The coverage domain consists of all paths that traverse each of the three loops zero times *and* once in the same or different executions of the program. We leave the enumeration of all such paths in Program P6.5 as an exercise. For this example, let us consider path p that begins execution at line 1, reaches the outermost `while` at line 10 , then the first `if` at line 12, followed by the statements that compute the factorial starting at line 20, and then the code to compute the exponential starting at line 13. For this example, we do not care what happens after the exponential has been computed and output has been generated.

Our tricky path is traversed when the program is launched and the first input request is to compute the factorial of a number followed by a request to compute the exponential. It is easy to verify that the sequence of requests in T does not exercise p. Therefore, T is inadequate with respect to the path coverage criterion.

To cover p, we construct the following test:

$$T' = \{< request = 2, x = 4 >, < request = 1, x = 2, y = 3 >, \\ < request = 3 >\}$$

When the values in T' are input to our example program in the sequence given, the program correctly outputs 24 as the factorial of 4 but incorrectly outputs 192 as the value of 2^3. This happens because T' traverses our tricky path that makes the computation of the exponentiation begin without initializing `product`. In fact, the code at line 14 begins with the value of `product` set to 24.

Note that in our effort to increase the path coverage, we constructed T'. Execution of the test program on T did cover a path that was not covered earlier and revealed an error in the program.

6.1.6 Single and Multiple Executions

In Example 6.9, we constructed two test sets T and T'. Note that both T and T' contain three tests one for each value of variable `request`. Should T be considered a single test or a set of three tests? The same question applies also to T'. The answer depends on how the values in T are input to the test program. In our example, we assumed that all three tests, one for each value of `request`, are input in a sequence during *a single execution of the test program.* Hence, we consider T as a test set containing one test case and write it as follows:

$$T = \left\{ \begin{array}{l} t_1: << request = 1,\ x = 2,\ y = 3 > \rightarrow \\ \quad < request = 2,\ x = 4 > \qquad \rightarrow < request = 3 >> \end{array} \right\}$$

Note the use of the outermost angular brackets to group all values in a test. Also, the right arrow ($\rightarrow$) indicates that the values of variables are changing in the same execution. We can now rewrite T' also in a way similar to T. Combining T and T', we get a set T'' containing two test cases written as follows:

$$T'' = \left\{ \begin{array}{ll} t_1: << request = 1,\ x = 2,\ y = 3 > \rightarrow & \\ \quad < request = 2, x = 4 > & \rightarrow < request = 3 >> \\ t_2: << request = 2, x = 4 > & \rightarrow \\ \quad < request = 1, x = 2, y = 3 > & \rightarrow < request = 3 >> \end{array} \right\}$$

Test set T'' contains two test cases, one that came from T and the other from T'. You might wonder why so much fuss regarding whether something is a test case or a test set. In practice, it does not matter. Thus, you may consider T as a set of three test cases or simply as a set of one test case. However, we do want to stress the point that distinct values of all program inputs may be input in separate runs or in the same run of a program. Hence, a set of test cases might be input in a single run or in separate runs.

In older programs that were not based on GUIs, it is likely that all test cases were executed in separate runs. For example, while testing a program to sort an array of numbers, a tester usually executed the sort program with different values of the array in each run. However, if the same sort program is *modernized* and a GUI added for ease of use and marketability, one may test the program with different arrays input in the same run.

In the next section, we introduce various criteria based on the flow of control for the assessment of test adequacy. These criteria are applicable to any program written in a procedural language such as C. The criteria can also be used to measure test adequacy for programs written in object-oriented languages such as Java and C++. Such criteria include plain method coverage as well as method coverage within context. The criteria presented in this chapter can also be applied to programs written in low-level languages such as an assembly language. We begin by introducing *control-flow-based* test-adequacy criteria.

6.2 Adequacy Criteria Based on Control Flow

6.2.1 Statement and Block Coverage

Any program written in a procedural language consists of a sequence of statements. Some of these statements are declarative, such as the `#define` and `int` statements in C, while others are executable, such as the assignment, `if` and `while` statements in C and Java. Note that a statement such as

```
int count=10;
```

could be considered declarative because it declares the variable `count` to be an integer. This statement could also be considered as executable because it assigns 10 to the variable `count`. It is for this reason that in C, we consider all declarations as executable statements when defining test adequacy criteria based on the flow of control.

Recall from Chapter 1 that a *basic block* is a sequence of consecutive statements that has exactly one entry point and one exit point. For any procedural language, adequacy with respect to the statement coverage and block coverage criteria are defined as follows:

Statement coverage

The statement coverage of T with respect to (P, R) is computed as $|S_c|/(|S_e| - |S_i|)$, where S_c is the set of statements covered, S_i the set of unreachable statements, and S_e the set of statements in the program, that is the coverage domain. T is considered adequate with respect to the statement coverage criterion if the statement coverage of T with respect to (P, R) is 1.

Block coverage

The block coverage of T with respect to (P, R) is computed as $|B_c|/(|B_e| - |B_i|)$, where B_c is the set of blocks covered, B_i the set of unreachable blocks, and B_e the blocks in the program, that is the block coverage domain. T is considered adequate with respect to the block coverage criterion if the block coverage of T with respect to (P, R) is 1.

In the definitions above, the coverage domain for statement coverage is the set of all statements in the program under test. Similarly, the coverage domain for block coverage is the set of all basic blocks in the program under test. Note that we use the term *unreachable* to refer to statements and blocks that fall on an infeasible path.

The next two examples explain the use of the statement and block coverage criteria. In these examples, we use line numbers in a program to refer to a statement. For example, the number 3 in S_e for Program P6.2 refers to the statement on line 3 of this program, that is to the `input (x, y)` statement. We will refer to blocks by block numbers derived from the flow graph of the program under consideration.

Example 6.10: The coverage domain corresponding to statement coverage for Program P6.4 is given below:

$$S_e = \{2, 3, 4, 5, 6, 7, 7b, 9, 10\}$$

Here, we denote the statement $z = z + 1$ as 7b. Consider a test set T_1 that consists of two test cases against which Program P6.4 has been executed.

$$T_1 = \{t_1: < x = -1, y = -1 >, t_2: < x = 1, y = 1 >\}$$

Statements 2, 3, 4, 5, 6, 7, and 10 are covered upon the execution of P against t_1. Similarly, the execution against t_2 covers statements 2, 3, 4, 5, 9, and 10. Neither of the two tests covers statement 7b that is unreachable as we can conclude from Example 6.8. Thus, we obtain $|S_c| = 8$, $|S_i| = 1$, $|S_e| = 9$. The statement coverage for T is $8/(9 - 1) = 1$. Hence, we conclude that T is adequate for (P, R) with respect to the statement coverage criterion.

Example 6.11: The five blocks in Program P6.4 are shown in Figure 6.4. The coverage domain for the block coverage criterion $B_e = \{1, 2, 3, 4, 5\}$. Consider now a test set T_2 containing three tests against which Program P6.4 has been executed.

$$T_2 \left\{ \begin{array}{l} t_1 < x = -1,\ y = -1 > \\ t_2 < x = -3,\ y = -1 > \\ t_3 < x = -1,\ y = -3 > \end{array} \right\}$$

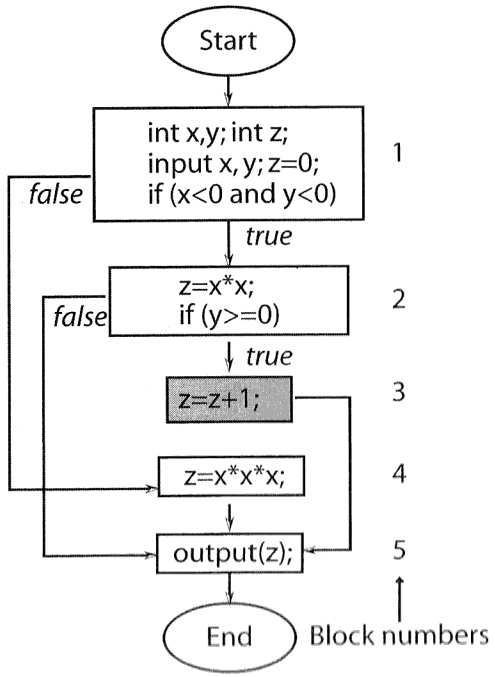

Fig. 6.4 CFG of Program P6.4. Blocks are numbered 1 through 5. The shaded block 3 is infeasible because the condition in block 2 will never be true.

Blocks 1, 2, and 5 are covered when the program is executed against test t_1. Tests t_2 and t_3 also execute exactly the same set of blocks. For T_2 and Program P6.4, we obtain $|B_c| = 5$, $|B_c| = 3$, and $|B_i| = 1$. The block coverage can now be computed as $3/(5-1) = 0.75$. As the block coverage is less than 1, T_2 is not adequate with respect to the block coverage criterion.

It is easy to check that the test set of Example 6.10 is indeed adequate with respect to the block coverage criterion. Also, T_2 can be made adequate with respect to the block coverage criterion by the addition of test t_2 from T_1 in the previous example.

The formulas given in this chapter for computing various types of code coverage yield a coverage value between 0 and 1. However, while specifying a coverage value, one might instead use percentages. For example, a statement coverage of 0.65 is the same as 65% statement coverage.

6.2.2 Conditions and Decisions

To understand the remaining adequacy measures based on control flow, we need to know what exactly constitutes a condition and a decision. Any expression that evaluates to `true` or `false` constitutes a condition. Such an expression is also known as a *predicate.* Given that A, B, and D are Boolean variables, and `x` and `y` are integers, A, $x > y$, A `or` B, A `and` $(x < y)$, (A `and` B) `or` (A `and` D) `and` (D), (A `xor` B) `and` $(x \geq y)$ are all conditions. In these examples, `and`, `or` , `xor`, are known as Boolean, or logical, operators. Note that in programming language C, x and $x + y$ are valid conditions, and the constants 1 and 0 correspond to, respectively, `true` and `false`.

Simple and compound conditions: A condition could be *simple or compound.* A simple condition does not use any Boolean operators except for the $\neg$ operator. It is made up of variables and at most one relational operator from the set $\{<, <, >, \geq, ==, \neq\}$. A compound condition is made up of two or more simple conditions joined by one or more Boolean operators. In the above examples, A and $x > y$ are two simple conditions, while the others are compound. Simple conditions are also referred to as *atomic* or *elementary* because they cannot be parsed any further into two or more conditions. Often, the term *condition* refers to a compound condition. In this book, we will use *condition* to mean any simple or compound condition.

Conditions as decisions: Any condition can serve as a decision in an appropriate context within a program. As shown in Figure 6.5, most high-level languages provide `if`, `while`, and `switch` statements to serve as

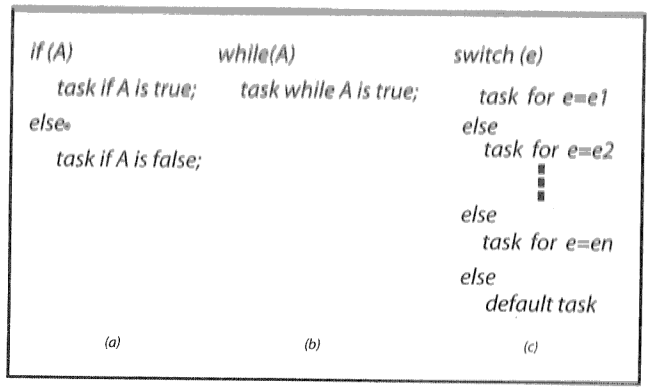

Fig. 6.5 Decisions arise in several contexts within a program. Three common contexts for decisions in a C or a Java program are (a) if, (b) while, and (c) switch statements. Note that the if and while statements force the flow of control to be diverted to one of two possible destinations, while the switch statement may lead the control to one or more destinations.

contexts for decisions. Whereas an `if` and a `while` contains exactly one decision, a `switch` may contain more.

A decision can have three possible outcomes—`true`, `false`, and undefined. When the condition corresponding to a decision evaluates to `true` or `false`, the decision to take one or the other path is taken. In the case of a `switch` statement, one of several possible paths gets selected and the control flow proceeds accordingly. However, in some cases the evaluation of a condition might fail in which the corresponding decision's outcome is undefined.

Example 6.12: Consider the following sequence of statements:

Program P6.6

```
1  bool foo(int a_parameter){
2    while (true) { // An infinite loop.
3      a_parameter=0;
4    }
5  } // End of function foo().
   :
6  if(x< y and foo(y)){ // foo() does not terminate.
7    compute(x,y);
   :
```

The condition inside the `if` statement on line 6 will remain undefined because the loop at lines 2–4 will never end. Thus, the decision on line 6 evaluates to `undefined`.

Coupled conditions: There is often the question of how many simple conditions are there in a compound condition. For example, $C' = (A \texttt{ and } B) \texttt{ or } (C \texttt{ and } A)$ is a compound condition. Does C' contain three or four simple conditions? Both answers are correct depending on one's point of view. Indeed, there are three distinct conditions A, B, and C. However, the answer is four when one is interested in the number of occurrences of simple conditions in a compound condition. In the example expression above, the first occurrence of A is said to be *coupled* to its second occurrence.

Conditions within assignments: Conditions may occur within an assignment statement as in the following examples:

1. $A = x < y$; // A simple condition assigned to a Boolean variable A.
2. $X = P$ or Q; // A compound condition assigned to a Boolean variable x.
3. $x = y + z * s$; `if (x)...` // Condition `true` if $x = 1$, `false` otherwise.
4. $A = x < y$; $x = A * B$; // A is used in a subsequent expression for x but not as a decision.

A programmer might want a condition to be evaluated before it is used as a decision in a selection or a loop statement, as in the previous examples 1–3. Strictly speaking, a condition becomes a decision only when it is used in the appropriate context such as within an `if` statement. Thus, in the example at line 4, $x < y$ does not constitute a decision and neither does $A * B$. However, as we shall see in the context of modified condition/decesion coverage (MC/DC) coverage, a decision is not synonymous with a branch point such as that created by an `if` or a `while` statement. Thus, in the context of the MC/DC coverage, the conditions at lines 1, 2, and the first one on line 4 are all decisions too!

6.2.3 Decision Coverage

A decision coverage is also known as a *branch decision coverage*. A decision is considered *covered* if the flow of control has been diverted to all possible destinations that correspond to this decision, that is all outcomes of the decision have been taken. This implies that, for example, the expression in the `if` or `while` statement has been evaluated to `true` in some execution of the program under test and to `false` in the same or another execution.

A decision implied by the `switch` statement is considered covered if during one or more executions of the program under test the flow of control has been diverted to all possible destinations. Covering a decision within a program might reveal an error that is not revealed by covering all statements and all blocks. The next example illustrates this fact.

Example 6.13: To illustrate the need for decision coverage, consider Program P6.7. This program inputs an integer x and, if necessary, transforms it into a positive value before invoking function `foo-1` to compute the output z. However, as indicated, this program has an error. As per its requirements, the program is supposed to compute z using `foo-2` when $x \geq 0$. Now consider the following test set T for this program:

$$T = \{t_i :< x = -5 >\}$$

It is easy to verify that when Program P6.7 is executed against the sole test case in T, all statements and all blocks in this program are covered. Hence, T is adequate with respect to both the statement and the block coverage criteria. However, this execution does not force the condition inside the `if` to be evaluated to `false` thus avoiding the need to compute z using `foo-2`. Hence, T does not reveal the error in this program.

Program P6.7

```
1  begin
2    int x, z;
3    input (x);
4    if(x<0)
5      z=-x;
6      z=foo-1(x);
7      output(z); ⟵ There should have been an else
                    clause before this statement.
8  end
```

Suppose that we add a test case to T to obtain an enhanced test set T'.

$$T' = \{t_1 :< x = -5 > t_2 :< x = 3 >\}$$

When Program P6.7 is executed against all tests in T', all statements and blocks in the program are covered. In addition, the sole decision in the program is also covered because condition $x < 0$ evaluates to `true` when the program is executed against t_1 and to `false` when executed against t_2. Of course, control is diverted to the statement at line 6 without executing line 5. This causes the value of z to be computed using `foo-1` and not `foo-2` as required. Now, `if foo-1(3)` $\neq$ `foo-2(3)`, then the program will give an incorrect output when executed against test t_2.

The above example illustrates how decision coverage might help a tester discover an incorrect condition and a missing statement by forcing the coverage of a decision. As you may have guessed, covering a decision does not necessarily imply that an error in the corresponding condition will always be revealed. As indicated in the example above, certain other program-dependent conditions must also be true for the error to be revealed. We now formally define adequacy with respect to decision coverage.

Decision coverage

The decision coverage of T with respect to (P, R) is computed as $|D_c|/(|D_e| - |D_i|)$, *where* D_c *is the set of decisions covered,* D_i *the set of infeasible decisions, and* D_e *the set of decisions in the program, that is the decision coverage domain. T is considered adequate with respect to the decision coverage criterion if the decision coverage of T with respect to (P, R) is 1.*

The domain of the decision coverage consists of all decisions in the program under test. Note that each `if` and each `while` contribute to one decision, whereas a `switch` may contribute to more than one. For the program in Example 6.13, the decision coverage domain is $D_e = \{3\}$ and hence $|D_c| = 1$.

6.2.4 Condition Coverage

A decision can be composed of a simple condition such as $x < 0$, or of a more complex condition such as $((x < 0$ and $y < 0)$ or $(p \geq q))$. Logical operators `and`, `or`, and `xor` connect two or more simple conditions to form a *compound* condition. In addition, ¬(pronounced as "not") is a unary logical operator that negates the outcome of a condition.

A simple condition is considered covered if it evaluates to `true` and `false` in one or more executions of the program in which it occurs. A compound condition is considered covered if each simple condition it is composed of is also covered. For example, $(x < 0$ and $y < 0)$ is considered covered when both $x < 0$ and $y < 0$ have evaluated to true and false during one or more executions of the program in which they occur.

The decision coverage is concerned with the coverage of decisions regardless of whether a decision corresponds to a simple or a compound condition. Thus, in the statement

```
1   if (x < 0 and y < 0) {
2       z = foo (x, y);
```

there is only one decision that leads control to line 2 if the compound condition inside the `if` evaluates to `true`. However, a compound condition might evaluate to `true` or `false` in one of several ways. For example, the condition at line 1 above evaluates to `false` when $x > 0$ regardless of the value of y. Another condition such as $x < 0$ or $y < 0$ evaluates to true regardless of the value of y when $x < 0$. With this evaluation characteristic in view, compilers often generate code that uses *short-circuit* evaluation of compound conditions. For example, the `if` statement in the above code segment might get translated into the following sequence.

```
1  if (x < 0)
2     if (y < 0)   // Note that y < 0 is evaluated only if x < 0 is true.
3        z = foo(x,y);
```

In the code segment above, we see two decisions, one corresponding to each simple condition in the if statement. This leads us to the following definition of condition coverage.

Condition coverage

The condition coverage of T with respect to (P, R) is computed as $|C_c|/(|C_e| - |C_i|)$, *where* C_c *is the set of simple conditions covered,* C_i *is the set of infeasible simple conditions, and* C_e *is the set of simple conditions in the program, that is the condition coverage domain. T is considered adequate with respect to the condition coverage criterion if the condition coverage of T with respect to (P, R) is 1.*

Sometimes the following alternate formula is used to compute the condition coverage of a test:

$$\frac{|C_c|}{2 \times (|C_e| - |C_i|)}$$

where each simple condition contributes 2, 1, or 0 to C_c depending on whether it is completely covered, partially covered, or not covered, respectively. For example, when evaluating a test set T, if $x < y$ evaluates to `true` but never to `false`, then it is considered partially covered and contributes a 1 to C_c.

Example 6.14: Consider the following program that inputs values of x and y and computes the output z using functions `foo1` and `foo2`. Partial specifications for this program are given in Table 6.1. This table lists how z is computed for different combinations of x and y. A quick examination of Program P6.8 against Table 6.1 reveals that for $x \geq 0$ and $y \geq 0$ the program incorrectly computes z as `foo2(x,y)`.

Table 6.1 Truth table for the computation of z in Program P6.8

x < 0	y < 0	Output (z)
true	true	foo1(x, y)
true	false	foo2(x, y)
false	true	foo2(x, y)
false	false	foo2(x, y)

Program P6.8

```
1  begin
2    int x, y, z;
3    input (x, y);
4    if(x<0 and y<0)
5      z=fool(x, y);
6    else
7      z=foo2(x, y);
8    output(z);
9  end
```

Consider T designed to test Program P6.8.

$$T = \{t_1: < x = -3, y = -2 > t_2: < x = -4, y = 2 >\}$$

T is adequate with respect to the statement coverage, block coverage, and the decision coverage criteria. You may verify that Program P6.8 behaves correctly on t_1 and t_2.

To compute the condition coverage of T, we note that $C_e = \{x < 0, y < 0\}$. Tests in T cover only the second of the two elements in C_e. As both conditions in C_e are feasible, $| C_i | = 0$. Plugging these values into the formula for condition coverage, we obtain the condition coverage for T to be $1/(2 - 0) = 0.5$.

We now add the test $t_3: < x = 3, y = 4 >$ to T. When executed against t_3, Program P6.8 incorrectly compute z as `foo2(x, y)`. The output will be incorrect if `fool`$(3, 4) \neq$ `foo2`$(3, 4)$. The enhanced test set is adequate with respect to the condition coverage criterion and possibly reveals an error in the program.

6.2.5 Condition/Decision Coverage

In the previous two sections, we learned that a test set is adequate with respect to decision coverage if it exercises all outcomes of each decision in the program during testing. However, when a decision is composed of a compound condition, the decision coverage does not imply that each simple condition within a compound condition has taken both values `true` and `false`.

Condition coverage ensures that each component simple condition within a condition has taken both values `true` and `false`. However, as illustrated in the next example, condition coverage does not require each decision to have taken both outcomes. The condition/decision coverage is also known as *branch condition coverage*.

Example 6.15: Consider a slightly different version of Program P6.8 obtained by replacing `and` by `or` in the `if` condition. For Program P6.9, we consider two test sets T_1 and T_2.

Program P6.9

```
1  begin
2    int x,y,z;
3    input (x,y);
4    if (x<0 or y<0)
5      z=fool(x,y);
6    else
7      z=foo2(x,y);
8    output(z);
9  end
```

$$T_1 = \left\{ \begin{array}{l} t_1: < x = -3 \;\; y = 2 > \\ t_2: < x = 4 \quad y = 2 > \end{array} \right\}$$

$$T_2 = \left\{ \begin{array}{l} t_1: < x = -3, \; y = 2 > \\ t_2: < x = 4, \quad y = -2 > \end{array} \right\}$$

Test set T_1 is adequate with respect to the decision coverage criterion because test t_1 causes the `if` condition to be `true` and test t_2 causes it to be `false`. However, T_1 is not adequate with respect to the condition coverage criterion because condition `y` < 0 never evaluates to `true`. In contrast, T_2 is adequate with respect to the condition coverage criterion but not with respect to the decision coverage criterion.

The condition/decision coverage-based adequacy criterion is developed to overcome the limitations of using the condition coverage and decision coverage criteria independently. The definition is as follows:

Condition/decision coverage

The condition/decision coverage of T with respect to (P, R) is computed as $(| C_e | + | D_e |)/((| C_e | - | C_i |) + (| D_e | - | D_i |))$, where

C_c denotes the set of simple conditions covered, D_c the set of decisions covered, C_e and D_e the sets of simple conditions and decisions, respectively, and C_i and D_i the sets of infeasible simple conditions and decisions, respectively. T is considered adequate with respect to the multiple-condition-coverage criterion if the condition/decision coverage of T with respect to (P, R) is 1.

Example 6.16: For Program P6.8, a simple modification of T_1 from Example 6.15 gives us T that is adequate with respect to the condition/decision coverage criteria.

$$T = \left\{ \begin{array}{l} t_1: < x = -3,\ y = -2 > \\ t_2: < x = 4,\ \ \ y = 2 > \end{array} \right\}$$

6.2.6 Multiple Condition Coverage

Multiple condition coverage is also known as *branch condition combination coverage*. To understand multiple condition coverage, consider a compound condition that contains two or more simple conditions. Using condition coverage on some compound condition C implies that each simple condition within C has been evaluated to `true` and `false`. However, it does not imply that all combinations of the values of the individual simple conditions in C have been exercised. The next example illustrates this point.

Example 6.17: Consider $D = (A < B)$ or $(A > C)$ composed of two simple conditions $A < B$ and $A > C$. The four possible combinations of the outcomes of these two simple conditions are enumerated in Table 6.2.

Now consider test set T containing two tests:

$$T = \left\{ \begin{array}{l} t_1: < A = 2,\ B = 3,\ C = 1 > \\ t_2: < A = 2,\ B = 1,\ C = 3 > \end{array} \right\}$$

The two simple conditions in D are covered when evaluated against tests in T. However, only two combinations in Table 6.2, those at lines 1 and 4, are covered. We need two more tests to cover the remaining two combinations at lines 2 and 3 in Table 6.2. We modify T to T' by adding two tests that cover all combinations of values of the simple conditions in D.

Table 6.2 Combinations in $D = (A < B)$ or $(A > C)$

	$A < B$	$A > C$	D
1	`true`	`true`	`true`
2	`true`	`false`	`true`
3	`false`	`true`	`true`
4	`false`	`false`	`false`

$$T' = \left\{ \begin{array}{l} t_1: < A = 2,\ B = 3,\ C = 1 > \\ t_2: < A = 2,\ B = 1,\ C = 3 > \\ t_3: < A = 2,\ B = 3,\ C = 5 > \\ t_4: < A = 2,\ B = 1,\ C = 1 > \end{array} \right\}$$

To define test adequacy with respect to the multiple-condition-coverage criterion, suppose that the program under test contains a total of n decisions. Assume also that each decision contains $k_1, k_2, \ldots, k_n$ simple conditions. Each decision has several combinations of values of its constituent simple conditions. For example, decision i will have a total of 2^{ki} combinations. Thus, the total number of combinations to be covered is $(\Sigma_{i=1}^{n} 2^{ki})$. With this background, we now define test adequacy with respect to multiple condition coverage.

Multiple Condition Coverage

The multiple condition coverage of T with respect to (P, R) is computed as $|C_c|/(|C_e| - |C_i|)$, where C_c denotes the set of combinations covered, C_i the set of infeasible simple combinations, and $|C_e| = \Sigma_{i=1}^{n} 2^{ki}$ the total number of combinations in the program. T is considered adequate with respect to the multiple-condition-coverage criterion if the condition coverage of T with respect to (P, R) is 1.

Example 6.18: It is required to write a program that inputs values of integers A, B, and C and computes an output S as specified in Table 6.3. Note from this table the use of functions $f1$ through $f4$ used to compute S for different combinations of the two conditions $A < B$ and $A > C$. Program P6.10 is written to meet the desired specifications. There is an obvious error in the program, computation

Table 6.3 Computing S for Program P6.10

	$A < B$	$A > C$	S
1	true	true	$f1(P, Q, R)$
2	true	false	$f2(P, Q, R)$
3	false	true	$f3(P, Q, R)$
4	false	false	$f4(P, Q, R)$

of S for one of the four combinations, line 3 in the table, has been left out.

Program P6.10

```
1  begin
2    int A,B,C,S=0;
3    input (A,B,C);
4    if(A<B and A>C) S=f1(A,B,C);
5    if(A<B and A≤C) S=f2(A,B,C);
6    if(A≥B and A≤C) S=f4(A,B,C);
7    output(S);
8  end
```

Consider test set T developed to test Program P6.10; this is the same test used in Example 6.17.

$$T = \left\{ \begin{array}{l} t_1: < A = 2,\ B = 3,\ C = 1 > \\ t_2: < A = 2,\ B = 1,\ C = 3 > \end{array} \right\}$$

Program P6.10 contains 3 decisions, 6 conditions, and a total of 12 combinations of simple conditions within the 3 decisions. Note that because all three decisions use the same set of variables, A, B, and C, the number of distinct combinations is only four. Table 6.4 lists all four combinations.

When Program P6.10 is executed against tests in T, all simple conditions are covered. Also, the decisions at lines 4 and 6 are covered. However, the decision at line 5 is not covered. Thus, T is adequate with respect to the condition coverage but not with respect to the decision coverage. To improve the decision coverage of T, we obtain T' by adding test t_3 from Example 6.17.

$$T' = \left\{ \begin{array}{l} t_1: < A = 2,\ B = 3,\ C = 1 > \\ t_2: < A = 2,\ B = 1,\ C = 3 > \\ t_3: < A = 2,\ B = 3,\ C = 5 > \end{array} \right\}$$

T' is adequate with respect to the decision coverage. However, none of the three tests in T' reveals the error in Program P6.10. Let us now evaluate whether T' is adequate with respect to the multiple-condition-coverage criteria.

Table 6.4 lists the 12 combinations of conditions in three decisions and the corresponding coverage with respect to tests in T'. From this table, we find that one combination of conditions in each of the three decisions remains uncovered. For example, at line 3 the (`false`, `true`) combination of the two conditions $A < B$ and $A > C$ remains uncovered. To cover this pair, we add t_4 to T' and get the following modified test set.

Table 6.4 Condition coverage for Program P6.10

	$A<B$	$A>C$	T	$A=B$	$A=C$	T	$A\neq B$	$A\neq C$	T
1	true	true	t_1	true	true	t_3	true	true	t_2
2	true	false	t_3	true	false	t_1	true	false	–
3	false	true	–	false	true	t_2	false	true	t_3
4	false	false	t_2	false	false	–	false	false	t_1

T, test.

$$T' = \left\{ \begin{array}{l} t_1: < A = 2,\ B = 3,\ C = 1 > \\ t_2: < A = 2,\ B = 1,\ C = 3 > \\ t_3: < A = 2,\ B = 3,\ C = 5 > \\ t_4: < A = 2,\ B = 1,\ C = 1 > \end{array} \right\}$$

Test t_4 in T'' does cover all of the uncovered combinations in Table 6.4 and hence renders T'' adequate with respect to the multiple condition criterion.

You might have guessed that our analysis in Table 6.4 is redundant. As all three decisions in Program P6.10 use the same set of variables, A, B, and C, we need to analyze only one decision in order to obtain a test set that is adequate with respect to the multiple-condition-coverage criterion.

6.2.7 Linear Code Sequence and Jump (LCSAJ) Coverage

Execution of sequential programs that contain at least one condition proceeds in pairs where the first element of the pair is a sequence of statements (a block), executed one after the other, and terminated by a jump to the next such pair (another block). The first element of this pair is a sequence of statements that follow each other textually. The last such pair contains a jump to program exit, thereby terminating program execution. An execution path through a sequential program is composed of one or more of such pairs.

A *linear code sequence and jump* is a program unit composed of a textual code sequence that terminates in a jump to the beginning of another code sequence and jump. The textual code sequence may contain one or more statements. An LCSAJ is represented as a triple (X, Y, Z), where X and Y are, respectively, locations of the first and the last statements and