```
fragment P:
  if (A || B || C)
  {
    m();
  }
  return;
```

```
fragment Q:
  if (A)
  {
    m();
    return;
  }
  if (B)
  {
    m();
    return;
  }
  if (C)
  {
    m();
  }
```

- 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.

4. (**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)
5. Repeat the prior exercise, but for the TestPat program in Chapter 2.
6. Repeat the prior exercise, but for the Quadratic program in Chapter 2.

3.4 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,

```
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
//          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 3.4. Calendar method.

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 3.4. The method lists explicit preconditions in natural language. These can be translated into predicate form as follows:

$$month1 >= 1 \wedge month1 <= 12 \wedge month2 >= 1 \wedge month2 <= 12 \wedge month1 <= month2$$
$$\wedge\, day1 >= 1 \wedge day1 <= 31 \wedge day2 >= 1 \wedge day2 <= 31 \wedge year >= 1 \wedge year <= 10000$$

The comment about *day*1 and *day*2 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 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.

Table 3.6. Correlated active clause coverage for cal() preconditions

		$m1 \geq 1$	$m1 \leq 12$	$m2 \geq 1$	$m2 \leq 12$	$m1 \leq m2$	$d1 \geq 1$	$d1 \leq 31$	$d2 \geq 1$	$d2 \leq 31$	$y \geq 1$	$y \leq 10000$
1.	$m1 \geq 1 = T$	T	t	t	t	t	t	t	t	t	t	t
2.	$m1 \geq 1 = F$	F	t	t	t	t	t	t	t	t	t	t
3.	$m1 \leq 12 = F$	t	F	t	t	t	t	t	t	t	t	t
4.	$m2 \geq 1 = F$	t	t	F	t	t	t	t	t	t	t	t
5.	$m2 \leq 12 = F$	t	t	t	F	t	t	t	t	t	t	t
6.	$m1 \leq m2 = F$	t	t	t	t	F	t	t	t	t	t	t
7.	$d1 \geq 1 = F$	t	t	t	t	t	F	t	t	t	t	t
8.	$d1 \leq 31 = F$	t	t	t	t	t	t	F	t	t	t	t
9.	$d2 \geq 1 = F$	t	t	t	t	t	t	t	F	t	t	t
10.	$d2 \leq 31 = F$	t	t	t	t	t	t	t	t	F	t	t
11.	$y \geq 1 = F$	t	t	t	t	t	t	t	t	t	F	t
12.	$y \leq 10000 = F$	t	t	t	t	t	t	t	t	t	t	F

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, *monthl* 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: $(monthl = -1, month2 = -2, day1 = 0, day2 = 0, year = 0)$ and $(monthl = 13, month2 = 14, day1 = 32, day2 = 32, year = 10500)$.

We must first find how to make each clause determine the predicate to apply the ACC 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 3.6. (To save space, we use abbreviations of the variable names.)

EXERCISES

Section 3.4.

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.

1. Formalize the precondition.
2. Find (or write) an implementation of an Iterator. The Java Collection classes are a good place to search.
3. Develop and run CACC tests on the implementation.

3.5 LOGIC COVERAGE OF FINITE STATE MACHINES

Chapter 2 discussed the application of graph coverage criteria to finite state machines (FSMs). 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 the 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 2, the trigger and thus the predicate is *openButton = pressed*. Tests are created by applying the criteria from Section 3.2 to these predicates.

Consider the example in Figure 3.5. This FSM models the behavior of the memory seat in a car (Lexus 2003 ES300). The memory seat has two configurations for two separate drivers and controls the side mirrors (sideMirrors), the vertical height of the seat (seatBottom), the horizontal distance of the seat from the steering wheel (seatBack), and the lumbar support (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.

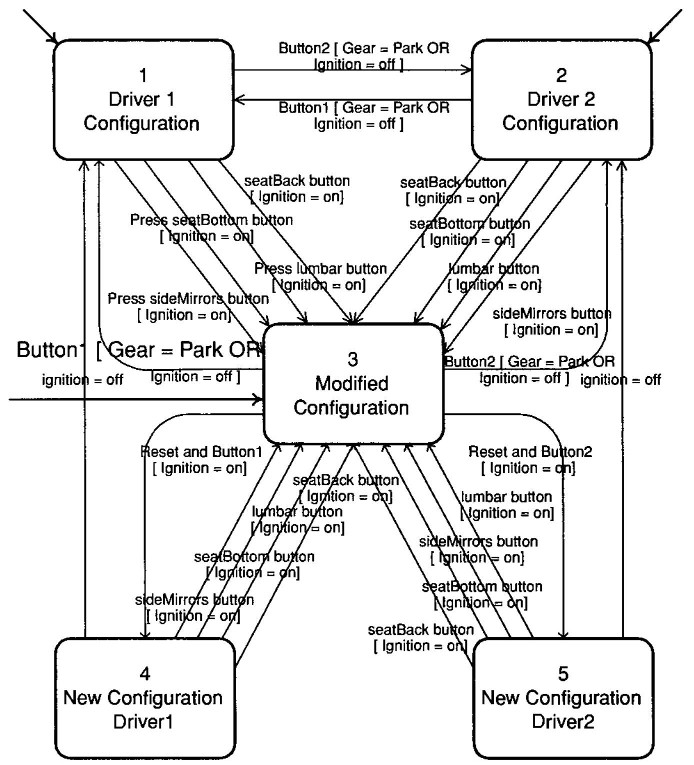
1
Driver 1
Configuration
2
Driver 2
Configuration
3
Modified
Configuration
4
New Configuration
Driver1
5
New Configuration
Driver2
Button2 [Gear = Park OR Ignition = off]
Button1 [Gear = Park OR Ignition = off]
seatBack button [Ignition = on]
Press seatBottom button [Ignition = on]
Press lumbar button [Ignition = on]
Press sideMirrors button [Ignition = on]
Button1 [Gear = Park OR ignition = off Ignition = off]
seatBack button [Ignition = on]
seatBottom button [Ignition = on]
lumbar button [Ignition = on]
sideMirrors button [Ignition = on]
Button2 [Gear = Park OR Ignition = off] ignition = off
Reset and Button1 [Ignition = on]
seatBack button [Ignition = on]
lumbar button [Ignition = on]
seatBottom button [Ignition = on]
sideMirrors button [Ignition = on]
Reset and Button2 [Ignition = on]
lumbar button [Ignition = on]
sideMirrors button [Ignition = on]
seatBottom button [Ignition = on]
seatBack button [Ignition = on]

Figure 3.5. FSM for a memory car seat – Lexus 2003 ES300.

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 ignition is on (a guard). The driver can also change to the other configuration by pressing either Button1 or Button2 when the ignition is on. In these cases, the guards allow the configuration to be changed only if the Gear is in Park or the 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 Reset button and either Button1 or Button2 when the ignition is on. The new configuration is saved permanently when 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

Table 3.7. Predicates from memory seat example

Pre-state	Post-state	Predicate
1	2	*Button2* $\wedge$ (*Gear* = *Park* $\vee$ *ignition* = *off*)
1	3	*sideMirors* $\wedge$ *ignition* = *on*
1	3	*seatButton* $\wedge$ *ignition* = *on*
1	3	*lumbar* $\wedge$ *ignition* = *on*
1	3	*seatBack* $\wedge$ *ignition* = *on*
2	1	*Button1* $\wedge$ (*Gear* = *Park* $\vee$ *ignition* = *off*)
2	3	*sideMirors* $\wedge$ *ignition* = *on*
2	3	*seatButton* $\wedge$ *ignition* = *on*
2	3	*lumbar* $\wedge$ *ignition* = *on*
2	3	*seatBack* $\wedge$ *ignition* = *on*
3	1	*Button1* $\wedge$ (*Gear* = *Park* $\vee$ *ignition* = *off*)
3	2	*Button2* $\wedge$ (*Gear* = *Park* $\vee$ *ignition* = *off*)
3	4	*Reset* $\wedge$ *Button1* $\wedge$ *ignition* = *on*
3	5	*Reset* $\wedge$ *Button2* $\wedge$ *ignition* = *on*
4	1	*ignition* = *off*
4	3	*sideMirors* $\wedge$ *ignition* = *on*
4	3	*seatButton* $\wedge$ *ignition* = *on*
4	3	*lumbar* $\wedge$ *ignition* = *on*
4	3	*seatBack* $\wedge$ *ignition* = *on*
5	2	*ignition* = *off*
5	3	*sideMirors* $\wedge$ *ignition* = *on*
5	3	*seatButton* $\wedge$ *ignition* = *on*
5	3	*lumbar* $\wedge$ *ignition* = *on*
5	3	*seatBack* $\wedge$ *ignition* = *on*

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: *ignition* = *on* $\wedge$ *ignition'* = *off*. *ignition'* is the after value.

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 3.7 (using the state numbers from Figure 3.5).

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 (using a depth first search), 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.

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. It is 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 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 output) 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 3.3. Sometimes this step is a simple syntactic rewriting of predicate assignments (Button1 to program input *button1*). Other times, the input values can be directly encoded as method calls and embedded into a program (for example, Button1 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.

EXERCISES

Section 3.5.

1. For the Memory Seat finite state machine, complete the test sets for the following coverage criteria by satisfying the predicates, ensuring reachability, and computing the expected output.
   - Predicate coverage
   - Correlated active clause coverage
   - General inactive clause coverage