)$.

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

2.2. **Edge Coverage (EC)** Each reachable path p of length 2 in G is a test requirement $p \in TR_{EC}(G)$

Node vs. Edge Coverage




```
if ( x < y ) then n1; n2
```

$p_1 = n_0, n_1, n_2$

$p_2 = n_0, n_2$

Test requirements

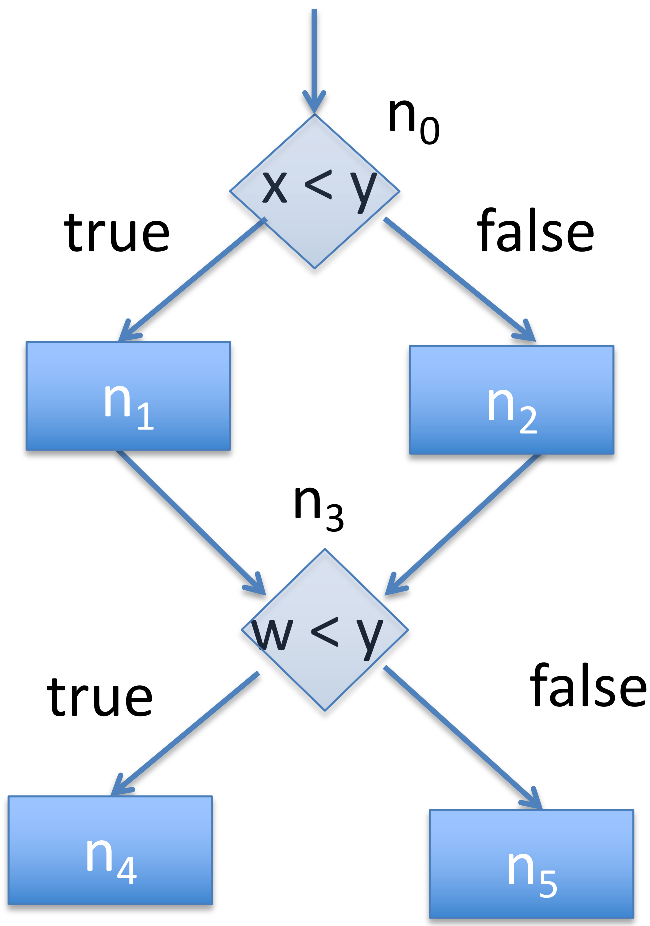
$TR_1 = \{p_1\}$ gives 100% node coverage for G

$TR_2 = \{p_1, p_2\}$ gives 100% edge coverage for G

2.3 **Edge-Pair Coverage** **(EPC = EC^2)** Each reachable path p of length ≤ 3 in G is a test requirement $p \in TR_{EC^2}(G)$.

- 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.

```
if ( x < y ) then n1 else n2;
if ( w < y ) then n4 else n5;
```



$p_1 = n_0, n_1, n_3, n_4$

$p_2 = n_0, n_2, n_3, n_5$

$p_3 = n_0, n_2, n_3, n_4$

$p_4 = n_0, n_1, n_3, n_5$

$TR_1 = \{p_1, p_2\}$ gives 100% edge coverage

$TR_2 = \{p_1, p_2, p_3, p_4\}$ gives 100% edge-pair coverage

Simple Paths

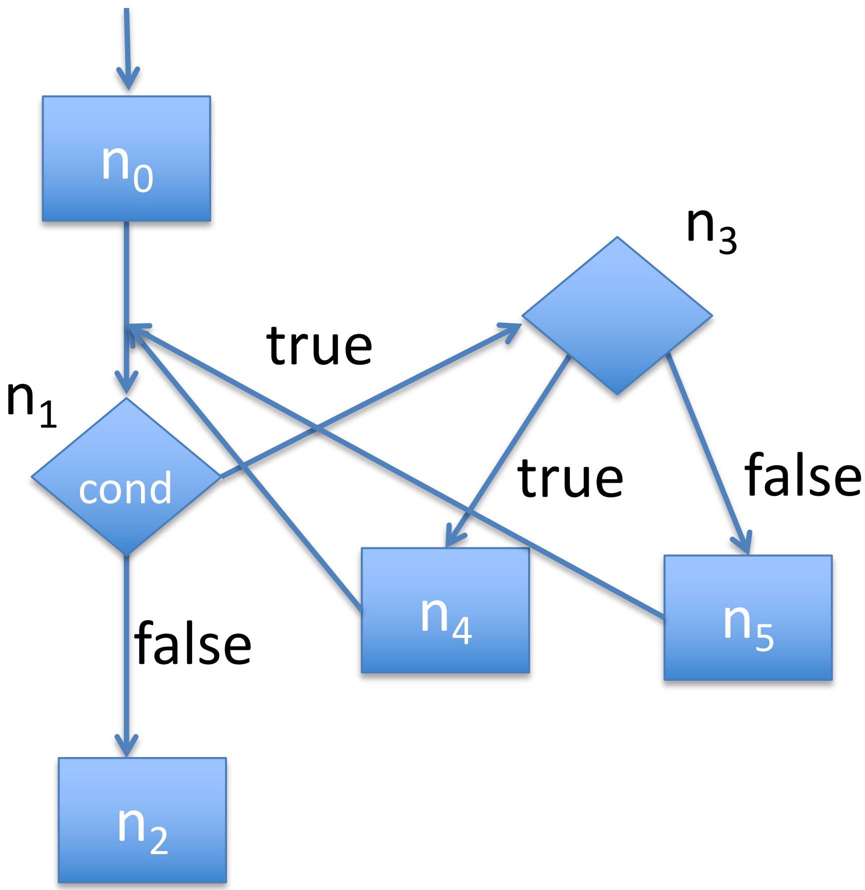
- How to deal with code loops?
- A path p is simple if it has no repetitions of nodes other than (possibly) the first and last node.
- *So a simple path p has no internal loops, but may itself be a loop*
- **Problem**: there are too many simple paths, since many are just sub-paths of longer simple paths.

Prime Paths

- 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 p in G is a test requirement:

$p \in TR_{PPC}(G)$



Prime Paths =
Maximal simple paths *<u>includes</u>*
(n_0, n_1, n_2),
(n_1,n_3,n_4,n_1),(n_1,n_3,n_5,n_1),
(n_3,n_4,n_1,n_3),(n_3,n_5,n_1,n_3),
(n_4,n_1,n_3,n_4), (n_4,n_1,n_3,n_5),
(n_5,n_1,n_3,n_4), (n_5,n_1,n_3,n_5),

Exercise: complete the set above with any <u>missing</u> maximal simple paths.

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 exactly 1 times.

DD2459 Software Reliability
Lecture 2

Glass-box Testing and
Glass-box Coverage
(see Amman and Offut, Chapter 2,
also my online lecture notes)

Part 2 Logic and Data Flow Coverage

Type 2 Examples:
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.
- For this, we consider exercising Boolean conditions in different ways.

Clauses and Predicates

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

Type 2 Coverage

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

Satisfying a Logic Coverage Model

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

- Again assumes 100% coverage.
- Maybe can only achieve < 100%?

Logic Coverage Models

- Look at some well known coverage models
- Increasingly sophisticated and subtle,
- Powerful for exactly these reasons!
- Should produce better testing results?
- 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

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 I b

Satisfy Coverage

- 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
- Notice p is <u>always</u> true, so we satisfy CC but <u>not</u> 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.
- See my online lecture notes for more detail

Brute Force Approach to Combining 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.

- 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 l 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
- Can we find something *more expressive than PC or CC* but *less expensive than CoC* which handles this?
- 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.

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 the truth table for p.

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

When R1(C) = 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

Improved 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: (1) c is reached and evaluates to true, and (2) 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 Assignments

- **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: (1) c is reached and evaluates to true, and (2) c is reached and evaluates to false. The 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 <u>not</u> satisfied although GACC is!

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

Same Assignments

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: (1) c is reached and evaluates to true, and (2) 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.

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

Problem: Determination is a global property!

Consider the code

```
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.

Combining GACC and PC
(Assignments are not the same, but not too different either)

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: (1) c is reached and evaluates to true, and (2) 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).

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

Definition/Use Paths (du-paths)

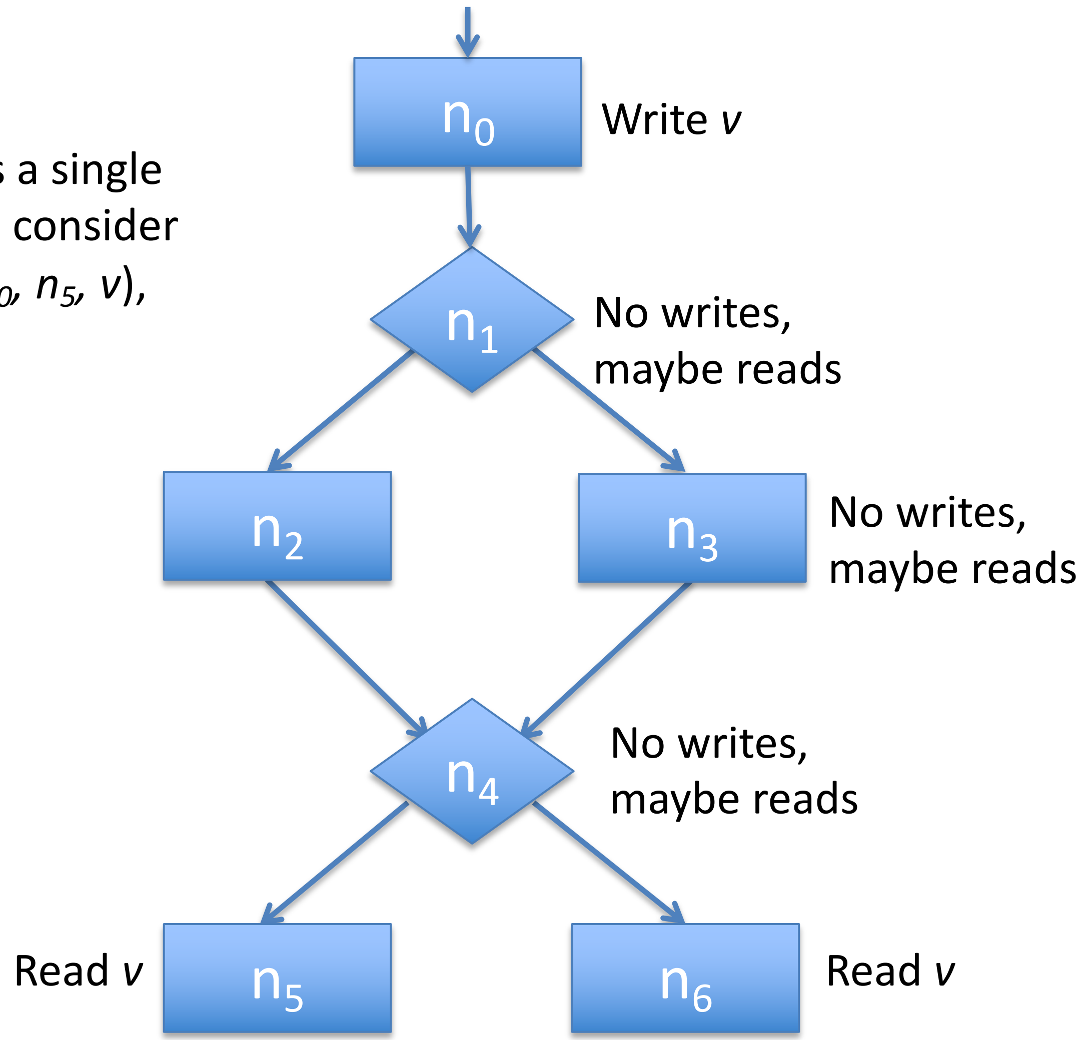
- A du-path w.r.t. v is a simple path

  $p = (n_1, .., n_k)$
- 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
- $du(n,v)$ = 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.
- **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.

Suppose G has a single variable v, and consider $du(n_0, v)$, $du(n_0, n_5, v)$, $du(n_0, n_6, v)$



All-defs = $\{(n_0, n_1, n_2, n_4, n_5)\}$

All-uses = $\{(n_0, n_1, n_2, n_4, n_5),$
$(n_0, n_1, n_2, n_4, n_6)\}$

All-du-paths = $\{(n_0, n_1, n_2, n_4, n_5),$
$(n_0, n_1, n_2, n_4, n_6)$
$(n_0, n_1, n_3, n_4, n_5)$
$(n_0, n_1, n_3, n_4, n_6)\}$

Lecture Summary

- We have looked at Glass-box testing
- Most commonly used approach in industry
- Compared glass and black-box testing
- Looked at 3 types of coverage
  - Control flow
  - Logic
  - Data flow
- For each type we have seen several models
- Compared model advantages and disadvantages
- Studied in more detail in Lab 1
- Learn which to choose in a given situation.