Chapter 8

Logic Coverage

Date last generated: September 29, 2014.

It is easy to make things hard. It is hard to make things easy.
— Al Chapanis

This chapter uses logical expressions to define criteria and design tests. This continues our progression into the RIPR model by ensuring that tests not only reach certain locations, but the internal state is infected by trying multiple combinations of truth assignments to the expressions. While logic coverage criteria have been known for a long time, their use has been steadily growing in recent years. One cause for their use in practice has been standards such as used by the US Federal Aviation Administration (FAA) for safety critical avionics software in commercial aircraft.

As in Chapter 7, we start with a sound theoretical foundation for logic predicates and clauses with the goal of making the subsequent testing criteria simpler. As before, we take a generic view of the structures and criteria, then discuss how logic expressions can be derived from various software artifacts, including code, specifications, and finite state machines.

This chapter presents two complementary approaches to logic testing. The first, which we call semantic logic coverage, considers what logic expressions mean regardless of how they are formulated. The strength of the semantic approach is that we get the same tests even if the predicate is rewritten in a different but equivalent form. The semantic approach is more common and more likely to be familiar to readers. The second approach, which we call syntactic logic coverage, develops tests specifically tailored to how a logic expression is formulated. The strength of the syntactic approach to logic coverage is that it addresses the specific ways in which an engineer might incorrectly formulate a given logic expression.
Studies have found that the syntactic approach usually detects more faults, but the test criteria are relatively complicated and can be quite expensive. In recent years, the research community has found ways to reduce the number of tests required without sacrificing fault detection. Specifically, the number of tests required for syntactic coverage has dropped substantially—to the point where it is competitive with the semantic approach. While the safety community still relies on the semantic approach, it may be time for this community to consider the syntactic approach.

This chapter presents both approaches, but in such a way that the syntactic approach can be omitted. Section 8.1 presents the semantic approach and Section 8.2 presents the syntactic approach. Subsequent sections show how to apply the semantic approach to artifacts from various parts of the lifecycle. The application of the syntactic approach to these same artifacts is presented in the exercises. The intent is that users of this textbook can cover both approaches, or choose to omit the syntactic approach by skipping Section 8.2 and associated exercises.

Readers who are already familiar with some of the common criteria may have difficulty recognizing them at first. This is because we introduce a generic collection of test criteria, and thus choose names that best help articulate all of the criteria. That is, we are abstracting several existing criteria that are closely related, yet use conflicting terminology. When we deviate, we mention the more traditional terminology and give detailed pointers in the bibliographic notes.

### 8.1 Semantic Logic Coverage Criteria (Active)

Before introducing the semantic logic coverage criteria, we introduce terms and notation. There are no standard terms or notations for these concepts, but they vary in different subfields, books, and papers. We formalize logical expressions in a way that is common in discrete mathematics textbooks.

A *predicate* is an expression that evaluates to a boolean value, and is our topmost structure. A simple example is \(((a > b) \lor C) \land p(x)\). Predicates may contain boolean variables, non-boolean variables that are compared with comparator operators \{\(>, <, =, \geq, \leq, \neq\}\}, and function calls. The internal structure is created by the logical operators:

- \(\neg\)–the *negation* operator
- \(\land\)–the *and* operator
- \(\lor\)–the *or* operator
- \(\rightarrow\)–the *implication* operator
- \(\oplus\)–the *exclusive or* operator
- \(\leftrightarrow\)–the *equivalence* operator
8.1. SEMANTIC LOGIC COVERAGE CRITERIA (ACTIVE)

Some of these operators (→, ⊕, ↔) may seem unusual for readers with a bias toward source code, but they turn out to be common in some specification languages and very handy in our computations. Short circuit versions of the and and or operators are also sometimes useful, and will be addressed when necessary. We adopt a typical precedence, which, from highest to lowest, matches the order listed above. When the order might not be obvious, we use parentheses for clarity.

A clause is a predicate that does not contain any logical operators. For example, the predicate \((a = b) \lor C \land p(x)\) contains three clauses: a relational expression \((a = b)\), a boolean variable \(C\), and the function call \(p(x)\). Because they may contain a structure of their own, relational expressions require special treatment.

A predicate may be written in a variety of logically equivalent ways. For example, the predicate \(((a = b) \lor C) \land ((a = b) \lor p(x))\) is logically equivalent to the predicate given in the previous paragraph, but \(((a = b) \land p(x)) \lor (C \land p(x))\) is not. The rules of boolean algebra (summarized in Section 8.1.5) can be used to convert boolean expressions into equivalent forms.

Logical expressions come from a variety of sources. The most familiar to most readers will probably be source code of a program. For example, the following if statement:

```java
if ((a > b) || C) && (x < y)
    o.m();
else
    o.n();
```

will yield the expression \(((a > b) \lor C) \land (x < y)\). Other sources of logical expressions include transitions in finite state machines. A transition such as `button2 = true (when gear = park)` will yield the expression `gear = park \land button2 = true`. Similarly, a precondition in a specification such as "pre: stack Not full AND object reference parameter not null" will result in a logical expression such as \(\neg stackFull() \land newObj \neq null\).

In the prior material we treat logical expressions according to their semantic meanings, not their syntax. As a consequence, a given logical expression yields the same test requirements for a given coverage criterion no matter which form of the logic expression is used.

8.1.1 Simple Logic Expression Coverage Criteria

Clauses and predicates are used to introduce a variety of coverage criteria. Let \(P\) be a set of predicates and \(C\) be a set of clauses in the predicates in \(P\). For each predicate \(p \in P\), let \(C_p\) be the clauses in \(p\), that is \(C_p = \{c | c \in p\}\). \(C\) is the union of the clauses in each predicate in \(P\), that is \(C = \bigcup_{p \in P} C_p\).

**Criterion 8.1 Predicate Coverage (PC):** For each \(p \in P\), \(TR\) contains two requirements: \(p\) evaluates to true, and \(p\) evaluates to false.
Predicate coverage is also known as *decision coverage*. The graph version of Predicate Coverage was introduced in Chapter 7 as Edge Coverage; this is where the graph coverage criteria overlap the logic expression coverage criteria. For control flow graphs where \( P \) is the set of predicates associated with branches, Predicate Coverage and Edge Coverage are the same. For the predicate given above, \(((a > b) \lor C) \land p(x)\), two tests that satisfy Predicate Coverage are \((a = 5, \ b = 4, \ C = true, \ p(x) = true)\) and \((a = 5, \ b = 6, \ C = false, \ p(x) = false)\).

An obvious failing of this criterion is that the individual clauses are not always exercised. Predicate coverage for the above clause could also be satisfied with the two tests \((a = 5, \ b = 4, \ C = true, \ p(x) = true)\) and \((a = 5, \ b = 4, \ C = true, \ p(x) = false)\), in which the first two clauses never have the value *false*. To rectify this problem, we move to the clause level.

**Criterion 8.2 Clause Coverage (CC):** For each \(c \in C\), TR contains two requirements: \(c\) evaluates to *true*, and \(c\) evaluates to *false*.

Clause coverage is also known as *condition coverage*. Our predicate \(((a > b) \lor C) \land p(x)\) requires different values to satisfy CC. Clause Coverage requires that \((a > b) = true\) and \(false, \ C = true\) and \(false\), and \(p(x) = true\) and \(false\). These requirements can be satisfied with two tests: \(((a = 5, \ b = 4), (C = true), p(x) = true)\) and \(((a = 5, \ b = 6), (C = false), p(x) = false)\).

Clause Coverage does not subsume Predicate Coverage, and Predicate Coverage does not subsume Clause Coverage, as we show with the predicate \(p = a \lor b\). The clauses \(C\) are \{\(a, \ b\}\). The four test inputs that enumerate the combinations of logical values for the clauses:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a ∨ b</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>F</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Consider two test sets, each with a pair of test inputs. Test set \(T_{23} = \{2, \ 3\}\) satisfies Clause Coverage, but not Predicate Coverage, because \(p\) is never false. Conversely, test set \(T_{24} = \{2, \ 4\}\) satisfies Predicate Coverage, but not Clause Coverage, because \(b\) is never true. These two test sets demonstrate that neither Predicate Coverage nor Clause Coverage subsumes the other.

From the testing perspective, we would certainly like a coverage criterion that tests individual clauses and that also tests the predicate. The most direct approach to rectify this problem is to try all combinations of clauses:

**Criterion 8.3 Combinatorial Coverage (CoC):** For each \(p \in P\), TR has test requirements for the clauses in \(C_p\) to evaluate to each possible combination of truth values.
Combinatorial Coverage has also been called *multiple condition coverage*. For the predicate \((a \lor b) \land c\), the complete truth table contains eight rows:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>((a \lor b) \land c)</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>F</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 predicate \(p\) with \(n\) independent clauses has \(2^n\) possible assignments of truth values. Thus Combinatorial Coverage is unwieldy at best, and impractical for predicates with more than a few clauses. What we need are criteria that capture the effect of each clause, but do so in a reasonable number of tests. These observations lead, after some thought\(^1\), to a powerful collection of test criteria that are based on the notion of making individual clauses “active” as defined in the next subsection. Specifically, we check to see that if we vary a clause in a situation where the clause should affect the predicate, then, in fact, the clause does affect the predicate. Later we turn to the complementary problem of checking to see that if we vary a clause in a situation where it should *not* affect the predicate, then it, in fact, does not affect the predicate.

### 8.1.2 Active Clause Coverage

The lack of subsumption between Clause and Predicate Coverage is unfortunate, but Clause and Predicate Coverage have deeper problems. Specifically, when we introduce tests at the clause level, we want also to have an effect on the predicate. The key notion is that of **determination**, the conditions under which a clause influences the outcome of a predicate. Although the formal definition is a bit messy, the basic idea is simple: if you flip the clause, and the predicate changes value, then the clause determines the predicate. To distinguish the clause in which we are interested from the remaining clauses, we adopt the following convention. The *major* clause, \(c_i\), is the clause on which we are focusing. All of the other clauses \(c_j, j \neq i\), are *minor* clauses. Typically, to satisfy a given criterion, each clause is treated in turn as a major clause. Formally:

**Definition 8.1 Determination:** \(\text{Given a major clause } c_i \text{ in predicate } p, \text{ we say that } c_i \text{ determines } p \text{ if the minor clauses } c_j \in p, j \neq i \text{ have values so that changing the truth value of } c_i \text{ changes the truth value of } p.\)

\(^1\)In practice, this “thought” turned out to be the collective effort of many researchers, who published dozens of papers over a period of several decades.
Note that this definition explicitly does not require that $c_i = p$. This issue has been left ambiguous by previous definitions, some of which require the predicate and the major clause to have the same value. This interpretation is not practical. When the negation operator is used, for example, if the predicate is $p = \neg a$, it becomes impossible for the major clause and the predicate to have the same value.

Consider the example where $p = a \lor b$. If $b$ is false, then clause $a$ determines $p$, because the value of $p$ is exactly the value of $a$. However if $b$ is true, then $a$ does not determine $p$, since $p$ is true regardless of the value of $a$.

From the testing perspective, we would like to test each clause under circumstances where the clause determines the predicate. Consider this as putting different members of a team in charge of the team. We do not know if they can be effective leaders until they try. Consider again the predicate $p = a \lor b$. If we do not vary $b$ under circumstances where $b$ determines $p$, then we have no evidence that $b$ is used correctly. For example, test set $T = \{TT, FF\}$, which satisfies both Clause and Predicate Coverage, tests neither $a$ nor $b$ effectively.

In terms of criteria, we develop the notion of active clause coverage in a general way first with the definition below, and then refine out the ambiguities in the definition to arrive at the resulting formal coverage criteria. This treats active clause coverage as a framework that generalizes several similar criteria, including the several variations of MCDC.

**Definition 8.2 Active Clause Coverage (ACC):** 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.

For example, for $p = a \lor b$, we end up with a total of four requirements in $TR$, two for clause $a$ and two for clause $b$. For clause $a$, $a$ determines $p$ if and only if $b$ is false. So we have the two test requirements $\{(a = true, b = false), (a = false, b = false)\}$. For clause $b$, $b$ determines $p$ if and only if $a$ is false. So we have the two test requirements $\{(a = false, b = true), (a = false, b = false)\}$. This is summarized in the partial truth table below (the values for the major clauses are in bold face).

<table>
<thead>
<tr>
<th>$c_i = a$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$T$</td>
<td>$f$</td>
</tr>
<tr>
<td>$F$</td>
<td>$F$</td>
<td>$f$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$c_i = b$</th>
<th>$f$</th>
<th>$T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f$</td>
<td>$f$</td>
<td>$F$</td>
</tr>
</tbody>
</table>

Two of these requirements are identical, so we end up with three distinct test requirements for Active Clause Coverage for the predicate $a \lor b$, namely $\{(a = true, b = false), (a = false, b = true), (a = false, b = false)\}$. Such overlap is common; a predicate with $n$ clauses needs at least $n$ tests, but no more than $2n$ tests, to satisfy Active Clause Coverage.

ACC is almost identical to the way early papers described another technique called MCDC. It turns out that this criterion has some ambiguity, which has led to a fair amount
of confusion about how to interpret MCDC over the years. The most important question is whether the minor clauses $c_j$ need to have the same values when the major clause $c_i$ is true as when $c_i$ is false. Resolving this ambiguity leads to three distinct and interesting flavors of Active Clause Coverage. For a simple predicate such as $p = a \lor b$, the three flavors turn out to be identical, but differences appear for more complex predicates. The most general flavor allows the minor clauses to have different values.

**Criterion 8.4 General Active Clause Coverage (GACC):** For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses $c_j$, $j \neq i$ 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.

Unfortunately, it turns out that General Active Clause Coverage does not subsume Predicate Coverage, as the following example shows.

Consider the predicate $p = a \leftrightarrow b$. Clause $a$ determines $p$ for any assignment of truth values to $b$. So, when $a$ is true, we choose $b$ to be true as well, and when $a$ is false, we choose $b$ to be false as well. We make the same selections for clause $b$. We end up with only two test inputs: $\{TT, FF\}$. $p$ evaluates to true for both of these cases, so Predicate Coverage is not achieved. GACC also does not subsume PC when an exclusive or operator is used. We save that example for an exercise.

Many testing researchers have a strong feeling that ACC should subsume PC, thus the second flavor of ACC requires that $p$ evaluates to true for one assignment of values to the major clause $c_i$, and false for the other. Note that $c_i$ and $p$ do not have to have the same values, as discussed with the definition for determination.

**Criterion 8.5 Correlated Active Clause Coverage (CACC):** 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.

So for the predicate $p = a \leftrightarrow b$ above, CACC can be satisfied with respect to clause $a$ with the test set $\{TT, FT\}$ and with respect to clause $b$ with the test set $\{TT, TF\}$. Merging these yields the CACC test set $\{TT, TF, FT\}$.

Consider the example $p = a \land (b \lor c)$. For $a$ to determine the value of $p$, the expression $b \lor c$ must be true. This can be achieved in three ways: $b$ true and $c$ false, $b$ false and $c$ true, and both $b$ and $c$ true. So, it would be possible to satisfy Correlated Active Clause Coverage with respect to clause $a$ with the two test inputs: $\{TT, FFT\}$. Other choices are possible with respect to $a$. The following truth table helps enumerate them. The row numbers are taken from the complete truth table for the predicate given previously. Specifically, CACC
can be satisfied for \( a \) by choosing one test requirement from rows 1, 2 and 3, and the second from rows 5, 6 and 7. Of course, nine possible ways exist to do this.

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

The final flavor forces the non-major clauses \( c_j \) to be identical for both assignments of truth values to the major clause \( c_i \).

**Criterion 8.6 Restricted Active Clause Coverage (RACC):** 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.

Note that the definition for RACC does not explicitly say that the value of the predicate has to be different for each value of \( c_i \), even though the definition for CACC did. It is true that the RACC tests will cause the predicate to be different for each value of the major clause, however this is a direct consequence of the definition of determination. That is, if you change the value of a major clause \( a \) under conditions where \( P_a \) is true, and you leave the minor clauses the same, this must change the value of the predicate.

For the example \( p = a \land (b \lor c) \), only three of the nine sets of test requirements that satisfy Correlated Active Clause Coverage with respect to clause \( a \) will satisfy Restricted Active Clause Coverage with respect to clause \( a \). In terms of the previously given complete truth table, row 2 can be paired with row 6, row 3 with row 7, or row 1 with row 5. Thus, instead of the nine ways to satisfy CACC, only three can satisfy RACC.

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

**CACC versus RACC**
Examples of satisfying a predicate for each of these three criteria are given later. One point that may not be immediately obvious is how CACC and RACC differ in practice.
It turns out that some logical expressions can be completely satisfied under CACC, but have infeasible test requirements under RACC. These expressions are a little subtle and only exist if dependency relationships exist among the clauses, that is, some combinations of values for the clauses are prohibited. Since this often happens in real programs, because program variables frequently depend upon one another, we introduce the following example.

Consider a system with a valve that might be either open or closed, and several modes, two of which are “Operational” and “Standby.” Assume the following two constraints:

1. The valve must be open in “Operational” and closed in all other modes.
2. The mode cannot be both “Operational” and “Standby” at the same time.

This leads to the following clause definitions:

\[ a = \text{"The valve is closed"} \]
\[ b = \text{"The system status is Operational"} \]
\[ c = \text{"The system status is Standby"} \]

Suppose that a certain action can be taken only if the valve is closed and the system status is either in Operational or Standby. That is:

\[ p = \text{valve is closed AND (system status is Operational OR system status is Standby)} \]
\[ = a \land (b \lor c) \]

This is exactly the predicate that was analyzed above. The constraints above can be formalized as:

\[ 1 \quad \neg a \leftrightarrow b \]
\[ 2 \quad \neg (b \land c) \]

These constraints limit the feasible values in the truth table. As a reminder, the complete truth table for this predicate is:

<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>
Recall that for \( a \) to determine the value of \( P \), either \( b \) or \( c \) or both must be true. Constraint 1 rules out the rows where \( a \) and \( b \) have the same values, that is, rows 1, 2, 7, and 8. Constraint 2 rules out the rows where \( b \) and \( c \) are both true, that is, rows 1 and 5. Thus, the only feasible rows are 3, 4, and 6. Recall that CACC can be satisfied by choosing one from rows 1, 2 or 3 and one from rows 5, 6 or 7. But RACC requires one of the pairs 2 and 6, 3 and 7, or 1 and 5. Thus, RACC is infeasible for \( a \) in this predicate.

8.1.3 Inactive Clause Coverage

The Active Clause Coverage Criteria in Section 8.1.2 focus on making sure the major clauses do affect their predicates. Inactive Clause Coverage ensures that changing a major clause that should not affect the predicate does not, in fact, affect the predicate.

**Definition 8.3 Inactive Clause Coverage (ICC):** 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 \( c_i \) under these circumstances: (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.

Although Inactive Clause Coverage (ICC) has some of the same ambiguity as ACC, only two distinct flavors can be defined, namely General Inactive Clause Coverage (GICC) and Restricted Inactive Clause Coverage (RICC). The notion of correlation is not relevant for Inactive Clause Coverage because \( c_i \) cannot correlate with \( p \) since \( c_i \) does not determine \( p \). Also, Predicate Coverage is guaranteed, subject to feasibility, in all flavors due to the structure of the definition.

The following example illustrates the value of the inactive clause coverage criteria. Suppose you are testing the control software for a shutdown system in a reactor, and the specification states that the status of a particular valve (open vs. closed) is relevant to the reset operation in Normal mode, but not in Override mode. That is, the reset should perform identically in Override mode when the valve is open and when the valve is closed. The skeptical test engineer will want to test reset in Override mode for both positions of the valve, since a reasonable implementation mistake would be to take into account the setting of the valve in all modes.

The formal versions of GICC and RICC are as follows.

**Criterion 8.7 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 \( p \). TR has four requirements for \( c_i \) under these circumstances: (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. The values chosen for the minor clauses \( c_j \) may vary amongst the four cases.
8.1. **SEMANTIC LOGIC COVERAGE CRITERIA (ACTIVE)**

**Criterion 8.8 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 \( p \). TR has four requirements for \( c_i \) under these circumstances: (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. The values chosen for the minor clauses \( c_j \) must be the same in cases (1) and (2), and the values chosen for the minor clauses \( c_j \) must also be the same in cases (3) and (4).

8.1.4 **Infeasibility and Subsumption**

A variety of technical issues complicate the Active Clause Coverage criteria. As with many criteria, the most vexing is the issue of infeasibility. Infeasibility is often a problem because clauses are sometimes related to one another. That is, choosing the truth value for one clause may affect the truth value for another clause. Consider, for example, a common loop structure, which assumes short circuit semantics:

```plaintext
while (i < n && a[i] != 0) {do something to a[i]}
```

The idea here is to avoid evaluating \( a[i] \) if \( i \) is out of range, and short circuit evaluation is not only assumed, but depended on. Clearly, it is not going to be possible to run a test case where \( i < n \) is false and \( a[i] \neq 0 \) is true.

In principle, the issue of infeasibility for clause and predicate criteria is no different from that for graph criteria. In both cases, the solution is to satisfy test requirements that are feasible, and then decide how to treat infeasible test requirements. The simplest solution is to simply ignore infeasible requirements, which usually does not affect the quality of the tests. The difficulty here is in knowing whether a test requirement is truly infeasible or simply hard to satisfy. Theoretically, recognizing infeasibility is a formally undecidable problem.

However, a better solution for some infeasible test requirements is to consider the counterparts of the requirements in a subsumed coverage criterion. For example, if RACC coverage with respect to clause \( a \) in predicate \( p \) is infeasible (due to additional constraints between the clauses), but CACC coverage is feasible, then it makes sense to replace the infeasible RACC test requirements with the feasible CACC test requirements. This approach is similar to that of Best Effort Touring developed in the graph coverage chapter.

Figure 8.1 shows the subsumption relationships among the logic expression criteria. Note that the Inactive Clause Coverage criteria do not subsume any of the Active Clause Coverage criteria, and vice versa. The diagram assumes that infeasible test requirements are treated on a best effort basis, as explained above. Where such an approach does not result in feasible test requirements, the diagram assumes that the infeasible test requirements are ignored.
The next question we address is how to find values for the minor clauses $c_j$ to ensure the major clause $c_i$ determines the value of $p$. A variety of approaches solve this problem effectively. We suggest that each student adopt an approach that resonates well with her mathematical background and experience. We give a direct definitional method here, first using a mathematical approach, then using a simplified tabular shortcut. The bibliographic notes give pointers to all methods of which the authors are aware.

**A Direct Definitional Method for Determination**

For a predicate $p$ with clause (or boolean variable) $c$, let $p_{c=true}$ represent the predicate $p$ with every occurrence of $c$ replaced by $true$ and $p_{c=false}$ be the predicate $p$ with every occurrence of $c$ replaced by $false$. For the rest of this development, we assume no duplicates (that is, $p$ contains only one occurrence of $c$). Note that neither $p_{c=true}$ nor $p_{c=false}$ contains any occurrences of the clause $c$. Now we connect the two expressions with an exclusive or:

$$p_c = p_{c=true} \oplus p_{c=false}$$

It turns out that $p_c$ describes the exact conditions under which the value of $c$ determines that of $p$. That is, if values for the clauses in $p_c$ are chosen so that $p_c$ is true, then the truth value of $c$ determines the truth value of $p$. If the clauses in $p_c$ are chosen so that $p_c$ evaluates
to false, then the truth value of $p$ is independent of the truth value of $c$. This is exactly what we need to implement the various flavors of Active and Inactive Clause Coverage.

As a first example, we try $p = a \lor b$. $p_a$ is, by definition:

$$p_a = p_{a=true} \oplus p_{a=false}$$
$$= (true \lor b) \oplus (false \lor b)$$
$$= true \oplus b$$
$$= \neg b$$

That is, for the major clause $a$ to determine the predicate $p$, the only minor clause $b$ must be false. This should make sense intuitively, since the value of $a$ will have an effect on the value of $p$ only if $b$ is false. By symmetry, it is clear that $p_b$ is $\neg a$.

If we change the predicate to $p = a \land b$, we get

$$p_a = p_{a=true} \oplus p_{a=false}$$
$$= (true \land b) \oplus (false \land b)$$
$$= b \oplus false$$
$$= b$$

That is, we need $b = true$ to make $a$ determine $p$. By a similar analysis, $p_b = a$.

The equivalence operator is a little less obvious and brings up an interesting point. Consider $p = a \leftrightarrow b$.

$$p_a = p_{a=true} \oplus p_{a=false}$$
$$= (true \leftrightarrow b) \oplus (false \leftrightarrow b)$$
$$= b \oplus \neg b$$
$$= true$$

That is, for any value of $b$, $a$ determines the value of $p$ without regard to the value for $b$! This means that for a predicate $p$, such as this one, where the value of $p_c$ is the constant true, the Inactive Clause Criteria are infeasible with respect to $c$. Inactive Clause Coverage is likely to result in infeasible test requirements when applied to expressions that use the equivalence or exclusive-or operators.

A more general version of this conclusion can be drawn that applies to the Active Clause Coverage criteria as well. If a predicate $p$ contains a clause $c$ such that $p_c$ evaluates to the constant false, the Active Clause Coverage criteria are infeasible with respect to $c$. The ultimate reason is that the clause in question is redundant; the predicate can be rewritten without it. While this may sound like a theoretical curiosity, it is actually a very useful result for testers. If a predicate contains a redundant clause, that is a very strong signal that something is wrong with the predicate!

Consider $p = a \land b \lor a \land \neg b$. This is really just the predicate $p = a$; $b$ is irrelevant. Computing $p_b$, we get

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

so it is impossible for \( b \) to determine \( p \).

We need to consider how to make clauses determine predicates for a couple of more complicated expressions. For the expression \( p = a \land (b \lor c) \), we get

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

This example ends with an undetermined answer, which points out the key difference between CACC and RACC. Three choices of values make \( b \lor c \) true, \((b = c = \text{true})\), \((b = \text{true}, c = \text{false})\), and \((b = \text{false}, c = \text{true})\). For Correlated Active Clause Coverage, we could pick one pair of values when \( a \) is true and another when \( a \) is false. For Restricted Active Clause Coverage, we must choose the same pair for both values of \( a \).

The derivation for \( b \) and equivalently for \( c \) is slightly more complicated:

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

The last step in the simplification shown above may not be immediately obvious. If it is not, try constructing the truth table for \( a \oplus (a \land c) \). The computation for \( p_c \) is equivalent and yields the solution \( a \land \lnot b \).
### Side Bar

**Boolean Algebra Laws**

You might have learned logic a long time ago. While a software tester does not need to be an expert logician, it sometimes helps to have a “toolbox” of Boolean algebra laws to help reduce predicates during determination. For that matter, the Boolean laws can help simplify predicates during design and development. The following summarizes some of the most useful laws. They are taken from standard logic and discrete mathematics textbooks.

Some books will use ‘+’ for “or” (our $\lor$) and a dot (‘.’) or multiplication symbol (‘*’) for “and” (our $\land$). It is often common to imply “and” by simply placing the two symbols adjacent to each other, that is, $a \land b$ can be written as $ab$.

In the following, $a$ and $b$ are boolean. The precedence from high to low is $\land$, $\lor$, $\lor$.

- **Negation Laws**
  
  \[
  \neg(\neg a) = a \\
  \neg a \lor a = \text{true} \\
  \neg a \land a = \text{false} \\
  a \lor \neg a \land b = a \lor b
  \]

- **AND Identity Laws**
  
  \[
  \text{false} \land a = \text{false} \\
  \text{true} \land a = a \\
  a \land a = a \\
  a \land \neg a = \text{false}
  \]

- **OR Identity Laws**
  
  \[
  \text{false} \lor a = a \\
  \text{true} \lor a = true \\
  a \lor a = a \\
  a \lor \neg a = true
  \]

- **XOR Identity Laws**
  
  \[
  \text{false} \oplus a = a \\
  \text{true} \oplus a = \neg a \\
  a \oplus a = \text{false} \\
  a \oplus \neg a = true
  \]

- **XOR Equivalence Laws**
  
  \[
  a \oplus b = (a \land \neg b) \lor (\neg a \land b) \\
  a \oplus b = (a \lor b) \land (\neg a \lor \neg b) \\
  a \oplus b = (a \lor b) \land \neg(a \land b)
  \]
Side Bar (part 2)

Boolean Algebra Laws continued

- **Commutativity Laws**
  \[ a \lor b = b \lor a \]
  \[ a \land b = b \land a \]
  \[ a \oplus b = b \oplus a \]

- **Associativity Laws**
  \[ (a \lor b) \lor c = a \lor (b \lor c) \]
  \[ (a \land b) \land c = a \land (b \land c) \]
  \[ (a \oplus b) \oplus c = a \oplus (b \oplus c) \]

- **Distributive Laws**
  \[ a \land (b \lor c) = (a \land b) \lor (a \land c) \]
  \[ a \lor (b \land c) = (a \lor b) \land (a \lor c) \]

- **DeMorgan’s Laws**
  \[ \neg(a \lor b) = \neg a \land \neg b \]
  \[ \neg(a \land b) = \neg a \lor \neg b \]

### A Tabular Shortcut for Determination

The previous method to find the values for minor clauses to make a major clause determine the value of a predicate is a general method that works in all cases for all predicates. However, the math can be challenging to some, so we present a simple shortcut.

This is done using a truth table. First, we draw the complete truth table for a predicate, including a column for the predicate result. Then for each pair of rows where the minor clauses have identical values, but the major clause differs, we check whether the predicate results are different. If they are, those two rows cause the major clause to determine the value of the predicate. This technique, in effect, shortcuts the above computation in a tabular form.

As an example, consider the predicate \( p = a \land (b \lor c) \). The complete truth table contains eight rows.

<table>
<thead>
<tr>
<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>
</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>T</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Now we add columns for each of $p_a$, $p_b$, and $p_c$. Under $p_a$, we note that when $b$ is true and $c$ is true (rows 1 and 5, where $b$ and $c$ have identical values), the predicate is true when $a$ is true but false when $a$ is false. Thus, TTT and FTT cause $a$ to determine the value of $p$. The same is true when $b$ is true and $c$ is false (rows 2 and 6) and when $b$ is false and $c$ is true (rows 2 and 6). However, when both $b$ and $c$ are false (rows 4 and 8), $p$ is false, so those two rows do not cause $a$ to determine the value of $p$. Thus, $a$ determines the value of $p$ when either $b$ is true or $c$ is true, or both. Mathematically, $p_a = b \lor c$, which matches what we showed in the previous subsection.

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

The determinations for $p_b$ and $p_c$ are similar, although fewer rows allow them to determine the value of the predicate. For $b$, rows 2 (TTF) and 4 (TFF) have different values for $p$. However, for the other pairs of rows where $a$ and $c$ are identical (rows 1 and 3, rows 5 and 7, and rows 6 and 8), the value of $p$ is the same, so they do not allow $b$ to determine the value of the predicate. Likewise, rows 3 (TFT) and 4 (TFF) allow $c$ to determine the value of the predicate. Thus, $b$ determines the value of $p$ when $a$ is true and $c$ is false, and $c$ determines the value of $p$ when $a$ is true and $b$ is false.

The tabular approach allows direct calculation of RACC, CACC, and GACC. RACC, CACC, and GACC are the same for clauses $b$ and $c$, because only one pair of rows allow them to determine the value of $p$. For $a$, GACC pairs are the cross product of tests where $a$ is true and $p_a$ is true, namely rows $\{1, 2, 3\}$, and tests where $a$ is false and $p_a$ is true, namely, rows $\{5, 6, 7\}$. This cross product yields nine pairs. CACC, which adds the requirement of different truth values for $p$, is simply the subset of GACC where the predicate differs: for this predicate, it is still all nine pairs for $a$. RACC pairs for $a$ requires “matching rows,” that is, rows 1 and 5, 2 and 6, and 3 and 7, a total of three pairs. The tabular approach is used in the web tool on the book website.

### 8.1.6 Finding Satisfying Values

The final step in applying the logic coverage criteria is to choose values that satisfy the criteria. This section shows how to generate values for one example; more cases are explored in the exercises and the application sections later in the chapter. The example is from Section 8.1.1:
\[ p = (a \lor b) \land c \]

Finding values for **Predicate Coverage** is easy and was already shown in Section 8.1.1. Two test requirements are:

\[ TR_{PC} = \{ p = true, p = false \} \]

and they can be satisfied with the following values for the clauses:

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>f</td>
</tr>
</tbody>
</table>

To run the test cases, we need to refine these truth assignments to create values for clauses \( a \), \( b \), and \( c \). Suppose that clauses \( a \), \( b \), and \( c \) were defined in terms of Java program variables as follows:

- **a** \( x < y \), a relational expression for program variables \( x \) and \( y \)
- **b** \( done \), a primitive boolean value
- **c** \( \text{list.contains(str)} \), for \( \text{List} \) and \( \text{String} \) objects

Thus, the complete expanded predicate is actually:

\[ p = (x < y \lor done) \land \text{list.contains(str)} \]

Then the following values for the program variables satisfy the test requirements for Predicate Coverage.

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x=3 ) ( y=5 )</td>
<td>( \text{done} = \text{true} )</td>
<td>( \text{list} = [\text{&quot;Rat&quot;}, \text{&quot;Cat&quot;}, \text{&quot;Dog&quot;}] ) ( \text{str} = \text{&quot;Cat&quot;} )</td>
</tr>
<tr>
<td>( x=0 ) ( y=7 )</td>
<td>( \text{done} = \text{true} )</td>
<td>( \text{list} = [\text{&quot;Red&quot;}, \text{&quot;White&quot;}] ) ( \text{str} = \text{&quot;Blue&quot;} )</td>
</tr>
</tbody>
</table>

Note that the values for the program variables need not be the same as another test if the goal is to set a clause to a particular value. For example, clause \( a \) is true in both tests, even though program variables \( x \) and \( y \) have different values.

Values to satisfy **Clause Coverage** were also shown in Section 8.1.1. The test requirements are:

\[ TR_{CC} = \{ a = true, a = false, b = true, b = false, c = true, c = false \} \]

and they can be satisfied with the following values for the clauses (blank cells represent “don’t-care” values):
8.1. SEMANTIC LOGIC COVERAGE CRITERIA (ACTIVE)

Refining the truth assignments to create values for program variables \( x, y, done, list, \) and \( str \) is left as an exercise for the reader.

Before proceeding with the other criteria, we first choose values for minor clauses to ensure that the major clauses will determine the value of \( p \). We gave a method of calculating \( p_a, p_b, \) and \( p_c \) earlier. The computations for this particular predicate \( p \) are left as an exercise. However, the results are:

<table>
<thead>
<tr>
<th></th>
<th>( p_a )</th>
<th>( p_b )</th>
<th>( p_c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = true )</td>
<td>( \neg b \land c )</td>
<td>( \neg a \land c )</td>
<td>( a \lor b )</td>
</tr>
<tr>
<td>( a = false )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b = true )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b = false )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( c = true )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( c = false )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now we can turn to the other clause coverage criteria. The first is Combinatorial Coverage, requiring all combinations of values for the clauses. In this case, we have eight test requirements, which can be satisfied with the following values:

<table>
<thead>
<tr>
<th></th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( (a \lor b) \land 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>( f )</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>( t )</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>

Recall that General Active Clause Coverage requires that each major clause be true and false and the minor clauses be such that the major clause determines the value of the predicate. Similarly to Clause Coverage, three pairs of test requirements can be defined:

\[
TR_{GACC} = \{(a = true \land p_a, a = false \land p_a), (b = true \land p_b, b = false \land p_b), (c = true \land p_c, c = false \land p_c)\}
\]

The test requirements can be satisfied with the following values for the clauses. Note that these can be the same as with Clause Coverage with the exception that the blank cells from Clause Coverage are replaced with the values from the determination analysis. In the following (partial truth) table, values for major clauses are indicated with upper case letters.
in boldface.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = true ∧ p&lt;sub&gt;a&lt;/sub&gt;</td>
<td>T</td>
<td>f</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>a = false ∧ p&lt;sub&gt;a&lt;/sub&gt;</td>
<td>F</td>
<td>f</td>
<td>t</td>
<td>f</td>
</tr>
<tr>
<td>b = true ∧ p&lt;sub&gt;b&lt;/sub&gt;</td>
<td>f</td>
<td>T</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>b = false ∧ p&lt;sub&gt;b&lt;/sub&gt;</td>
<td>f</td>
<td>F</td>
<td>t</td>
<td>f</td>
</tr>
<tr>
<td>c = true ∧ p&lt;sub&gt;c&lt;/sub&gt;</td>
<td>t</td>
<td>f</td>
<td>T</td>
<td>t</td>
</tr>
<tr>
<td>c = false ∧ p&lt;sub&gt;c&lt;/sub&gt;</td>
<td>f</td>
<td>t</td>
<td>F</td>
<td>f</td>
</tr>
</tbody>
</table>

Note the duplication; the first and fifth rows are identical, and the second and fourth are identical. Thus, only four tests are needed to satisfy GACC.

A different way of looking at GACC considers all of the possible pairs of test inputs for each pair of test requirements. Recall that the active clause coverage criteria always generate test requirements in pairs, with one pair generated for each clause in the predicate under test. To identify these test inputs, we will use the row numbers from the truth table. Hence, the pair (3, 7) represents the first two tests listed in the table above.

It turns out that (3, 7) is the only pair that satisfies the GACC test requirements with respect to clause a (when a is major), and (5, 7) is the only pair that satisfies the GACC test requirements with respect to clause b. For clause c, the situation is more interesting. Nine pairs satisfy the GACC test requirements for clause c, namely

\{(1, 2), (1, 4), (1, 6), (3, 2), (3, 4), (3, 6), (5, 2), (5, 4), (5, 6)\}

Recall that Correlated Active Clause Coverage requires that each major clause be true and false, the minor clauses be such that the major clause determines the value of the predicate, and the predicate must have both the value true and false. As with GACC, three pairs of test requirements can be defined: For clause a, the pair of test requirements is:

\[ a = true \land p_a \land p = x \]
\[ a = false \land p_a \land p = \neg x \]

where \( x \) may be either true or false. The point is that \( p \) must have a different truth value in the two test cases. We leave the reader to write out the corresponding CACC test requirements with respect to \( b \) and \( c \).

For our example predicate \( p \), a careful examination of the pairs of test cases for GACC reveals that \( p \) takes on both truth values in each pair. Hence, GACC and CACC are the same for predicate \( p \), and the same pairs of test inputs apply. In the exercises the reader will find predicates where a test pair that satisfies GACC with respect to some clause \( c \) turns out not to satisfy CACC with respect to \( c \).

The situation for RACC is quite different, however, in the example \( p \). Recall that Restricted Active Clause Coverage is the same as CACC except that it requires the values
for the minor clauses $c_j$ to be identical for both assignments of truth values to the major clause, $c_i$. For clause $a$, the pair of test requirements that RACC generates is:

$$a = \text{true} \land p_a \land b = B \land c = C$$

$$a = \text{false} \land p_a \land b = B \land c = C$$

for some boolean constants $B$ and $C$. An examination of the pairs given above for GACC reveals that with respect to clauses $a$ and $b$, the pairs are the same. So pair $(3, 7)$ satisfies RACC with respect to clause $a$ and pair $(5, 7)$ satisfies RACC with respect to $b$. However, with respect to $c$, only three of the pairs satisfy RACC, namely,

$$\{(1, 2), (3, 4), (5, 6)\}$$

This example does leave one question about the different flavors of the Active Clause Coverage criteria, namely, what is the practical difference among them? That is, beyond the subtle difference in the arithmetic, how do they affect practical testers? The real differences do not show up very often, but when they do they can be dramatic and quite annoying.

GACC does not require that Predicate Coverage be satisfied on the pair of tests for each clause, so use of that flavor may mean we do not test our program as thoroughly as we might like. In practical use, it is easy to construct examples where GACC is satisfied but Predicate Coverage is not when the predicates are very small (one or two terms), but difficult with three or more terms, since for one of the clauses, it is likely that the chosen GACC tests will also be CACC tests.

The restrictive nature of RACC, on the other hand, can sometimes make it hard to satisfy the criterion. This is particularly true when some combinations of clause values are infeasible. Assume that in the predicate used above, the semantics of the program effectively eliminate rows 2, 3, and 6 from the truth table. Then RACC cannot be satisfied with respect to clause $\text{list.contains(str)}$ (that is, we have infeasible test requirements), but CACC can. Additionally, we have no evidence that RACC gives more or better tests. Wise readers, (that is, if still awake) will by now realize that Correlated Active Clause Coverage is often the most practical flavor of ACC.

---

**Exercises, Section 8.1.**

1. List all the clauses for the predicate below:
   $$((f \leq g) \land (X > 0)) \lor (M \land (e < d + c))$$

2. Write the predicate (only the predicate) to represent the requirement: “List all the wireless mice that either retail for more than $100 or for which the store has more than 20 items. Also list non-wireless mice that retail for more than $50.”
3. Use predicates (i) through (x) to answer the following questions. Verify your computations with the logic coverage tool on the book website.

i. \( p = a \land (\neg b \lor c) \)

ii. \( p = a \lor (b \land c) \)

iii. \( p = a \land b \)

iv. \( p = a \rightarrow (b \rightarrow c) \)

v. \( p = a \oplus b \)

vi. \( p = a \leftrightarrow (b \land c) \)

vii. \( p = (a \lor b) \land (c \lor d) \)

viii. \( p = (\neg a \land \neg b) \lor (a \land \neg c) \lor (\neg a \land c) \)

ix. \( p = a \lor b \lor (c \land d) \)

x. \( p = (a \land b) \lor (b \land c) \lor (a \land c) \)

(a) Identify the clauses that go with predicate \( p \).

(b) Compute (and simplify) the conditions under which each the clause determines predicate \( p \).

(c) Write the complete truth table for all clauses. Label your rows starting from 1. Use the format in the example underneath the definition of Combinatorial Coverage in Section 8.1.1. That is, row 1 should be all clauses true. You should include columns for the conditions under which each clause determines the predicate, and also a column for the predicate itself.

(d) Identify all pairs of rows from your table that satisfy General Active Clause Coverage (GACC) with respect to each clause.

(e) Identify all pairs of rows from your table that satisfy Correlated Active Clause Coverage (CACC) with respect to each clause.

(f) Identify all pairs of rows from your table that satisfy Restricted Active Clause Coverage (RACC) with respect to each clause.

(g) Identify all 4-tuples of rows from your table that satisfy General Inactive Clause Coverage (GICC) with respect to each clause. Identify any infeasible GICC test requirements.

(h) Identify all 4-tuples of rows from your table that satisfy Restricted Inactive Clause Coverage (RICC) with respect to each clause. Identify any infeasible RICC test requirements.

4. Show that GACC does not subsume PC when the exclusive or operator is used. Assume \( p = a \oplus b \).

5. In Section 8.1.6, we introduced the example \( p = (a \lor b) \land c \), and provided expanded versions of the clauses using program variables. We then gave specific values to satisfy PC. We also gave truth values to satisfy CC. Find values for the program variables given to satisfy CC, that is, refine the abstract tests into concrete test values.

6. Refine the GACC, CACC, RACC, GICC, and RICC coverage criteria so that the constraints on the minor clauses are made more formal.
7. (Challenging!) Find a predicate and a set of additional constraints so that CACC is infeasible with respect to some clause, but GACC is feasible.
8.2 Syntactic Logic Coverage Criteria (DNF)

The semantic logic coverage criteria (active) apply to logic predicates, no matter how they are written. This approach has the advantage of testing the software’s logic irrespective of the way the predicates are written, but this same advantage has the disadvantage of sometimes creating tests that are blind to certain types of faults. This section introduces an approach that results in criteria that are stronger than the semantic criteria, but that are also more complicated to understand and used.

Specifically, this section considers testing predicates expressed in a particular form known as Disjunctive Normal Form or (DNF). DNF is a common choice for expressing logic expressions because it allows complex situations to be captured in small, independent chunks. Suppose a specifier thinks of some action as happening under one of several (possibly overlapping) conditions. Then a DNF formalization directly captures the specifier’s mental model. The fact that the format of the given DNF expression closely tracks the specifier’s understanding of the problem has important implications for testing. Specifically, it suggests that testing should focus on the details of the representation. In other words, it provides a strong motivation for approaching logic coverage criteria from a syntactic perspective.

This section uses different terms and notations than the previous section. This is both to match the still very active research literature and because the notation works better with DNF predicates. Readers familiar with DNF may be familiar with the dual formulation of Conjunctive Normal Form or CNF. Every result for DNF has an equivalent result for CNF. CNF tends to be used less than DNF, both in practice and in the research literature, so we do not treat it here.

We use the same notion of a clause as in the treatment of semantic coverage. For much of this section, it may be helpful to think of a clause simply as a boolean variable. A literal is a clause or the negation of a clause. A term is a set of literals connected only by logical ANDs. A Disjunctive Normal Form (DNF) predicate is a set of terms connected by logical ORs. Terms in DNF predicates are also called implicants, because if a single term is true, that implies the entire predicate is true.

For example, this predicate is in disjunctive normal form:
\[(a \land \neg c) \lor (b \land \neg c)\]
but this (equivalent) one is not:
\[(a \lor b) \land \neg c\]
This example has three clauses: a, b, and c; three literals: a, b, and \(\neg c\); and two terms: \((a \land \neg c)\) and \((b \land \neg c)\).

In general, the DNF representation of a predicate is not unique. For example, the above predicate can be rewritten in the following way, which is also in DNF:
\[(a \land b \land \neg c) \lor (a \land \neg b \land \neg c) \lor (\neg a \land b \land \neg c)\]
This section follows the convention from the DNF testing literature and uses adjacency for the \(\land\) operator, “+” for the \(\lor\) operator, and an overstrike for the negation operator. This
8.2. SYNTACTIC LOGIC COVERAGE CRITERIA (DNF)

approach makes the sometimes long expressions easier to read. So, the last DNF predicate above will be written:

\[ ab \overline{c} + a\overline{b}c + \overline{a}b\overline{c} \]

8.2.1 Implicant Coverage

The next three subsections explain how disjunctive normal form expressions are used to design tests. As with the semantic logic coverage criteria, we start small and build to a very strong coverage criterion, MUMCUT.

One simple way of testing with respect to DNF representations is to assign values to clauses so that each implicant in the DNF representation is satisfied on at least one test. All of these tests result in the predicate evaluating to true, so we never test the false case. We address this problem by formulating a DNF expression for the negation of the predicate in question, and evaluating tests for the negated predicate with the same coverage criteria used for the predicate itself. These ideas are enough to define our first DNF coverage criterion:

**Criterion 8.9 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.

As an example of IC, consider the following DNF expression for a predicate \( f \) in three clauses \((a, b, \text{ and } c)\) and two terms \( ab \) and \( b\overline{c} \).

\[ f(a, b, c) = ab + b\overline{c} \]

Its negation can be computed algebraically as follows:

\[
\begin{align*}
\overline{f}(a, b, c) &= \overline{ab} \lor \overline{b}\overline{c} \\
&= \overline{ab} \land \overline{b}\overline{c} \quad \text{– DeMorgan’s Law} \\
&= (\overline{a} \lor \overline{b}) \land (\overline{b} \lor c) \quad \text{– DeMorgan’s Law} \\
&= \overline{a}b \lor \overline{a}c \lor \overline{b}b \lor \overline{b}c \quad \text{– Distributive Law} \\
&= (\overline{a}b \lor \overline{b}b) \lor \overline{b}c \lor \overline{a}c \quad \text{– Commutativity Law} \\
&= (\overline{b} \lor b\overline{c}) \lor \overline{a}c \quad \text{– Absorption Law} \\
&= \overline{b} \lor \overline{a}c \quad \text{– Absorption Law}
\end{align*}
\]

Collectively, \( f \) and \( \overline{f} \) have a total of four implicants:

\[ \{ab, b\overline{c}, \overline{b}, \overline{a}c\} \]

An obvious but simple way to generate tests for these four implicants would be to choose one test for each. However, they can be satisfied with fewer tests. Consider the following table, which indicates the truth assignments required for each of the four implicants.
The first and second row can be satisfied simultaneously, as can the third and fourth. Thus only two tests are needed to satisfy IC for this example:

\[ T_1 = \{TTF, FFT\} \]

IC guarantees that the predicate will be both true and false, thus it subsumes Predicate Coverage. However it does not subsume any of the Active Clause Coverage criteria.

One problem with IC is that tests might be chosen so that a single test satisfies multiple implicants. Indeed, this is how the two element test set \( T_1 \) above was chosen. Although this lets testers minimize the size of test suites, it makes it harder to test each implicant individually. Another problem with IC is that the arbitrary nature of choosing a specific DNF representation for the negation of a predicate. In short, IC is fairly weak, and there are much stronger DNF coverage criteria available. Before we can develop these criteria, we need to introduce a bit more mathematical machinery.

### 8.2.2 Minimal DNF

Just as with the active clause criteria, we would like each implicant in a DNF expression to “matter.” That is, we want a DNF form where each implicant can be satisfied without satisfying any other implicant. Fortunately, standard approaches already exist that can be used. A proper subterm of an implicant is an implicant with one or more subterms removed. For example, the proper subterms of \( abc \) are \( ab \), \( bc \), \( ac \), \( a \), \( b \), and \( c \). A prime implicant is an implicant such that no proper subterm of the implicant is also an implicant of the same predicate. That is, in a prime implicant, it is not possible to remove a term without changing the value of the predicate. For example, in the following reformulation of the previous example

\[ f(a, b, c) = abc + ab\bar{c} + b\bar{c} \]

\( abc \) is not a prime implicant, because a proper subterm, namely \( ab \), is an implicant. \( ab\bar{c} \) is not a prime implicant either, because the proper subterm \( ab \) is an implicant, as is the proper subterm \( b\bar{c} \).

We need one additional concept. An implicant is redundant if it can be omitted without changing the value of the predicate. As an example, the formula

\[ f(a, b, c) = ab + ac + b\bar{c} \]
has three prime implicants, but the first one, \( ab \), is redundant because \( ac + b\bar{c} \) is exactly the same function as \( ab + ac + b\bar{c} \). A DNF representation is minimal if every implicant is prime and no implicant is redundant. Minimal DNF representations can be computed algebraically or by hand with Karnaugh maps, as discussed in section 8.2.4. Since non-prime implicants mean unnecessary constraints and redundant implicants are, by definition, unnecessary, there is good reason for the software engineer to refactor DNF predicates into minimal form.

With the above definitions, we can assume that we have a minimal DNF representation of a predicate. Given a minimal DNF representation for \( f \), a unique true point with respect to the \( ith \) implicant is an assignment of truth values to clauses such that the \( ith \) implicant is true and all other implicants are false. It should be noted that if it is impossible to make all of the “other” implicants false, then the implicant is redundant, violating our assumption that \( f \) is in minimal DNF form. We illustrate unique true points with an example. If \( f \) is:

\[
f(a, b, c, d) = ab + cd
\]

then with respect to implicant \( ab \), \( TTFT \), \( TTTF \), and \( TTFF \) are all unique true points. \( TTTT \) is also a true point, but it is not a unique true point, because both implicants \( ab \) and \( cd \) are true for \( TTTT \).

There is a corresponding notion for false points. Given a DNF representation of a predicate \( f \), a near false point for \( f \) with respect to clause \( c \) in implicant \( i \) is an assignment of truth values to clauses such that \( f \) is false, but if \( c \) is negated and all other clauses are left as is, \( i \) (and hence \( f \)) evaluates to true. For example, if \( f \) is:

\[
f(a, b, c, d) = ab + cd
\]

then the near false points are \( FTFF \), \( FTFT \), and \( FTTF \) for clause \( a \) in the implicant \( ab \), and \( TFFF \), \( TFFT \), and \( TFTF \) for clause \( b \) in the implicant \( ab \).

### 8.2.3 The MUMCUT Coverage Criterion

The literature contains many DNF coverage criteria. The motivation for many of these criteria is their ability to detect certain categories of faults. In this section, we develop MUMCUT, the most important of these criteria in the sense that it guarantees detection of single instances of all possible faults in a certain fault hierarchy. First, we need to introduce fault types for logic expressions, then several preliminary criteria.

Table 8.1 defines nine syntactic faults on predicates in DNF form\(^2\). These faults capture typical ways in which one might fail to express the correct predicate by making a single mistake. For example, the Literal Insertion Fault or LIF represents the case where an additional constraint is mistakenly included in a term. This set of fault classes has received considerable scrutiny in the literature, and is regarded as reasonably complete. There are

---

\(^2\)The notion of mutation operators developed in the Chapter 9 is closely related to the notion of fault classes presented here.
some obvious faults, such as “stuck-at” faults, that are not included explicitly in the list. These faults are not included because if the faults that are included are found, then they will also be found.

Figure 8.2 gives a detection relationship between the types of faults in 8.1. If a test set is guaranteed to detect a given type of fault, then the test set is also guaranteed to detect the types of faults “downstream” from that fault. For example, a test set guaranteed to detect all LIFs is also guaranteed to all TOFs and all LRFs, and by implication, all ORF+s, LNFs, TNFs, and ENFs. Note that any test detects ENFs.

<table>
<thead>
<tr>
<th>Fault</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression Negation Fault (ENF)</td>
<td>An expression incorrectly written as its negation: $f = ab + c$ written as $f' = \overline{ab} + c$</td>
</tr>
<tr>
<td>Term Negation Fault (TNF)</td>
<td>A term is incorrectly written as its negation: $f = ab + c$ written as $f' = \overline{ab} + c$</td>
</tr>
<tr>
<td>Term Omission Fault (TOF)</td>
<td>A term is incorrectly omitted: $f = ab + c$ written as $f' = ab$</td>
</tr>
<tr>
<td>Literal Negation Fault (LNF)</td>
<td>A literal is incorrectly written as its negation: $f = ab + c$ written as $f' = \overline{ab} + c$</td>
</tr>
<tr>
<td>Literal Reference Fault (LRF)</td>
<td>A literal is incorrectly replaced by another literal: $f = ab + bcd$ written as $f' = ad + bcd$</td>
</tr>
<tr>
<td>Literal Omission Fault (LOF)</td>
<td>A literal is incorrectly omitted: $f = ab + c$ written as $f' = a + c$</td>
</tr>
<tr>
<td>Literal Insertion Fault (LIF)</td>
<td>A literal is incorrectly added to a term: $f = ab + c$ written as $f' = ab + \overline{bc}$</td>
</tr>
<tr>
<td>Operator Reference Fault (ORF+)</td>
<td>An ‘Or’ is incorrectly replaced by ‘And’: $f = ab + c$ written as $f' = abc$</td>
</tr>
<tr>
<td>Operator Reference Fault (ORF*)</td>
<td>An ‘And’ is incorrectly replaced by ‘Or’: $f = ab + c$ written as $f' = a + b + c$</td>
</tr>
</tbody>
</table>

Table 8.1: DNF fault classes.

The first coverage criterion we introduce detects LIF faults, which is targeted because LIF is at the top of the fault hierarchy. Multiple Unique True Points (MUTP) is defined:

**Criterion 8.10 Multiple Unique True Points Coverage (MUTP):** Given a minimal DNF representation of a predicate $f$, for each implicant $i$, choose unique true points (UTPs) such that clauses not in $i$ take on values $T$ and $F$.

By way of example, consider:

$$f(a, b, c, d) = ab + cd$$

For implicant $ab$, if we choose the unique true points $TTFT$ and $TTTF$, then $c$ and $d$, the clauses not in implicant $ab$, take on the values $T$ and $F$. Similarly, for implicant $cd$, if we
choose the unique true points $FTTT$ and $TFTT$, then $a$ and $b$, the clauses not in implicant $cd$, take on the values $T$ and $F$. The resulting MUTP set for predicate $ab + cd$ is:

$$\{TTFT, TTTF, FTTT, TFTT\}$$

MUTP is a powerful criterion in terms of detecting faults. As mentioned earlier, MUTP is engineered to detect Literal Insertion Faults or LIFs, a fault class that sits atop the fault hierarchy in figure 8.2. If MUTP is feasible, that is, if there exist unique true points such that literals not in each implicant can take on the values $T$ and $F$, then MUTP detects all LIF faults. Applying this fact to the fault hierarchy, we can see that if MUTP is feasible, it detects seven of the nine fault classes. The only fault classes not detected are LOF and ORF*.

To see why MUTP is so powerful, consider what happens when a literal is inserted into a term. Because MUTP forces the clauses not in the implicant to take on the values $T$ and $F$ on different tests, the inserted literal is guaranteed to take on the value $F$ on some test. That means that the entire implicant is false at what is supposed to be a true point, and hence the MUTP test fails by evaluating to false instead of true.

To make this concrete, consider the implicant $ab$ in our earlier predicate $ab + cd$. As we saw, MUTP is feasible for every implicant in this predicate, which means that we found UTPs where $c$ and $d$ takes on both truth values, namely $TTFT$ and $TTTF$.

Now consider what happens if we insert a literal $l$ into implicant $ab$:

$$abl$$

If $l$ is $a$, then the literal is redundant, and there is no change to the function, and hence no fault to detect. If $l$ is $\bar{a}$, then both MUTP tests $TTFT$ and $TTTF$ will evaluate to false, and
the LIF is detected. Similar behavior occurs if \( l \) is a \( b \) or \( \bar{b} \). If \( l \) is \( c \), then test \( TTFT \) evaluates to false, and the LIF is detected. Similarly, if \( l \) is \( \bar{c} \), test \( TTTF \) evaluates to false, and again the LIF is detected. Similar behavior happens if \( l \) is \( d \) or \( \bar{d} \). Of course, this argument breaks down for predicates where MUTP is not feasible for all implicants, and hence there is no guarantee that MUTP detects all LIFs for arbitrary predicates.

To summarize, MUTP is good, but it is not complete with respect to the fault hierarchy. In particular, it cannot detect any LOF or ORF* faults, since these faults require false points for detection, and, by definition, MUTP generates only true points. MUTP also has blind spots were MUTP is infeasible. The next criterion, CUTPNFP, includes false points to address the first of these concerns:

<table>
<thead>
<tr>
<th>Criterion 8.11 Corresponding Unique True Point and Near False Point Pair Coverage (CUTPNFP):</th>
</tr>
</thead>
</table>

Given a minimal DNF representation of a predicate \( f \), for each literal \( c \) in each implicant \( i \), \( TR \) contains a unique true point for \( i \) and a near false point for \( c \) in \( i \) such that the two points differ only in the truth value of \( c \).

By way of example, for:

\[
f(a, b, c, d) = ab + cd
\]

if we consider clause \( a \) in the implicant \( ab \), we can choose one of three unique true points, namely, \( TTFF \), \( TTFT \), and \( TTTF \), and pair each, in turn, with the corresponding near false points \( FTFF \), \( FTFT \), and \( FTTF \). So, for example, to satisfy CUTPNFP with respect to clause \( a \) in implicant \( ab \), we could choose the first pair, \( TTFF \) and \( FTFF \). Likewise, to satisfy CUTPNFP with respect to clause \( b \) in implicant \( ab \), we could choose the pair \( TTFF \) and \( TFFF \), to satisfy CUTPNFP with respect to clause \( c \) in implicant \( cd \), the pair \( FTTT \) and \( FFST \), and to satisfy CUTPNFP with respect to clause \( d \) in implicant \( cd \), the pair \( FFFF \) and \( FFFT \). The resulting CUTPNFP set is:

\[
( TTFF, FFFF, TFFF, TFFT, FFFT, FFFT )
\]

Note that the first two tests are unique true points, and the remaining four are corresponding near false points.

Unlike MUTP, CUTPNFP effectively detects LOF faults if CUTPNFP is feasible. The reason is that for every clause \( c \) in term \( i \), CUTPNFP demands a unique true point and a near false point. These two tests differ only in the value of the clause \( c \). Hence if \( c \) (or \( \bar{c} \)) is incorrectly deleted in the implementation, both of these tests will produce the same truth value, thereby revealing the fault. Given the detection relationships in Figure 8.2, we can infer that CUTPNFP, if feasible, also detects ORF*, LNF, TNF, and ENF faults. It’s worth pointing out that CUTPNFP does subsume RACC, which is not suprising if you consider the way in which CUTPNFP picks pairs of tests. Also, CUTPNFP does not guarantee the detection of LIF faults and hence cannot replace MUTP.

There are some cases where MUTP and CUTPNFP are infeasible, and hence additional tests are needed. The MNFP criterion supplies these tests:
8.2. SYNTACTIC LOGIC COVERAGE CRITERIA (DNF)

**Criterion 8.12 Multiple Near False Point Coverage (MNFP):** Given a minimal DNF representation of a predicate \( f \), for each literal \( c \) in each implicant \( i \), choose near false points (NFPs) such that clauses not in \( i \) take on values T and F.

Consider again:

\[ f(a, b, c, d) = ab + cd \]

For implicant \( ab \), consider literal \( a \). If we choose \( FTFT \) and \( TFFT \) as near false points (NFPs) with respect to \( a \), then \( c \) and \( d \), the literals not in \( ab \), take on the values T and F. Similarly for literal \( b \) in implicant \( ab \), we can choose \( TFFT \) and \( TFTF \). For implicant \( cd \), if we choose \( TFTF \) and \( TFFT \) as near false points (NFPs) with respect to \( c \), then \( a \) and \( b \), the literals not in \( cd \), take on the values T and F. Similarly for literal \( d \) in implicant \( cd \), we can choose \( FTTF \) and \( TFTF \). There is overlap in these choices: only 4 tests are needed. The resulting MNFP set for predicate \( ab + cd \) is:

\[ \{TFTF, TFFT, FTTF, FTFT\} \]

It turns out that if you apply all of MUTP, CUTPNFP, and MNFP, the resulting test set detects the entire fault hierarchy, even in those cases where some test requirements are infeasible. Basically, feasible test requirements from one criterion compensate for infeasible test requirements from other criteria. Hence MUMCUT combines these three criteria:

**Criterion 8.13 MUMCUT:** Given a minimal DNF representation of a predicate \( f \), apply MUTP, CUTPNFP, and MNFP to \( f \).

Compared to a semantic coverage criterion such as RACC, MUMCUT is quite expensive in terms of the number of tests needed for a given predicate. But less expensive variants of MUMCUT have been developed, and these variants require far fewer tests, although still more than the semantic (ACC) criteria. But there is a significant benefit to these extra tests. Let’s consider the effectiveness of RACC in detecting faults in the fault hierarchy.

From a theoretical perspective, RACC is only guaranteed to detect all instances of the TNF and ENF faults. RACC tests are not guaranteed to detect the faults for the other seven fault classes.

In practice, researchers have found that RACC tests only detect about one third of the faults from the fault hierarchy, failing to detect two thirds. Thus, MUMCUT should be considered when testing applications where the consequences of failures are especially severe.

8.2.4 Karnaugh Maps

This section reviews Karnaugh maps, which are exceedingly useful for producing DNF representations for predicates with a modest number of clauses. Students looking for an in-depth treatment of Karnaugh maps can turn to a wide variety of textbooks or internet resources.
A Karnaugh map is a tabular representation of a predicate with the special property that groupings of adjacent table entries correspond to simple DNF representations. Karnaugh maps are useful for predicates of up to 4 or 5 clauses; beyond that, they become cumbersome. A Karnaugh map for a predicate in 4 clauses is given below:

![Karnaugh map table for the predicate “ab + cd”.](image)

For now, suppose that entries in the table are restricted to truth values. Truth values can be assigned in $2^n$ possible ways to the $2^n$ entries in a table for $n$ clauses. So, the four clauses represented in the table above have $2^4$ or 16 entries, and $2^{16} = 65,536$ possible functions. The reader will be relieved to know that we will not enumerate all of these in the text. Notice the labeling of truth values along the columns and rows. In particular, notice that any pair of adjacent cells differ in the truth value of exactly one clause. It might help to think of the edges of the Karnaugh map as being connected as well, so that the top and bottom rows are adjacent, as are the left and right columns (that is, a toroidal mapping from 2-space to 3-space).

The particular function represented in the Karnaugh map above can be spelled out in full:

$$ab\bar{c}\bar{d} + ab\bar{c}d + abcd + ab\bar{c}d + \bar{a}bcd + \bar{a}bcd + \overline{a}bcd$$

The expression simplifies to:

$$ab + cd$$

The simplification can be read off the Karnaugh map by grouping together adjacent cells into rectangles of size $2^k$ for some $k > 0$ and forming rectangles of size 1 for cells with no adjacent cells. Overlaps among the groupings are fine. We give an example in three clauses to illustrate. Consider the following Karnaugh map:

![Karnaugh map for three clauses](image)

Four rectangles of size 2 can be extracted from this graph. They are the functions $b\bar{c}$, $ab$, $ac$, and $bc$, and are represented by the following Karnaugh maps:

![Karnaugh map for three clauses](image)
8.2. SYNTACTIC LOGIC COVERAGE CRITERIA (DNF)

At first, the last of these might be a bit hard to see as a rectangle, but remember that the Karnaugh map is joined at the edges, left and right, as well as top and bottom. We could write the original function out as the disjunction of these four Karnaugh maps, each of which gives a prime implicant, but notice that the second, representing \( ab \), is, in fact, redundant with the other three implicants, since all of its entries are covered by another Karnaugh map. The resulting minimal DNF expression is:

\[
f = \overline{b}c + ac + \overline{bc}
\]

One can also note that all of the entries of \( ac \) are covered by other Karnaugh maps, so \( ac \) is redundant with the remaining three implicants. So a different minimal DNF representation is:

\[
f = \overline{b}c + ab + \overline{bc}
\]

Negations in DNF form are also easy to pull from a Karnaugh map. Consider again the function \( f \) given above. We can negate \( f \) by changing all blank entries to ‘1’s and all ‘1’s to blank:

\[
\overline{f} = \overline{b}\overline{c} + \overline{a}bc
\]

Karnaugh maps are extremely convenient notations to derive test sets for many of the logic coverage criteria. For example, consider again the predicate \( ab + cd \). Unique true points are simply true points covered by a single rectangle. Hence, of all the true points in \( ab + cd \), all but \( TTTT \) are unique true points. Near false points for any given true point are simply those false points that are immediately adjacent in the Karnaugh map. For MUTP, we can identify unique true points where clauses not in the term take on both truth values.
For CUTPNFP, pair up near false points with unique true points, being careful to obtain a pairing for each clause in \( f \). For MNFP, we identify near false points for each literal such that clauses not in the term under analysis take on both truth values. Karnaugh maps are an easy way to compute determination: simply identify all pairs of adjacent cells where the truth value of the variable in question and the truth value of the predicate both change. Pairing of true points with near false points is also an easy way to develop RACC tests. Note that for RACC tests, it does not matter if the true points are unique or not. Slide animations of all of these uses of Karnaugh maps are available on the book website, as are some video illustrations.

**Exercises, Section 8.2.**

1. Use predicates (i) through (iv) to answer the following questions.

   i. \( f = ab\bar{c} + \bar{a}b\bar{c} \)
   
   ii. \( f = \bar{a}\bar{b}\bar{c}d + abcd \)
   
   iii. \( f = ab + abc + \bar{a}\bar{b} \)
   
   iv. \( f = \bar{a}\bar{c}d + \bar{c}d + bcd \)

   (a) Draw the Karnaugh maps for \( f \) and \( \bar{f} \).
   
   (b) Find the nonredundant prime implicant representation for \( f \) and \( \bar{f} \).
   
   (c) Give a test set that satisfies Implicant Coverage (IC) for \( f \).
   
   (d) Give a test set that satisfies Multiple Unique True Points (MUTP) for \( f \).
   
   (e) Give a test set that satisfies Corresponding Unique True Point and Near False Point Pair Coverage (CUTPNFP) for \( f \).
   
   (f) Give a test set that satisfies Multiple Near False Points (MNFP) for \( f \).
   
   (g) Give a test set that is guaranteed to detect all faults in Figure 8.2.

2. Use the following predicates to answer questions (a) through (f).

   - \( W = (b \land \neg c \land \neg d) \)
   - \( X = (b \land d) \lor (\neg b \land \neg d) \)
   - \( Y = (a \land b) \)
   - \( Z = (\neg b \land d) \)

   (a) Draw the Karnaugh map for the predicates. Put \( ab \) on the top and \( cd \) on the side. Label each cell with \( W, X, Y, \) and/or \( Z \) as appropriate.
   
   (b) Find the minimal DNF expression that describes all cells that have more than one definition.
   
   (c) Find the minimal DNF expression that describes all cells that have no definitions.
   
   (d) Find the minimal DNF expression that describes \( X \lor Z \).
   
   (e) Give a test set for \( X \) that uses each prime implicant once.
   
   (f) Give a test set for \( X \) that is guaranteed to detect all faults in Figure 8.2.
3. **(Challenging!)** Consider “stuck-at” faults, where a literal is replaced by the constant *true* or the constant *false*. These faults do not appear in the fault list given in table 8.1 or the corresponding fault detection relationships given in Figure 8.2.

   (a) Which fault type “dominates” the stuck-at fault for the constant *true*? That is, find the fault in Figure 8.2 such that if a test set is guaranteed to detect every occurrence of that fault, then the test set also detects all stuck-at *true* faults. Explain your answer.

   (b) Which fault type dominates the stuck-at fault for the constant *false*? That is, find the fault in Figure 8.2 such that if a test set is guaranteed to detect every occurrence of that fault, then the test set also detects all stuck-at *false* faults. Explain your answer.

---

### 8.3 Structural Logic Coverage of Programs

As with graph coverage criteria, the logic coverage criteria apply to programs in a straightforward way. Predicates are derived directly from decision points in the programs (if, case, and loop statements). The higher-end criteria, such as active clause coverage, are most useful as the number of clauses in the predicates grow. However, the vast majority of predicates in real programs have only one clause, and programmers tend to write predicates with a maximum of two or three clauses. It should be clear that when a predicate only has one clause, all of the logic coverage criteria collapse to Predicate Coverage.

The primary complexity of applying logic coverage to programs has more to do with reachability than with the criteria. That is, a logic coverage criterion imposes test requirements that are related to specific decision points (statements) in the program. Getting values that satisfy those requirements is only part of the problem; getting to the statement is sometimes more difficult. Two issues are associated with getting there. The first is simply that of reachability from Chapter 3; the test case must include values to reach the statement. In small programs (that is, most methods) this problem is not hard, but when applied within the context of an entire arbitrarily large program, satisfying reachability can be enormously complex. The values that satisfy reachability are prefix values in the test case.

The other part of “getting there” can be even harder. The test requirements are expressed in terms of program variables that may be defined locally to the unit or locally to the statement block under test. Test cases, on the other hand, can include values only for inputs to the program that we are testing. Therefore these internal variables have to be resolved to be in terms of the input variables. Although the values for the variables in the test requirements should ultimately be a function of the values of the input variables, this relationship may be arbitrarily complex. In fact, this internal variable problem is formally undecidable.

Consider an internal variable *X* that is derived from a table lookup, where the index to the table is determined by a complex function whose inputs are program inputs. To choose a particular value for *X*, the tester has to work backward from the statement where the
decision appears, to the table where $X$ was chosen, to the function, and finally to an input that would cause the function to compute the desired value. This controllability problem has been explored in depth in the automatic test data generation literature and will not be discussed in detail here, except to note that this problem is a major reason why the use of program-level logic coverage criteria is usually limited to unit and module testing activities.

We illustrate the logic coverage concepts through an example. Figure 8.3 shows the class Thermostat, which is part of a household programmable thermostat. It contains one principle method, turnHeaterOn(), which uses several class variables to decide whether to turn the heater on. The class variables each have a short “setter” method, so can be considered to be “half-beans.” Although a small example, Thermostat has several advantages: Its purpose is relatively easy to understand, it is small enough to fit in a classroom exercise, and its logic structure is complicated enough to illustrate most of the concepts. Line numbers have been added to the figure to allow us to reference specific decision statements in the text.

When applying logic criteria to programs, predicates are taken from decision points in the program, including if statements, case / switch statements, for loops, while loops, and do-unltil loops. This is illustrated with the turnHeaterOn() method in the Thermostat class. turnHeaterOn() has the following predicates (line numbers are shown on the left, and the else statement at line 40 does not have its own predicate):

28-30: (((curTemp < dTemp - thresholdDiff) ||
   (Override && curTemp < overTemp - thresholdDiff)) &&
   timeSinceLastRun.greaterThan (minLag))
34: (Override)

The predicate on lines 28-30 has four clauses and uses seven variables (two are used twice). We use the following substitutions to simplify the discussion.

a: curTemp < dTemp - thresholdDiff
b: Override
c: curTemp < overTemp - thresholdDiff
d: timeSinceLastRun > minLag

Thus we get:

28-30: (a || (b && c)) && d
31: b

The turnHeaterOn() method has one input parameter, an object that contains the temperature settings the user has programmed. turnHeaterOn() also uses the class variables controlled by setter methods. dTemp is an internal variable that determines the desired temperature. It uses the period of the day and the type of day to ask the ProgrammedSettings
8.3. STRUCTURAL LOGIC COVERAGE OF PROGRAMS

1 // Introduction to Software Testing
2 // Authors: Paul Ammann & Jeff Offutt
3 // Chapter 8, page ??
4 // See ThermostatTest.java for JUnit tests
5
6 import java.io.*;
7 import java.util.*;
8
9 // Programmable Thermostat
10 public class Thermostat
11 {
12    private int curTemp;  // current temperature reading
13    private int thresholdDiff;  // temp difference until we turn heater on
14    private int timeSinceLastRun;  // time since heater stopped
15    private int minLag;  // how long I need to wait
16    private boolean override;  // has user overridden the program
17    private int overTemp;  // overriding temperature
18    private int runTime;  // output of turnHeaterOn - how long to run
19    private boolean heaterOn;  // output of turnHeaterOn - whether to run
20    private Period period;  // period
21    private DayType day;  // daytype
22
23    // Decide whether to turn the heater on, and for how long.
24    public boolean turnHeaterOn (ProgrammedSettings pSet)
25    {
26        int dTemp = pSet.getSetting (period, day);
27
28        if (((curTemp < dTemp - thresholdDiff) ||
29                (override && curTemp < overTemp - thresholdDiff)) &&
30                (timeSinceLastRun > minLag))
31        {  // Turn on the heater
32            // How long? Assume 1 minute per degree (Fahrenheit)
33            int timeNeeded = curTemp - dTemp;
34            if (override)
35               timeNeeded = curTemp - overTemp;
36            setRunTime (timeNeeded);
37            setHeaterOn (true);
38            return (true);
39        }
40        else
41        {
42            setHeaterOn (false);
43            return (false);
44        }
45    } // End turnHeaterOn
46
47    public void setCurrentTemp (int temperature) { curTemp = temperature; }
48    public void setThresholdDiff (int delta) { thresholdDiff = delta; }
49    public void setTimeSinceLastRun (int minutes) { timeSinceLastRun = minutes; }
50    public void setMinLag (int minutes) { minLag = minutes; }
51    public void setOverride (boolean value) { override = value; }
52    public void setOverTemp (int temperature) { overTemp = temperature; }
53
54    // for the ProgrammedSettings
55    public void setDay (DayType curDay) { day = curDay; }
56    public void setPeriod (Period curPeriod) { period = curPeriod; }
57
58    // outputs from turnHeaterOn - need corresponding getters to activate heater
59    void setRunTime (int minutes) { runTime = minutes; }
60    void setHeaterOn (boolean value) { heaterOn = value; }
61 } // End Thermostat class

Figure 8.3: Thermostat Class
object for the current desired temperature. The rest of this section illustrates how to satisfy the logic coverage criteria on `turnHeaterOn()`. Before addressing the actual criteria, it is first necessary to analyze the predicates to find values that will reach the predicates (the reachability problem) and to understand how to assign particular values to the internal variable `dTemp` (the internal variable problem).

First we consider reachability. The predicate on lines 28-30 is always reached, so the condition that must be satisfied to reach lines 28-30 (its reachability condition) is `True`, as shown in Table 8.2. The predicate on line 34 is inside the `if` block that starts on line 24, so is only reached if the predicate on lines 28-30 is true. Thus, its reachability condition is `(a || (b && c)) && d`. The `else` part of the `if` block transfers control to line 42. Its reachability condition is the negation of the reachability condition to enter the `if` block: `!(a || (b && c)) && d`, which can be simplified to `!c || (!a && (!b || !d))`.

Table 8.2: Reachability for Thermostat predicates.

<table>
<thead>
<tr>
<th>Line</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>28-30</td>
<td><code>True</code></td>
</tr>
<tr>
<td>34</td>
<td>`(a</td>
</tr>
<tr>
<td>40</td>
<td>`!(a</td>
</tr>
</tbody>
</table>

Note that clause `a` is an abbreviation for `curTemp < dTemp - thresholdDiff`, which uses the local (internal) variable `dTemp`. We cannot pass a value directly to `dTemp` as part of the test inputs, so we have to control its value indirectly. So the next step in generating test values is to discover how to assign specific values to `dTemp`.

Line 26 uses the `programmedSettings` object to call the method `getSetting()` with the parameters `period` and `day`. Let’s suppose we want the desired temperature to be a room comfortable 69F (about 20.5C). This is an issue of controllability that complicates test automation. A naive solution would be to change the method under test by replacing the method call with a direct assignment. This has two disadvantages: (1) we must recompile the `Thermostat` class before running each test, and (2) we are testing a different method than we plan to deploy.

A more robust approach is to learn how to set the program state so that the call in `turnHeaterOn()` will return the desired value. In the Thermostat program, this is accomplished with a call to the `setSetting()` method in the `programmedSettings` object. The `period` and `day` are Java enum types. The source for `Thermostat.java`, `ProgrammedSettings.java`, `Period.java`, and `DayType.java` are all available on the book website. We choose to set the temperature in the morning on a weekday, so our test needs the following three calls:

- `setSetting (Period.MORNING, DayType.WEEKDAY, 69);`
- `setPeriod (Period.MORNING);`
8.3. STRUCTURAL LOGIC COVERAGE OF PROGRAMS

- `setDay (DayType.WEEKDAY);`

These statements must appear in the automated test before the call to `turnHeaterOn()`. This also illustrates an implicit requirement for automated testing—the test team must include programmers who can understand the software well enough to create these kinds of calls.

8.3.1 Satisfying Predicate Coverage

Finding values to satisfy Predicate Coverage for the predicate on lines 28-30 in `turnHeaterOn()` involves four clauses and seven variables, including the internal variable `dTemp`. To set the predicate `(a || (b && c)) && d` to be true, `d` must be true and the left side, `(a || (b && c))`, must also be true. Let’s make it simple and try to assign all four clauses, `a`, `b`, `c`, and `d`, to be true, as shown in Table 8.3.

<table>
<thead>
<tr>
<th>Clause Label</th>
<th>Clause Detail</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td><code>curTemp &lt; dTemp - thresholdDiff</code></td>
<td>true</td>
</tr>
<tr>
<td>b</td>
<td><code>Override</code></td>
<td>true</td>
</tr>
<tr>
<td>c</td>
<td><code>curTemp &lt; overTemp - thresholdDiff</code></td>
<td>true</td>
</tr>
<tr>
<td>d</td>
<td><code>timeSinceLastRun &gt; minLag</code></td>
<td>true</td>
</tr>
</tbody>
</table>

Table 8.3: Clauses in the Thermostat predicate on lines 28-30.

Clause `a` is straightforward, although we must remember that we fixed `dTemp` to be 69 to solve the internal variable problem. If we set the current temp (`curTemp`) to be 63 and the threshold difference (`thresholdDiff`) to be 5, then 63 is less than 69-5 and `a` is true. (The threshold difference is the maximum we allow the current temperature to deviate from the desired temperature before cycling the heater on again.)

Clause `b` is even simpler: an override means a human has entered a new desired temperature that will temporarily override the programming. So the variable `Override` is simply given the value true.

Clause `c` is associated with an override. An override must come with a new temperature (`overTemp`), and the heater is only turned on if the current temperature is less than the new overriding temperature, minus the threshold. We have already fixed `thresholdDiff` at 5 and `curTemp` at 63, so clause `c` can be set true by setting `overTemp` to be 70.

Finally, clause `d` compares `timeSinceLastRun` with `minLag`. The `minLag` variable defines how long the heater must be off before it can be turned on again (a safety or engineering constraint from the heater manufacturer). We will assume it is 10 minutes. Then we must set `timeSinceLastRun` to be greater than 10, for example, 12.

Putting all of these decisions together results in the executable test in Figure 8.4.
// Partial test for method turnHeaterOn() in class Thermostat
// Criterion: PC
// Value: True
// Predicate: lines 28-30
// Expected Output: true

// Instantiate needed objects
thermo = new Thermostat();
settings = new ProgrammedSettings();

// Setting internal variable dTemp
settings.setSetting (Period.MORNING, DayType.WEEKDAY, 69);
thermo.setPeriod (Period.MORNING);
thermo.setDay (DayType.WEEKDAY);

// Clause a: curTemp < dTemp - thresholdDiff : true
thermo.setCurrentTemp (63);
thermo.setThresholdDiff (5);

// Clause b: Override : true
thermo.setOverride (true);

// Clause c: curTemp < overTemp - thresholdDiff : true
thermo.setOverTemp (70);

// Clause d: timeSinceLastRun.greaterThan (minLag) : true
thermo.setMinLag (10);
thermo.setTimeSinceLastRun (12);

// Run the test
assertTrue(thermo.turnHeaterOn (settings));

Figure 8.4: PC true test for Thermostat class
The expected result is true, as stated in the comments. Analysis for the false case is similar and is left as an exercise. We also include an exercise to complete the automated test in a framework such as JUnit.

It should be obvious from this example that Predicate Coverage on programs is simply another way to formulate the Edge Coverage criterion. It is not necessary to draw a graph for the logic criteria, but the control flow graph can be used to find values for reachability.

Previously we said that selection of values for “don’t care” inputs should be postponed until reachability is determined. This is because of potential interactions with the requirements for reachability and the selection of values. That is, some inputs may be “don’t care” for the test requirements, but may need specific values to reach the decision. Thus, if we select values too early, it may become difficult or impossible to satisfy reachability.

8.3.2 Satisfying Clause Coverage

We have already done most of the work to satisfy clause coverage for the predicate on lines 28-30 when satisfying predicate coverage. We use the same clause abbreviations from Table 8.3 (a, b, c, and d). To satisfy CC, we need to set each clause to be both true and false. Since we set each clause to be true for the PC tests, half our work is already done.

For clause a, we already discovered how to set dTemp to be 69, so we can reuse that part of the test. We can also set thresholdDiff at 5 again. If we set curTemp to be 66, then clause a evaluates to false.

All we need to do for clause b is set the variable to be false.

For clause c, we have already fixed thresholdDiff at 5 and curTemp at 66. We make clause c false by setting overTemp to be 72.

Finally, clause d compares timeSinceLastRun with minLag. The minLag variable defines how long the heater must be off before it can be turned on again (a safety or engineering constraint from the heater manufacturer). For consistency with other tests, we will again assume it is 10 minutes. Then we must set timeSinceLastRun to be less than or equal to 10, for example, 8.

The definition for CC does not specify whether the values for each clause should be in separate tests, or combined into one test. We can satisfy CC on this predicate with two tests—one where all clauses are true, and another where all clauses are false. The first disadvantage of this approach is that PC and CC become the same. The second is that short-circuit evaluation means some clauses will never be evaluated. With the predicate \((a \lor (b && c)) && d\), if \((a \lor (b && c))\) is false, then d is not evaluated. If a is true, \((b && c)\) is not evaluated. Thus, if all true clauses are combined into one test, yielding \((a=\text{true} \lor (b=\text{true} && c=\text{true})) && d=\text{true}, b\) and \(c\) are not even evaluated. Likewise, if all false clauses are combined into one test, yielding \((a=\text{false} \lor (b=\text{false} && c=\text{false})) && d=\text{false}, c\) and \(d\) are not evaluated.

Rather than resolve this question, we simply list the Java statements needed to automate the clause assignments in Figure 8.5. Each clause is listed separately and the tester can combine them as desired.
CHAPTER 8. LOGIC COVERAGE

Figure 8.5: CC test assignments for Thermostat class

```java
// Instantiate needed objects
thermo = new Thermostat();
settings = new ProgrammedSettings();

// Setting internal variable dTemp
settings.setSetting (Period.MORNING, DayType.WEEKDAY, 69);
thermo.setPeriod (Period.MORNING);
thermo.setDay (DayType.WEEKDAY);

// Clause a=true: curTemp < dTemp - thresholdDiff : true
thermo.setCurTemp (63);
thermo.setThresholdDiff (5);
// Clause a=false: curTemp < dTemp - thresholdDiff : false
thermo.setCurTemp (66);
thermo.setThresholdDiff (5);

// Clause b=true: Override : true
thermo.setOverride (true);
// Clause b=false: Override : false
thermo.setOverride (false);

// Clause c=true: curTemp < overTemp - thresholdDiff : true
thermo.setOverTemp (70);
// Clause c=false: curTemp < overTemp - thresholdDiff : false
thermo.setOverTemp (65);

// Clause d=true: timeSinceLastRun > minLag : true
thermo.setMinLag (10);
thermo.setTimeSinceLastRun (12);
// Clause d=false: timeSinceLastRun > minLag : false
thermo.setMinLag (10);
thermo.setTimeSinceLastRun (8);
```
8.3. **STRUCTURAL LOGIC COVERAGE OF PROGRAMS**

8.3.3 **Satisfying Active Clause Coverage**

Rather than going through all of the active clause criteria, we focus on **Correlated Active Clause Coverage**. Our predicate is \( p = (a \lor (b \land c)) \land d \). Computing \( p_a \), we get:

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

That is, clause \( a \) determines the value of the predicate exactly when \( d \) is true, and either \( b \) or \( c \) is false. We suggest students verify this computation with the tabular method and with the online tool on the book website. Similar computations for clauses \( b \), \( c \), and \( d \) yield:

\[
p_b = \neg a \land c \land d \\
p_c = \neg a \land b \land d \\
p_d = a \lor (b \land c)
\]

Table 8.4 shows the truth assignments needed to satisfy CACC for all four clauses, based on the determination computations. The table shows the truth assignments for the various clauses. The major clauses are in the left column, and major clause values are shown with capital ‘T’s and ‘F’s.

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T</strong></td>
<td><strong>F</strong></td>
<td><strong>F</strong></td>
<td><strong>T</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td><strong>T</strong></td>
<td><strong>F</strong></td>
<td><strong>T</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td><strong>F</strong></td>
<td><strong>F</strong></td>
<td><strong>T</strong></td>
</tr>
<tr>
<td><strong>T</strong></td>
<td><strong>T</strong></td>
<td><strong>T</strong></td>
<td><strong>T</strong></td>
</tr>
<tr>
<td><strong>T</strong></td>
<td><strong>T</strong></td>
<td><strong>T</strong></td>
<td><strong>F</strong></td>
</tr>
</tbody>
</table>

In Table 8.4, the second truth assignment for \( a \) as a major clause is the same as the second truth assignment for \( c \). Likewise, the first truth assignment for \( b \) is the same as the first for \( c \). These are duplicated truth assignments and can be removed, so we only need six tests to satisfy CACC on this predicate.
The six tests specified in Table 8.4 can be turned into executable tests by using the appropriate values for the clauses as worked out in sections 8.3.1 and 8.3.2. Putting these all together results in six tests. Again, each test must start by instantiating the objects, and then setting the internal variable \texttt{dTemp}:

```java
// Instantiate needed objects
thermo = new Thermostat();
settings = new ProgrammedSettings();

// Setting internal variable \texttt{dTemp}
settings.setSetting (Period.MORNING, DayType.WEEKDAY, 69);
thermo.setPeriod (Period.MORNING);
thermo.setDay (DayType.WEEKDAY);
```

Since these will be common to all tests, we would expect them to be in a JUnit \texttt{@Setup} method (or something similar in another test framework). Putting the setting for \texttt{dTemp} into \texttt{@Setup} must be done with care, however, in case another test needs a different value. It is possible for a test to override what happens in the \texttt{@Setup} methods, but it can be confusing for tests that have a long life span.

The key assignments for the tests are listed below. Test number five has all clauses true, so can be taken directly from Figure 8.4 in section 8.3.1. The following list includes short notes about the clauses that are set to be false.

1. \texttt{T t f t} — Major clause \texttt{a} is true.
   // Clause \texttt{c} is set to false by setting \texttt{overTemp} to 65.
   thermo.setCurrentTemp (63);
   thermo.setThresholdDiff (5);
   thermo.setOverride (true);
   thermo.setOverTemp (65);
   thermo.setMinLag (10);
   thermo.setTimeSinceLastRun (12);

2. \texttt{F t f t} — Major clause \texttt{a} is false, major clause \texttt{c} is false.
   // Clause \texttt{a} is set to be false by setting \texttt{curTemp} to 66.
   // Clause \texttt{c} is set to be false by setting \texttt{overTemp} to 65.
   thermo.setCurrentTemp (66);
   thermo.setThresholdDiff (5);
   thermo.setOverride (true);
   thermo.setOverTemp (65);
   thermo.setMinLag (10);
   thermo.setTimeSinceLastRun (12);

3. \texttt{f T t t} — Major clause \texttt{b} is true, major clause \texttt{c} is true.
   // Clause \texttt{a} is set to be false by setting \texttt{curTemp} to 66.
   // But this makes clause \texttt{c} false, so we set \texttt{overTemp} to 72.
   thermo.setCurrentTemp (66);
   thermo.setThresholdDiff (5);
   thermo.setOverride (true);
thermo.setOverTemp (72);
thermo.setMinLag (10);
thermo.setTimeSinceLastRun (12);

4. // f F t t — Major clause b is false.
   // Clause a is set to be false by setting curTemp to 66.
   // Setting thresholdDiff to 5 makes clause c true.
   // Clause d is set to be true by setting timeSinceLastRun to be 12.
thermo.setCurrentTemp (66);
thermo.setThresholdDiff (5);
thermo.setOverride (false);
thermo.setOverTemp (70);
thermo.setMinLag (10);
thermo.setTimeSinceLastRun (12);

5. // t t t T — Major clause d is true.
thermo.setCurrentTemp (63);
thermo.setThresholdDiff (5);
thermo.setOverride (true);
thermo.setOverTemp (70);
thermo.setMinLag (10);
thermo.setTimeSinceLastRun (12);

6. // t t t F — Major clause d is false.
thermo.setCurrentTemp (63);
thermo.setThresholdDiff (5);
thermo.setOverride (true);
thermo.setOverTemp (70);
thermo.setMinLag (10);
thermo.setTimeSinceLastRun (8);

The tester can be very confident that these six tests will exercise the turnHeaterOn() method thoroughly, and test the predicate with great rigor.

8.3.4 Predicate Transformation Issues

ACC criteria are considered to be expensive for testers, and attempts have been made to reduce the cost. One approach is to rewrite the program to eliminate multi-clause predicates, thus reducing the problem to branch testing. A conjecture is that the resulting tests will be equivalent to ACC. However, we explicitly advise against this approach for two reasons. One, the resulting rewritten program may have substantially more complicated control structure than the original (including repeated statements), thus endangering both reliability and maintainability. Second, as the following examples demonstrate, the transformed program may not require tests that are equivalent to the tests for ACC on the original program.
Consider the following program segment, where a and b are arbitrary boolean clauses and S1 and S2 are arbitrary statements. S1 and S2 could be single statements, block statements, or function calls.

```java
if (a && b)
    S1;
else
    S2;
```

The Correlated Active Clause Coverage criterion requires the test specifications (t, t), (t, f), and (f, t) for the predicate $a \land b$. However, if the program segment is transformed into the following functionally equivalent structure:

```java
if (a)
    { 
        if (b)
            S1;
        else
            S2;
    } 
else
    S2;
```

the Predicate Coverage criterion requires three tests: (t, t) to reach statement S1, (t, f) to reach the first occurrence of statement S2, and either (f, f) or (f, t) to reach the second occurrence of statement S2. Choosing (t, t), (t, f), and (f, f) means that our tests do not satisfy CACC in that they do not allow a to determine fully the predicate’s value. Moreover, the duplication of S2 in the above example has been taught to be poor programming for years, because of the potential for mistakes when duplicating code.

A slightly larger example reveals the flaw even more clearly. Consider the simple program segment:

```java
if ((a && b) || c)
    S1;
else
    S2;
```

A straightforward rewrite of this program fragment to remove the multi-clause predicate results in this complicated ugliness:

```java
if (a)
    if (b)
        if (c)
            S1;
        else
            S1;
    else
        if (c)
            S1;
        else
            S2;
    else
        if (b)
            if (c)
8.3. STRUCTURAL LOGIC COVERAGE OF PROGRAMS

This fragment is cumbersome in the extreme, and likely to be error-prone with five occurrences of S1 and two of S2. Applying the Predicate Coverage criterion to this would be equivalent to applying Combinatorial Coverage to the original predicate. A reasonably clever programmer (or good optimizing compiler) would simplify it as follows:

```plaintext
if (a)
  if (b)
    S1;
  else
    if (c)
      S1;
    else
      S2;
else
  if (c)
    S1;
  else
    S2;
```

This fragment is still much harder to understand than the original. Imagine a maintenance programmer trying to change this thing!

The following table illustrates truth assignments that can be used to satisfy CACC for the original program segment and predicate testing for the modified version. An ‘X’ under CACC or Predicate indicates that truth assignment is used to satisfy the criterion for the appropriate program fragment. Clearly, Predicate Coverage on an equivalent program is not the same as CACC testing on the original. Predicate coverage on this modified program does not subsume CACC, and CACC does not subsume Predicate Coverage.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>((a \land b) \lor c)</th>
<th>CACC</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>T</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>t</td>
<td>t</td>
<td>f</td>
<td>T</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>t</td>
<td>f</td>
<td>t</td>
<td>T</td>
<td>X X</td>
</tr>
<tr>
<td>4</td>
<td>t</td>
<td>f</td>
<td>f</td>
<td>F</td>
<td>X X</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
<td>t</td>
<td>t</td>
<td>T</td>
<td>X</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>t</td>
<td>f</td>
<td>F</td>
<td>X</td>
</tr>
<tr>
<td>7</td>
<td>f</td>
<td>f</td>
<td>t</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>F</td>
<td>X</td>
</tr>
</tbody>
</table>
8.3.5 Side Effects in Predicates

One more difficult issue comes up when applying logic criteria to predicates. If a predicate contains the same clause twice, and a clause in between has a sideeffect that can change the value of the clause that appears twice, the test values get much harder to create.

Consider the predicate $A \land (B \lor A)$, where $A$ appears twice. We might assume that the runtime system will first check $A$, then check $B$. If $B$ is false, then $A$ is checked again. However, suppose $B$ is actually a method call, $\text{changeVar}(A)$, which has a sideeffect of changing the value of $A$.

This introduces a very difficult controllability problem—how can we write the test to control for two different values of $A$ in the same predicate? Neither the literature on logic testing nor the literature on test automation give a clear answer to this problem, so the tester probably needs to handle this as a special case.

Our best suggestion is social, rather than technical. Go ask the programmer if she really wants to do that. Perhaps the best solution to this example would be to replace the predicate $A \land (B \lor A)$ with the equivalent $A$.

---

Exercises, Section 8.3.

1. Complete and run the tests to satisfy PC for the Thermostat class.

2. Complete and run the tests to satisfy CC for the Thermostat class.

3. Complete and run the tests to satisfy CACC for the Thermostat class.

4. For the Thermostat class, check the computations for how to make each major clause determine the value of the predicate by using the online tool, then the tabular method.

5. Answer the following questions for the method $\text{checkIt}()$ below:

```java
public static void checkIt (boolean a, boolean b, boolean c)
{
    if (a && (b || c))
        System.out.println ("P is true");
    else
        System.out.println ("P isn't true");
}
```

- Transform $\text{checkIt}()$ to $\text{checkItExpand}()$, a method where each if statement tests exactly one boolean variable. Instrument $\text{checkItExpand}()$ to record which edges are traversed. (“print” statements are fine for this.)
- Derive a GACC test set $T_1$ for $\text{checkIt}()$. Derive an Edge Coverage test set $T_2$ for $\text{checkItExpand}()$. Build $T_2$ so that it does not satisfy GACC on the predicate in $\text{checkIt}()$.
- Run both $T_1$ and $T_2$ on both $\text{checkIt}()$ and $\text{checkItExpand}()$.

6. Answer the following questions for the method $\text{twoPred}()$ below:
8.3. **STRUCTURAL LOGIC COVERAGE OF PROGRAMS**

```java
public String twoPred (int x, int y)
{
    boolean z;
    if (x < y)
        z = true;
    else
        z = false;
    if (z && x+y == 10)
        return "A";
    else
        return "B";
}
```

- Identify test inputs for `twoPred()` that achieve Restricted Active Clause Coverage (RACC).
- Identify test inputs for `twoPred()` that achieve Restricted Inactive Clause Coverage (RICC).

7. Answer the following questions for the program fragments below:

<table>
<thead>
<tr>
<th>fragment P:</th>
<th>fragment Q:</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (A</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>m();</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>return;</td>
<td>if (A)</td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>m();</td>
<td></td>
</tr>
<tr>
<td>return;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>if (B)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>m();</td>
<td></td>
</tr>
<tr>
<td>return;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>if (C)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>m();</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

- Give a GACC test set for fragment P. (Note that GACC, CACC, and RACC yield identical test sets for this example.)
- Does the GACC test set for fragment P satisfy Edge Coverage on fragment Q?
- Write down an Edge Coverage test set for fragment Q. Make your test set include as few tests from the GACC test set as possible.

8. For the `index()` program in Chapter 7, complete the test sets for the following coverage criteria by filling in the “don’t care” values, ensuring reachability, and deriving the expected output. Download the program, compile it, and run it with your resulting test cases to verify correct outputs.

- Predicate Coverage (PC)
- Clause Coverage (CC)
- Combinatorial Coverage (CoC)
- Correlated Active Clause Coverage (CACC)

9. For the `Quadratic` program in Chapter 7, complete the test sets for the following coverage criteria by filling in the “don’t care” values, ensuring reachability, and deriving the expected output. Download the program, compile it, and run it with your resulting test cases to verify correct outputs.
10. The following program **TriTyp** is an old and well used example from the unit testing literature. **TriTyp** is used as a teaching tool for the same reasons it has staying power in the literature: the problem is familiar; the control structure is interesting enough to illustrate most issues; and it does not use language features that make this analysis really hard, such as loops and indirect references. This version of **TriTyp** is more complicated than some, but that helps illustrate the concepts. **TriTyp** is a simple triangle classification program. Line numbers were added to allow us to refer to specific decision statements in the the answers.

Use **TriTyp**, a numbered version of which is available on the book web site, to answer the questions below. Only the **triang()** method is considered.

(a) List all predicates in the **triang()** method. Index them by the line numbers in the program listing.

(b) Compute reachability for each of **triang()**’s predicates. You may abbreviate the input variables as **S1**, **S2**, and **S3**.

(c) Many of the reachability predicates contain an internal variable (**triOut**). Resolve the internal variable in terms of input variables. That is, determine what values the input variables need to have to give **triOut** each possible value.

(d) Rewrite the reachability predicates by solving for **triOut**. That is, the reachability predicates should be completely in terms of the input variables.

(e) Find values for each predicate to satisfy predicate coverage (PC).

(f) Find values for each predicate to satisfy clause coverage (CC).

(g) Find values for each predicate to satisfy correlated active clause coverage (CACC).

11. (Challenging!) For the **TriTyp** program, complete the test sets for the following coverage criteria by filling in the “don’t care” values, ensuring reachability, and deriving the expected output. Download the program, compile it, and run it with your resulting test cases to verify correct outputs.

- Predicate Coverage (PC)
- Clause Coverage (CC)
- Combinatorial Coverage (CoC)
- Correlated Active Clause Coverage (CACC)

12. Consider the **GoodFastCheap** class, available on the book web site. This class implements the old engineering joke: “Good, Fast, Cheap: Pick any two!”

(a) Develop tests that achieve RACC for the predicate in the **isSatisfactory()** method. Implement these tests in JUnit.

(b) Suppose we refactor the **isSatisfactory()** method as shown below:
8.4 Specification-based Logic Coverage

Software specifications, both formal and informal, appear in a variety of forms and languages. They almost invariably include logical expressions, allowing the logic coverage criteria to be applied. We start by looking at their application to simple preconditions on methods.

Programmers often include preconditions as part of their methods. The preconditions are sometimes written as part of the design, and sometimes added later as documentation. Specification languages typically make preconditions explicit with the goal of analyzing the preconditions in the context of an invariant. A tester may consider developing the preconditions specifically as part of the testing process if preconditions do not exist. For a variety of reasons, including defensive programming and security, transforming preconditions into exceptions is common practice. In brief, preconditions are common and rich sources of predicates in specifications, and so we focus on them here. Of course, other specification constructs, such as postconditions and invariants, also are rich sources of complex predicates.

Consider the cal() method in Figure 8.6. The method lists explicit preconditions in natural language. These can be translated into predicate form as follows:

\[
\begin{align*}
\text{month1} & \geq 1 \land \text{month1} \leq 12 \land \text{month2} \geq 1 \land \text{month2} \leq 12 \land \text{month1} \leq \text{month2} \\
\land \text{day1} & \geq 1 \land \text{day1} \leq 31 \land \text{day2} \geq 1 \land \text{day2} \leq 31 \land \text{year} \geq 1 \land \text{year} \leq 10000
\end{align*}
\]

The comment about day1 and day2 being in the same year can be safely ignored, because that prerequisite is enforced syntactically by the fact that only one parameter appears for year. It is probably also clear that these preconditions are not complete. Specifically, a day of 31 is valid only for some months. This requirement should be reflected in the specifications or in the program.

This predicate has a very simple structure. It has eleven clauses (which sounds like a lot!) but the only logical operator is "and." Satisfying Predicate Coverage for cal() is simple—all
public static int cal (int month1, int day1, int month2, 
int day2, int year) 
{
//***********************************************************
// Calculate the number of Days between the two given days in
// the same year.
// preconditions: day1 and day2 must be in same year
// 1 <= month1, month2 <= 12
// 1 <= day1, day2 <= 31
// day2 >= day1
// month1 <= month2
// The range for year: 1 ... 10000
//***********************************************************
int numDays;
if (month2 == month1) // in the same month
    numDays = day2 - day1;
else
{
    // Skip month 0.
    int daysIn[] = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    // Are we in a leap year?
    int m4 = year % 4;
    int m100 = year % 100;
    int m400 = year % 400;
    if ((m4 != 0) || ((m100 == 0) && (m400 != 0)))
        daysIn[2] = 28;
    else
        daysIn[2] = 29;
    // start with days in the two months
    numDays = day2 + (daysIn[month1] - day1);
    // add the days in the intervening months
    for (int i = month1 + 1; i <= month2-1; i++)
        numDays = daysIn[i] + numDays;
}
return (numDays);
}

Figure 8.6: Calendar method.

clauses need to be true for the true case and at least one clause needs to be false for the 
false case. So (month1 = 4, month2 = 4, day1 = 12, day2 = 30, year = 1961) satisfies 
the true case, and the false case is satisfied by violating the clause month1 <= month2, 
with (month1 = 6, month2 = 4, day1 = 12, day2 = 30, year = 1961). Clause coverage 
requires all clauses to be true and false. We might try to satisfy this requirement with only 
two tests, but some clauses are related and cannot both be false at the same time. For 
example, month1 cannot be less than 1 and greater than 12 at the same time. The true 
test for Predicate Coverage allows all clauses to be true, then we use the following tests to 
make each clause false: (month1 = −1, month2 = −2, day1 = 0, day2 = 0, year = 0) and 
(month1 = 13, month2 = 14, day1 = 32, day2 = 32, year = 10500).

We must first find how to make each clause determine the predicate to apply the active 
clause coverage criteria. This turns out to be simple with disjunctive normal form predicates— 
all we have to do is make each minor clause true. To find the remaining tests, each other 
clause is made to be false in turn. Therefore, CACC (also RACC and GACC) is satisfied by
the tests that are specified in Table 8.5. (To save space, we use abbreviations of the variable names.)

**Exercises, Section 8.4.**

1. Consider the `remove()` method from the Java `Iterator` interface. The `remove()` method has a complex precondition on the state of the `Iterator`, and the programmer can choose to detect violations of the precondition and report them as `IllegalStateException`.

   (a) Formalize the precondition.

   (b) Find (or write) an implementation of an `Iterator`. The Java `Collection` classes are a good place to search.

   (c) Develop and run CACC tests on the implementation.

### 8.5 Logic Coverage of Finite State Machines

Chapter 7 discussed the application of graph coverage criteria to Finite State Machines. Recall that FSMs are graphs with nodes that represent states and edges that represent transitions. Each transition has a pre-state and a post-state. FSMs usually model behavior of software and can be more or less formal and precise, depending on the needs and inclinations of the developers. This text views FSMs in the most generic way, as graphs. Differences in notations are considered only in terms of the effect they have on applying the criteria.

The most common way to apply logic coverage criteria to FSMs is to use logical expressions from the transitions as predicates. In the Elevator example in Chapter 7, the trigger and thus the predicate is `openButton = pressed`. Tests are created by applying the criteria from Section 8.1.1 to these predicates.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Truth Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_1 \geq 1$</td>
<td>$T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$</td>
</tr>
<tr>
<td>$m_1 \leq 12$</td>
<td>$T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$</td>
</tr>
<tr>
<td>$m_1 \leq m_2$</td>
<td>$T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$</td>
</tr>
<tr>
<td>$d_1 \geq 1$</td>
<td>$T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$</td>
</tr>
<tr>
<td>$d_1 \leq 31$</td>
<td>$T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$ $T$</td>
</tr>
</tbody>
</table>
Consider the example in Figure 8.7. This FSM models the behavior of the memory seat in a car (Nissan Maxima 2012). The memory seat has two configurations for two separate drivers and controls the side mirrors ($\text{sideMirrors}$), the vertical height of the seat ($\text{seatBottom}$), the horizontal distance of the seat from the steering wheel ($\text{seatBack}$), and the lumbar support ($\text{lumbar}$). The intent is to remember the configurations so that the drivers can conveniently switch configurations with the press of a button. Each state in the figure has a number for efficient reference.

The initial state of the FSM is whichever configuration it was in when the system was last shut down, either Driver 1, Driver 2, or Modified Configuration. The drivers can modify the configuration by changing one of the four controls; changing the side mirrors, moving the seat backwards or forwards, raising or lowering the seat, or modifying the lumbar support (triggering events). These controls work only if the $\text{ignition}$ is on (a guard). The driver can also change to the other configuration by pressing either $\text{Button1}$ or $\text{Button2}$ when the $\text{ignition}$ is on. In these cases, the guards allow the configuration to be changed only if the Gear is in Park or the $\text{ignition}$ is off. These are safety constraints, because it would be dangerous to allow the driver’s seat to go flying around when the car is moving.

When the driver changes one of the controls, the memory seat is put into the modified configuration state. The new state can be saved by simultaneously pressing the $\text{Reset}$ button and either $\text{Button1}$ or $\text{Button2}$ when the ignition is on. The new configuration is saved permanently when
8.5. LOGIC COVERAGE OF FINITE STATE MACHINES

the ignition is turned off.

This type of FSM provides an effective model for testing software, although several issues must be understood and dealt with when creating predicates and then test values. Guards are not always explicitly listed as conjuncts, but they are conjuncts in effect and so should be combined with the triggers using the AND operator. In some specification languages, most notably SCR, the triggers actually imply two values. In SCR, if an event is labeled as triggering, it means that the value of the resulting expression must explicitly change. This implies two values—a before value and an after value, and is modeled by introducing a new variable. For example, in the memory seat example, the transition from New Configuration Driver 1 to Driver 1 Configuration is taken when the ignition is turned off. If that is a triggering transition in the SCR sense, then the predicate needs to have two parts: \( \text{ignition} = \text{on} \land \text{ignition}' = \text{off}. \)

The transitions from Modified Configuration to the two New Configuration states demonstrate another issue. The two buttons Reset and Button1 (or Button2) must be pressed simultaneously. In practical terms for this example, we would like to test for what happens when one button is pressed slightly prior to the other. Unfortunately, the mathematics of logical expressions used in this chapter do not have an explicit way to represent this requirement, thus it is not handled explicitly. The two buttons are connected in the predicate with the AND operator. In fact, this is a simple example of the general problem of timing, and needs to be addressed in the context of real-time software.

The predicates for the memory seat example are in Table 8.6 (using the state numbers from Figure 8.7).

The tests to satisfy the various criteria are fairly straightforward and are left to the exercises. Several issues must be addressed when choosing values for test cases. The first is that of reachability; the test case must include prefix values to reach the pre-state. For most FSMs, this is just a matter of finding a path from an initial state to the pre-state (a depth first search can be used), and the predicates associated with the transitions are solved to produce inputs. The memory seat example has three initial states, and the tester cannot control which one is entered because it depends on the state the system was in when it was last shut down. In this case, however, an obvious solution presents itself. We can begin every test by putting the Gear in park and pushing Button 1 (part of the prefix). If the system is in the Driver 2 or the Modified Configuration state, these inputs will cause the system to transition to the Driver 1 state. If the system is in the Driver 1 state, these inputs will have no effect. In all three cases, the system will effectively start in the Driver 1 state.

To automate the tests, we must also define a complete execution through the FSM. Some FSMs also have exit states that must be reached with postfix values. Finding these values is essentially the same as finding prefix values; that is, finding a path from the post-state to a final state. The memory seat example does not have an exit state, so this step can be skipped. We also need a way to see the results of the test case (verification values). This might be possible by giving an input to the program to print the current state, or causing some other output that is dependent on the state. The exact form and syntax this takes depends on the implementation, and so it cannot be finalized until the input-output behavior syntax of the software is designed.

One major advantage of this form of testing is determining the expected output. This might be possible by giving an input to the program to print the state simply the post-state of the transition for the test case values that cause the transition to be true, and the pre-state for the test case values that cause the transition to be false (the system should remain in the current state). The only
Table 8.6: Predicates from memory seat example.

<table>
<thead>
<tr>
<th>Pre-state</th>
<th>Post-state</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>$Button_2 \land (\text{Gear} = \text{Park} \lor \text{ignition} = \text{off})$</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>sideMirrors $\land$ ignition = on</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>seatBottom $\land$ ignition = on</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>lumbar $\land$ ignition = on</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>seatBack $\land$ ignition = on</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>$Button_1 \land (\text{Gear} = \text{Park} \lor \text{ignition} = \text{off})$</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>sideMirrors $\land$ ignition = on</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>seatBottom $\land$ ignition = on</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>lumbar $\land$ ignition = on</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>seatBack $\land$ ignition = on</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>$Button_1 \land (\text{Gear} = \text{Park} \lor \text{ignition} = \text{off})$</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>$Button_2 \land (\text{Gear} = \text{Park} \lor \text{ignition} = \text{off})$</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>Reset $\land$ Button1 $\land$ ignition = on</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>Reset $\land$ Button2 $\land$ ignition = on</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>ignition = off</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>sideMirrors $\land$ ignition = on</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>seatBottom $\land$ ignition = on</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>lumbar $\land$ ignition = on</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>seatBack $\land$ ignition = on</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>ignition = off</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>sideMirrors $\land$ ignition = on</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>seatBottom $\land$ ignition = on</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>lumbar $\land$ ignition = on</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>seatBack $\land$ ignition = on</td>
</tr>
</tbody>
</table>

exception to this rule is that occasionally a false predicate might coincidentally be a true predicate for another transition, in which case the expected output should be the post-state of the alternate transition. This situation can be recognized automatically. Also, if a transition is from a state back to itself, then the pre-state and the post-state are the same and the expected output is the same whether the transition is true or false.

The final problem is that of converting a test case (composed of prefix values, test case values, postfix values, and expected results) into an executable test script. The potential problem here is that the variable assignments for the predicates must be converted into inputs to the software. This has been called the mapping problem with FSMS and is analogous to the internal variable problem of Section 8.3. Sometimes this step is a simple syntactic rewriting of predicate assignments ($Button_1$ to program input $button1$). Other times, the input values can be directly encoded as method calls and embedded into a program (for example, $Button_1$ becomes $pressButton1()$). At other times, however, this problem is much greater and can involve turning seemingly small inputs at the FSM modeling level into long sequences of inputs or method calls. The exact situation depends on the software implementation; thus a general solution to this problem is elusive at best.
8.6. **BIBLIOGRAPHIC NOTES**

Exercises, Section 8.5.

1. For the *Memory Seat* finite state machine, complete the test sets for the predicate coverage criterion (PC) by satisfying the predicates, ensuring reachability, and computing the expected output.

2. For the *Memory Seat* finite state machine, complete the test sets for the correlated active clause coverage criterion (CACC) by satisfying the predicates, ensuring reachability, and computing the expected output.

3. For the *Memory Seat* finite state machine, complete the test sets for the general inactive active clause coverage criterion (GICC) by satisfying the predicates, ensuring reachability, and computing the expected output.

4. Redraw Figure 8.7 to have fewer transitions, but more clauses. Specifically, nodes 1, 2, 4, and 5 each has four transitions to node 3. Rewrite these transitions to have only one transition from each of nodes 1, 2, 4, and 5 to node 3, and the clauses are connected by ORs. Then derive tests to satisfy CACC for the four resulting predicates. (You can omit the other predicates.) How do these tests compare with the tests derived from the original graph?

5. Consider the following deterministic finite state machine:

<table>
<thead>
<tr>
<th>Current State</th>
<th>Condition</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>$a \lor b$</td>
<td>Active</td>
</tr>
<tr>
<td>Active</td>
<td>$a \land b$</td>
<td>Idle</td>
</tr>
<tr>
<td>Active</td>
<td>$\neg b$</td>
<td>WindDown</td>
</tr>
<tr>
<td>WindDown</td>
<td>$a$</td>
<td>Idle</td>
</tr>
</tbody>
</table>

(a) Draw the finite state machine.

(b) This machine does not specify which conditions cause a state to transition back to itself. However, these conditions can be derived from the existing conditions. Derive the conditions under which each state will transition back to itself.

(c) Find CACC tests for each transition from the Active state.

6. Pick a household appliance such as a watch, calculator, microwave, VCR, clock-radio or programmable thermostat. Draw the FSM that represents your appliance’s behavior. Derive tests to satisfy Predicate Coverage, Correlated Active Clause Coverage, and General Inactive Clause Coverage.

7. Implement the memory seat FSM. Design an appropriate input language to your implementation and turn the tests derived for question 1, 2, or 3 into test scripts. Run the tests.

8.6 Bibliographic Notes

The active clause criteria seem to have their beginnings in Myers’ 1979 book [34]. A more accessible paper is by Zhu [42]. He defined decision and condition coverage, which this
book calls predicate and clause coverage. Chilenski and Miller later used these definitions as a conceptual basis for MCDC [9, 39]. The definitions as originally given correspond to GACC in this book and did not address whether minor clauses had to have the same value for both values of the major clause. Chilenski also emphasized that the abbreviation should be “MCDC,” not “MC/DC,” and he has never put the ‘/’ in the middle [11]. Most members of the aviation community originally interpreted MCDC to mean that the values of the minor clauses had to be the same, an interpretation that is called “unique-cause MCDC” [11]. Unique-cause MCDC corresponds to our RACC. More recently, the FAA has accepted the view that the minor clauses can differ, which is called “masking MCDC” [10]. Masking MCDC corresponds to our CACC. Our previous paper [2] clarified the definitions in the form used in this book and introduced the “ACC” terms.

The inactive clause criteria are adapted from the RC/DC method of Vilkomir and Bowen [40].

The result that the internal variable problem is formally undecidable is from Offutt’s PhD dissertation [13, 35]. The problem is of primary importance in the automatic test data generation literature [6, 7, 13, 14, 18, 20, 22, 29, 30, 33, 38, 36].

Jasper et al. presented techniques for generating tests to satisfy MCDC [21]. They took the definition of MCDC from Chilenski and Miller’s paper with the “default” interpretation that the minor clauses must be the same for both values of the major clauses. They went on to modify the interpretation so that if two clauses are coupled, which implies it is impossible to satisfy determination for both, the two clauses are allowed to have different values for the minor clauses. The fact that different values are allowed only when clauses are coupled puts their interpretation of MCDC between the RACC and CACC of this book.

Weyuker, Goradia and Singh presented techniques for generating test data for software specifications that are limited to boolean variables [41]. The techniques were compared in terms of the ability of the resulting test cases to kill mutants (introduced in Chapter 9) [12, 13]. The results were that their technique, which is closely related to MCDC, performed better than any of the other techniques. Weyuker et al. incorporated syntax as well as meaning into their criteria. They presented a notion called meaningful impact, which is related to the notion of determination, but which has a syntactic basis rather than a semantic one.

Kuhn investigated methods for generating tests to satisfy various decision-based criteria, including MCDC tests [31]. He used the definition from Chilenski and Miller [9, 39], and proposed the boolean derivative to satisfy MCDC. In effect, this interpreted MCDC in such a way to match CACC.

Dupuy and Leveson’s 2000 paper evaluated MCDC experimentally [15]. They presented results from an empirical study that compared pure functional testing with functional testing augmented by MCDC. The experiment was performed during the testing of the attitude control software for the HETE-2 (High Energy Transient Explorer) scientific satellite. The definition of MCDC from their paper is the traditional definition given in the FAA report and Chilenski and Miller’s paper: “Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible
outcomes at least once, and each condition has been shown to affect that decision outcome independently. A condition is shown to affect a decision’s outcome independently by varying just that decision while holding fixed all other possible conditions.”

Note the misstatement in last line: “varying just that decision” should be “varying just that condition”. This does not say that the decision has a different value when the condition’s value changes. “Holding fixed” can be assumed to imply that the minor clauses cannot change with different values for the major clause (that is, RACC, not CACC).

The full predicate method of Offutt, Liu, Abdurazik and Ammann [37] explicitly relaxes the requirement that the major clauses have the same value as the predicate. This is equivalent to CACC and almost the same as masking MCDC.

Jones and Harrold developed a method for reducing the regression tests that were developed to satisfy MCDC [23]. They defined MCDC as follows: “MC/DC is a stricter form of decision (or branch) coverage. ... MC/DC requires that each condition in a decision be shown by execution to independently affect the outcome of the decision.” This is taken directly from Chilenski and Miller’s original paper, and their interpretation of the definition is the same as CACC.

SCR was first discussed by Henninger [19] and its use in model checking and testing was introduced by Atlee [3, 4].

The cost of active clause coverage was originally reported by Chilenski and Miller [9, 39], who claimed that the minimum test set size for MCDC is $n + 1$, and the maximum is $2n$. In his dissertation, Kaminski [24] confirmed that in general, MCDC and the RACC criteria need at least $n + 1$ tests, but always fewer than $2n$ tests. Kaminski also showed that $n + 1$ is enough when $n < 4$, because of the overlap among tests, but the number of tests needed for some functions gets closer to $2n$ as $n$ grows.

The statement that “the vast majority of predicates in real programs have only one clause” is due to Durelli et al. [16], who counted the number of clauses in 400,811 predicates in 63 open-source Java programs. They found that 88.02% of the predicates had only one clause, 9.97% had two clauses, 1.29% had three clauses, 0.47% had four clauses, 0.11% had five clauses, and less than 0.15% had more than five clauses.

The method of determining $p_c$ given in this book uses the boolean derivative developed by Akers [1]. Both Chilenski and Richey [10] and Kuhn [31] applied Akers’s derivative to exactly the problem given in this chapter. The other methods are the pairs table method of Chilenski and Miller and the tree method, independently discovered by Chilenski and Richey [10] and Offutt et al. [37]. The tree method can be thought of as implementing the boolean derivative method in a procedural way.

Ordered Binary Decision Diagrams (OBDDs) offer another way to determine $p_c$. In particular, consider any OBDD in which clause $c$ is ordered last. Then any path through the OBDD that reaches a node labeled $c$ (there will be exactly zero, one, or two such nodes) is, in fact, an assignment of values to the other variables so that $c$ determines $p$. Continuing the path on to the constants $T$ and $F$ yields a pair of tests satisfying RACC with respect to $c$. Selecting two different paths that reach the same node labeled $c$, and then extending each so that one reaches $T$ and the other reaches $F$ yields a pair of tests that satisfy CACC,
but not $RACC$, with respect to $c$. Finally, if two nodes are labeled $c$, then it is possible to satisfy $GACC$ but not $CACC$ with respect to $c$ by selecting paths to each of the two nodes labeled $c$, extending one path by choosing $c$ true, and extending the other by choosing $c$ false. Both paths will necessarily end up in the same node, namely, either $T$ or $F$. ICC tests with respect to $c$ can be derived by considering paths to $T$ and $F$ in the OBDD where the paths do not include variable $c$. The attractive aspect of using OBDDs to derive ACC or ICC tests is that a variety of existing tools can handle a relatively large number of clauses. The unattractive aspect is that for a predicate with $N$ clauses, $N$ different OBDDs for a given function are required, since the clause being attended to needs to be the last in the ordering. To the knowledge of the authors, the use of OBDDs to derive ACC or ICC tests does not appear in the literature.

Beizer’s [5] book includes a chapter on DNF testing, including a variant of IC coverage for $f$, but not $\bar{f}$, and an extensive development of Karnaugh maps. Kuhn [31] developed the first fault detection relations; this work was greatly expanded by Yu, Lau and Chen, who developed much of the key material relating DNF coverage criteria to fault detecting ability. Two good papers to begin study of this topic are by Chen and Lau [8], which develops MUMCUT, and Lau and Yu [32], which is the source for the fault class hierarchy shown in Figure 8.2. Kaminski et al. [25, 27] developed a minimal version of MUMCUT, and then later [26] used optimization techniques to develop a minimum version of MUMCUT. Kaminski, Offutt, and Ammann [28] presented the results with respect to test set size and fault detection for RACC vs. MUMCUT. Gargantini and Fraser [17] developed a different algorithm for reducing MUMCUT sets. In personal communications, Greg Williams and Gary Kaminski provided the authors with valuable assistance in organizing and expanding the DNF fault detection material.
Bibliography


active clause coverage, 13
active clause coverage (ACC)
  ambiguity, 6–7
  cost, 6, 59
  definition, 6
automatic test data generation, 36
Best Effort Touring, 11
Boolean algebra laws, 14–16
  associativity, 16
  commutativity, 16
  distributive, 16
  Identity, 15
  negation, 15
  xor, 15
clause
  definition, 3
CNF, 24
condition coverage, 4, 57
Conjunctive Normal Form, 24
controllability, 35–36, 38, 48
criteria
  CACC, 7–12, 14, 20–21, 43–47, 50, 52, 53, 57–60
  CC, 4, 12, 18–20, 41, 50, 52
  CoC, 5, 12, 19
  CUTPNFP, 12, 30–31, 34
  GACC, 7, 12, 19–21, 52, 58, 60
  GICC, 10, 57
  IC, 12, 25–26, 60
  MNFP, 31, 34
  MUMCUT, 31, 60
  MUTP, 28–30, 33
  PC, 4–7, 10, 12, 18, 21, 26, 35, 39–41, 46–47, 50–52, 57
  RACC, 8–12, 14, 20–21, 30, 34, 52, 58–60
  RICC, 10–12
decision coverage, 4, 57
determination
  definition, 5
  examples, 6–10, 19–21, 33–34, 43–45, 52–53
  explanation, 5, 12–14
direct definitional method, 12–16
Disjunctive Normal Form, 24
DNF, 24
example Java
  cal(), 52
  checkIt(), 48
  GoodFastCheap, 50, 51
  GoodFastCheapRefactored, 51
  Thermostat, 37, 40, 42, 48
  TriTyp, 50
  twoPred(), 48
expected results, 55, 56
fault, 30
  specific types, 27–31
Federal Aviation Administration, 1, 58, 59
implicant, 24
  prime, 26
  redundant, 26
inactive clause coverage, 11, 13, 22, 49, 58
inactive clause coverage (ICC)
  ambiguity, 10
  definition, 10
  examples, 10
infeasible, 11
  CACC and RACC, 8–9
infection, 1
internal variable, 35, 38, 39, 56, 58
  solving for, 38–39
Karnaugh maps, 26–27, 31–34
INDEX

literal, 24
logic coverage, 1

major clause, 5–8, 10–14, 43–45
   definition, 5
mapping problem
   definition, 56
MCDC, 6, 58, 59
minor clause, 5–8, 10–14, 43–45
   definition, 5

near false point, 27

postfix values, 55, 56
predicate, 3–4
   definition, 2
   examples, 3
   minimal, 27
prefix values, 35, 55, 56
proper subterm, 26

reachability, 1, 35, 38, 41, 49, 55, 57
RIPR, 1

safety constraints, 54
security, 51
semantic logic coverage, 1–21
subsumption, 11
   clause and predicate coverage, 4–5
   graph, 28–31
   predicate coverage and ACC, 7, 47
   predicate coverage and IC, 26
syntactic logic coverage, 1, 2, 24–35

term, 24
test case values, 56
test engineer, 10
test script, 56
tour
   best effort, 11

undecidable, 11, 35, 58
unique true point, 27
unit test, 50

verification values, 55