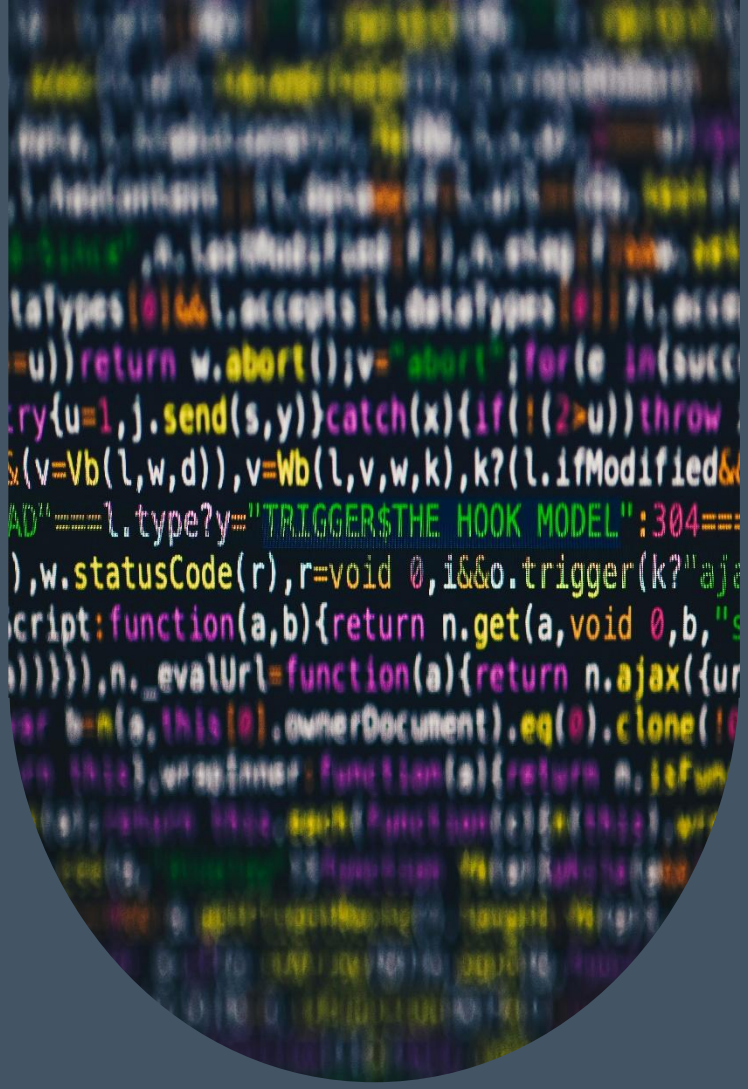
Games ENG II
Chapter 3,
Logic Coverage



Dr. Omer Ali
Adapted from
Introduction to Software Testing (Ammann & Offutt)

SETU
Ollscoil Teicneolaíochta an Oirdheiscirt
South East Technological University

setu.ie
INSPIRING FUTURES

Can Graph Coverage identify bugs?

```
#include <iostream>// Function to determine user access
bool hasAccess(bool isAuthenticated, bool isAdmin, bool isVip) {
        if (isAuthenticated && (isAdmin || isVip))
              {
                return true;    }
          return false;
}
```

We need only two test cases to get 100% branch coverage: one that enters the if block (**true** branch) and one that skips it (**false** branch).

Test Case 1 : True branch

isAuthenticated = true, isAdmin = true, isVip = false

Result = hasAccess (true) → Test Passed

Test Case 2 : False branch

isAuthenticated = false, isAdmin = true, isVip = false

Result = hasAccess (false) → Test Passed

Based on that we have 100% branch coverage.

Hidden Bug?

- What if a VIP user is not authenticated but should still get access?
- The business logic requires that **if a user is a VIP, they should have access regardless of their authentication status**.
- The developer made a mistake and included *isAuthenticated* in the condition, preventing this VIP access rule from working.

- Why graph coverage missed it?
- Our branch coverage tests didn't cover the specific logical combination that reveals this bug.
- We need to test the ***individual conditions*** within the if statement more thoroughly.

Testing with Logic Coverage?

- Forcing all logical combinations to identify flaws/bugs
- This is sometimes referred to as Multiple Condition Coverage (MCC)
- MCC requires testing all Boolean conditions within a scope (predicate, more to that later)

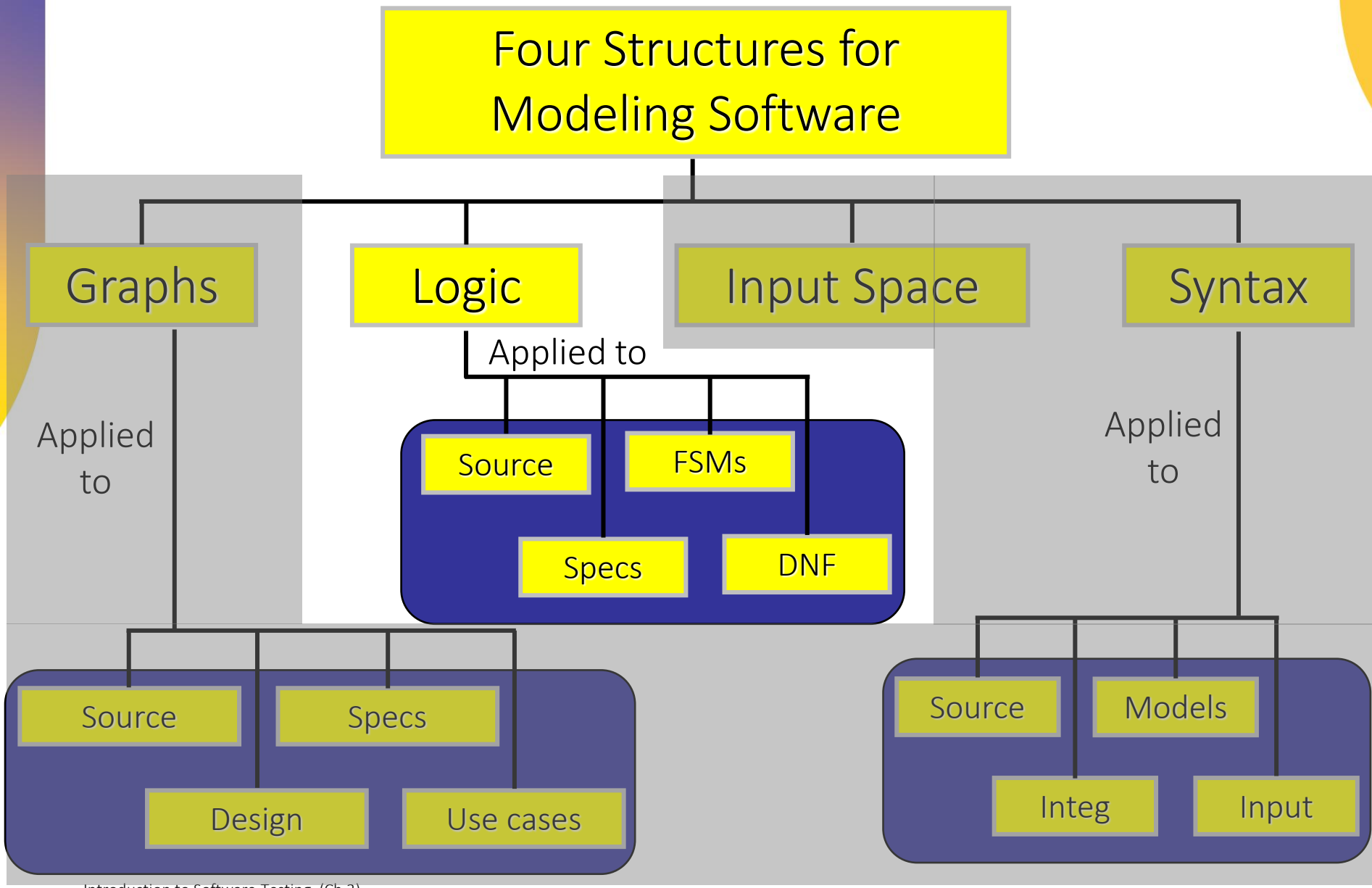
Test Case 3 : MCC Coverage

isAuthenticated = false, isAdmin = false, isVip = true

Result = hasAccess (false) → Test Passed

- **The Problem:** The test for a VIP user who is not authenticated ***failed***.
- **Expected:** true (if a user is a VIP, they should have access regardless of authentication status).
- **Actual:** false.
- **Conclusion:** The code has a bug! Logic coverage forced us to create a test case that exposed the defect.

Logic Coverage



Covering Logic Expressions (3.1)

- Logic expressions show up in many situations
- Covering logic expressions is required by the *US Federal Aviation Administration for safety critical software (FAA)*

- Logical expressions can come from many sources
  - Decisions in programs
  - FSMs and statecharts
  - Requirements

- Tests are intended to choose some subset of the total number of truth assignments to the expressions

Logic Predicates and Clauses

- A *predicate* is an expression that evaluates to a boolean value
- Predicates can contain
- boolean variables
- non-boolean variables that contain >, <, ==, >=, <=, !=
- boolean function calls
- Internal structure is created by logical operators

  ¬ : the *negation* operator

  ∧ : the *and* operator

  ∨ : the *or* operator

  → : the *implication* operator

  ⊕ : the *exclusive or* operator

  ↔ : the *equivalence* operator
- A *clause* is a predicate with no logical operators

  ((a **<** b) ∧ C) ∨ ((a **>=** b) ∧ p(x))

Review: Truth Table for Logical Operators

p	q	$p \wedge q$	$p \vee q$	$p \oplus q$	$p \rightarrow q$	$p \leftrightarrow q$
T	T	T	T	F	T	T
T	F	F	T	T	F	F
F	T	F	T	T	T	F
F	F	F	F	F	T	T

Review: the Laws of Logic

1. Commutative Laws
   - $p \wedge q \equiv q \wedge p$
   - $p \vee q \equiv q \vee p$
2. Associative Laws
   - $(p \wedge q) \wedge r \equiv p \ (q \wedge r)$
   - $(p \vee q) \vee r \equiv p \vee (q \vee r)$
3. Distributive Laws
   - $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
   - $p \vee (q \ r) \equiv (p \vee q) \ (p \vee r)$
4. Idempotent Laws
   - $p \wedge p \equiv p$
   - $p \vee p \equiv p$
5. Identity Laws
   - $p \wedge F \equiv F$
   - $p \vee F \equiv p$
   - $p \wedge T \equiv p$
   - $p \vee T \equiv T$
6. Involution Law
   - $\neg(\neg p) \equiv p$
7. De Morgan's Laws
   - $\neg(p \vee q) \equiv (\neg p) \wedge (\neg q)$ (sometimes written p NOR q)
   - $\neg(p \wedge q) \equiv (\neg p) \vee (\neg q)$ (sometimes written p NAND q)
8. Complement Laws
   - $p \wedge \neg p \equiv F$
   - $p \vee \neg p \equiv T$
   - $\neg T \equiv F$
   - $\neg F \equiv T$

Examples

- **P = (a < b) ∨ f (z) ∧ D ∧ (m >= n*o)**
- P is a predicate
- P has four clauses:
  1. (a < b) – relational expression
  2. f (z) – boolean-valued function
  3. D – boolean variable
  4. (m >= n*o) – relational expression
- Most predicates have few clauses
- It would be nice to quantify that claim !!!
- Sources of predicates
- Decisions in programs
- Guards in finite state machines
- Decisions in UML activity graphs
- Requirements, both formal and informal
- SQL queries (*i.e where conditions*)

From a study of 63 open-source programs, with > 400,000 predicates:

- 88.5% have 1 clause
- 9.5% have 2 clauses
- 1.35% have 3 clauses
- Only 0.65% have 4 or more!

Translating from English

- "I am interested in Games ENG II and OOP I"
- *course = Games2 OR OOP1 = ?*

Humans have trouble translating from English to Logic

- "If you leave before 6:30 AM, take Braddock to 495, if you leave after 7:00 AM, take Prosperity to 50, then 50 to 495"
- $time < 6{:}30 \rightarrow path = Braddock \vee time > 7{:}00 \rightarrow path = Prosperity$
- Hmm … this is incomplete !
- $time < 6{:}30 \rightarrow path = Braddock \vee time >= 6{:}30 \rightarrow path = Prosperity$

Source Code

```
if ((gear == 5 || gear == 6) && speed >= 100)
    print("highway")
else
    print("some other road");
```

- P : (g = 5 ∨ g = 6) ∧ s **>=** 100

Testing and Covering Predicates

We use predicates in testing as follows :

- Developing a model of the software as one or more predicates
- Requiring tests to satisfy some combination of clauses

Abbreviations:

- P is the set of predicates
- p is a single predicate in P
- C is the set of clauses in P
- C_p is the set of clauses in predicate p
- c is a single clause in C

Predicate and Clause Coverage

The first (*and simplest*) two criteria require that each predicate and each clause be evaluated to both true and false

Predicate Coverage (PC) : For each *p* in *P*, *TR* contains two requirements: *p* evaluates to true, and *p* evaluates to false.

- When predicates come from conditions on edges, this is equivalent to edge coverage
- PC does not evaluate all the clauses, so …

Clause Coverage (CC) : For each *c* in *C*, *TR* contains two requirements: *c* evaluates to true, and *c* evaluates to false.

Predicate Coverage Example

$$((a < b) \vee D) \wedge (m >= n*o)$$

predicate coverage

Predicate = true

a = 5, b = 10, D = true, m = 1, n = 1, o = 1

= (5 < 10) ∨ true ∧ (1 >= 1*1)

= true ∨ true ∧ TRUE

= true

Predicate = false

a = 10, b = 5, D = false, m = 1, n = 1, o = 1

=(10< 5) ∨ false ∧ (1>=1*1)

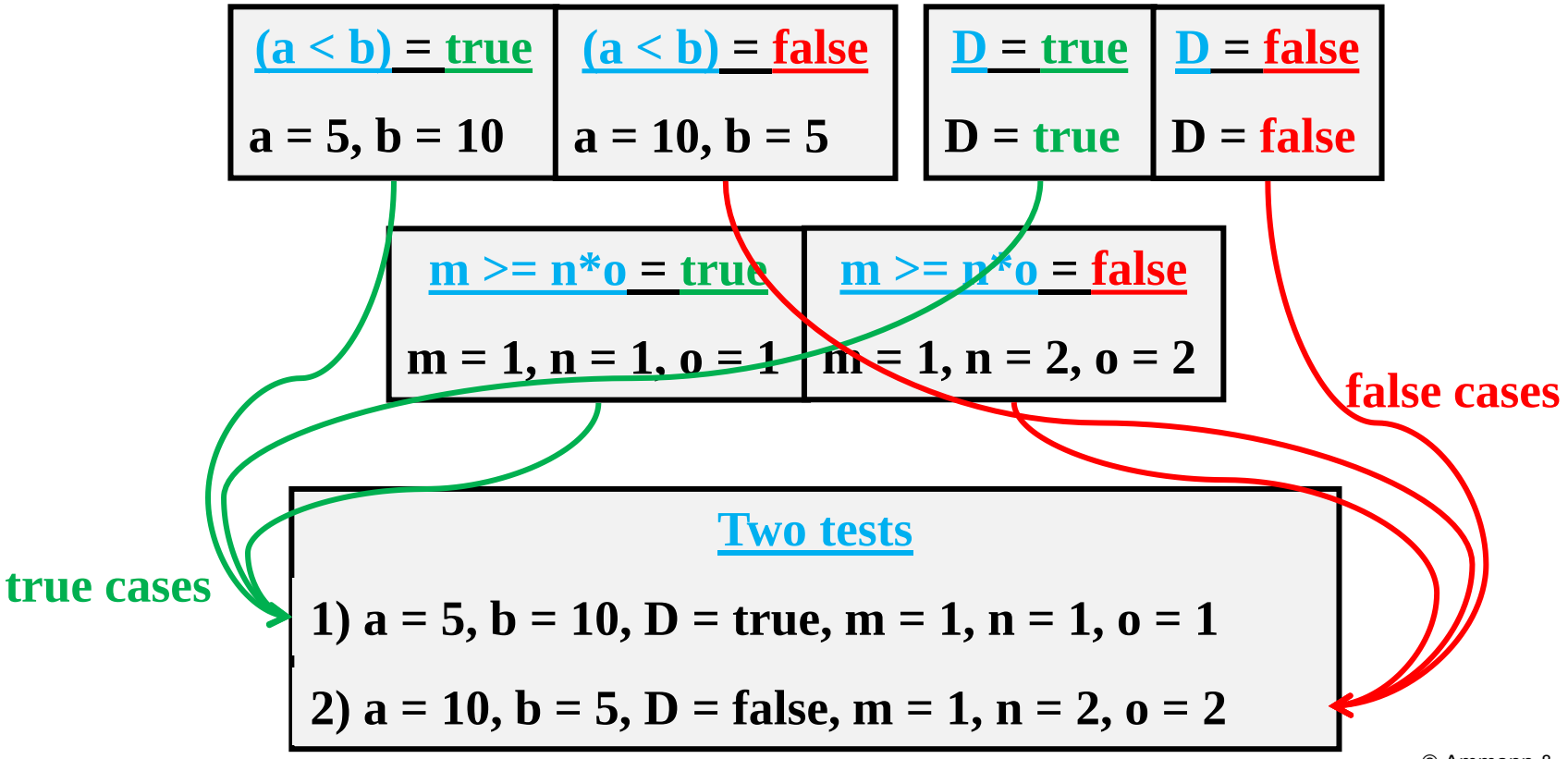
= false ∨ false ∧ TRUE

= false

Clause Coverage Example

$$((a < b) \vee D) \wedge (m >= n*o)$$

Clause coverage



Problems with PC and CC

	a	b	p = a ∨ b
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

Examples:

T_{23} : Satisfies CC but not PC, *because p is never false*

T_{24} : Satisfies PC but not CC, *because b is never true*

T_{14} : Satisfies both CC and PC coverage

So, neither PC nor CC subsumes the other in all cases.

- **PC does not fully exercise all the clauses, especially in the presence of *short circuit evaluation***
- **CC does not always ensure PC**
- That is, we can **satisfy** CC without causing the predicate to be both true and false
- This is definitely not what we want !
- **The simplest solution is to test all combinations …**

Combinatorial Coverage

- CoC requires every possible combination
- Sometimes called *Multiple Condition Coverage*

Combinatorial Coverage (CoC) : For each *p* in *P*, TR has test requirements for the clauses in C_p to evaluate to each possible combination of truth values.

	a < b	D	m >= n*o	P = ((a < b) ∨ D) ∧ (m >= n*o)
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Combinatorial Coverage

This is simple, neat, clean, and comprehensive …

- But quite expensive!
- *Requires* 2^N tests, where N is the number of clauses
  - Impractical for predicates with *more than 3 or 4 clauses*
- The literature has lots of suggestions
- The general idea is simple:

Test each clause independently from the other clauses

- Getting the details right is hard
- What exactly does "independently" mean ?
- The book presents this idea as "*making clauses active*" …

Active Clauses

- Clause Coverage has a weakness :
  - The values do not always make a difference
- Consider the first test for clause coverage, which caused each clause to be true: $(5 < 10) \vee true \wedge (1 >= 1*1)$
- Only the last clause counts !
- To really test the results of a clause, the clause should be the determining factor in the value of the predicate

Determination :	**A clause C_i in predicate p, called the major clause that determines p *iff* the values of the remaining minor clauses C_j are such that changing C_i changes the value of p**

- This is considered to *make the clause active*

Active Clauses

```
int HonorRollStatus (double CGPA, double termGPA,
int creditsCompleted, Boolean fullTimeStatus)
{
   if ((creditsCompleted > 30) && (CGPA >3.4) &&
(fullTimeStatus == true) && (termGPA >2.5)){
   cout<<"Made it to the Dean's list"<<endl;}
   else if ((creditsCompleted > 30) && (CGPA >3.7)
&& (fullTimeStatus == true) && (termGPA > 2.5)){
   cout<<"The student is on High honor list"<<endl;}
   else if ((creditsCompleted > 30) && (CGPA > 2.5)
&& (fullTimeStatus == true) && (termGPA > 2.5)){
    cout<<"You're on honors list");}
    else{
     cout<<"You're on a 'hit list' somewhere"<<endl;
  }
}
```

CC	CGPA	FT	TGPA	Honor's List?
29	33	T	2.4	F
31	33	T	2.4	T
31	30	T	2.4	F
31	33	T	2.4	T
31	33	F	2.4	F
31	33	T	2.4	T
31	33	T	1.9	F
31	33	T	2.4	T

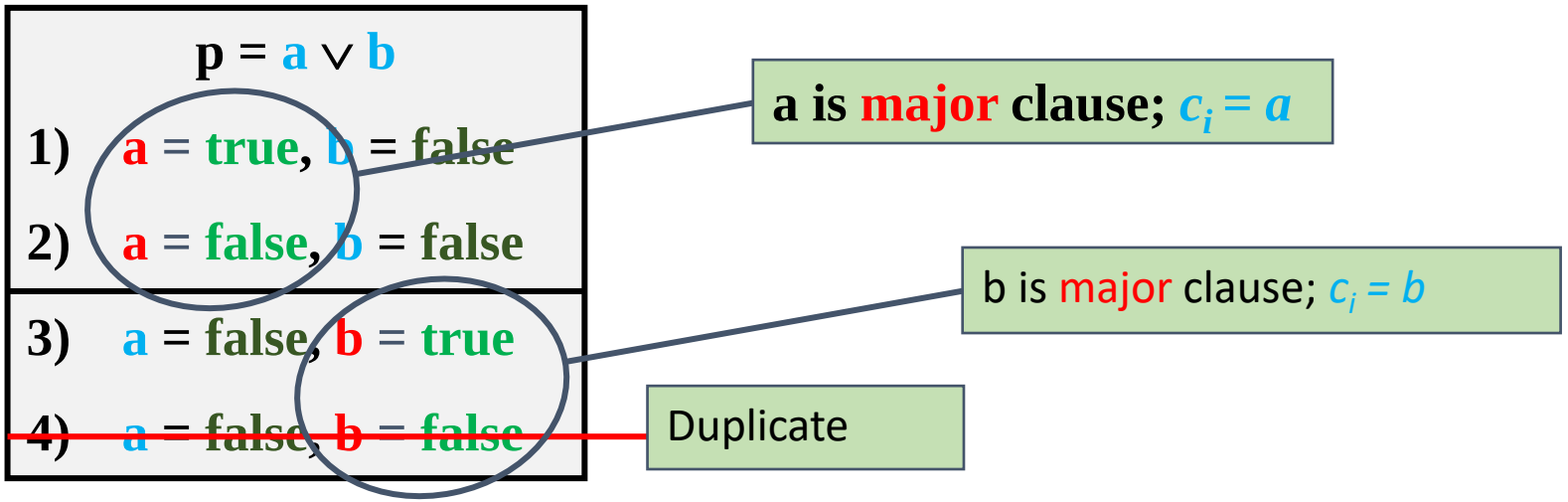
Where,
CC = creditsCompleted, FT = fullTimeStatus
TGPA = termGPA

Determining Predicates

- Determination, the conditions under which a clause influences the outcome of a predicate
- If you flip the clause, and the predicate changes value, then the clause determines the predicate.
- The **major** clause, c_i , is the clause on which we are focusing.
- All of the other clauses c_j , j != i, are **minor** clauses.
- **Typically**, to satisfy a given criterion, each clause is treated in turn (*One at a time, One by one*) as a **major** clause.
- **From the testing perspective**, we would like to test each clause under circumstances where the clause determines the predicate

Active Clause Coverage

Active Clause Coverage (ACC) : For each p in P and each major clause c_i in C_p, choose minor clauses c_j, j != i, so that c_i determines p. TR has two requirements for each c_i : c_i evaluates to true and c_i evaluates to false.



- This is a form of MCDC (*Modified Condition/Decision Coverage*), which is required by the FAA for safety critical software
- **Ambiguity** : Do the minor clauses have to have the same values when the major clause is true and false?

Resolving the Ambiguity

$p = a \vee (b \wedge c)$

Major clause : a (or $c_i = a$)

a = true, b = false, c = true

a = false, b = false, c = false

Is this allowed ?

- Is the minor clauses c_j need to have the same values when the major clause c_i is true as when c_i is false ???
- This question caused confusion among testers for years
- Considering this carefully leads to three separate criteria :
  1. Minor clauses do not need to be the same (GACC)
  2. Minor clauses do need to be (must be) the same (RACC)
  3. Minor clauses force the predicate to become both true and false (CACC)

General Active Clause Coverage

General Active Clause Coverage (GACC) : For each p in P and each major clause c_i in C_p, choose minor clauses c_j, j != i, so that c_i determines p. TR has two requirements for each c_i : c_i evaluates to true and c_i evaluates to false.

Condition: The values chosen for the minor clauses c_j do not need to be the same when c_i is true as when c_i is false, that is,
$c_j(c_i = true) = c_j(c_i = false)$ for all c_j OR
$c_j(c_i = true)$!= $c_j(c_i = false)$ for all c_j

- This is complicated !
- It is possible to satisfy GACC without satisfying predicate coverage (PC) . i.e. p = a ↔ b
- We really want to cause predicates to be both true and false !