Ch. 3 : Logic Coverage

Four Structures for Modeling Software

- Graphs
- Logic
- Input Space
- Syntax

Applied to:
- Source
- FSMs
- Specs
- DNF

Applied to:
- Source
- Specs
- Design
- Use cases

Applied to:
- Source
- Models
- Integ
- Input

Introduction to Software Testing (Ch 3)
Framework of logics

• **Propositional logics, Boolean logics**
  - $p, \land, \lor, \neg, \rightarrow, \ldots$
  - ex: happy $\land$ (rain $\rightarrow$ smoke)

• **1st-order logics**
  - $x, p(x), \land, \lor, \neg, \forall, \exists, \rightarrow, \ldots$
  - ex: $(\text{human}(\text{Socrates}) \land \forall x(\text{human}(x) \rightarrow \text{mortal}(x)))$
    $\rightarrow \text{mortal}(\text{Socrates})$

• **Higher-order logics, 2nd-order logics**
  - $x, p(x), \land, \lor, \neg, \forall, \exists, F, \rightarrow, \ldots$
  - Ex: $\exists F(F(\text{Socrates}) \land \forall x \exists y((F(x) \land \text{friend}(x,y)) \rightarrow F(y)))$
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

- 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 with >, <, ==, >=, <=, !=
  - **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 non-standard term, should be called ATOM.**
Semantics

- Truth table, meaning of propositional logics

<table>
<thead>
<tr>
<th>a &lt; b</th>
<th>D</th>
<th>m &gt;= n*o</th>
<th>((a &lt; b) ∨ D) ∧ (m &gt;= n*o)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

- a truth assignment,
- an interpretation,
- a row
2014/12/05 stopped here.
Examples

• \((a < b) \lor f(z) \land D \land (m \geq n \times o)\)
• Four clauses:
  – \((a < b)\) – relational expression
  – \(f(z)\) – boolean-valued function
  – \(D\) – boolean variable
  – \((m \geq n \times o)\) – 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: \(\{\,(a,b) \mid a \in A \land b \in B \land age(a) = age(b)\}\),
    \(\exists a \in A \exists b \in B \, age(a) = \text{agen}(b)\)
Humans have trouble translating from English to Logic

- “I am interested in SWE 637 and CS 652”
  - course = swe637 OR course = cs652
- “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 → path = Braddock ∨ time > 7:00 → path = Prosperity
  - Hmm … this is incomplete!
  - time < 6:30 → path = Braddock ∨ time ≥ 6:30 → path = Prosperity
Testing and Covering Predicates 
(3.2)

• 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.

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

• When predicates come from conditions on edges, this is equivalent to edge coverage

• PC does not evaluate all the clauses, so …
Predicate Coverage Example

\[(a < b) \lor D) \land (m >= n*o)\]

**Predicate coverage**

**Predicate = true**

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

\[
= (5 < 10) \lor true \land (1 >= 1*1)
\]

\[
= true \lor true \land TRUE
\]

\[
= true
\]

**Predicate = false**

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

\[
= (10 < 5) \lor false \land (1 >= 1*1)
\]

\[
= false \lor false \land TRUE
\]

\[
= false
\]
Clause Coverage Example

\((a < b) \lor D) \land (m \geq n \times o)\)

Clause coverage

<table>
<thead>
<tr>
<th>(a &lt; b) = true</th>
<th>(a &lt; b) = false</th>
<th>D = true</th>
<th>D = false</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 5, b = 10</td>
<td>a = 10, b = 5</td>
<td>D = true</td>
<td>D = false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>m \geq n \times o = true</th>
<th>m \geq n \times o = false</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = 1, n = 1, o = 1</td>
<td>m = 1, n = 2, o = 2</td>
</tr>
</tbody>
</table>

Two tests

1) a = 5, b = 10, D = true, m = 1, n = 1, o = 1
2) a = 10, b = 5, D = false, m = 1, n = 2, o = 2
Problems with PC and CC

• 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)

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

For each $p$ in $P$, TR has test requirements for the clauses in $C_p$ to evaluate to each possible combination of truth values.

<table>
<thead>
<tr>
<th>$a &lt; b$</th>
<th>$D$</th>
<th>$m \geq n \ast o$</th>
<th>$((a &lt; b) \lor D) \land (m \geq n \ast o)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Combinatorial Coverage

• This is simple, neat, clean, and comprehensive …

• But quite expensive!

• $2^N$ tests, where $N$ is the number of clauses
  – Impractical for predicates with more than 3 or 4 clauses

• The literature has lots of suggestions – some confusing

• 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:

\[(5 < 10) \lor true \land (1 \geq 1*1)\]

- Only the first 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**, **determines** \( p \) if and only if 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**.
Determining Predicates (呂宗翰)

\[ p = A \lor B \]

- if \( B = true \), \( p \) is always true.
- so if \( B = false \), \( A \) determines \( p \).
- if \( A = false \), \( B \) determines \( p \).

\[ p = A \land B \]

- if \( B = false \), \( p \) is always false.
- so if \( B = true \), \( A \) determines \( p \).
- if \( A = true \), \( B \) determines \( p \).

- **Goal**: Find tests for each clause when the clause determines the value of the predicate
- This is formalized in **several criteria** that have subtle, but very important, differences
Active Clause Coverage (ACC) (吴家贤)

For each $p$ in $P$ and each major clause $c_i$ in $C_p$, choose minor clauses $c_j$, $j \neq i$, 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

$p = a \lor (b \land c)$

Major clause: $a$

- $a = true, b = false, c = true$
- $a = false, b = false, c = false$

Is this allowed?

- This question caused confusion among testers for years
- Considering this carefully leads to three separate criteria:
  - Minor clauses do not need to be the same
  - Minor clauses do need to be the same
  - Minor clauses force the predicate to become both true and false
General Active Clause Coverage (GACC)

Motivation: For each \( p \) in \( P \) and each major clause \( c_i \) in \( C_p \), choose minor clauses \( c_j, j \neq i \), 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.

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
- \( c_j(c_i = true) \neq c_j(c_i = false) \) for all \( c_j \).

- This is complicated!
- It is possible to satisfy GACC without satisfying predicate coverage
- We really want to cause predicates to be both true and false!
Restricted Active Clause Coverage (RACC)

Motivation: For each $p$ in $P$ and each major clause $c_i$ in $C_p$, choose minor clauses $c_j, j \neq i$, 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.

The values chosen for the minor clauses $c_j$ must be the same when $c_i$ is true as when $c_i$ is false, that is, it is required that $c_j(c_i = true) = c_j(c_i = false)$ for all $c_j$.

- This has been a common interpretation by aviation developers
- RACC often leads to infeasible test requirements
- There is no logical reason for such a restriction
Correlated Active Clause Coverage (CACC)

Motivation: For each \( p \) in \( P \) and each major clause \( c_i \) in \( C_p \), choose minor clauses \( c_j, j \neq i \), 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.

The values chosen for the minor clauses \( c_j \) must cause \( p \) to be true for one value of the major clause \( c_i \) and false for the other, that is, it is required that \( p(c_i = \text{true}) \neq p(c_i = \text{false}) \).

- **A more recent interpretation**
- **Implicitly** allows minor clauses to have different values
- **Explicitly** satisfies (subsumes) predicate coverage
### CACC and RACC

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>( a \land (b \lor c) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

#### Major Clause

\( P_a \): b = true or c = true

CACC can be satisfied by choosing any of rows 1, 2, 3 AND any of rows 5, 6, 7 – a total of nine pairs.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>( a \land (b \lor c) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

#### RACC

RACC can only be satisfied by row pairs (1, 5), (2, 6), or (3, 7)

Only three pairs
Inactive Clause Coverage (ICC)

- The active clause coverage criteria ensure that “major” clauses do affect the predicates.
- Inactive clause coverage takes the opposite approach – major clauses do not affect the predicates.

For each \( p \) in \( P \) and each major clause \( c_i \) in \( C_p \), choose minor clauses \( c_j, j \neq i \), so that \( c_i \) does not determine \( p \).

TR has four requirements for each \( c_i \):

1. \( c_i \) evaluates to true with \( p \) true,
2. \( c_i \) evaluates to false with \( p \) true,
3. \( c_i \) evaluates to true with \( p \) false, and
4. \( c_i \) evaluates to false with \( p \) false.
General and Restricted ICC

- Unlike ACC, the notion of correlation is not relevant
  - $c_i$ does not determine $p$, so cannot correlate with $p$

- Predicate coverage is always guaranteed

*General Inactive Clause Coverage (GICC):* For each $p$ in $P$ and each major clause $c_i$ in $C_p$, choose minor clauses $c_j$, $j \neq i$, so that $c_i$ does not determine $p$.

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, $c_j(c_i = \text{true}) = c_j(c_i = \text{false})$ for all $c_j$ or $c_j(c_i = \text{true}) \neq c_j(c_i = \text{false})$ for all $c_j$.

*Restricted Inactive Clause Coverage (RICC):* For each $p$ in $P$ and each major clause $c_i$ in $C_p$, choose minor clauses $c_j$, $j \neq i$, so that $c_i$ does not determine $p$.

The values chosen for the minor clauses $c_j$ must be the same when $c_i$ is true as when $c_i$ is false, that is, it is required that $c_j(c_i = \text{true}) = c_j(c_i = \text{false})$ for all $c_j$. 


Logic Coverage Criteria Subsumption

- Combinatorial Clause Coverage (COC)
  - Restricted Active Clause Coverage (RACC)
  - Restricted Inactive Clause Coverage (RICC)
  - Correlated Active Clause Coverage (CACC)
  - General Active Clause Coverage (GACC)
- Clause Coverage (CC)
- Predicate Coverage (PC)
- General Inactive Clause Coverage (GICC)
Making Clauses Determine a Predicate

• Finding values for minor clauses \( c_j \) is easy for simple predicates
• But how to find values for more complicated predicates?
• Definitional approach:

\[- p_c=\text{true} \text{ is predicate } p \text{ with every occurrence of } c \text{ replaced by true} \]

\[- p_c=\text{false} \text{ is predicate } p \text{ with every occurrence of } c \text{ replaced by false} \]

\[ p = ((a < b) \lor D) \land (m \geq n \times o) \]

\[ p_{(a<b)=true} = (true \lor D) \land (m \geq n \times o) \]

\[ = (m \geq n \times o) \]

\[ p_{(a<b)=false} = (false \lor D) \land (m \geq n \times o) \]

\[ = D \land (m \geq n \times o) \]
Making Clauses Determine a Predicate

• Finding values for minor clauses $c_j$ is easy for simple predicates
• But how to find values for more complicated predicates?
• Definitional approach:
  \[ p_{c=\text{true}} \] is predicate $p$ with every occurrence of $c$ replaced by true
  \[ p_{c=\text{false}} \] is predicate $p$ with every occurrence of $c$ replaced by false

• To find values for the minor clauses, connect $p_{c=\text{true}}$ and $p_{c=\text{false}}$ with exclusive OR

\[ p_c = p_{c=\text{true}} \oplus p_{c=\text{false}} \]

• After solving, $p_c$ describes exactly the values needed for $c$ to determine $p$
Examples

\[ p = a \lor b \]

\[ p_a = p_a=\text{true} \oplus p_a=\text{false} \]

\[ = (\text{true} \lor b) \text{ XOR } (\text{false} \lor b) \]

\[ = \text{true XOR } b \]

\[ = \neg b \]

\[ p = a \land b \]

\[ p_a = p_a=\text{true} \oplus p_a=\text{false} \]

\[ = (\text{true} \land b) \oplus (\text{false} \land b) \]

\[ = b \oplus \text{false} \]

\[ = b \]

\[ p = a \lor (b \land c) \]

\[ p_a = p_a=\text{true} \oplus p_a=\text{false} \]

\[ = (\text{true} \lor (b \land c)) \oplus (\text{false} \lor (b \land c)) \]

\[ = \text{true } \oplus (b \land c) \]

\[ = \neg (b \land c) \]

\[ = \neg b \lor \neg c \]

• “\text{NOT } b \lor \text{NOT } c”\text{ means either } b \text{ or } c \text{ can be false}\

• \text{RACC requires the same choice for both values of } a, \text{ CACC does not}
Repeated Variables

- The definitions in this chapter yield the same tests no matter how the predicate is expressed

- \((a \lor b) \land (c \lor b) == (a \land c) \lor b\)

- \((a \land b) \lor (b \land c) \lor (a \land c)\)
  - Only has 8 possible tests, not 64

- Use the simplest form of the predicate, and ignore contradictory truth table assignments
2013/12/05 stopped here.
A More Subtle Example

\[ p = (a \land b) \lor (a \land \neg b) \]

\[ p_a = p_{a=true} \oplus p_{a=false} \]
\[ = ((true \land b) \lor (true \land \neg b)) \oplus ((false \land b) \lor (false \land \neg b)) \]
\[ = (b \lor \neg b) \oplus false \]
\[ = true \oplus false \]
\[ = true \]

\[ p = (a \land b) \lor (a \land \neg b) \]

\[ p_b = p_{b=true} \oplus p_{b=false} \]
\[ = ((a \land true) \lor (a \land \neg true)) \oplus ((a \land false) \lor (a \land \neg false)) \]
\[ = (a \lor false) \oplus (false \lor a) \]
\[ = a \oplus a \]
\[ = false \]

- \textit{a} always determines the value of this predicate
- \textit{b} never determines the value – \textit{b} is \textit{irrelevant}!
Infeasible Test Requirements

- Consider the predicate:
  
  \[(a > b \land b > c) \lor c > a\]

- \((a > b) = true, (b > c) = true, (c > a) = true\) is infeasible

- As with graph-based criteria, infeasible test requirements have to be recognized and ignored

- Recognizing infeasible test requirements is hard, and in general, undecidable

- Software testing is inexact – engineering, not science
Logic Coverage Summary

- Predicates are often very simple—in practice, most have less than 3 clauses
  - In fact, most predicates only have one clause!
  - With only clause, PC is enough
  - With 2 or 3 clauses, CoC is practical
  - Advantages of ACC and ICC criteria significant for large predicates
    - CoC is impractical for predicates with many clauses

- Control software often has many complicated predicates, with lots of clauses

- Question … why don’t complexity metrics count the number of clauses in predicates?
Logic Expressions from Source

• Predicates are derived from **decision** statements in programs

• In programs, most predicates have **less than four** clauses
  – Wise programmers actively strive to keep predicates simple

• When a predicate only has one clause, COC, ACC, ICC, and CC all collapse to **predicate coverage** (PC)

• Applying logic criteria to program source is hard because of **reachability** and **controllability**:
  – **Reachability**: Before applying the criteria on a predicate at a particular statement, we have to **get to** that statement
  – **Controllability**: We have to **find input values** that indirectly assign values to the variables in the predicates
  – Variables in the predicates that are not inputs to the program are called **internal variables**

• These issues are illustrated through an example in the following slides ...
Thermostat (pg 1 of 2)（庭橻講）

1 // Jeff Offutt--October 2010
2 // Programmable Thermostat
3 import java.io.*;
4 class thermostat
5 {
6 private Heater myHeater;
7 // Decide whether to turn the heater on, and for how long.
8 public boolean turnHeaterOn ( 
9     int curTemp, /* Current temperature reading */
10     int thresholdDiff, /* Temp difference until we turn heater on */
11     Minutes timeSinceLastRun, /* Time since heater stopped */
12     Minutes minLag, /* How long I need to wait */
13     Time timeOfDay, /* current time (Hours and minutes) */
14     Day dayOfWeek, /* Monday, Tuesday, ... */
15     Settings programmedSettings [], /* User's program, by day */
16     boolean Override, /* Has user overridden the program */
17     int overTemp /* OverridingTemp */
18 )
Thermostat (pg 2 of 2)（庭楘講）

19 {
20     int desiredTemp;
21     // getPeriod() translates time into Morning, Day, Evening, Night
22     desiredTemp = programmedSettings [dayOfWeek].getDesiredTemp
23       (getPeriod [TimeOfDay]);
24     if (((curTemp < desiredTemp - thresholdDiff) ||
25         (Override && curTemp < overTemp - thresholdDiff)) &&
26         timeSinceLastRun.greaterThan (minLag))
27     { // Turn on the heater
28         // How long? Assume 1 minute per degree (Fahrenheit)
29         int timeNeeded = curTemp - desiredTemp;
30         if (Override)
31             timeNeeded = curTemp - overTemp;
32         myHeater.setRunTime (timeNeeded);
33         return (true);
34     }
35     else
36         return (false);
37 } // End turnHeaterOn
38 } // End class
Two Thermostat Predicates （庭榮講）

24-26: (((curTemp < desiredTemp - thresholdDiff) || (Override && curTemp < overTemp - thresholdDiff)) && timeSinceLastRun.greaterThan (minLag))

30: (Override)

Simplify

a : curTemp < desiredTemp - thresholdDiff
b : Override
c : curTemp < overTemp - thresholdDiff
d : timeSinceLastRun.greaterThan (minLag)

24-26: (a || (b && c)) && d

30: b
Reachability for Thermostat Predicates

24-26 : True

49: \((a \lor (b \&\& c)) \&\& d\)

\[\text{curTemp} < \text{desiredTemp} - \text{thresholdDiff}\]

Need to solve for the internal variable \(\text{desiredTemp}\)

\[\text{programmedSettings [dayOfWeek].getDesiredTemp (getPeriod [TimeOfDay])}\]

\[\text{programmedSettings [Monday].setDesiredTemp (Morning, 69)}\]
Predicate Coverage \((true)\)

\[(a \| (b \&\& c)) \&\& d\]

\begin{align*}
\text{a} : & \text{true} \\
\text{b} : & \text{true} \\
\text{c} : & \text{true} \\
\text{d} : & \text{true}
\end{align*}

\begin{align*}
\text{curTemp} & < \text{desiredTemp} - \text{thresholdDiff} : \text{true} \\
\text{Override} & : \text{true} \\
\text{curTemp} & < \text{overTemp} - \text{thresholdDiff} : \text{true} \\
\text{timeSinceLastRun.greaterThan} & (\text{minLag}) : \text{true}
\end{align*}

\begin{align*}
\text{programmedSettings [Monday].setDesiredTemp} & (\text{Morning, 69}) \\
// \text{dayOfWeek} & = \text{Monday} \\
// \text{timeOfDay} & = 8:00 \\
\text{curTemp} & = 63 \\
63 & < 69 - 5 \\
\text{Override} & : \text{true} \\
\text{overTemp} & = 70 \\
63 & < 70 - 5 \\
\text{timeSinceLastRun.setValue} & (12) \\
\text{minLag} & = 10
\end{align*}
Predicate Coverage \textit{(false)}

\[(a \| (b \&\& c)) \&\& d\]

\begin{itemize}
  \item a : false
  \item b : false
  \item c : false
  \item d : false
\end{itemize}

curTemp < desiredTemp – thresholdDiff : false 
Override : false 
curTemp < overTemp – thresholdDiff : false 
timeSinceLastRun.greaterThan (minLag) : false

programmedSettings \texttt{[Monday].setDesiredTemp} \texttt{(Morning, 69)}
// dayOfWeek = Monday 
// timeOfDay = 8:00 
curTemp = 66
66 < 69 – 5 
Override : false 
overTemp = 70 
66 < 70 – 5 
timeSinceLastRun.setValue (8) 
minLag = 10
Correlated Active Clause Coverage (1 of 5)

\[ P_a = ((a \lor (b \land c)) \land d) \oplus ((a \lor (b \land c)) \land d) \]

\[ ((T \lor (b \land c)) \land d) \oplus ((F \lor (b \land c)) \land d) \]

\[ (T \land d) \oplus ((b \land c) \land d) \]

\[ d \oplus ((b \land c) \land d) \]

\[ !(b \land c) \land d \]

\[ (\neg b \lor \neg c) \land d \]

Check with the logic coverage web app
http://cs.gmu.edu:8080/offutt/coverage/LogicCoverage
Correlated Active Clause Coverage (2 of 5)

Six tests needed for CACC on Thermostat

(a || (b && c)) && d

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Correlated Active Clause Coverage (3 of 5)

curTemp desiredTemp thresholdDiff
curTemp < desiredTemp - thresholdDiff  63  69  5
!(curTemp < desiredTemp - thresholdDiff)  66  69  5

desiredTemp = programmedSettings [Monday].setDesiredTemp (Morning, 69)
dayOfWeek = Monday
timeOfDay = 8:00

Override
Override T
!Override F

curTemp overTemp thresholdDiff
curTemp < overTemp - thresholdDiff  63  70  5
!(curTemp < overTemp - thresholdDiff)  66  70  5

timeSinceLastRun minLag
timeSinceLastRun.greaterThan (minLag)  12  10
!(timeSinceLastRun.greaterThan (minLag))  8  10

These values then need to be placed into calls to turnHeaterOn() to satisfy the 6 tests for CACC.
Correlated Active Clause Coverage (4 of 5)

desiredTemp = programmedSettings [Monday].setDesiredTemp (Morning, 69)

1. T t f t
   a = T : curTemp = 63;  c = f : curTemp = 66
   turnHeaterOn ( 63/66, 5, 12, 10, 8:00, Monday, programmedSettings, true, 70 )

2. F t f t
   turnHeaterOn ( 66, 5, 12, 10, 8:00, Monday, programmedSettings, true, 70 )

3. f T t t
   a = f : curTemp = 66;  c = t : curTemp = 63
   turnHeaterOn ( 63/66, 5, 12, 10, 8:00, Monday, programmedSettings, true, 70 )

4. f t F f
   turnHeaterOn ( 66, 5, 8, 10, 8:00, Monday, programmedSettings, true, 70 )

5. t t t T
   turnHeaterOn ( 63, 5, 12, 10, 8:00, Monday, programmedSettings, true, 70 )

6. t t t F
   turnHeaterOn ( 63, 5, 8, 10, 8:00, Monday, programmedSettings, true, 70 )
Correlated Active Clause Coverage (5 of 5)

• Tests 1 and 3 are infeasible with the values we chose
• But we can choose different values for clause $c$
• $curTemp$ is fixed by the solution to clause $a$
• $thresholdDiff$ is also fixed by the solution to clause $a$
• So we choose different values for $overtemp$ ...

1. T t f t
   turnHeaterOn (63, 5, 12, 10, 8:00, Monday, programmedSettings, true, 62)

3. f T t t
   turnHeaterOn (66, 5, 12, 10, 8:00, Monday, programmedSettings, true, 66)
Program Transformation Issues

if ((a && b) || c) {
    S1;
}
else {
    S2;
}

Transform (1)?

d = a && b;
e = d || c;
if (e) {
    S1;
}
else {
    S2;
}

Transform (2)?

if (a) {
    if (b)
        S1;
    else {
        if (c)
            S1;
        else
            S2;
    }
}
else {
    if (c)
        S1;
    else
        S2;
}

if (a) {
    if (b)
        S1;
    else {
        if (c)
            S1;
        else
            S2;
    }
}
else {
    if (c)
        S1;
    else
        S2;
}

Transform (1)?
Problems with Transformed Programs

- Maintenance is certainly harder with Transform (1)
  - Not recommended!
- Coverage on Transform (1)
  - PC on transform does not imply CACC on original
  - CACC on original does not imply PC on transform
- Coverage on Transform (2)
  - Structure used by logic criteria is “lost”
  - Hence CACC on transform 2 only requires 3 tests
  - Note: Mutation analysis (Chapter 5) addresses this problem
- Bottom Line: Logic coverage criteria are there to help you!

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>(a&amp;b)\lor c</th>
<th>CACC</th>
<th>PC</th>
<th>CACC (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary: Logic Coverage for Source Code

- **Predicates** appear in decision statements
  - if, while, for, etc.
- Most predicates have less than **four clauses**
  - But some applications have predicates with many clauses
- The hard part of applying logic criteria to source is resolving the **internal variables**
- Sometimes setting variables requires calling **other methods**
- **Non-local variables** (class, global, etc.) are also input variables if they are used
- If an input variable is changed within a method, it is treated as an **internal variable** thereafter
- To maximize effect of logic coverage criteria:
  - Avoid transformations that hide predicate structure
Specifications in Software

- Specifications can be **formal** or **informal**
  - Formal specs are usually expressed *mathematically*
  - Informal specs are usually expressed in *natural language*

- Lots of **formal languages** and **informal styles** are available

- Most specification languages include **explicit logical expressions**, so it is very easy to apply logic coverage criteria

- Implicit logical expressions in natural-language specifications should be **re-written** as explicit logical expressions as part of test design
  - You will often find mistakes

- One of the most common is **preconditions** …
Preconditions

- Programmers often include **preconditions** for their methods
- The preconditions are often expressed in **comments** in method headers
- Preconditions can be in **javadoc**, “requires”, “pre”, ...

**Example – Saving addresses**

// name must not be empty
// state must be valid
// zip must be 5 numeric digits
// street must not be empty
// city must not be empty

**Rewriting to logical expression**

name != "" \(\wedge\) state in stateList \(\wedge\) zip >= 00000 \(\wedge\) zip <= 99999 \(\wedge\)
street != "" \(\wedge\) city != ""
Shortcut for Predicates in Conjunctive Normal Form

- Conjunctive clauses are connected only by the **and** operator
  - \( A \land B \land C \land \ldots \)

- Each major clause is made active by making all other clauses **true**

- ACC tests are “**all true**” and then a “**diagonal**” of false values:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>
Shortcut for Predicates in Disjunctive Normal Form

- Disjunctive clauses are connected only by the or operator
  \[ A \lor B \lor C \lor \ldots \]
- Each major clause is made active by making all other clauses false
- ACC tests are “all false” and then a “diagonal” of true values:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>
Summary : Logic Coverage for Specs

- Logical specifications can come from **lots of places**:
  - Preconditions
  - Java asserts
  - Contracts (in design-by-contract development)
  - OCL conditions
  - Formal languages

- Logical specifications can describe behavior at **many levels**:
  - Methods and classes (unit and module testing)
  - Connections among classes and components
  - System-level behavior

- Many predicates in specifications are in **disjunctive** normal or **conjunctive** normal form – simplifying the computations
Covering Finite State Machines

- **FSMs are graphs**
  - nodes represent state
  - edges represent transitions among states

- Transitions often have logical expressions as guards or triggers

- As we said:

  **Find a *logical expression* and cover it**
Example – Subway Train

- All Doors Open
  - secondPlatform = right
  - ~emergencyStop \land ~overrideOpen \land doorsClear (all three transitions)

- Left Doors Open
  - trainSpeed = 0 \land platform=left \land (inStation \lor (emergencyStop \land overrideOpen))

- Right Doors Open
  - trainSpeed = 0 \land platform=right \land (inStation \lor (emergencyStop \land overrideOpen))

- All Doors Closed
  - secondPlatform = left
### Determination of the Predicate (郭世揚)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trainSpeed = 0 ∧ platform=left ∧ (inStation ∨ (emergencyStop ∧ overrideOpen))</td>
<td></td>
</tr>
<tr>
<td>platform = left : trainSpeed = 0 ∧ (inStation ∨ (emergencyStop ∧ overrideOpen))</td>
<td></td>
</tr>
<tr>
<td>inStation : trainSpeed = 0 ∧ platform = left ∧ (¬ emergencyStop ∨ ¬ overrideOpen)</td>
<td></td>
</tr>
<tr>
<td>emergencyStop : trainSpeed = 0 ∧ platform = left ∧ (¬ inStation ∧ overrideOpen)</td>
<td></td>
</tr>
<tr>
<td>overrideOpen : trainSpeed = 0 ∧ platform = left ∧ (¬ inStation ∧ emergencyStop)</td>
<td></td>
</tr>
</tbody>
</table>
Test Truth Assignments (CACC)(鄭惟浩)

\[ trainSpeed = 0 \land platform=\text{left} \land (inStation \lor (emergencyStop \land overrideOpen)) \]

<table>
<thead>
<tr>
<th></th>
<th>trainSpeed=0</th>
<th>platform=left</th>
<th>inStation</th>
<th>emergencyStop</th>
<th>overrideOpen</th>
</tr>
</thead>
<tbody>
<tr>
<td>trainSpeed = 0</td>
<td>T</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>trainSpeed != 0</td>
<td>F</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>platform = left</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>platform != left</td>
<td>t</td>
<td>F</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>inStation</td>
<td>t</td>
<td>t</td>
<td></td>
<td>F</td>
<td>f</td>
</tr>
<tr>
<td>\neg inStation</td>
<td>t</td>
<td>t</td>
<td>F</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>emergencyStop</td>
<td>t</td>
<td>t</td>
<td>f</td>
<td>T</td>
<td>t</td>
</tr>
<tr>
<td>\neg emergencyStop</td>
<td>t</td>
<td>t</td>
<td>f</td>
<td>F</td>
<td>t</td>
</tr>
<tr>
<td>overrideOpen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>overrideOpen</td>
<td>t</td>
<td>t</td>
<td>F</td>
<td>t</td>
<td>T</td>
</tr>
<tr>
<td>\neg overrideOpen</td>
<td>t</td>
<td>t</td>
<td>F</td>
<td>t</td>
<td>F</td>
</tr>
</tbody>
</table>

Note: other choices are possible
Complicating Issues

• Some buttons must be pressed **simultaneously** to have effect – so timing must be tested

• **Reachability** : The tests must reach the state where the transition starts (the prefix)

• **Exit** : Some tests must continue executing to an end state

• **Expected output** : The expected output is the state that the transition reaches for true values, or same state for false values

• **Accidental transitions** : Sometimes a false value for one transition happens to be a true value for another
  – The alternate expected output must be recognized
Reachability via weakest precondition (1/2)

- State (a) $\rightarrow$ (b): precondition: $x < 5$; action: $x := 0$;
- State (b) $\rightarrow$ (c): precondition: $x > y$; action: $x := 3$;
- Assume that at (c), we want to test $x > z$.
- Assume initial state is (a).

Weakest precondition:

1. The precondition of $x := 3$ to $x > z$: $x > z[x = 3] = 3 > z$
2. The precondition of $(x > y)$ to $3 > z$: $x > y && 3 > z$
3. The precondition of $x := 0$ to $x > y && 3 > z$: $0 > y && 3 > z$
4. The precondition: $(x < 5)$ to $0 > y && 3 > z$: $x < 5 && 0 > y && 3 > z$
5. So we need input of $x$, $y$, $z$ to make $x < 5 && 0 > y && 3 > z$ true at initial state (a).
6. $x = 0$, $y = -1$, $z = 0$ as test input to check $x > z$ at (c).
Reachability via weakest precondition (2/2)

- Assume that at (c), we want to test $\cos(x)+a*b > z$.
- Assume initial state is (a).

Weakest precondition:

1. The precondition of $x:=3$ to $\cos(x)+a*b>z$: $>z[x=3] = \cos(3)+a*b>z$
2. The precondition of $(x>y)$ to $3>z$: $x>y&&\cos(3)+a*b>z$
3. The precondition of $x:=0$ to $x>y&&3>z$: $0>y&&\cos(3)+a*b>z$
4. The precondition: $(x<5)$ to $0>y&&3>z$: $x<5&&0>y&&\cos(3)+a*b>z$
5. So we need input of $x$, $y$, $z$ to make $x<5&&0>y&&\cos(3)+a*b>z$ true at initial state (a).
6. $x=0$, $y = -1$, $z = 0$, $a=b=2$ as test input to check $x>z$ at (c).
Test Scripts

- **Test scripts** are executable sequences of value assignments

- **Mapping problem**: The names used in the FSMs may not match the names in the program
  - Sometimes a direct name-to-name mapping can be found
  - Sometimes more complicated actions must be taken to assign the appropriate values
  - *Simulation*: Directly inserting value assignments into the middle of the program

- The solution to this is implementation-specific
Summary FSM Logic Testing

- FSMs are widely used at all levels of abstraction
- Many ways to express FSMs
  - Statecharts, tables, Z, decision tables, Petri nets, …
- Predicates are usually explicitly included on the transitions
  - Guards
  - Actions
  - Often represent safety constraints
- FSMs are often used in embedded software
Introduction to Software Testing
Chapter 3.6
Disjunctive Normal Form Criteria

Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/
2013/12/19 stopped here.
Disjunctive Normal Form (DNF)

- Common Representation for Boolean Functions
  - Slightly Different Notation for Operators
  - Slightly Different Terminology

- Basics:
  - A literal is a clause or the negation (overstrike) of a clause
    - Examples: \( a, \overline{a} \)
  - A term is a set of literals connected by logical “and”
    - “and” is denoted by adjacency instead of \( \land \)
    - Examples: \( ab, a\overline{b}, \overline{a}\overline{b} \) for \( a \land b, a \land \neg b, \neg a \land \neg b \)
  - A (disjunctive normal form) predicate is a set of terms connected by “or”
    - “or” is denoted by \( + \) instead of \( \lor \)
    - Examples: \( abc + \overline{a}b + a\overline{c} \)
    - Terms are also called “implicants”
      - If a term is true, that implies the predicate is true
Disjunctive Normal Form (DNF)

Examples:
• abc + ~ab +b~c
• ab + ~b~c
• a(b+~c) not DNF $\Rightarrow$ distributivity: ab + a~c: DNF
• ~(a+~c) not DNF $\Rightarrow$ de Morgan’s law: ~a~~c $\Rightarrow$ ~ac: DNF

Properties of DNF:
• a + ab $\equiv$ a
• a + a~b $\equiv$ a
• a~b + b $\equiv$ a + b
Implicant Coverage (張唯霖講)

- Obvious coverage idea: Make each implicant evaluate to “true”.
  - Problem: Only tests “true” cases for the predicate.
  - Solution: Include DNF representations for negation.

**Implicant Coverage (IC):** Given DNF representations of a predicate \( f \) and its negation \( \overline{f} \), for each implicant in \( f \) and \( \overline{f} \), TR contains the requirement that the implicant evaluate to true.

- Example: \( f = ab + b\overline{c} \quad \overline{f} = \overline{b} + \overline{ac} \)
  - Implicants: \( \{ ab, b\overline{c}, \overline{b}, \overline{ac} \} \)
  - Possible test set: \( \{TTF, FFT\} \)
- Observation: IC is relatively weak
Improving on Implicant Coverage （Martin講）

• Additional Definitions:
  – A proper subterm is a term with one or more clauses removed
    • Example: $abc$ has 6 proper subterms: $a, b, c, ab, ac, bc$
  – A prime implicant is an implicant such that no proper subterm is also an implicant (in any DNF of the predicate).
    • Example: $f = ab + abc$
    • Implicant $ab$ is a prime implicant
    • Implicant $a\overline{bc}$ is not a prime implicant (due to proper subterm $ac$)
      $$-f = ab + \overline{a}bc = a(b + \overline{bc}) = a(b + c) = ab + ac$$
  – A redundant implicant is an implicant that can be removed without changing the value of the predicate
    • Example: $f = ab + ac + \overline{bc}$
    • $ab$ is redundant
    • Predicate can be written: $ac + bc$
unique true points

• A minimal DNF representation is one with only prime, nonredundant implicants.
• A unique true point with respect to a given implicant is an assignment of truth values so that
  – the given implicant is true, and
  – all other implicants are false
• Hence a unique true point test focuses on just one implicant
• A minimal representation guarantees the existence of at least one unique true point for each implicant

Unique True Point Coverage (UTPC) : Given minimal DNF representations of a predicate $f$ and its negation $\overline{f}$, TR contains a unique true point for each implicant in $f$ and $\overline{f}$. 
Unique True Point Example

- Consider again: \( f = ab + bc \) \( \bar{f} = \bar{b} + \bar{ac} \)
  - Implicants: \( \{ ab, bc, \bar{b}, \bar{ac} \} \)
  - Each of these implicants is prime
  - None of these implicants is redundant

- Unique true points:
  - \( ab: \{ TTT \} \)
  - \( bc: \{ FTF \} \)
  - \( \bar{b}: \{ FFF, TFF, TFT \} \)
  - \( \bar{ac}: \{ FTT \} \)

- Note that there are three possible (minimal) tests satisfying UTPC

- UTPC is fairly powerful
  - Exponential in general, but reasonable cost for many common functions
  - No subsumption relation wrt any of the ACC or ICC Criteria
Near False Points

- **A near false point** with respect to a clause $c$ in implicant $i$ is an assignment of truth values such that $f$ is false, but if $c$ is negated (and all other clauses left as is), $i$ (and hence $f$) evaluates to true.

- **Relation to determination**: at a near false point, $c$ determines $f$
  - Hence we should expect relationship to ACC criteria

---

**Unique True Point and Near False Point Pair Coverage (CUTPNFP)**: Given a minimal DNF representation of a predicate $f$, for each clause $c$ in each implicant $i$, TR contains a unique true point for $i$ and a near false point for $c$ such that the points differ only in the truth value of $c$.

- Note that definition only mentions $f$, and not $\overline{f}$.
- Clearly, CUTPNFP subsumes RACC
CUTPNFP Example

- Consider \( f = ab + cd \)
  - For implicant \( ab \), we have 3 unique true points: \{TTFF, TTFT, TTTF\}
    - For clause \( a \), we can pair unique true point TTFF with near false point FTFF
    - For clause \( b \), we can pair unique true point TTFF with near false point TFFF
  - For implicant \( cd \), we have 3 unique true points: \{FFTT, FTTT, TFTT\}
    - For clause \( c \), we can pair unique true point FFTT with near false point FFFT
    - For clause \( d \), we can pair unique true point FFTT with near false point FFTF
- CUTPNFP set: \{TTFF, FFTT, TFFF, FTFF, FFFT, FFFT\}
  - First two tests are unique true points; others are near false points
- Rough number of tests required: \# implicants * \# literals
DNF Fault Classes

- ENF: Expression Negation Fault  \( f = ab+c \quad f' = \overline{ab}+c \)
- TNF: Term Negation Fault  \( f = ab+c \quad f' = \overline{ab}+c \)
- TOF: Term Omission Fault  \( f = ab+c \quad f' = ab \)
- LNF: Literal Negation Fault  \( f = ab+c \quad f' = \overline{a}\overline{b}+c \)
- LRF: Literal Reference Fault  \( f = ab + bcd \quad f' = ad + bcd \)
- LOF: Literal Omission Fault  \( f = ab + c \quad f' = a + c \)
- LIF: Literal Insertion Fault  \( f = ab + c \quad f' = ab + bc \)
- ORF+: Operator Reference Fault  \( f = ab + c \quad f' = abc \)
- ORF*: Operator Reference Fault  \( f = ab + c \quad f' = a + b + c \)

Key idea is that fault classes are related with respect to testing: Test sets guaranteed to detect certain faults are also guaranteed to detect additional faults.
Fault Detection Relationships

- Literal Insertion Fault (LIF)
- Term Omission Fault (TOF)
- Operator Reference Fault (ORF+)
- Literal Reference Fault (LRF)
- Literal Negation Fault (LNF)
- Term Negation Fault (TNF)
- Expression Negation Fault (ENF)
- Literal Omission Fault (LOF)
- Operator Reference Fault (ORF*)
Understanding The Detection Relationships

- Consider the TOF (Term Omission Fault) class
  - UTPC requires a unique true point for every implicant (term)
  - Hence UTPC detects all TOF faults
  - From the diagram, UTPC also detects:
    - All LNF faults (Unique true point for implicant now false)
    - All TNF faults (All true points for implicant are now false points)
    - All ORF+ faults (Unique true points for joined terms now false)
    - All ENF faults (Any single test detects this…)

- Although CUTPNFP does not subsume UTPC, CUTPNFP detects all fault classes that UTPC detects (Converse is false)

- Consider what this says about the notions of subsumption vs. fault detection

- Literature has many more powerful (and more expensive) DNF criteria
  - In particular, possible to detect entire fault hierarchy (MUMCUT)
Karnaugh Maps for Testing Logic Expressions

• Fair Warning
  – We use, rather than present, Karnaugh Maps
  – Newcomer to Karnaugh Maps probably needs a tutorial
    • Suggestion: Google “Karnaugh Map Tutorial”

• Our goal: Apply Karnaugh Maps to concepts used to test logic expressions
  – Identify when a clause determines a predicate
  – Identify the negation of a predicate
  – Identify prime implicants and redundant implicants
  – Identify unique true points
  – Identify unique true point / near false point pairs

• No new material here on testing
  – Just fast shortcuts for concepts already presented
K-Map: A clause determines a predicate

- Consider the predicate: \( f = b + \overline{a} \overline{c} + ac \)
- Suppose we want to identify when \( b \) determines \( f \)
- The dashed line highlights where \( b \) changes value
  - If two cells joined by the dashed line have different values for \( f \), then \( b \) determines \( f \) for those two cells.
  - \( b \) determines \( f \): \( \overline{a} \overline{c} + ac \) (but NOT at \( ac \) or \( \overline{a} \overline{c} \))
- Repeat for clauses \( a \) and \( c \)
K-Map: Negation of a predicate

• Consider the predicate: \( f = ab + bc \)

• Draw the Karnaugh Map for the negation
  – Identify groups
  – Write down negation: \( \bar{f} = \bar{b} + \bar{a} \bar{c} \)
K-Map: Prime and redundant implicants

• Consider the predicate: \( f = abc + abd + abcd + abcd + acd \)
• Draw the Karnaugh Map
• Implicants that are not prime: \( abd, \ abcd, \ abcd, \ acd \)
• Redundant implicant: \( abd \)
• Prime implicants
  – Three: \( ad, bcd, abc \)
  – The last is redundant
  – Minimal DNF representation
    • \( f = ad + bcd \)
K-Map: Unique True Points

- Consider the predicate: \( f = ab + cd \)
- Three unique true points for \( ab \)
  - TTFF, TTFT, TTTT
  - TTTT is a true point, but not a unique true point
- Three unique true points for \( cd \)
  - FFFT, FTTT, TFTT
- Unique true points for \( \overline{f} \)
  \( \overline{f} = \overline{a}c + \overline{b}c + \overline{a}d + \overline{b}d \)
  - FTFT, TFFT, FTTF, TFTF
- Possible UTPC test set
  - \( f \): \{TTFT, FFFT\}
  - \( \overline{f} \): \{FTFT, TFFT, FTTF, TFTF\}
**K-Map: Unique True Point/Near False Point Pairs**

- Consider the predicate: \( f = ab + cd \)

- For implicant \( ab \)
  - For \( a \), choose UTP, NFP pair
    - TTFF, FTFF
  - For \( b \), choose UTP, NFP pair
    - TTFT, TFFT

- For implicant \( cd \)
  - For \( c \), choose UTP, NFP pair
    - FFTT, FFFT
  - For \( d \), choose UTP, NFP pair
    - FFTT, FFTF

- Possible CUTPNFP test set
  - \{TTFF, TTFT, FFTT \ //UTPs
       FTFF, TFFT, FFFT, FFTF\} //NFPs