Black Box Testing Techniques - part 4 (Combinatorial testing, Use Case testing, K-maps, Logical expressions)

Dr. John H Robb, PMP, IEEE SEMC
UTA Computer Science and Engineering

Combinatorial Testing

Consider This Example

- As a simple example a car insurance quotation (Lloyd Roden - stickyminds.com):
  - Three policy types (third party; third party, fire, theft; fully comprehensive)
  - Three storage modes (garage, driveway, road)
  - Four no-claims-discount types (NCD) (0 years, 1 year, 2 years, and 3+ years)
  - Two license types (provisional, full)
  - Five age categories (17-21, 22-30, 31-40, 41-50, 50+)
  - Five engine sizes (<1,001 cc, 1,001 cc:1,600 cc, 1,601 cc:2,000 cc, 2,001 cc:2,999 cc, 3,000 cc+)
- To test all combinations we would need: 3 x 3 x 4 x 2 x 5 x 5 = 1,800 test cases
- Pair-wise testing is a type of combinatorial testing - One of the reasons that pairwise testing can find a large number of defects is that they are:
  - Single Modal - in which something either works or it fails all by itself
  - Dual Modal - in which even though two things work by themselves, they fail when paired (connected) together
  - Multi Modal - in which three or more things in combination don't work together - it takes all three to cause an error

Combinational Error Example

- One possible example (can be many types):

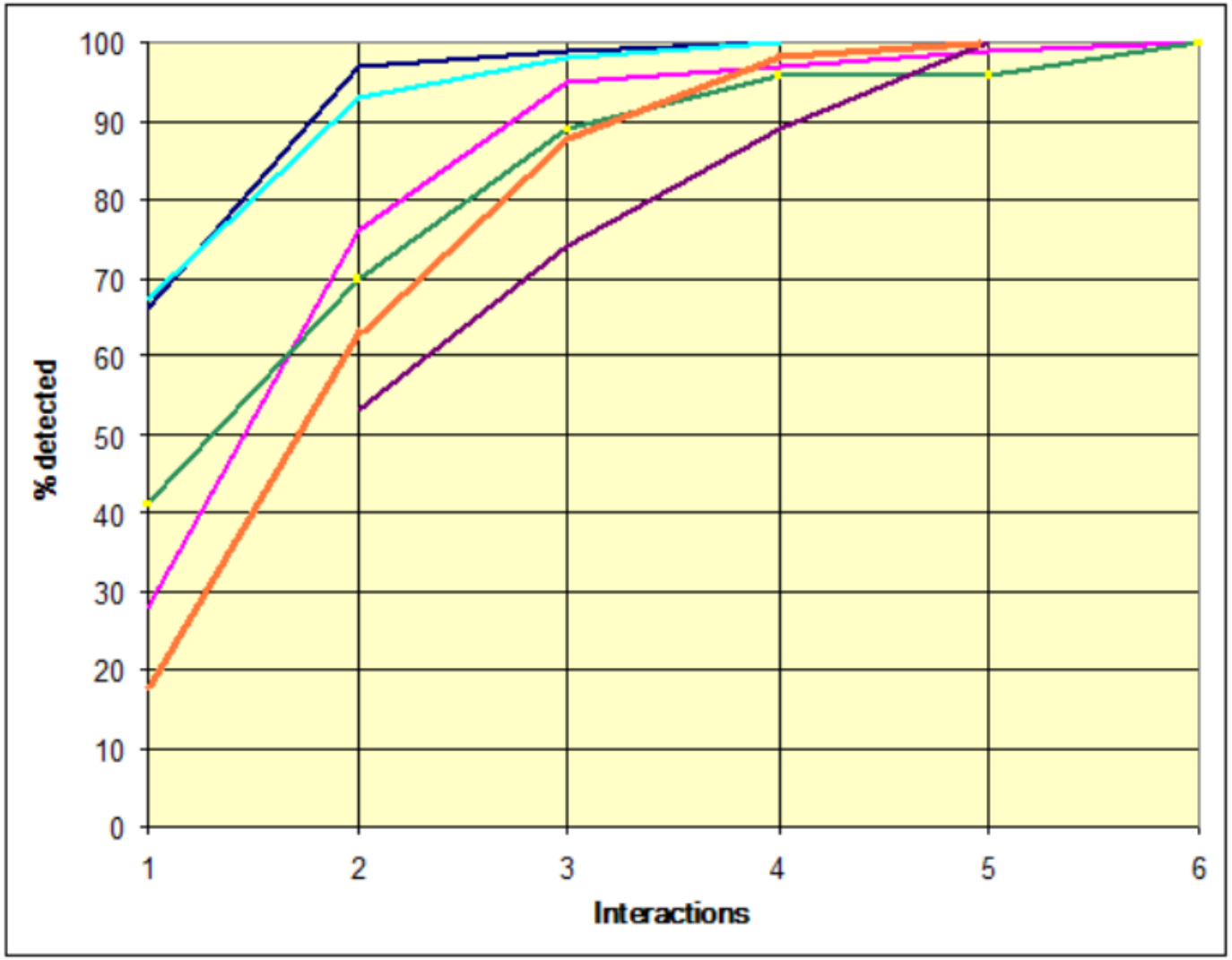
```
if (pressure < 10) {
      single modal error here or in if statement above
      if (volume > 300)
            dual modal error here or in if statement above
      else
            if (temp > 50)
                  multi modal error here or in if statement above

      }
      else {
      // do something else
      {
```

- Not limited to just code - can be configurations, actions - all kinds of combination types

Why Does Pairwise Testing Work?

- Identify a minimum set of tests that will find all multi-mode defects (particularly serious defects)
  - All single-mode defect will be found if every option is tested at least once (unit elements)
  - 2-way or "Pair-wise" is combinations of 2 items (parameters) that cause a defect
  - 3-way or "Tri-wise" is combinations of 3 items (parameters) that cause a defect
    - " Pair-wise" defect detection (pair-wise testing) finds most defects, ex. testing all pairs typically finds **75%** of defects Source: Kuhn
    - NASA Deep Space Mission - study showed that **88%** of bugs discovered using "pair-wise" defect detection testing
    - U.S. Food and Drug Administration – study showed **98%** bugs discovered using " pair-wise " defect detection Source: 27 th NASA/ IEEE Software Engineering Workshop, NASA Goddard Space Flight Center, 4-6 Dec, 2002.
    - " The Combinatorial Design Approach to Automatic Test Generation" Source: IEEE, Software 1996, Vol. 13, No. 5
      - » Baseline application under study resulted in a reduction in test plan time from 1 month to less than 1 week
      - » In several experiments, "pair-wise" demonstrated good code coverage and defect detection ability

Why Does Pairwise Testing Work (cont.)?



Kuhn, NIST 2010

Consider This Example

- Let's say that we have a Website that is hosted on a number of serves and operating systems and is viewed by a number of browsers with various plug ins (example from Systematic Software Testing, ISBN 1580535089)
  - Web Browser (Firefox 38.0.5, IE 11, Opera 30.0)
  - Plug-in (None, RealPlayer, MediaPlayer)
  - Application Server (IIS, Apache, Firefox Enterprise)
  - OS (Win10, Win 8, Linux)
- How many distinct combinations could be tested? 3x3x3x3 = 81
- What if we wanted to test one of each?
  - We have 4x3 unique values
  - We could reduce this down to the following test combinations

Browser	Plug-in	Server	OS
Firefox 38.0.5	None	IIS	Win10
IE 11	RealPlayer	Apache	Win 8
Opera 30.0	MediaPlayer	Firefox Enterp	Linux

  - But we would have very little isolation capability to determine what interactions might have caused errors

Consider This Example

- What could we do to increase the ability to isolate errors across interactions?
- We could test
  - Each browser with each plug-in, with each server, and with each OS
  - Each plug-in with each browser, with each server, and with each OS
  - Each server with each browser, each plug-in and each OS
  - Each OS with each browser, each plug-in and each server
- This would let us isolate down to problems associated with a single browser or problems with a browser interaction with a OS.
- It would cost us 3x3 tests do look at this interaction if we design the tests in a certain way

Test Case	Browser	Plug-in	Server	OS
1	Firefox 38.0.5	None	IIS	Win10
2	Firefox 38.0.5	RealPlayer	Apache	Win 8
3	Firefox 38.0.5	MediaPlayer	Firefox Ent	Linux
4	IE 11	None	Apache	Linux
5	IE 11	RealPlayer	Firefox Ent	Win10
6	IE 11	MediaPlayer	IIS	Win 8
7	Opera 30.0	None	Firefox Ent	Win 8
8	Opera 30.0	RealPlayer	IIS	Linux
9	Opera 30.0	MediaPlayer	Apache	Win10

How Do We Select the Test Cases?

- For the Pair-wise example one approach is to look at orthogonal arrays
- We will use a $L_9(3^4)$ orthogonal array

	1	**2**	**3**	**4**
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

- The 9 means it has 3x3=9 rows, with 4 columns and each cell contains a value between 1 and 3 (inclusive)
- In many practical applications, there is more than one factor, each taking on a different set of values. Mixed orthogonal arrays are useful in designing test configurations for such applications.

More On Orthogonal Arrays

- $L_x(n^y)$
  - Where x = number of rows (# of test cases)
  - y = number of columns (# of variables)
  - n = maximum number choices within each category/variable (level)
- Orthogonal array terms
- An orthogonal array is a rectangular matrix where rows represent **test cases** (runs) and columns represent **variables** (factors) being tested.
- The different states or values that each variable can assume are called **levels**.
- There are only certain orthogonal arrays - so those problems that don't have an exact fit require finding the nearest best fit (sometimes by filling in extra parameters with the best fit)
  - $L_4 2^3$, $L_8 2^7$, $L_9 3^4$, $L_{16} 4^5$, $L_{16} 2^{15}$, $L_{18} 3^8$, etc.

		Number of Parameters (P)																													
		2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Number of Levels	2	L4	L4	L8	L8	L8	L8	L12	L12	L12	L12	L16	L16	L16	L16	L32	L32	L32	L32	L32	L32	L32	L32	L32	L32	L32	L32	L32	L32	L32	L32
	3	L9	L9	L9	L18	L18	L18	L18	L27	L27	L27	L27	L27	L36	L36	L36	L36	L36	L36	L36	L36	L36	L36								
	4	L'16	L'16	L'16	L'16	L'32	L'32	L'32	L'32	L'32																					
	5	L25	L25	L25	L25	L25	L50	L50	L50	L50	L50	L50																			

All-Pairs Algorithm

- An alternative method of testing all-pairs combinations is by using an “all-pairs algorithm” or utility. These do not use orthogonal arrays to generate pairs using other algorithms.
- They do not generate the balanced exposures that orthogonal arrays do, but they still generate pairs of interactions
  - For software test, we don't care about balanced interactions, we just care about having an interaction between the pair
- Some try to get these down to a minimum set (optimum solution).
- We can approximate how many tests it will take by selecting the two largest levels and multiplying them together

Browser	Plug-Ins	OS	Server	Server OS
IE5	None	WME	IIS	NT
IE6	RealPlayer	W2000	Apache	XP
IE7	MediaPlayer	NT	WebLogic	Linux
Netscape 6.0		XP		
Netscape 6.1		XP-Pro		
Netscape 7.0		Vista		
Mozilla 1.1				
Opera 7				

Example from "A Practitioner's Guide to Software Test Design - Lee Copeland

Number of test cases for all-pairs => 8 x 6 = 48

All-Pairs vs. Orthogonal Arrays

- There is a difference - for the previous problem what we need is a $8^1 6^1 3^3$ array- but one does not exist so we pick the closest fitting Orthogonal array would be a $L_{64} 8^2 4^3$
  - The requirement of $8^1 6^1$ is met by 8^2 (2 columns of 1 through 8) and the requirement of 3^3 is met by 4^3 (3 columns of 1 through 4)
  - Means that we have some duplicated levels for the 6-level OS selection and for the three 3-level (Plug-ins, Server, Server-OS)
  - The Orthogonal array solution is 64 test cases, All-pairs was 48 test cases
  - All combinations are 1,296 test cases, so either one is a significant reduction in tests with 60-90 percent defect detection
- Other combinations and numbers of tests (from the Previous slide)
  - 1-way -> 8 tests, 2-way -> 48 tests
  - 3-way -> 159 tests, 4-way -> 520 tests
  - 5-way -> 1,296 tests

All-Pairs vs. Orthogonal Arrays (cont.)

- For the first problem (Insurance problem) $5^2 4 3^2 2$- there are 6 variables with a maximum level of 5
  - The Orthogonal array solution uses a $L_{25}5^6$ array - 25 test cases
  - All-pairs would require 28 test cases (requires a tool because the number of variables and levels)
  - Total number of combinations would require 1,800 test cases
- In the first case (previous slide) All-pairs produces the minimum set, in the second above Orthogonal arrays produces the minimum set
- Other combinations and numbers of tests (from the Insurance problem)
  - 1-way -> 5 tests
  - 2-way -> 25 tests
  - 3-way -> 103 tests
  - 4-way -> 341 tests
  - 5-way -> 900 tests
  - 6-way -> 1,800 tests

Pair-Wise Testing in Summary

- Pair-wise testing is a subset of combinatorial testing, obviously of single and dual modal failures
- Orthogonal arrays or All-pairs are used to design the pair interactions
- Orthogonal arrays provide a completely balanced exposure of pair wise interactions
- All-pairs produces a complete set of pair-wise interactions, but does not go the extra step of ensuring balance - for software test we don't care about balanced exposures - we just want at least one pair exposure
- Either approach (OATs or All-pairs) significantly reduces the problem space
- The primary strength in pair-wise testing is to provide a good coverage quick check with a smaller set of tests prior to performing more comprehensive tests with the complete set

Multi-Modal Combinations

- For deterministic applications the balance requirement can be **relaxed - covering arrays**, or **mixed level covering arrays** can be used for combinatorial designs.
- Covering arrays do not meet the balance requirement that is met by orthogonal arrays. This difference leads to combinatorial approaches that are often smaller in size.
- This smaller size allows for testing of multi-modal interactions with a low-cost approach. This can be valuable in areas where safety or security is paramount.
- Combinatorial testing tools develop the test combinations automatically and in a small amount of time. Model checker's can be used to develop automated expected outputs once the model has been specified.
- This is an area of significant research.

Combinatorial Testing Resources

- For problems with more than about 5 variables or with problems with levels of 5-6 or higher a tools is the mot useful to id the test cases needed
- Combinatorial testing tools can be found here:
  - http://www.pairwise.org/tools.asp
- Dr Jeff Lei (UTA) is a recognized expert in the field of combinatorial testing
  - This site includes Dr Lei's tool ACTS

- Dr Lei's book is:
  - **Introduction to Combinatorial Testing,** D. Richard Kuhn, Raghu N. Kacker, Yu (Jeff) Lei, ISBN 1466552298

Exercise

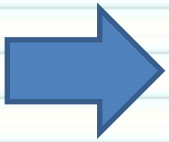
- Identify the pair-wise tests for the following combination

Comm Class Parameters		
Received	**Connection**	**Method**
Ack	1	Wireless1
Nack	2	Wireless2
Time out	3	Wireless3

Answer

– Identify the pair-wise tests for the following combination

Comm Class Parameters		
Received	**Connection**	**Method**
Ack	1	Wireless1
Nack	2	Wireless2
Time out	3	Wireless3



Comm Class Parameters		
Received	**Connection**	**Method**
Ack	1	Wireless3
Ack	2	Wireless2
Ack	3	Wireless1
Nack	1	Wireless2
Nack	2	Wireless1
Nack	3	Wireless3
Time out	1	Wireless1
Time out	2	Wireless3
Time out	3	Wireless2

Pair-wise testing gives us 3x3=9 two-wise pairs to test

For sets of levels (above) take the product of the two largest numbers of levels - for complex sets of levels you will need a tool

Use Case Testing

Use Case Based Testing

Agile Approach

Step 1. Prioritize the use cases in the requirement-use case traceability matrix.

Step 2. Generate test scenarios from the expanded use cases.

Step 3. Identify test cases.

Step 4. Generate test data (values).

Step 5. Generate test scripts.

- Why do we prioritize use cases?
  - We may not have needed resources or time.
  - Prioritizing ensures that high-priority use cases are tested.
- The requirement-use case traceability matrix
  - rows are requirements and columns are use cases
  - an entry is checked if the use case realizes the requirement
  - the bottom row sums up the priority of each use case.

Requirement-Use Case Traceability Matrix

	Priority Weight	UC1	UC2	UC3	UC4	UC5	UC6
R1	3	X	X				
R2	2					X	
R3	2	X					
R4	1		X	X			
R5	1				X		X
R6	1		X			X	
Score		5	5	1	1	3	1

Generating Use Case Scenarios

- A scenario is an instance of a use case.
- A scenario is a concrete example showing how a user would use the system.
- A use case has a primary scenario (the normal case) and a number of secondary scenarios (the rare or exceptional cases).
- This step should generate a sufficient set of scenarios for testing the use case.

Example: Register for Company Events

ID: UC5

Name: Register for Event

Actor: Event Participant

Precondition:

Participant has an account on the system and participant is not already registered for the event.

Primary Scenario: (see next slide)

Register for Event: Primary Scenario

Actor: Participant	System: Web App.
1. TUCBW the participant clicks the "Register" button on homepage.	2. Systems ask the participant to enter email and password.
3. Participant enter email and password.	4. System checks that login is correct and displays list of events.
5. Participant selects an event.	6. System asks for event related info.
7. Participant enters event related info.	8. System verifies info and displays a confirmation.
9. TUCEW Participant acknowledges by clicking the OK button (confirmation dialog)	

Identifying Secondary Scenarios/Exceptions

Base test cases on user input (we see some partitions here):

User Input	Normal Case	Abnormal Cases
email	valid	invalid
password	valid	invalid
selected event	still open	• registration closed • duplicate registration
event related info	valid	invalid

Other exceptions:

- **user quit before completing registration**

Identifying Test Cases

- Identify test cases from the scenarios using a test case matrix:
  - the rows list the test cases identified
  - the columns list the scenarios, the user input variables and system state variables, and the expected outcome for each scenario
  - We're going to apply techniques such as Boundary Value Analysis, Equivalence classes, decision tables as before to identify test cases

- This is where a technique like pair-wise testing or OATS can be very beneficial

Use Case Based Test Case Generation

		Inputs					
Test Case ID	Scenario	Email ID	Password	Registered	Event Info	Event Open	Expected Result
TC1	Successful Registration	V	V	V	V	V	Display confirma-tion
TC2	User Not Found	I	NA	NA	NA	NA	Error msg
TC3	Invalid Info	V	V	NA	I	NA	Error msg
TC4	User Quits	V	V	NA	NA	NA	Back to login
TC5	Registration Closed	V	V	NA	NA	I	Error msg
TC6	Duplicate Registration	V	V	I	NA	NA	Error msg

Identifying Test Data Values

Test Case ID	Scenario	Email ID	Pass-Word	Registered	Event Info	Event Open	Expected Result
TC1	Successful Registration	lee@ca.com	Lee123	No	Yes	Yes	Display confirma-tion
TC2	User Not Found	unknow@ca.com	NA	NA	NA	NA	Error msg
TC3	Invalid Info	lee@ca.com	Lee123	NA	I	NA	Error msg
TC4	User Quits	lee@ca.com	Lee123	NA	NA	NA	Back to login
TC5	System Unavailable	lee@ca.com	Lee123	NA	NA	NA	Error msg
TC6	Registration Closed	lee@ca.com	Lee123	NA	NA	No	Error msg
TC7	Duplicate Registration	lee@ca.com	Lee123	Yes	NA	NA	Error msg

Use Case Testing Disclaimer

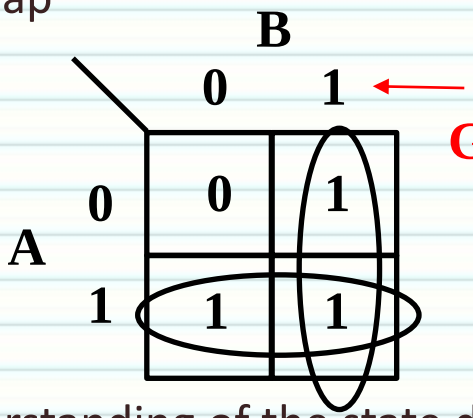
- Frankly, I think use case testing is pretty lightweight - this is because it is working devoid of requirements and at a higher level than code.

- I cover this because some companies use this approach - but I don't think it's a good set of tests - it's like a cloud floating between the warm sun of requirements and the firm ground of code

- How can we make this a better test?
  - Develop tests to the requirements - go back to the requirements/use case traceability matrix and identify requirements relevant to the user case under test
  - Develop test cases from those requirements - the total set of test cases are then use cases + requirements tests
  - This provides a much stronger tie between the UC and its requirements - it also gives more concrete things to test

Karnaugh Maps

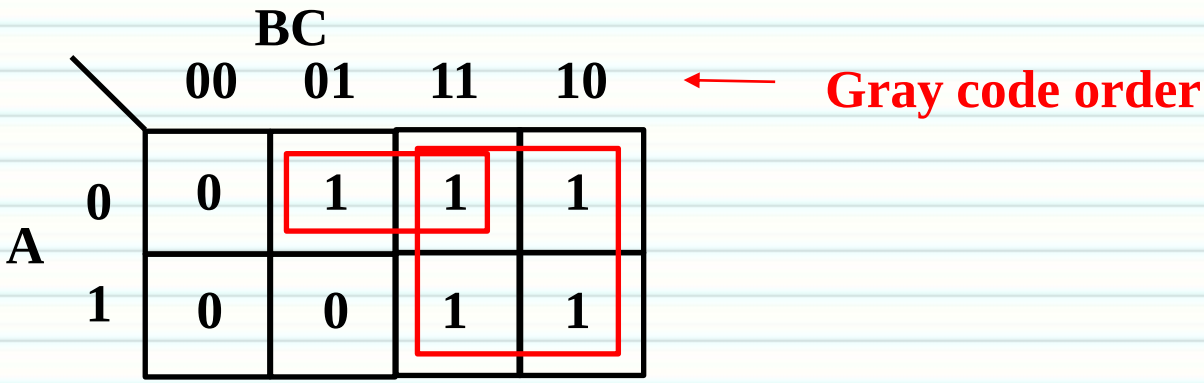
- Karnaugh maps are useful when simplifying logical expressions - this is very helpful in reducing the test associated with logical expressions
- It also helps to reduce the logic to make clearer the dependencies on specific operands - K-maps were used in the State tables to reduce expressions for the state diagrams
- A Karnaugh map is a two-dimensional truth-table. Unlike ordinary (i.e., one-dimensional) truth tables, however, certain logical network simplifications can be easily recognized from a Karnaugh map
- The expression AB' + A'B + AB
- Evaluates to A + B



- Simplification of the logic can help understanding of the state diagram
- Try A'B' + AB' + A'B as an exercise

Karnaugh Maps (2)

- 3-variable Karnaugh maps



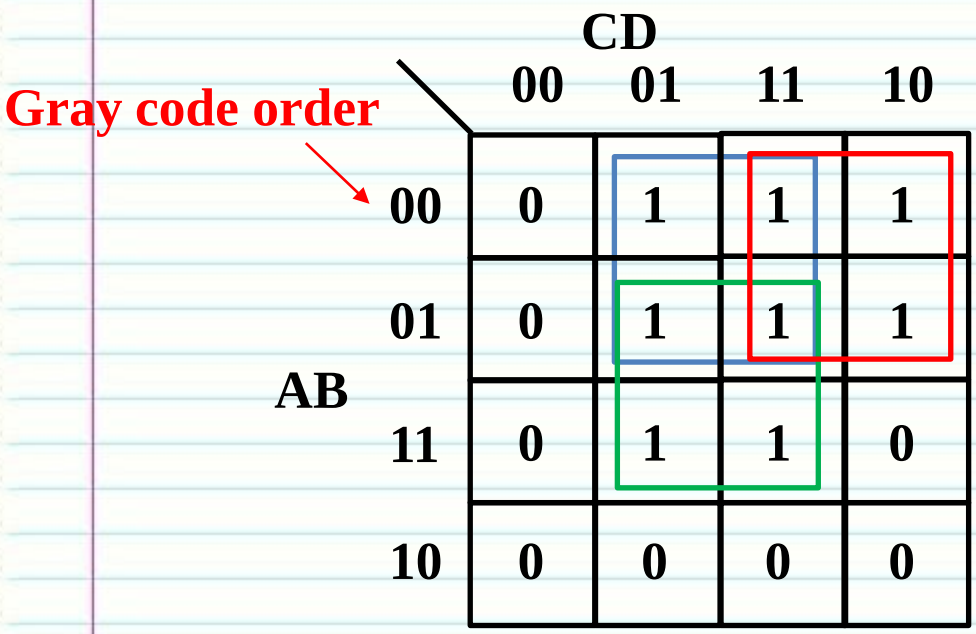
Gray code order

- So, A'B'C + A'BC + A'BC' + ABC + ABC' = A'C + B

- Try A'B'C + A'BC + A'BC' + ABC + ABC' + AB'C as an exercise

Karnaugh Maps (3)

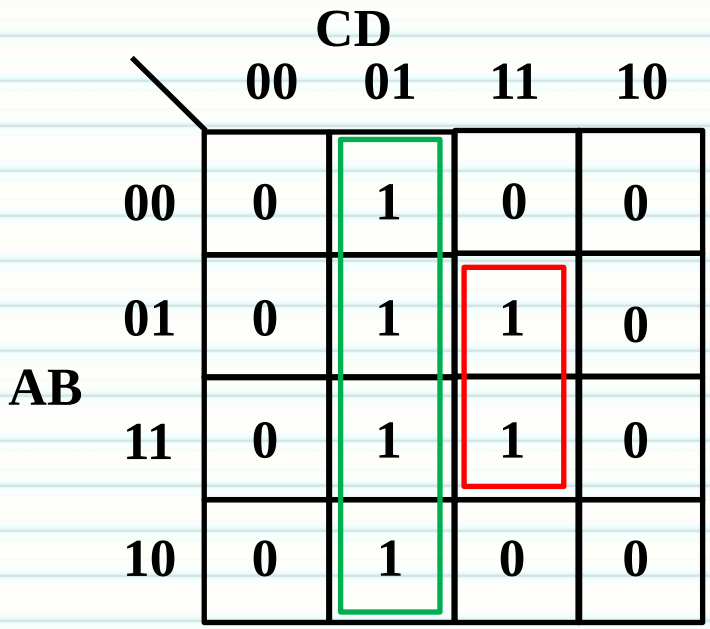
- Reduce A’B’C’D + A’B’CD + A’B’CD’ + A’BC’D + A’BCD + A’BCD’ + ABC’D + ABCD = A’D + A’C + BD



- Class exercise - reduce A’B’C’D + A’B’CD + A’B’CD’ + A’BC’D + A’BCD + A’BCD’ + ABC’D + ABCD + ABCD’

Karnaugh Maps (4)

- Is the following correct?



Karnaugh Maps (5)

- Reduce the truth table

w	x	y	z	f
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	1
0	1	0	1	1
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	1
1	1	0	1	1
1	1	1	0	
1	1	1	1	

Karnaugh Maps (5)

- Reduce the truth table

w	x	y	z	f
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	1
0	1	0	1	1
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	1
1	1	0	1	1
1	1	1	0	
1	1	1	1	

f=xy'

Karnaugh Maps (6)

- Reduce the truth table

w	x	y	z	f
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	1
0	1	0	1	
0	1	1	0	1
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	1
1	1	0	1	
1	1	1	0	1
1	1	1	1	

Karnaugh Maps (6)

- Reduce the truth table

w	x	y	z	f
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	1
0	1	0	1	
0	1	1	0	1
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	1
1	1	0	1	
1	1	1	0	1
1	1	1	1	

AB \ CD	00	01	11	10
00	0	0	0	0
01	1	0	0	1
11	1	0	0	1
10	0	0	0	0

Boolean Logic Identities

Identity Name	AND Form	OR Form
Commutative Law	$xy = yx$	$x+y = y+x$
Associative Law	$(xy)z = x(yz)$	$(x+y)+z = x + (y+z)$
Distributive Law	$x+yz = (x+y)(x+z)$	$x(y+z) = xy+xz$

Identity Name	AND Form	OR Form
Absorption Law	$x(x+y) = x$	$x + xy = x$
DeMorgan's Law	$\overline{(xy)} = \bar{x} + \bar{y}$	$\overline{(x+y)} = \bar{x}\bar{y}$
Double Complement Law	$\overline{(\bar{x})} = x$	

a XOR b = a'b + ab'

Expressions and Logical Terms

- Tautology
  - something that is always true -> a + a'
  - a definition, e.g. a XOR b = a'b + ab'
  - logically we read this as "it is always true that **a XOR b = a'b + ab'** "
  - "a bachelor is an unmarried man" notice that we cannot say "all unmarried men are bachelors"
- Contradiction (sometimes called an absurdity)
  - something that is always false -> a&!a
  - "Rich is running" and "Rich is not running" at the same time/place
- Verification of a positive statement
  - "There is gold in Alaska" - this requires only one case of gold being in Alaska
- Verification of a negative statement
  - "There is no gold in Alaska" - impossible to prove - requires examination of all possible places in Alaska
  - In software test we cannot verify that something did not happen

Expressions and Logical Terms (cont.)

- Let P be “he is a bachelor”, Q “he is unmarried”
  if P -> Q
  P
  ∴Q (by modus ponens)
- Conversion (the converse) - this is one of the most frequent logical errors made, using the logical rules above
  if P -> Q (unchanged from above)
  Q
  ∴ P (wrong!)
- If he is unmarried he must be a bachelor
- “If it is a cat then it is a mammal” becomes
  - “it is a mammal therefore it is a cat”
- Also known as “affirming the consequent”
- In computer science “an abduction” is frequently used in expert systems and is very similar to affirming the consequence
  - “it is a mammal therefore perhaps it is a cat”

Logical Expressions and Test Coverage

Logical Expressions - Terms

- We are still looking at expressions independent of source code but will use source code constructs as they might be specified in requirements
- A logical expression is a **decision**, and the decision consists of **conditions**, conjoined by Boolean operators (and, or, ...). A condition contains no Boolean operators.
- A decision is a Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition.

- Logic Coverage

```
If ( (a>b) || C) && p(x))
    X=1
else
    X=2
If (x>9)
...
```

Decision

Condition

Condition/Decision Coverage

- We start with condition/decision coverage first.
- How do we select condition/decision coverage? Use the following steps
  1. Start at the top row of the truth table and at the bottom row
  2. If the decision result is the **opposite** this pair is a c/d pair.
  3. Select the second from the top and bottom. Repeat step 2 above
  4. Select the nth row from the top and the nth row from the bottom - use the evaluation step 2 to determine if this is a c/d pair.
  5. Continue selecting rows in this manner until the entire table has been traversed.

Condition Coverage (cont.)

- Do Condition coverage as the second coverage step.
- How do we select condition coverage? Use the following steps
  1. Start at the top row of the truth table and at the bottom row
  2. If the decision result is the **same** this pair is a c pair.
  3. Select the second from the top and bottom. Repeat step 2 above
  4. Select the nth row from the top and the nth row from the bottom - use the evaluation step 2 to determine if this is a c pair.
  5. Continue selecting rows in this manner until the entire table has been traversed.

Decision Coverage (cont.)

- We do decision coverage as the last step ini the coverage assessment
- How do we select decision coverage? Use the following steps
  1. Pick the first row in the table - determine what its decision result is.
  2. Pick the first row in the table where the decision result is the opposite of the previous step. This is a possible d pair.
  3. Determine if the possible d pair has not been selected as either a c pair or a c/d pair, if not it is a d pair.

Basic Coverage Exercises (1)

- a + bc

a b c	a + bc	
FFF	F	**c/d 1**
FFT	F	**c/d 2**
FTF	F	**c/d 3**
FTT	T	**c1**
TFF	T	**c1**
TFT	T	**c/d 3**
TTF	T	**c/d 2**
TTT	T	**c/d 1**

- we look for a pair of 3-tuples (in this case) that satisfy the previous conditions
  - condition/decision: (FFF,TTT), (FFT,TTF), or (FTF,TFT)
  - condition (only): (FTT,TFF)
  - decision (only): (FFF,FTT), (FFF,TFF), (FFF,TFT), (FFF, TTF), (FFT, FTT)...

Basic Coverage Exercises (2)

- ab + c

a b c	ab + c	
FFF	F	**c/d 1**
FFT	T	**c1**
FTF	F	**c/d 2**
FTT	T	**c/d 3**
TFF	F	**c/d 3**
TFT	T	**c/d 2**
TTF	T	**c1**
TTT	T	**c/d 1**

- we look for a pair of 3-tuples (in this case) that satisfy the previous conditions
  - condition/decision: (FFF,TTT), (FTF,TFT), or (FTT,TFF)
  - condition (only): (FFT,TTF)
  - decision (only): (FFF,FFT), (FFF,FTT), (FFF,TFT), (FFF, TTF), (FFT, FTF)...

Coverage Example

- What is the coverage criteria below?

a	b	c	(a || b) && c
TRUE	TRUE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	TRUE	TRUE
TRUE	TRUE	FALSE	TRUE
FALSE	TRUE	FALSE	FALSE
TRUE	FALSE	FALSE	FALSE
TRUE	FALSE	TRUE	TRUE

Coverage Example

- What is the coverage criteria below?

a	b	c	(a || b) && c
TRUE	TRUE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	TRUE	TRUE
TRUE	TRUE	FALSE	TRUE
FALSE	TRUE	FALSE	FALSE
TRUE	FALSE	FALSE	FALSE
TRUE	FALSE	TRUE	TRUE

Condition/decision

None - no coverage

Decision coverage

Does Coverage Criteria Really Matter?

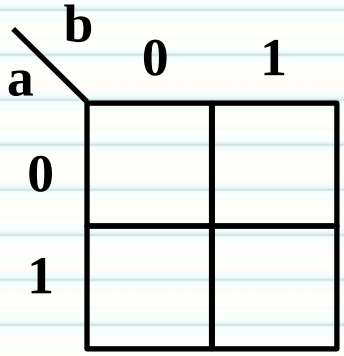
- Security and/or safety applications are complex and code coverage is simply one of many means needed to ensure correct operation.
- At this point in the course we are studying requirements specifications, but we will answer coverage in terms of both requirements and code
- Coverage criteria is vitally important - its importance is best emphasized with the following questions
  - "What acceptance tests" or "What requirements based tests do I need to test a logical expression in the requirements?"
  - "What completion criteria should a software engineer who is developing code for a security or safety critical application have?"
  - So coverage criteria in essence tell us when we have enough test cases
- The typical approach is then to have an overarching coverage criteria and then to design test cases to meet this criteria - when we've done this - we are completed with test case design

Fault Taxonomies

- It is important to examine fault taxonomies when studying test - we looked at several regarding requirements
- We now look at logical expressions - fault taxonomies tell us how things fail and this is important because we want to focus our test effort
- Throughout the class we will use fault taxonomies of various kinds to drive our software test effort
- This will ensure that we are designing test cases to detect defects

What Kind of Errors Can We Make with Logic?

- If we look at a simple logical expression f(a,b) = a + b we can determine the total number of ways that we can implement a two variable boolean expression from the following K-map



- We have terms a'b' + a'b + ab' + ab from the map – each of these terms can be either a 1 or a 0 on the map, so we end up with the following

  $a_0a'b' + a_1a'b + a_2ab' + a_3ab$

- We also see that the map can have the following values for a_0, a_1, a_2, a_3

  0 0 0 0
  0 0 0 1
  …
  1 1 1 1

  16 possible values = 2^{2^n} , n = # of conditions

What Kind of Errors Can We Make with Logic?

- We end up with the following set of possible errors for f(a,b) = a + b

a_0	a_1	a_2	a_3		Result	Notes	# Mistakes
0	0	0	0		0	aa', bb'	2
0	0	0	1		ab	and instead of or	1
0	0	1	0		ab'	omit operator and invert literal	2
0	0	1	1		a	omit 2nd term	1
0	1	0	0		a'b	omit operator and invert literal	2
0	1	0	1		b	omit 1st term	1
0	1	1	0		a X b	XOR instead of or	1
0	1	1	1		a + b	original expression	0
1	0	0	0		a'b'	invert expression	1
1	0	0	1		a EQV b	invert XOR instead of or	2
1	0	1	0		b'	omit term and invert literal	2
1	0	1	1		a + b'	invert literal	1
1	1	0	0		a'	omit term and invert literal	2
1	1	0	1		a' + b	invert literal	1
1	1	1	0		a' + b'	invert both literals	2
1	1	1	1		1	a + a', b + b'	2

- If we look at the **number of mistakes** we discover that some errors are more likely than others

What Kind of Errors Can We Make with Logic?

- If we look at higher level functions f(a,b,c) = a + b + c

- We discover that some errors (in fact many are very low probability errors)

a'b'c + a'bc' + abc
ac' + b'c + ab'
a(b XOR c) + a'(b XOR c)
a'b + bc + a'c + ab'c'

p(n,t) is the probability of detecting an error given n conditions and t tests

$$p(n,t) = 1 - \left(\frac{2^{(2^n - t)} - 1}{2^{2^n}} \right)$$

our examples to the left show that many of these would not be human induced

- Ask not "What Kind of Errors Can We Make with Logic?" but "What Kind of Errors **Do** We Make with Logic?"

- We want to apply a smart fault taxonomy to look at errors that are
  1. most probable to be made by a programmer and
  2. those that would be most easily missed during a code review

What Kind of Errors Do We Make with Logic?

- Mutation testing has helped to significantly advance the state of the art of logical expressions and fault taxonomies.
  - It works by altering source code operators/operations one at a time (in a special way) to determine the adequacy of the test.
- The applicability of this approach is governed by two hypotheses:
  1. The Competent Programmer hypothesis. Software faults introduced by experienced programmers are due to small syntax errors.
  2. The Coupling Effect hypothesis. Complex faults are a series of combined simple faults.
- These two hypotheses together allow us to use single fault taxonomies of logical expressions to assess coverage criteria - because this is what most requirements or programming errors will look like!
- The next several slides will examine single fault taxonomies so that we can better understand typical logical errors and detection

Understanding Logical Coverage Criteria

- When we study single faults taxonomies for logical expressions we get the following (Lau and Yu) for the expression ab + cd

No.	Single Fault Type	Abbr	Example
1	Expression Negation Fault	(*ENF*)	Implementing *ab + cd* as *!(ab + cd)*
2	Term Negation Fault	(*TNF*)	Implementing *ab + cd* as *!(ab) + cd*
3	Term Omission Fault	(*TOF*)	Implementing *ab + cd* as *cd*
4	Literal Reference Fault	(*LRF*)	Implementing *ab + cd* as *cb + cd*
5	Literal Insertion Fault	(*LIF*)	Implementing *ab + cd* as *abc + cd*
6	Literal Omission Fault	(*LOF*)	Implementing *ab + cd* as *b + cd*
7	Literal Negation Fault	(*LNF*)	Implementing *ab + cd* as *!(a)b + cd*
8	Operator Reference Fault +	(*ORF+*)	Implementing *ab + cd* as *abcd*
9	Operator Reference Fault .	(*ORF.*)	Implementing *ab + cd* as *a + b + cd*

- There are 40 such single fault possibilities for the expression ab + cd

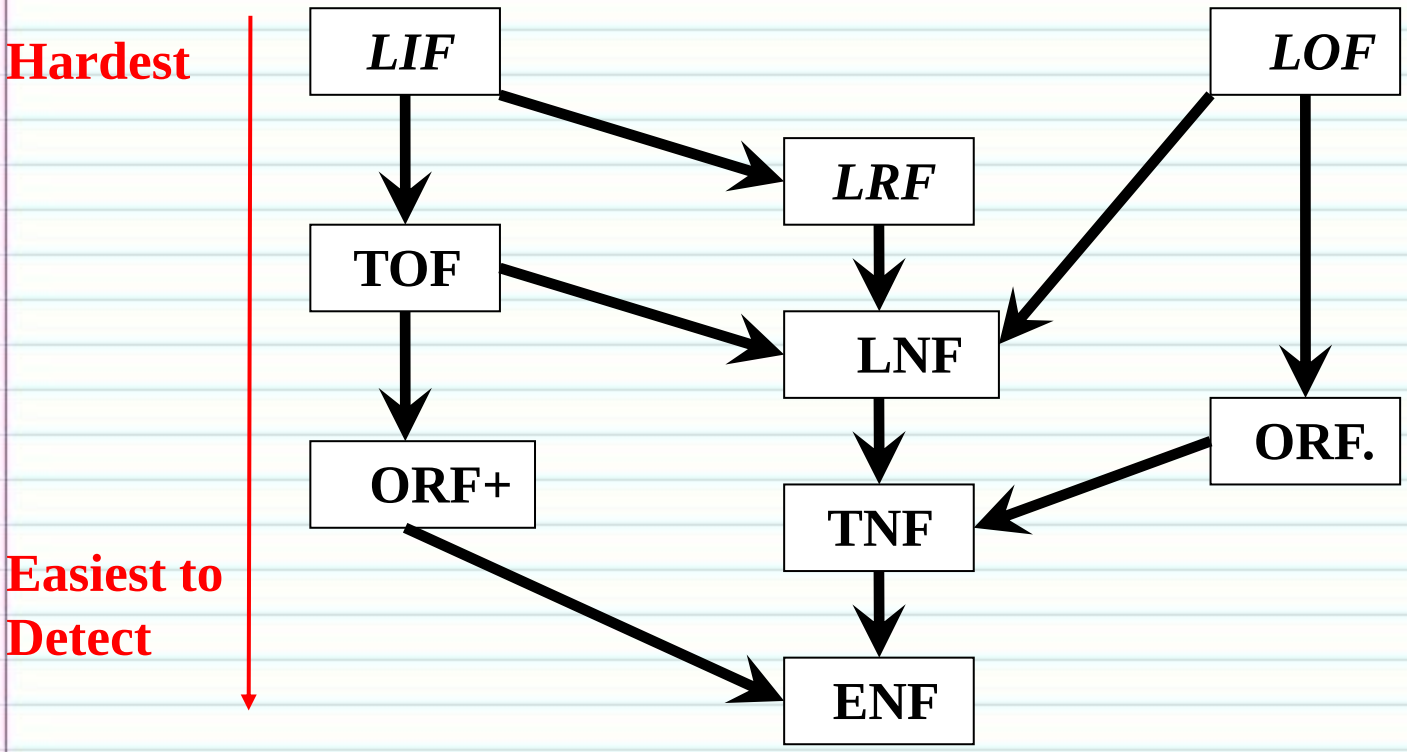
Fault Class	Implementation (mutant)	Fault Class	Implementation (mutant)
ENF	$\overline{ab + cd}$	LRF	$cb + cd, \bar{c}b + cd, db + cd, \bar{d}b + cd,$ $ac + cd, a\bar{c} + cd, ad + cd, a\bar{d} + cd,$ $ab + ad, ab + \bar{a}d, ab + bd, ab + \bar{b}d,$ $ab + ca, ab + c\bar{a}, ab + cb, ab + c\bar{b}$
TNF	$\overline{ab} + cd, ab + \overline{cd}$		
LNF	$\overline{a}b + cd, a\overline{b} + cd, ab + \overline{c}d, ab + c\overline{d}$		
ORF[+]	$abcd$		
ORF[·]	$a + b + cd, ab + c + d$	LIF	$abc + cd, ab\bar{c} + cd, abd + cd,$ $ab\bar{d} + cd, ab + acd, ab + \bar{a}cd,$ $ab + bcd, ab + \bar{b}cd$
TOF	cd, ab		
LOF	$b + cd, a + cd, ab + d, ab + c$		

Understanding Logical Coverage Criteria (cont.)

- The expression ab + cd has these parts (written in IDNF)
  1. Literals a, b, c, d (separated by Boolean operators)
  2. Terms ab, cd (separated by "+" operators since IDNF)
  3. Operators · and + (AND and OR)
  4. Expression: ab + cd (the logical expression)
- These are the things that can be done to each of the parts
  1. Negation
  2. Insertion
  3. Replacement
  4. Omission
- Some of these don't have a mapping because they require more than one fault at a time (e.g. TIF, TRF, ERF, EIF)
- These capture things like parenthesis faults implicitly because they are first translated to IDNF, then the equivalent fault is applied

Lau and Yu's DNF Fault Hierarchy

- Arrow means test for source fault also detects destination fault
- Ignores criterion feasibility



Irreducible DNF (IDNF)

This section of the slides looks at some advanced case studies from the referenced paper

DNF - Disjunctive normal form - a disjunction of conjunctive clauses - "an OR of ANDs"

A minimal-DNF predicate consists of literals connected by **AND** and terms separated by **OR**, where all terms are *prime implicants*.

Terms separated by OR, literals by AND

ab + ac' vs. a(b + c')

- Make each term true and other terms false

  ab + ac vs. ab + abc

- Can't remove a literal without changing predicate's truth value

  ab vs. abc + abc'

Green – in IDNF

Red – not in IDNF

A correct K-map solution will produce a IDNF expression

Comparing Levels of Coverage

- For the expression ab + cd and the 40 possible single faults the following techniques provide these levels of coverage

Coverage technique (for the expression ab + cd)	Percent single faults detected	Single fault types not detected
Decision level coverage	45.0%	-Some TOF, LRF, LIF -All LOF
Condition level coverage	52.5%	-Some ORF, LRF -All LOF
Condition/Decision level coverage	45.0%	-Some TOF, LRF, LIF, ORF, LOF
MC/DC coverage	90.0%	-4 LIF

- Of the widely available coverage levels reported by commercial tools, MCDC is by far the most comprehensive
- For this example, MCDC by itself will not pick up :
  - 4 LIFs - abd + cd, ab + bcd, abc' + cd, ab + a'cd
  - (these could easily be verified by inspection)

- We will study and use MC/DC in this class because of its superior coverage and linear cost (see later slide)

MCDC Coverage and Test Case Design

- Modified Condition/Decision Coverage
  - It is the highest level of test coverage of logical statements reported by commercial tools (at least 30 commercial tools at this point)

- Every condition in a decision has taken on all possible outcomes at least once, and each condition has been shown to affect that decision's outcome independently. A condition is shown to affect a decision's outcome independently by varying just that condition while holding fixed all other possible conditions.

- So, for a && b, here is the corresponding truth table

a	b	a && b
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

MCDC Coverage and Basic Logic

- How can we better test the expression a && b?
- What if we could set a to false and then to true and have the outcome (decision) change value - this would demonstrate that changing a only has an effect on the decision
- Since and's (&&) are sensitive to false we have to keep b set to true, otherwise it will hide a from affecting the decision
- a&&b, with a as the Condition of Interest
  - a=true, b=true, the outcome is true
  - a=false, b=true, the outcome is false
- a&&b, with b as the Condition of Interest, a must be true for the same reason
  - a=true, b=true, the outcome is true
  - a=true, b=false, the outcome is false

MCDC Coverage and Basic Logic (cont.)

- So we end up with four test cases - three of which are unique
  - a=false, b=true, the outcome is false
  - a=true, b=true, the outcome is true
  - a=true, b=false, the outcome is false

- In our expression, a&&b there are 2 conditions (a, b). For MCDC testing of n conditions, n+1 test cases are required.

Class Exercise

- Develop the test cases for MC/DC coverage of a || b

a	b	a || b
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

- Remember that ORs are sensitive to TRUE

Class Exercise (answer)

- Develop the test cases for MC/DC coverage of a || b

1. We know that or's (||) are sensitive to true. This means when we test a as the COI, b must be false (otherwise the outcome is always true)
2. a||b, with a as the COI
   a. a=true, b=false, outcome=true
   b. a=false, b=false, outcome=false
3. a||b, with b as the COI
   a. a=false, b=true, outcome=true
   b. a=false, b=false, outcome=false
4. Our three tests (2 conditions - 3 MCDC tests) are:
   a. a=true, b=false, outcome=true
   b. a=false, b=false, outcome=false
   c. a=false, b=true, outcome=true

More MCDC Examples

- For the expression a && b && c && d
  1. We know that &&'s are sensitive to false, this means one of the tests must be all true (a=true, b=true, c=true, d=true)
  2. We set one and only one condition to false at a time, which causes the outcome to become false
  3. Tests are for the tuple abcd
     a. TTTT
     b. FTTT
     c. TFTT
     d. TTFT
     e. TTTF
  4. We have demonstrated that by setting each condition to false the outcome becomes false - this is a very strong test approach
  5. We have 4 conditions - MCDC requires 5 test cases - which is what we got above

Class Exercise (answer)

- Test the logical expression a + b + c + d

1. Or's are sensitive to true, so we must have FFFF as the first test case
2. Then we set one and only one to true at a time
   a. FFFF
   b. TFFF
   c. FTFF
   d. FFTF
   e. FFFT
3. We have demonstrated that by setting each condition to true the outcome becomes true - this is a very strong test approach
4. We have 4 conditions - MCDC requires 5 test cases - which is what we got above

Three Methods of Applying MCDC

- In practice there are three ways of applying MC/DC coverage depending upon the logical expression being tested:
  1. Unique Cause MC/DC (the original intent) - where we change only one condition at a time (**we've been doing this**)
  2. Masking MC/DC - we allow the inactive clause to change false conditions (**we'll have examples in a few slides**)
  3. Unique Cause + Masking MC/DC - applies UC where it can and M where needed

Other Important MC/DC terms

- Condition of interest (COI) - the condition that is being examined by the independence pair - this is the term that is allowed to vary.
- Independence pair - these are the pairs of tests that show the condition of interest toggling between true and false
- Inactive term(s) - the terms that do NOT contain the COI
- Active term - the term that contains the COI
- Base set - the merged set of tests for the COIs that are IN a multiple condition term(s)

Multiple Condition Expressions

- How do we test a + bc?
  1. Each inactive term (term without the COI) must be **false**
  2. Each active term must be true
  3. Write an X for the COI - write each as n-tuples
  4. COI a (just setting condition a to T and then F)
     a. bc must be false - XFT, XTF, or XFF
  5. COI b
     a. a must be false, c must be true - FXT
  6. COI c
     a. a must be false, b must be true - FTX
  7. Merge COIs within the same term and definitize - this is the **base set**
     a. base set = (FFT,FTT) U (FTF,FTT) = (FFT,FTT,FTF) - COI b U COI c

"base set" because each COI only has two possible values

Multiple Condition Expressions

- How do we test a + bc?
  8. Evaluate the remaining COIs (here COI a)
     a. Determine which possibilities have a common term (when definitized) with the base set
     b. XTF and XFT each have a common term with the base set - there are two UC solutions
        i. base set U XTF = (FFT,FTT,FTF,TTF)
        ii. base set U XFT = (FFT,FTT,FTF,TFT)
     c. The left over set XFF is used for the masking solution
        i. since COI a is never exercised as true we select TFF, so the masking solution is (FFT,FTT,FTF,TFF)

MCDC of Mixed Operator Expressions (cont.)

- How do we test a + bc?
  - Unique Cause solution

	a.	TFT	T
COI a			
COI b	b.	FFT	F
	c.	FTT	T
COI c			
	d.	FTF	F

9. Masking solution - the inactive term is still false but can change

	a.	TFF	T
COI a			
COI b	b.	FFT	F
	c.	FTT	T
COI c			
	d.	FTF	F

Another MCDC example (cont.)

- Develop the UC MCDC solution for a (b+c)
  1. Convert to IDNF - ab + ac, we have a "strongly coupled condition" (a)
  2. Develop the base set
     a. COI b - TXF - TTF, TFF
     b. COI c - TFX - TFT, TFF (note: FTX or FFX is incorrect)
     c. Base set = (TTF,TFF,TFT)
  3. For strongly coupled conditions we need to count condition a twice so we have 4 conditions and 5 tests
  4. We could use
     a. COI a - XTT, XTF, or XFT
     b. We note that XTT is a poor choice - it doesn't tell us if the expression is implemented as ab, ac, or ab + ac
     c. So we choose FTF and FFT as our two remaining tests
  5. Our solution is TTF,TFF,TFT,FTF,FFT - this approach is the "strong MCDC" approach

Another MCDC example (cont.)

- Develop the MCDC solution for ab+ a‘c, this is a strongly coupled condition with inversion - the expression is already IDNF
  1. For COIa
     a. we cannot use XTT (always T) or XFF (always F)
     b. we are only left with XFT or XTF - if we expand this out we discover that we only need TTF,FFT,FTF or TTF,FFT,TFT because FTF or TFT cause both terms to be false similarly
     c. base set = (TTF,FFT,FTF) or (TTF,FFT,TFT)
  2. COI b: TX- c is a don’t care -> TXT or TXF which gives (TTT,TFT) or (TTF,TFF)
  3. COI c: F-X b is a don’t care -> FTX or FFX which gives (FTF,FTT) or (FFF,FFT)
  4. We have 4 conditions - 5 tests - 4 solutions

     Sol 1: TTF,FFT,FTF,TFF,FTT Sol 2: TTF,FFT,FTF,TFF,FFF

     Sol 3: TTF,FFT,TFT,TFF,FFF Sol 4: TTF,FFT,TFT,TTT,FFF

Notice that because of our choices this is not strictly UC

Another Example

- ab + c (UC)
  - COI a => XTF - TTF, FTF
  - COI b => TXF - TTF, TFF
  - COI c => ab=F - FFX, FTX, TFX

  } FTF, TFF, TTF
  - FTF, TFF, TTF, FTT or
  - FTF, TFF, TTF, TFT
- ab + c + d (UC)
  - COI a => XTFF
  - COI b => TXFF
  - COI c => ab=F, d=F - TFXF, FTXF, FFXF,
  - COI d => ab=F, c=F - TFFX, FTFX, FFFX,
  - TTFF, FTFF, TFFF, TFTF, TFFT or
  - TTFF, FTFF, TFFF, TFTF, FTFT or
  - TTFF, FTFF, TFFF, FTTF, TFFT or
  - TTFF, FTFF, TFFF, FTTF, FTFT

UC Solution for ab + cd

- How about ab + cd? (UC solution)
  1. This is in DNF already
  2. COI a
     a. b must be true, cd must be false (FT, TF, or FF) -
        so we get XTFT, XTTF, or XTFF
  3. COI b
     a. a must be true, cd must be false - use the same from before for cd - we get TXFT, TXTF, TXFF
  4. COI c
     a. ab must be false (FT, TF, or FF), d must be true - we get
        FTXT, TFXT, or FFXT
  5. COI d
     a. ab must be false (FT, TF, or FF), c must be true - we get
        FTTX, TFTX, or FFTX
  6. We have 4 conditions - 5 MCDC test cases

UC Solution for ab + cd (cont.)

- There are four possible solutions for the UC solution to ab + cd:
  1. TFFT, TTFT, FTFT, FTTT, FTTF : F, T, F, T, F
  2. TFFT, TFTT, TFTF, TTTF, FTTF : F, T, F, T, F
  3. TFTF, TFTT, TFFT, TTFT, FTFT : F, T, F, T, F
  4. TFTF, TTTF, FTTF, FTTT, FTFT : F, T, F, T, F

Masking Solution for ab + cd

- How about ab + cd? (Masking solution)
  1. This is in DNF already
  2. COI a
     a. b must be true, cd must be false (FT, TF, or FF) - but instead of holding cd as FT we can mix cd across cases, TTFT, FTTF
     b. cd is still false, but we're changing more than one condition at a time - this is the masking approach
  3. COI b
     a. a must be true, cd must be false - use the same from before for cd - we get TTFT, TFTF
  4. COI c
     a. ab must be false (FT, TF, or FF), d must be true - we chose FTTT, TFFT - we mix across ab this time
  5. COI d
     a. ab must be false (FT, TF, or FF), c must be true - we get FTTT, TFTF
  6. We have 4 conditions - 5 MCDC test cases: TFFT, TTTF, FTFT, FTTT, FTTF - this is one Masking solution (more than one condition at a time) - there are 9 total

Masking Solution for ab + cd

- How about ab + cd? (Masking solution)
  7. There are many more masking solutions than UC solutions because we can vary the non-COI terms - 9 total solutions
  8. It was thought that this would provide more flexibility

Cost of Using MC/DC Coverage

- As with any other test endeavor, the main question is how can we achieve good coverage of logical expressions without performing exhaustive testing?
- The modified condition/decision coverage criterion was developed to achieve many of the benefits of multiple-condition testing while retaining the linear growth in required test cases of condition/decision testing. The essence of the modified condition/decision coverage criterion is that each condition must be shown to independently affect the outcome of this decision, i.e., one must demonstrate that the outcome of a decision changes as a result of changing a single condition

Conditions	Possible Combinations	MC/DC Combinations
2	4	3
3	8	4
4	16	5
5	32	6
6	64	7
7	128	8
8	256	9
9	512	10
10	1,024	11

MC/DC achieves good coverage but grows only linearly - $(n+1)/2^n$

Technical Approach to Determine MCDC Tests

- So far we've used an intuitive method to determine the MCDC test cases
  - This is better for understanding what MCDC provides

- There is a technical method for solving this, use a + bc
  1. COI a -> (T + bc) XOR (F + bc) = T XOR (bc) = b' + c'
     i. solve b' + c' for true, we get that bc can be TF, FT, or FF
  2. COI b -> (a + Tc) XOR (a +Fc) = (a + c) XOR (a) = a’c
  3. COI c -> (a + bT) XOR (a + bF) = (a + b) XOR (a) = a’b

- These provide the same COI solutions for a, b, and c as the intuitive approach

Extending MC/DC

- What if we go back to our example of MC/DC coverage criteria for a || b and look at a slight modification

a	b	a || b
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

- What if the expression were (c>7) || b or from above a=c>7
- What are the two partitions for a? {-min,7} and {8,max}
- If I ignore robustness values (-min,max) then I have two values for c that I need c=7 and c=8 for which a evaluates to FALSE and TRUE respectively

a	b	a || b
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

c	b	(c>7) || b
7	TRUE	TRUE
7	FALSE	FALSE
8	FALSE	TRUE
8	TRUE	TRUE

- We're still changing one input at a time and also testing the boundary conditions

Analysis of MC/DC Coverage

Logical expression	MCDC Detection coverage[1]	Terms not detected	Detection with Code Inspection[2]
a + b	100.0%		100.0%
ab	100.0%		100.0%
abc	100.0%		100.0%
a + b + c	100.0%		100.0%
ab + c	91.3%	LIF - ab+bc, ab+a'c	98.7%
abcd	100.0%		100.0%
a + b + c + d	100.0%		100.0%
ab + cd	90.0%	LIF - abd+cd, ab+bcd, abc'+cd, ab+a'cd	98.5%
a + b + cd	96.0%	LIF - ac+b+cd, a+bc+cd, ad'+b+cd, a+bd'+cd	99.4%
ab + ac	100.0%[3]		100.0%
ab + a'c	90.9%[3]	LOF - ab+c	98.6%
ab + cde	90.6%	LIF - abd+cde, abc'+cde, ab+bcde, abe+cde	98.5%
ab + cd + e	86.3%	10 LIF - abd+cd+e, abc'+cd+e, abe'+cd+e, ab+a'cd+e …	97.9%
a + b + c + d + e	100.0%		100.0%
abcde	100.0%		100.0%

Undetected faults are easily detectable during code inspections

Analysis of MC/DC Coverage (cont)

Notes from previous table

Note 1: Single Fault Failure detection of Lau/Yu taxonomy

Note 2: Assuming the required code inspection with 85% defect removal efficiency (Capers Jones industry data)

Note 3: Strong MC/DC approach

Case Study 1 of Advanced Logic Coverage Minimal-MUMCUT

- Minimal-MUMCUT criterion
  - MUMCUT generates many unnecessary tests; several refinements has been addressed, resulting in minimal-MUMCUT.
  - Minimal-MUMCUT selects a test set such that,
    1. All faults in Lau and Yu's DNF fault hierarchy are guaranteed to be detected by the test set. (MCDC tests are only guaranteed to detect TNF and ENF faults)
    2. If even one test is removed from the test set, then at least one fault in Lau and Yu's fault hierarchy will go undetected by the remaining tests in the test set.
  - Note that "minimal" does not mean minimized test set.

Minimal MUMCUT

- Comparison of MCDC and Minimal-MUMCUT
- For the expression ab + cd
  - MCDC gives - TTFT, TTTF, TFTT, FTFT, TFTF
  - Minimal MUMCUT gives - MCDC + FTTT (one additional term)
    - So with one additional test case we get complete coverage of the previous single insertion faults
  - Remember that neither provides full coverage of the truth table - this is unnecessarily expensive
    - MCDC - 5/16 and Minimal-MUMCUT 3/8 from coverage respectively of the truth table (multiple condition coverage)
    - But MCDC and Minimal MUMCUT provide a reduced space coverage (90 percent and 100 percent)
- The much maligned MCDC does a very good job of single fault detection and is reported by commercial tools
  - Minimal MUMCUT can provide a complete coverage, but remember that a Technical Review could also pick up the four LIF faults here

Minimal MUMCUT Concerns

- Minimal MUMCUT/MUMCUT has been an excellent research tool, but
  - It ground-rules out the XOR operator "^" as an ORF candidate
    - XOR is not easily detected but is easily misused - this is an important omission
  - Requires the expression to be IDNF which is not always the case for industry (later researchers are looking at general format expressions)
- Escapes Minimal MUMCUT - the following table shows a few of the code expressions that Minimal MUMCUT cannot distinguish from the original

Logical expression [1]	Minimal MUMCUT miss [1]	# Mistakes
a + b	a ^ b	1
ab + c	ab ^ c [2]	1
a + b + c	a ^ b ^ c	2
	a + b ^ c [2]	1
	a ^ b + c [2]	1
	a ^ c + b [2]	1
ab + cd	ab ^ cd [2]	1
abcd	ab ^ cd [2]	1
	a ^ b ^ c ^ d	3

Note 1: All logical expressions shown as logical expression.

Note 2: Operators && || implemented as bitwise & | operators in the code

Research and MC/DC

- Many research papers show possible improvements to MC/DC - a big research field
- Data is somewhat questionable because of the variety of MC/DC approaches - not sure what the baseline MC/DC coverage is
- Researchers are looking at MC/DC by itself and not addressing the required code inspection element
  - Together they provide extremely good defect removal efficiencies
  - Most of the MC/DC escapes are LIFs which are easily detected in code inspections - add LIF and XOR checks to code inspection checklists
- No defect methodology is going to provide 100% coverage of typical defects but MC/DC with the required inspection is very high

Case Study 2 of Advanced Logic Coverage

- ROR - relational operator replacement - logic based testing is performed through mutation operator replacement - these are analyzed to determine what logic faults can be detected

Original relational operator	Mutants to create		
<	<=	!=	F
>	>=	!=	F
<=	<	==	T
>=	>	==	F
==	<=	>=	F
!=	<	>	T

- Logic based testing with RORG - this provides an assessment of what faults MCDC coverage does not detect
- for the expression (a1 < a2) & (b1 <= b2) | (c1 != c2) the following slide shows test cases the authors propose as MCDC compliant and those generated by the RORG approach

RORG Augmentation of MCDC (cont.)

- We know for the expression a & b |c the following test cases satisfy MCDC criteria: TTF, TFT, TFF, FTF
- For the expression (a1 < a2) & (b1 <= b2) | (c1 != c2), the authors suggest these test would satisfy MCDC criteria

Test Case	a1	a2	b1	b2	c1	c2
1	5	6	10	11	21	21
2	5	6	11	10	21	22
3	5	6	11	10	21	21
4	6	5	10	11	21	21

- Do these test satisfy our testing approach used in class?
- No,
  - none of the tests check for the a1=a2 or b1=b2 - BVs have not been fully tested for this
  - the tests for c1 != c2 do not fully test the BV - they have covered < and = but > is not covered - an "=" operator requires 3 test cases

RORG Augmentation of MCDC (cont.)

- The authors assert that the following three test should be added to achieve full coverage

Test Case	a1	a2	b1	b2	c1	c2	a	b	c
5	5	5	10	11	21	21	==		
6	5	6	10	10	21	21		==	
7	5	6	11	10	22	21			>

- The approach we are learning in class would require
  1. 7 test cases to satisfy all BVs
     a. 2 for the a1 < a2
     b. 2 for the b1 <= b2
     c. 3 for the c1!=c2
  2. We would satisfy MCDC criteria in achieving these BV conditions

- This analysis is not meant to criticize the authors worthy research but show the importance of proper test design and how the techniques used in class will meet these criteria

Achieving Boundary Coverage on Conditions

- How do we achieve boundary coverage on the following expression?

  if ((w>3) & (x<9) & (y>0) & (z<7))

- Count conditions and add 1 - we have 5 test cases

	w	x	y	z
T	4	8	1	6
F	3	9	0	7

	w	x	y	z
TTTT	4	8	1	6
FTTT	3	8	1	6
TFTT	4	9	1	6
TTFT	4	8	0	6
TTTF	4	8	1	7

- We have achieved full MCDC coverage
- Notice that between the TTTT test case and the other test cases we are **changing only one condition at a time** - this shows that changing only that condition causes the expression to change

- Student exercise - achieve full boundary coverage of the following condition

  if ((x>3) & (x<9) & (y>0) & (y<7))

Achieving Boundary Coverage on Conditions (cont.)

- How do we achieve boundary coverage on the following expression?
  if ((x>3) & (x<9) & (y>0) & (y<7))
- Count conditions and add 1 - we still have 5 test cases, but there are multiple partitions for each operand x and y
- This makes laying out the logic and test cases harder
- We need to examine the MCDC combinations and determine what is achievable
- The following table shows the possible combinations of x>3 & x<9 and y>0 & y<7 for x and y respectively

	Possible boundary values	
Combination	**X**	**Y**
FF	Illegal	Illegal
FT	3	0
TF	9	7
TT	4 or 8	1 or 6

4 and 8 are interchangeable for x

	x	y
TTTT	4	1
FTTT	3	6
TFTT	9	6
TTFT	8	0
TTTF	8	7

1 and 6 are interchangeable for y

so we are still only changing one condition at a time from the TTTT test

- The test cases in the table above provide full coverage of boundary values and condition/decision coverage but cannot provide full MCDC coverage because of the illegal combinations

Infeasibility

- For the expression (a > b) & (b > c) + (c > a)
- if a > b = true, b > c = true, then (c > a) is infeasible
- Do this as a class exercise
- Infeasible requirements have to be detected and corrected – sometimes these can be detected during technical reviews
- Sometimes these are complex and hard to detect – in some cases detecting infeasibility is undecidable
- When we get to the code level we'll find that tools such as Static Code Analysis tools can make detection of this issues in the code much easier

Backup Slides

Commercially Available MCDC Coverage Tools

- Ada-core
- AdaTest
- Aonix
- Aprobe
- Beacon AUTT
- BullseyeCoverage
- Cantata++
- CodeCover
- CodeSonar
- CodeTest
- CTC++
- DACS Object Coverage
- G-cover
- Hindsight
- McCabeIQ
- Parasoft
- RapiCover
- Rational RT Tester
- Seacon for Simulink
- Simulink
- TBRun (LDRA)
- TCMON
- Telelogic Tau
- TestMate
- TestWorksCoverage
- TestWorksCoverage TCAT
  T-VEC
- VectorCAST
- Verocel
- VerOCode