2.1 Introduction to Program Testing

Given a computer program, how can we determine whether or not it will do exactly what it is designed to do? This question is not only intellectually challenging, but also of primary importance in practice.

An ideal solution to this problem would be to develop certain techniques that can be used to systematically construct the formal proof (or disproof) of the correctness of a program. There have been considerable efforts to develop such techniques, and many different techniques for proving program correctness have been reported as the results. However, none of them has been developed to the point where it can be readily applied in practice. Most of the techniques are of only theoretical interest for the following reason: in developing these techniques, the basic approach taken is to translate the problem of proving program correctness into that of proving a certain statement is a theorem in a formal system. The difficulty is that all known automatic theorem-proving techniques require an intolerably large amount of computation to construct a proof. This renders all the known automatic techniques based on theorem proving impractical. These techniques can be made practical only if we can produce a theorem prover that is powerful and efficient enough to overcome this difficulty. But even the foremost experts in theorem proving today cannot optimistically foresee the possibility of achieving, in the near future, a major breakthrough that will enable us to produce such a theorem prover. Thus to fulfill the short-term needs we have to search for practical, and perhaps less idealistic, solutions to the problem.

A practical and more intuitive approach in which we attempt to improve our confidence in a program by testing the program for a set of test cases will be discussed in the following. This is perhaps one of the most common approaches taken by today's software producers in their attempts to assess the reliability of their products.

How do we go about testing a computer program for its correctness? Perhaps the most intuitive (and seemingly plausible) answer to this question is to consider the program as a black box and test it for all possible input cases to see if it will produce the correct outputs. Unfortunately, it is in general impractical for us to do this simply because of the number of possible input cases involved. For example, consider Program 1.6.4.2, which is a typical small program with three inputs and one output as depicted below:
If, for an assignment of values to the input variables i, j, and k, the output variable match will assume a correct value upon execution of the program, then we can assert that the program is correct for this particular test case. And if we can test the program for all possible assignments to i, j, and k, then we will be able to determine its correctness. The difficulty here is that, even for a small program like this, with only three input variables, the number of possible assignments will be prohibitively large. To see why this is so, let us recall that i, j, and k are integer variables in C, each of which can assume a value in the range from \(-32767\) to \(+32767\) (i.e., \(2^{16}\) different values) on most platforms. Then there are \(2^{16} \times 2^{16} \times 2^{16} = 2^{48} = 256 \times 10^{12}\) possible assignments to the input triple (i, j, k). Now suppose this program can be test-executed at the rate of one test per microsecond in average. Then it will take more than eight (8) years for us to complete an exhaustive test for this small program!

This example clearly indicates that exhaustive testing is utterly impractical: we will never have time or financial resources to do that.

This example also shows that, no matter how large a practically feasible set of test cases we may choose, it always constitutes an extremely small sample out of all possible test cases. If the test results are incorrect, it definitely indicates that the program contains errors. If, however, the test results are correct, we have an insignificantly weak statistical base to infer that the program is correct. This is the basis for the well-known maxim that program testing can be used to discover the presence of errors, but not their absence.

The point is that, superficially, the test results are insignificant. We say "superficially" because here we are assuming that all test cases are equally important, although in fact they are not. When we examine the structure or the specification of the program, we can find a small set of test cases which is significant. What we would like to stress at this point is that (1) a randomly selected set of test cases is statistically insignificant, and (2) a selection of test cases based on the program structure or specification can be significant in that the use of these test cases leads to a higher probability of error detection.

### 2.2 Structural Testing

#### 2.2.1 Statement and Branch Testing

How can a set of test cases selected by a criterion based on the program structure be significant? The reason is that, for most programs, not every statement will occur in a given execution path.
Therefore, if a program contains a statement in error and that statement is not executed during the test, we will not be able to detect any abnormality at all in the test result. Thus, a possible test criterion is to have each and every statement in the program executed at least once during the test. However, as will be shown later, this leaves some important classes of errors undetected. Therefore we shall consider a more stringent criterion which requires that every branch in the flowchart be traversed at least once during the test. In what follows we shall illustrate these points by using the examples of Figures 2.1-2.3.

Let us consider the program given in Figure 2.1.

```c
Main()
{
    float a, b, e, w, p, q, u, v;
    scanf("%5.2f %5.2f %1.4f", &a, &b, &e);
    w = b - a;
    while (w > e) {
        p = a + w / 3;
        u = f(p);
        q = b - w / 3;
        v = f(q);
        if (u < v)
            a = p;
        else
            b = q;
        w = b - a;
    }
    max = (a + b) / 2;
    printf("5.2f\n", max);
}
```

where \( \alpha \): \( \text{scanf("%5.2f %5.2f %1.4f", \&a, \&b, \&e);} \)
\[
\begin{align*}
    w &= b - a; \\
    \beta: &/\ w > e; \\
    p &= a + w / 3; \\
    u &= f(p); \\
    q &= b - w / 3; \\
    v &= f(q); \\
    \gamma: &/\ !(u < v); \\
    b &= q; \\
    \delta: &/\ u < v; \\
    a &= p; \\
    \varepsilon: &w = b - a; \\
    \eta: &/\ !(w > e); \text{ max } = (a + b) / 2; \\
    &\text{printf("5.2f\n", max);}
\end{align*}
\]

Figure 2.1. A program and its program graph.

This program is designed to find the abscissa within the interval (a, b) at which a function \( f(x) \) assumes the maximum value. The basic strategy used is that, given a continuous function that has a maximum in the interval (a, b), we can find the desired point on the x-axis by first dividing the interval into three equal parts. Then compare the values of the function at the dividing points \( a+w/3 \) and \( b-w/3 \), where \( w \) is the width of the interval being considered. If the value of the function at \( a+w/3 \) is less than that at \( b-w/3 \), then the leftmost third of the interval is eliminated for further consideration; otherwise the rightmost third is eliminated. This process is repeated until the width of the interval being considered becomes less than or equal to a predetermined small constant \( e \). When that point is reached, the location at which the maximum of the function occurs can be taken as the center of the interval, \( (a+b)/2 \), -- with an error less than \( e/2 \).

Now suppose we wish to test this program for three different test cases, and assume that the function \( f(x) \) can be plotted as shown in Figure 2.2. Let us first arbitrarily choose \( e \) to be equal to 0.1, and choose the interval (a, b) to be (3, 4), (5, 6), and (7, 8). Now suppose that the values of \( \text{max} \) for all three cases are found to be correct in the test. What can we say about the design of this test?

Figure 2.2. The function plot of \( f(x) \).
Observe that in all three intervals chosen the value of \( u \) will be always greater than \( v \) as we can see from the function plot. Consequently, the statement \( a=p \) in the program will never be executed during the test. Thus if this statement is for some reason erroneously written as, say, \( a=q \) or \( b=p \), we will never be able to discover the error in a test using the three test cases mentioned above. This is so simply because this particular statement is not "exercised" during the test.

The functional plot given in Figure 2.2 shows that \( u \) will always be less than \( v \) within the interval \((0, 1)\). Thus if the test cases used include the interval \((0, 1)\) we will be able to discover the error described above.

The point to be made here is that our chances of discovering errors through program testing can be significantly improved if we select the test cases in such a way that each and every statement will be executed at least once.

It must be emphasized here, however, that the use of such a set of test cases gives us no assurance that the presence of an error will be definitely reflected in the test result. This fact can be demonstrated by using a simple example. For instance, if a statement in the program, say, \( x=x+y \) is somehow erroneously written as \( x=x-y \), and if the test case used is such that it sets \( y = 0 \) prior to the execution of this statement, the test result certainly will not indicate the presence of this error.

![Figure 2.3. A type of programming error.](image)
The inadequacy of testing a program only to the extent that each and every statement is executed at least once is actually more serious than what we described above. There is a class of common programming errors that cannot be discovered in this way. For instance, consider the type of error illustrated in Figure 2.3, where the flow of control is transferred to a wrong place as indicated by the dotted line. This occurs when a C programmer mistakenly writes

```c
if (B)
  s1;
s2;
```

instead of

```c
if (B) {
  s1;
  s2;
}
```

In this case the program produces correct results as long as the input data cause B to be true when this program segment is entered. The requirement of having each and every statement executed at least once is trivially satisfied in this case by choosing input data so that B is true. Obviously, the error will not be detected in this case.

The problem is that a program may contain paths from the entry to the exit (in its control flow) which need not be traversed in order to have each and every statement executed at least once. Since the present test requirement can be satisfied without having such paths traversed during the test, it is only natural that we will not be able to discover errors that occur on those paths. An obvious solution to this problem would be to require that each and every control path in the program be traversed at least once during the test. However, this test requirement can be easily proved to be impractical because in practice almost every program contains loops, and a program with a loop contains at least as many different control paths as the number of times the loop can be iterated, which is prohibitively large in many cases. A more realistic solution is to require that each and every edge or branch (these two terms are used interchangeably throughout this article) in the program graph be traversed at least once during the test. In accordance with this new test requirement, we will have to use a new test case that makes B false, in addition to the one that satisfies B, in order to have every branch in Figure 2.3 traversed at least once. Hence our chances of discovering the error will be greatly improved, because the program will most likely produce an erroneous result for the test case that makes B false.

Observe that this new requirement of having each and every branch traversed at least once is more stringent than the previously stated requirement of having each and every statement executed at least once. In fact, satisfaction of the new requirement implies satisfaction of the previous one. This is so because every statement in the program is associated with some edge in the program graph. Thus each and every statement has to be executed at least once in order to have every branch traversed at least once (provided that there is no inaccessible code in the program text). Satisfaction of the previously stated requirement, however, does not necessarily entail satisfaction of the new one. This can be easily verified by using a counter-example readily obtainable from Figure 2.3.
A survey of the literature shows that there is no common agreement as to what can be considered as an adequate test criterion. However, the measure of thoroughness defined here appears to have been widely recognized as a basic test requirement. A test that requires every statement in the program to be exercised at least once (every branch in the control flow to be traversed at least once) is commonly referred to as a statement (branch) test.

The question now is how do we go about to do statement or branch test?

A simple and practical way to determine to what extent the test coverage has been achieved is to instrument the program to be tested by using a set of software counters as explained in a later chapter. After having the program tested for a number of test cases, we can determine the coverage achieved by examining the resulting counter values. If the test requirement is to have each branch traversed at least once, and if the program is instrumented in such a way that there is one counter on each decision-to-decision \* path, then the requirement is satisfied when the values of all counters are non-zero.

Alternatively, we can begin the test procedure by finding a (minimal) set of test cases that will test the program thoroughly. We then use this set of test cases, perhaps in conjunction with any other desirable test cases, to test the program. In this way the desired degree of thoroughness will be automatically achieved. Furthermore, by using a minimal set (i.e., a set with a minimal number of elements) of test cases, we can keep the required resources for program testing to a minimum.

The question now is: how can we find for a given program a minimal set of test cases for a branch test? Essentially, the desired set of test cases can be generated in three steps: (1) Find \( S \), a minimal set of paths from the entries to the exits in the program graph such that every branch is on some path in \( S \); (2) Find a path predicate for each path in \( S \); and (3) Find a set of assignments to the input variables, each of which satisfies a path predicate obtained in step 2. This set is the desired set of test cases.

We shall illustrate the process of test-case generation by using the program shown in Figure 2.1. First we need to find set \( S' \) of all paths in the flowchart. A flowchart is essentially a directed graph, and techniques for finding paths in a directed graph can be found in textbooks on graph theory. We then construct \( S \), a minimal subset of \( S' \) that covers every branch in the graph. In a simple flowchart like the one in the present example, we can find \( S \) simply by inspection, viz., \( S = \{ \alpha \beta \delta \epsilon \eta, \alpha \beta \gamma \eta \} \). These two paths correspond to the trace subprograms listed below.

\[
\alpha \beta \delta \epsilon \eta: \quad \text{scanf("%5.2f %5.2f %1.4f", &a, &b, &e);} \\
\text{w = b - a;} \\
/\ \text{w > e;} \\
\text{p = a + w / 3;} 
\]

\[
\alpha \beta \gamma \eta: \quad \text{scanf("%5.2f %5.2f %1.4f", &a, &b, &e);} \\
\text{w = b - a;} \\
/\ \text{w > e;} \\
\text{p = a + w / 3;} 
\]

\* A decision-to-decision path is an execution path that emanates from the entry of the program or a node in the control-flow graph with an out-degree greater than one, and terminates at another which is either an exit of the program, or a node with an out-degree greater than one. Furthermore, no node on the path, except the first and the last, has an out-degree greater than one.
\[ u = f(p); \]
\[ q = b - w / 3; \]
\[ v = f(q); \]
\[ /\ u < v; \]
\[ a = p; \]
\[ w = b - a; \]
\[ /\ !(w > e); \]
\[ max = (a + b) / 2; \]
\[ printf("5.2f\n", max); \]

Next, we need to determine the domains for which these two subprogram are defined, i.e., the conditions under which these two paths will be traversed in execution. This can be done by applying Theorem 1.6.2.12(a) repeatedly to move all constraints in each subprogram upstream until they reach the input statement (that defines the value of variables involved). As the result, we have

\[ \text{\texttt{αβγεη: scanf("%5.2f %5.2f %1.4f", &a, &b, &e);}} \]
\[ /\ w > e; \]
\[ p = a + w / 3; \]
\[ u = f(p); \]
\[ q = b - w / 3; \]
\[ v = f(q); \]
\[ /\ !(u < v); \]
\[ b = q; \]
\[ w = b - a; \]
\[ /\ !(w > e); \]
\[ max = (a + b) / 2; \]
\[ printf("5.2f\n", max); \]
\[ q = b - w / 3; \]
\[ v = f(q); \]
\[ b = q; \]
\[ w = b - a; \]
\[ \max = (a + b) / 2; \]
\[ \text{printf("5.2f\n", max);} \]

Thus, the path condition of path \( \alpha \beta \delta \epsilon \eta \) is

\[
\begin{align*}
\text{b - a} & > e \\
\&\& \ f\left(\frac{a + (b - a)}{3}\right) & < f\left(\frac{b - (b - a)}{3}\right) \\
\&\& \ !\left(2 \times \frac{(b - a)}{3} > e\right)
\end{align*}
\]

which can be simplified to

\[
\begin{align*}
\text{b - a} & > e \\
\&\& \ f\left(\frac{(b + 2a)}{3}\right) & < f\left(\texttt{a + 2b} \right) / 3) \\
\&\& \ !\left(2 \times \frac{(b - a)}{3} > e\right)
\end{align*}
\]

and that of path \( \alpha \beta \gamma \epsilon \eta \) is

\[
\begin{align*}
\text{b - a} & > e \\
\&\& \ !\left(f\left(\frac{a + (b - a)}{3}\right) < f\left(\frac{b - (b - a)}{3}\right)\right) \\
\&\& \ !\left(2 \times \frac{(b - a)}{3} > e\right)
\end{align*}
\]

which can be simplified to

\[
\begin{align*}
\text{b - a} & > e \\
\&\& \ !\left(f\left(\frac{(b + 2a)}{3}\right) < f\left(\texttt{a + 2b} \right) / 3) \\
\&\& \ !\left(2 \times \frac{(b - a)}{3} > e\right)
\end{align*}
\]

Since we do not have the exact specification of function \( f \), we need to rewrite the second atomic expression into a more definitive statement. It is observed that \( a < b \) because they stand for the lower and upper boundaries of an interval on the x-axis. Hence it is always true that \( (b+2a)/3 < (a+2b)/3 \). Now, from the function plot in Fig. 2.2 we see that \( \neg(f(x_1) < f(x_2)) \) will be true if \( x_1 < x_2 \) and \( x_1 \) is greater than or equal to 2. In other words, the second atomic expression will be true if \( (b + 2a)/3 = 2 \), or, equivalently, \( b + 2a = 6 \). Thus, instead of the path predicates shown above, we may consider

\[
\begin{align*}
\text{b - a} & > e \&\& \ b + 2a \geq 6 \&\& \ !\left(2 \times \frac{(b - a)}{3} > e\right)
\end{align*}
\]

as the path predicate of path \( \alpha \beta \delta \epsilon \eta \), and

\[
\begin{align*}
\text{b - a} & > e \&\& \ !\left((b + 2a) \geq 6\right) \&\& \ !\left(2 \times \frac{(b - a)}{3} > e\right)
\end{align*}
\]

as that of path \( \alpha \beta \gamma \epsilon \eta \) for the purpose of finding test cases.
How do we go about to find a test case (i.e., an assignment of value to the input variables \(a, b,\) and \(e\) that satisfy the path predicate)? A method is given in the last part of Sec. 1.2. Note that the path predicate for path \(\alpha\beta\delta\varepsilon\eta\) is the logical expression (A) therein.

Once the desired set of test cases is found, the question then is: what kind of test result will be produced if the program is correct, and what kind of abnormality may be observed if the program is in error? To fix the idea, let us consider the case in which we test the program shown in Figure 2.1 by using the test case:

\[
\begin{align*}
a &\leftarrow 0, \\
b &\leftarrow 0.5, \\
e &\leftarrow \frac{1}{3}.
\end{align*}
\]

If the program is correct, we should obtain as the result \(\text{max} = 0.5 - \Delta\), where \(\Delta\) is the error less than or equal to \(\frac{e}{2} = \frac{1}{6}\). If the program is in error, say, the assignment statement \(p = a + \frac{w}{3}\) is somehow written as \(p = a - \frac{w}{3}\), then the algorithm will not converge. Consequently, we have an infinite loop in the program and execution will not terminate. It is interesting to see that if the logical expression (associated with the second decision box in Figure 2.1) is erroneously written as \(v < u\) instead of \(u < v\), then variable \(\text{max}\) will contain the abscissa at which \(f(x)\) assumes the minimum value (instead of the maximum). In other words, we will obtain as the test result \(\text{max} = \Delta\), i.e., \(\text{max}\) is within the distance \(\Delta = \frac{e}{2} = \frac{1}{6}\) from \(a = 0\), which is clearly not a correct answer, as one can see from the function plot.

Although the presence of the two types of error mentioned above will be clearly reflected in the test results, we would like to emphasize here again that constructing a minimally thorough test is not sufficient to warrant detection of all possible programming errors. Here is another good example attesting to this. Suppose the assignment statement \(a = p\) is mistakenly written as \(a = q\). It is easy to see that the program will produce a correct result for the test case:

\[
\begin{align*}
a &\leftarrow 1.9, \\
b &\leftarrow 2.2, \\
e &\leftarrow 0.2,
\end{align*}
\]

because this statement will not be executed, assuming that \(f(x)\) is monotonously decreasing for \(x = 2\). Neither will the presence of this error be clearly reflected in the test result using test case:

\[
\begin{align*}
a &\leftarrow 0, \\
b &\leftarrow 0.5, \\
e &\leftarrow \frac{1}{3}.
\end{align*}
\]

The reader should be able to verify that we still will obtain as the result \(\text{max} = 0.5 - \Delta\) for some \(\Delta > \frac{1}{6}\), although the magnitude of error \(\Delta\) could be smaller than that produced in the absence of this programming error.

For the sake of clarity in presentation, we have purposely simplified the problems in program testing and used somewhat contrived examples to illustrate the ideas involved in solving the problems. This might have given the reader the impression that we have relatively simple and straightforward solutions to these problems. Therefore, in concluding this part of discussion it is deemed imperative for us to point out that the problems are actually much more complicated than we have described in this section. For instance, the problem of test-case generation is in general unsolvable in the sense that there does not exist a single algorithm that can be used to
find assignments to input variables that will satisfy any given path predicate, or even to
determine its satisfiability.

All that we can do at this stage of development is to identify a solvable subset of problems and
then develop effective methods for solving these problems. Even for those programs or parts of
a program that we know how to handle, the process of test-case generation is in practice much
more complicated than what we have described in this section. Here are some major factors that
contribute to the complexity of test-case generation.

1) \textit{Number of paths involved in a large program.}

2) \textit{Nontraversable paths in a program:} it is well known that some paths in a program may
never be traversed in execution. Such paths cannot be used as the test paths, and thus must
be excluded in the process of constructing the minimal covering set of paths. The fact that an
untraversable path cannot be identified on the basis of the graph structure of the flowchart
but rather by the fact that it has an unsatisfiable path predicate greatly complicates the
problem. To facilitate understanding this important problem, let us consider the example
program shown in Figure 2.4.

```c
main()
{
    int x, y, z;
    scanf("%d %d", &x, &y);
    z = 1;
    while (y != 0) {
        if (y \% 2 == 1)
            z = z * x;
        y = y / 2;
        x = x * x;
    }
    printf("%d\n", z);
}
```

![Diagram of program flow](image-url)
where \( \alpha: \) \( \text{scanf}("%d \ %d", \ \&x, \ \&y); \)
\[
\begin{align*}
z &= 1; \\
\beta: &/\!\!/ \ y \neq 0; \\
\gamma: &/\!\!/ \ !\ (y \ %\ 2 == 1); \\
\delta: &/\!\!/ \ y \ %\ 2 == 1; \\
&z = z \ * \ x; \\
\varepsilon: &y = y / 2; \\
&x = x \ * \ x; \\
\eta: &/\!\!/ \ !\ (y \ != 0); \\
&\text{printf}("%d\n", \ z); \\
\end{align*}
\]

Figure 2.4. A program.

This program computes \( x^y \) by a binary decomposition of \( y \) for integer \( y = 0 \). By inspection we see that every branch is on some path in the set given below:

\{\( \alpha\beta\delta\epsilon\eta, \alpha\beta\gamma\epsilon\eta \}\n
and thus is a candidate minimal covering set for test-case construction. These two paths correspond to the two trace subprograms listed below.

\( \alpha\beta\delta\epsilon\eta: \) \( \text{scanf}("%d \ %d", \ \&x, \ \&y); \)
\[
\begin{align*}
z &= 1; \\
/\!\!/ \ y \neq 0; \\
/\!\!/ \ y \ %\ 2 == 1; \\
&z = z \ * \ x; \\
&y = y / 2; \\
&x = x \ * \ x; \\
/\!\!/ \ !\ (y \ != 0); \\
&\text{printf}("%d\n", \ z); \\
\end{align*}
\]

\( \alpha\beta\gamma\epsilon\eta: \) \( \text{scanf}("%d \ %d", \ \&x, \ \&y); \)
\[
\begin{align*}
z &= 1; \\
/\!\!/ \ y \neq 0; \\
/\!\!/ \ !\ (y \ %\ 2 == 1); \\
&y = y / 2; \\
&x = x \ * \ x; \\
/\!\!/ \ !\ (y \ != 0); \\
&\text{printf}("%d\n", \ z); \\
\end{align*}
\]

Again, by applying Theorem 1.6.2.12(a) repeatedly to move all constraints in each subprogram upstream until they reach the input statement (that defines the value of variables involved), we have

\( \alpha\beta\delta\epsilon\eta: \) \( \text{scanf}("%d \ %d", \ \&x, \ \&y); \)
\[
/\!\!/ \ y \neq 0;
\]
\( \text{y} \% 2 == 1; \)
\( \text{!} (\text{y} / 2 != 0); \)
\( \text{z} = 1; \)
\( \text{z} = \text{z} * \text{x}; \)
\( \text{y} = \text{y} / 2; \)
\( \text{x} = \text{x} * \text{x}; \)
\( \text{printf("}\%d\text{\n", z}); \)

Thus the path predicates for these two paths are:

\( \text{y} != 0 \&\& \text{y} \% 2 == 1 \&\& \text{y} / 2 == 0 \)

and

\( \text{y} != 0 \&\& \text{y} \% 2 != 1 \&\& \text{y} / 2 == 0, \)

respectively. Some reflection will show that the first path predicate can be satisfied by letting \( \text{y} = 1 \), but the second predicate cannot be satisfied by any non-negative integer! This indicates that the second path \( \alpha\beta\gamma\epsilon\eta \) cannot be traversed at all. Our solution to this problem at the present is to construct the minimal covering set solely based on the graph structure of the flowchart, and then appropriately replace any member path which is found to be associated with an unsatisfiable path predicate. For example, path \( \alpha\beta\gamma\epsilon\eta \) in the above example can be replaced by path \( \alpha\beta\gamma\epsilon\beta\delta\epsilon\eta \), whose path predicate can be satisfied by letting \( \text{y} = 2 \).

EXERCISE: Can the second path predicate be satisfied by assigning a negative value to \( \text{y} \)?

EXERCISE: Find a way to verify experimentally that path \( \alpha\beta\gamma\epsilon\eta \) can be traversed in an execution.

3) **Loop structure in a program:** the process of constructing a minimal covering set of paths can be greatly simplified if we can use the fact that a loop needs to be iterated only once in order to have every branch traversed at least once. Unfortunately, some loops in a program must be iterated for a constant number (greater than one) of times. A loop formed by a statement of the form: "for \( i = 1 \) step 2 until 15 do S" is an example. Thus if we construct a test path by having such a loop traversed once, we will find the associated path predicate unsatisfiable. This necessitates a more elaborate path-finding method that has a provision to identify such a loop. Also, if a path consists of a loop that needs to be iterated for a great
number of times, then the associated path predicate will be a very long expression unless a special notational convention is used in the process.

4) **Subscripted variables:** to see what kind of problem the use of subscripted variables may introduce, let us consider the predicate: \( a[i+1] = a[j] \) as an example. This predicate can be satisfied by letting \( i + 1 = j \) or by assigning the same value to \( a[i+1] \) and \( a[j] \). Thus a degree of indeterminacy is added to the process of finding an assignment that satisfies a predicate.

5) **Block structure and call of procedure or subroutine:** if the program is written in a language (such as C) that permits the use of block structures, or if it contains a function (procedure) call, then we need to be able to tell whether a given variable is local or global. Since the same identifier can be used to denote two distinct variables in the same program, we must keep track of the scopes in which the variables are defined. We also need to know whether an identifier stands for a "call by name" or a "call by value" parameter in order to construct a path predicate correctly.

6) **Path predicates involving floating-point variables:** the truth values of such predicates may become unpredictable.

The complicating factors mentioned above by no means exhaust the list. By trying to apply the test-case generation procedure described in the preceding sections to practical problems, the reader will certainly encounter many other problems.

### 2.2.2 Path and Dataflow Testing

It is interesting to see what branch test means in terms of the tasks to be performed by a program. Mathematically speaking, a computer program may be considered as the definition of a function. This function usually is expressed as a set of partial functions, each of which is defined on a subset of the intended input domain. Each partial function is associated with an execution path in such a way that the sequence of non-control statements on the path is actually a subprogram that computes the values of that partial function. The condition that a set of input data has to satisfy in order for a path to be traversed in execution is generally referred to as the **path predicate (condition)** of that path. The path predicate essentially defines the membership of a subdomain in which the corresponding partial function is defined. If every branch in the program is traversed at least once, it implies that most, but not necessarily all, of the possible execution paths will be traversed at least once. Therefore, to test a program by having every branch traversed at least once is to test the correctness of most partial functions for at least one point in the subdomain in which each is defined.

If there is an error in the constituent statements of a certain path, it is most likely that we will discover the error because the corresponding partial function will be checked for at least one point in its domain. We must remember, however, that some possible execution paths may not be covered in a branch test. Furthermore, for some input data, some programs may produce results that are fortuitously correct, as we have illustrated before. This is why the requirement of
having every branch traversed at least once is still not sufficient to ensure that the presence of an error will be definitely indicated in the test result.

Obviously, we can make a test more thorough by requiring that every possible execution path in the program be exercised at least once. But it is infeasible in practice because most programs contain loop constructs, each of which yields a prohibitively large number of executable paths.

Having every branch traversed at least once can be seen as a practical way to obtain a well-distributed sample of execution paths.

In addition to branch testing, there are at least three other methods for selecting paths to be tested.

THE FIRST, called the boundary-interior testing [HOWD75], is designed to circumvent the problem presented by a loop construct. A boundary test of a loop construct causes it to be entered but not iterated. An interior test causes a loop construct to be entered and iterated at least once.

THE SECOND is proposed by McCabe based on his complexity measure [MCCA76]. It requires that at least a maximal set of linearly independent paths in the program be traversed during the test.

A graph is said to be strongly connected if there is a path from any node in the graph to any other node. It can be shown [BERG73] that, in a strongly connected graph $G = <E, N>$, where $E$ is the set of edges and $N$ is the set of nodes in $G$, there can be as many as $v(G)$ elements in a set of linearly independent paths, where

$$v(G) = |E| - |N| + 1.$$ 

The number $v(G)$ is also known as McCabe's cyclomatic number, a measure of program complexity [MCCA76].

Here we speak of a program (control) graph with one entry and one exit. It has the property that every node can be reached from the entry, and every node can reach the exit. In general, it is not strongly connected, but can be made so by adding an edge from the exit to the entry. For example, we can make the program graph in Fig. 2.4 strongly connected by adding the edge $\mu$ (in dashed line) as depicted below.
Since there are 7 edges and 5 nodes in this graph, \( v(G) = 7 - 5 + 1 = 3 \) in this example. Note that, for an ordinary program graph without that added edge, the formula for computing \( v(G) \) should be:

\[
v(G) = |E| - |N| + 2.
\]

Next, for any path in \( G \), we can associate it with a 1 by \( |N| \) vector, where the element on the i-th column is an integer equal to the number of times the i-th edge is used in forming the path. Thus, if we arrange the edges in the above graph in the order \( \alpha \beta \delta \varepsilon \gamma \mu \), then the vector representation of path \( \alpha \beta \gamma \varepsilon \eta \) is \( <1 \ 0 \ 1 \ 1 \ 0 \ 1> \), and that of \( \beta \gamma \varepsilon \beta \gamma \varepsilon \) is \( <0 \ 2 \ 0 \ 2 \ 0 \ 0> \). We shall write \( <\alpha \beta \gamma \varepsilon \eta> = <1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1> \), and \( <\beta \gamma \varepsilon \beta \gamma \varepsilon> = <0 \ 2 \ 0 \ 2 \ 0 \ 0> \).

A path is said to be a linear combination of others if its vector representation is equal that formed by a linear combination of their vector representations. Thus, path \( \beta \gamma \varepsilon \eta \) is a linear combination of \( \beta \gamma \) and \( \varepsilon \eta \) because \( <\beta \gamma \varepsilon \eta> = <\beta \gamma> + <\varepsilon \eta> = <0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1> + <0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1> = <0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1> \), and path \( \alpha \eta \) is a linear combination of \( \alpha \beta \delta \varepsilon \eta \) and \( \beta \delta \varepsilon \) because \( <\alpha \eta> = <\alpha \beta \delta \varepsilon \eta> - <\beta \delta \varepsilon> = <1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1> - <0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0> = <1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0> \).

A set of paths is said to be linearly independent if no path in the set is a linear combination of any other paths in the set. Thus \( \{\alpha \beta \delta \varepsilon \eta, \alpha \beta \gamma \varepsilon \eta, \alpha \eta\} \) is linearly independent, but \( \{\alpha \beta \delta \eta, \alpha \beta \delta \beta \delta \varepsilon \eta, \alpha \eta\} \) is not (because \( <\alpha \beta \delta \varepsilon \eta> + <\alpha \beta \delta \eta> - <\alpha \beta \delta \beta \delta \varepsilon \eta> = <\alpha \eta> \)).

A basis set of paths is a maximal set of linearly independent paths. In graph \( G \) given above, since \( v(G) = 3 \), \( \{\alpha \beta \delta \varepsilon \eta, \alpha \beta \gamma \varepsilon \eta, \alpha \eta\} \) constitutes a basis set of paths. Note that, although \( v(G) \) is fixed by the graph structure, the membership of a basis set is not unique. For example, \( \{\alpha \beta \delta \eta, \alpha \beta \delta \beta \delta \gamma \varepsilon \eta, \alpha \eta\} \) is also a basis set in \( G \).

It is interesting to observe that \( v(G) \) has the following properties:

(a) \( v(G) \geq 1 \).
(b) \( v(G) \) is the maximum number of linearly independent paths in \( G \), and it is the size of the basis set.

(c) Inserting or deleting a node with out-degree of 1 does not affect \( v(G) \).

(d) \( G \) has only one path if \( v(G) = 1 \).

(e) Inserting a new edge in \( G \) increases \( v(G) \) by 1.

(f) \( v(G) \) depends only on the decision structure of the program represented by \( G \).

THE THIRD method is based on the data flow in the program. When a program is executed along a path, the value of each variable involved is defined first and then used later. By requiring such "define-use" relations be exercised during the test, the probability of error detection can be increased.

Data flow testing is also a form of structure testing. The "component" that will be exercised during the test is a segment of control path that starts from the point where a variable is defined, and ends at the point where that definition is used. We need to introduce a few terms before we proceed to describe data-flow oriented methods for test case selection.

A path is said to be definition clear with respect to a variable, say, \( x \), if it begins at a point where \( x \) is defined, and contains no statement that causes \( x \) to be undefined or redefined.

A path is loop-free if every node on the path occurs only once.

A simple path is a path in which at most one node occurs twice.

A du path of a variable, say, \( x \), is a simple path that is definition clear with respect to \( x \).

Now we are ready to enumerate a number of test-case selection criteria that can be formulated based on the data flow in a program.

**All-du-path testing**

It requires that every du path from every definition of every variable in the program to every use of that definition be traversed at least once during the test.

**All-use testing**

It requires that at least one definition-clear path from every definition of every variable to every use of that definition be traversed during the test.

**All-p-use/some-c-use testing**

It requires that at least one definition-clear path from every definition of every variable to every p-use (i.e., the definition is used in a predicate) of that definition be traversed during the test. If there is no p-use of that definition, replace "every p-use" in the above sentence with "at least one c-use (i.e., the definition is used in a computation)."
**All-c-use/some-p-use testing**

It requires that at least one definition-clear path from every definition of every variable to every c-use of that definition be traversed during the test. If there is no c-use of that definition, replace "every c-use" in the above sentence with "at least one p-use."

**All-definition testing**

It requires that, for every definition of every variable in the program, at least one du-path emanating from that definition be traversed at least once during the test.

**All-p-use testing**

It derives from the all p-use/some c-use by dropping the "some c-use" requirement.

**All-c-use testing**

It derives from the all c-use/some p-use by dropping the "some p-use" requirement.

The following diagram shows the coverage relationship among test-case selection criteria based on the structure of the source code.


2.3 Domain Strategy Testing

Before proceeding to describe yet another test method, we shall digress momentarily to discuss a way to categorize possible programming errors. The reader may have already sensed that there are many different test methods, and each is effective only for detecting a certain type of errors. An error categorization scheme is useful if it enables us to characterize a test method in terms of error type for which it is effective.

In abstract, the intended function of a program can be viewed as a function \( f \) of the nature \( f: X \rightarrow Y \). The definition of \( f \) usually is expressed as a set of subfunctions \( f_1, f_2, \ldots, f_m \), where \( f_i: X_i \rightarrow Y \) (i.e., \( f_i \) is \( f \) restricted to \( X_i \) for all \( 1 \leq i \leq m \)), \( X = X_1 \cup X_2 \cup \ldots \cup X_m \), and \( f_i \neq f_j \) if \( i \neq j \). We shall use \( f(x) \) to denote the value of \( f \) evaluated at \( x \in X \), and suppose that each \( X_i \) can be described in the standard subset notation \( X_i = \{ x \mid x \in X \land C_i(x) \} \).

Note that, in the above, we require the specification of \( f \) to be compact, i.e., \( f_i \neq f_j \) if \( i \neq j \). This requirement makes it easier to construct the definition of a type of programming error in the following. In practice, the specification of a program may not be compact, i.e., \( f_i \) may be identical to \( f_j \) for some \( i \) and \( j \). Such a specification, however, can be made compact by merging \( X_i \) and \( X_j \).

Let \((P, S)\) denote a program, where \( P \) is the condition under which the program will be executed, and \( S \) is the sequence of statements to be executed. Furthermore, let \( D \) be the set of all possible inputs to the program. Then the (valid) input domain of this program should be \( X = \{ x \mid x \in D \land P(x) \} \), and the program should be composed of \( n \) paths, i.e.,

\[
(P, S) = (P_1, S_1) + (P_2, S_2) + \ldots + (P_n, S_n),
\]

such that for every \( 1 = i = n \), \( S_i \) is the sequence of statements designed to compute \( f_j \) for some \( 1 = j = m \) (note that \( n \) is not necessarily equal to \( m \)).

We shall use \( S(x) \) to denote the computation performed by an execution of \( S \) with \( x \) as input.

Two basic types of error may be committed in constructing the program \((P, S)\):

1. **Computational error:** the program has a computational error if

\[
(\exists i)(\exists j)((P_i \supset C_j \land S_i(x) \neq f_j(x)).
\]

2. **Domain error:** the program has a domain error if

\[
(\forall i)(\exists j)(P_i \supset C_j).
\]

Note that the above definition is not identical to that given by Goodenough and Gerhart [GOGE77], Howden [HOWD76], or White and Cohen [WHCO80]. More will be said about the differences at the end of this section.
The domain strategy testing [WHCO80] described below is designed to detect domain errors, and is based on a geometrical analysis of the domain boundary, taking advantage of the fact that points on or near the border are most sensitive to domain errors.

Test cases are to be selected for each border segment, which, if processed correctly, determine that both the relational operator and the position of the border are correct.

An important assumption made in this work is that the user or an "oracle" is available who can decide unequivocally if the output is correct for the specific input processed. The oracle decides only if the output values are correct, and not whether they are computed correctly. If they are incorrect, the oracle does not provide any information about the error and does not give the correct output values.

This test method is applicable only if the program has the following properties:

(a) It contains only simple linear predicates of the form \(a_1v_1 + a_2v_2 + \ldots + a_kv_k \, \text{ROP} \, C\), where \(a_i\)'s and \(C\) are constants, and \(\text{ROP}\) is a relational operator.
(b) The path predicate of every path in the program is composed of a conjunction of such simple linear predicates.
(c) Coincidental (fortuitous) correctness of the program will not occur for any test case.
(d) A missing path error is not associated with the path being tested.
(e) Each border is produced by a simple predicate.
(f) The path corresponding to each adjacent domain computes a different subfunction.
(g) Functions defined in two adjacent subdomains yield different values for the same test point near the border.
(h) Any border defined by the program is linear, and if it is incorrect, the correct border is also linear.
(i) The input space is continuous rather than discrete.

The essence of domain-boundary geometrical analysis to be performed in test-case selection can be stated as follows.

Each border is a line segment in a k-dimensional space, which can be open or closed, depending on the relational operator in the predicate.

A *closed* border segment of a domain is actually part of that domain and is formed by a predicate with \(\geq\), \(=\), or \(\leq\) operator.

An *open* border segment of a domain forms part of the domain boundary, but does not constitute part of that domain, and is formed by a \(<\), \(>\), or \(\neq\) operator.

The test points (cases) selected will be of two types defined by their relative position with respect to the given border. An *on* test point lies on the given border while an *off* test point is a small distance \(\varepsilon\) from, and lies on the open side of, the given border.
When testing a closed border of a domain, the *on* test points are in the domain being tested, and each *off* test point is in some adjacent domain.

When testing an open border, each *on* test point is in some adjacent domain while the *off* test points are in the domain being tested.

Three test points will be selected for each border segment in an on-off-on sequence as depicted in Fig. 2.5.

![Figure 2.5. A sequence of on-off-on test points.](image)

The test will be successful if the test points $a$ and $b$ are computed by the subfunction defined for domain $D_i$, and the test point $c$ is computed by that defined for the neighboring domain $D_j$. This will be the case if the correct border is a line that intersects the line segments $ac$ and $bc$ at any point except $c$. In order to verify that the given border is identical to the correct one, we need to select the test point $c$ in such a way that its distance from the given border is $\varepsilon$, an arbitrarily small number.

The strategy is reliable for all three types of domain errors depicted in Fig. 2.6. The domain border may be erroneously placed in parallel below (Fig. 2.6a) or above (Fig. 2.6b) the correct one, or may intersect with it as shown in Fig. 2.6c. Observe that in Fig. 2.6a, $f_i(z)$ will be computed as $f_j(z)$; in Fig. 2.6b, $f_j(x)$ and $f_j(y)$ will be computed as $f_i(x)$ and $f_i(y)$, respectively; and in Fig. 2.6c, $f_j(y)$ will be computed as $f_i(y)$ instead. Since it is assumed that $f_i(p) \neq f_j(p)$ for any point $p$ near the border, all three types of domain errors can be detected by using this strategy.
Recall that two important assumptions were made at the outset: (1) all path predicates are numerical and linear, and (2) functions defined in two adjacent subdomains yield different values for the same test point near the border. The class of real-world programs satisfying assumption (1) is probably small. Assumption (2) is contrary to requirements in most practical applications: the common requirement is to have the functions defined in the adjacent subdomains to produce approximately the same, if not identical, values near the border. These two assumptions limit the applicability of this test strategy.

Now let us return to the subject of error classification touched on earlier. There are at least three other different classification schemes for programming error:

The following discussion about programming errors is a direct quote from WHCO80.

The basic ideas behind the classification of errors that we use are due to Howden, but our approach to defining them is somewhat more operational than that given in his paper.

From the previous sections, it is clear that a program can be viewed as (1) establishing an exhaustive partition of the input space into mutually exclusive domains, each of which corresponds to an execution path, and (2) specifying, for each domain, a set of assignment statements that compute the domain computation.
Thus we have a canonical representation of a program, which is a (possibly infinite) set of pairs \{(D_1, f_1), (D_2, f_2), ..., (D_i, f_i), ...\} where \(D_i\) is the i-th domain and \(f_i\) is the corresponding domain computation function.

Given an incorrect program \(P\), let us consider the changes in its canonical representation as a result of modifications performed on \(P\). It is assumed that these modifications are made using only permissible language constructs and results in a legal program.

**Definition**: A *domain-boundary modification* occurs if the modification results in a change in the \(D_i\) component of some \((D_i, f_i)\) pair in the canonical representation.

**Definition**: A *domain-computation modification* occurs if the modification results in a change in the \(f_i\) component of some \((D_i, f_i)\) pair in the canonical representation.

**Definition**: A *missing-path modification* occurs if the modification results in the creation of a new \((D_i, f_i)\) pair such that \(D_i\) is a subset of \(D_j\) occurring in some \((D_j, f_j)\) pair in the canonical representation of \(P\).

Notice that a particular modification (say a change of some assignment statement) can be a modification of more than one type. In particular, a missing path modification is also a domain boundary modification.

The error that occurs in a program can be classified on the basis of the modifications needed to obtain a correct program and consequent changes in the canonical representation. In general, there will be many correct programs and multiple ways to get a particular correct program. Hence, the error classification is not absolute, but relative to the particular correct program that would result from the series of modifications.

**Definition**: An incorrect program \(P\) can be viewed as having a *domain error* (computational error) (missing path error) if a correct program \(P^*\) can be created by a sequence of modifications, at least one of which is a domain boundary modification (domain computation modification) (missing path modification).

Several remarks are in order. The operational consequence of the phrase "can be viewed as" in the above definition is that the error classification is relative not only to a particular correct program, but also to a particular sequence of modifications. For instance, consider an error in a predicate interpretation such that an incorrect relational operator is employed, e.g., use of > instead of <. This could be viewed as a domain error, leading to a modification of the predicate, or as a computational error, leading to a modification of the functions computed on the two branches. The fact that it might be more possible to change the relational operator rather than the function computations is a consequence of the language
constructs, and it is not directly captured in the definitions of the types or error. In this paper, we would regard an error due to an incorrect relational operator as a domain error; it is a simpler modification to change the relational operator in the predicate than to interchange the set of assignment statements.

More specific characterization of these errors can be made in the context of specific programming language constructs. In particular, the following informal description directly relates domain errors to the predicate constructs allowed in the language.

A path contains a domain error if an error in some predicate interpretation causes a border segment to be "shifted" from its correct position or to have an incorrect relational operator. A domain error can be caused by an incorrectly specified predicate or by an incorrect assignment statement, which affects a variable, used in the predicate. An incorrect predicate or assignment statement may affect many predicate interpretations, and consequently cause more than one border to be in error.

The following paragraphs contain Howden's definition of program errors (excerpts from [HOWD76]).

The classes of program $P$ for which we will characterize the reliability of the path analysis testing strategy are associated with different kinds of errors in programs. We will be concerned with programs which are either correct or can be considered deviations from a hypothetical correct program $P^*$. The "differences" between $P$ and $P^*$ define errors in $P$. Each class of programs $\mathcal{D}$ will consist of correct programs $P^*$ together with incorrect programs $P$ which differ from $P^*$ by some type of error.

A path through a program corresponds to some possible flow of control. A path may be infeasible in the sense that there is no input data that will cause the path to be executed. Flows of control involving different numbers of iterations of loops are considered to be different paths. In general, a program containing loops will have an infinite number of paths. The errors in a program can be categorized in terms of their effects on the paths through the program.

Associated with each path through a program is the subset of the input domain that causes the path to be followed and a sequence of computations that is carried out by the path.

**Definition:** Suppose $P_i$ is a path through a program $P$. Then the path domain $D_i = D(P_i)$ for $P_i$ is the subset of the input domain which causes $P_i$ to be executed. The path computation $C_i = C(P_i)$ for $P_i$ is the function which is computed by the sequence of computations in $P_i$. 
The domain of the functions $C_i$ is considered to be the domain $D$ of $P$. During execution of the program $P$, each computation $C(P_i)$ is only carried out over the path domain $D(P_i)$. In general $C_i$ may not be defined over all of $D$ or, since $P$ may contain errors, even over all of $D_i$. In comparing two computations $C_i$ and $C_j$, we say that $C_i$ and $C_j$ are equivalent ($C_i = C_j$) if $C_i$ and $C_j$ are defined for the same subset $D'$ of $D$ and $C_i(x) = C_j(x)$ for all $x \in D'$.

The effect of program errors on the paths through a program can be described in terms of their effects on the path domains and path computations of the paths. Three simple classes of errors will be studied. If there is an isomorphism (one-to-one correspondence) between the paths $P_i$ of $P$ and the paths $P_i^*$ of the correct version $P^*$ of $P$ such that $D(P_i) = D(P_i^*)$ and $C(P_i) = C(P_i^*)$ for all paths, the $P = P^*$ and $P$ is correct. If $P$ is not correct, no isomorphism having these properties can be constructed. Either the domains or the computations, or both, of $P$ and $P^*$ will be different.

**Definition:** Suppose $P$ is an incorrect program for computing a function $F$ and $P^*$ is a correct program. Suppose there is an isomorphism between the paths $P_i$ of $P$ and the paths $P_i^*$ of $P^*$ such that for all pairs of paths $(P_i, P_i^*)$, $D(P_i) = D(P_i^*)$ but that for some pairs $(P_k, P_k^*)$, $C(P_k) \neq C(P_k^*)$. Then $P$ contains a path computation or computation error.

**Definition:** Suppose $P$ is an incorrect program for computing a function $F$ and $P^*$ is a correct program. Suppose there is an isomorphism between the paths $P_i$ of $P$ and the paths $P_i^*$ of $P^*$ such that for all pairs of paths $(P_i, P_i^*)$, $C(P_i) = C(P_i^*)$ but that for some pairs $(P_k, P_k^*)$, $D(P_k) \neq D(P_k^*)$. Then $P$ contains a path domain or domain error.

**Definition:** Suppose $P$ is an incorrect program for computing a function $F$ and $P^*$ is a correct program. Suppose there is an isomorphism between the paths $P_i$ of $P$ and a subset of the paths $P_i^*$ of $P^*$ such that $C(P_i^*) = C(P_i)$ and $D(P_i) \supset D(P_i^*)$ for all paths $P_i$ in $P$. Then $P$ contains a subcase error.

When a program contains a computation error we assume that the paths in $P$ and $P^*$ have been indexed so that $D(P_i) = D(P_i^*)$ for all paths. When it contains a domain or a subcase error we assume they have been indexed so that $C(P_i) = C(P_i^*)$ for all paths $P_i$ in $P$.

Different relationships can be proved between classes of statement type errors and errors which are defined in terms of the domains and computations for a program.
**Theorem:** Suppose that P is an incorrect program and that the only difference between P and P* is in some statement which does not affect the flow of control in P. Then P has a computation error.

**Theorem:** Suppose that P is an incorrect program and that the only difference between P and a correct program P* is in some statement which affects the flow of control in P. Then P may have a computation, domain, or subcase error.

The following paragraphs contain Goodenough and Gerhart's definition of program errors [GOGE77].

**Missing Control Flow Paths.** This type of error arises from failure to examine a particular condition; it results in the execution (or no execution) of inappropriate actions. For example, failure to test for a zero divider before executing a division may be a missing-path error. Other examples will be given later. This type of error results from failing to see that some condition or combination of conditions requires a unique sequence of actions to be handled properly. When a program contains this type of error, it may be possible to execute all control flow paths through the program without detecting the error. This is why exercising all program paths does not constitute a reliable test.

**Inappropriate Path Selection.** This type of error occurs when a condition is expressed incorrectly, and therefore, an action is sometimes performed (or omitted) under inappropriate conditions. For example, writing IF A instead of IF A AND B means that when A is true and B false, an inappropriate action will be taken or omitted. When a program contains this type of error, it is quite possible to exercise all statements and all branch conditions without detecting the error. This error can occur not merely through failure to evaluate the right combination of conditions, but also through failure to see that the method of evaluation is not adequate (e.g., determining whether three numbers are equal by writing (X+Y+Z)/3 = X).

**Inappropriate or Missing Action.** Examples are calculating a value using a method that does not necessarily give the correct result (e.g., d*d instead of d+d), or failing to assign a value to a variable, or calling a function or procedure with the wrong argument list. Some of these errors are revealed when the action is executed under any circumstances. Requiring all statements in a program to be executed will catch such errors. But sometimes the action is incorrect only under certain combinations of conditions; in this case, merely exercising the action (or the part of the program where a missing action should appear) will not necessarily reveal the error. For example, this is the case if d*d is written instead of d+d.

This classification of errors is useful because our goal is to detect errors by constructing appropriate tests. Other classifications are useful for other purposes, for example, to understand the effect of a programming language on software reliability, or to understand why errors occur. But insight into test reliability is
given by the proposed classification. For example, consider the test-data selection criterion, "choose data to exercise all statements and branch conditions in an implementation." In evaluating the reliability of this criterion, we would ask, "Will all construction, specification, design, and requirements errors always be detected by exercising programs with data satisfying this criterion?" Clearly, if a design error, for example, is manifested as a missing path in an implementation, then this criterion for test data selection will not be reliable. So our typology of implementation errors is useful to help understand factors impairing test reliability.

To see how these error classification schemes can be applied, let us now consider P, the program (written in C) listed below:

```c
main()
{
    int i, j, k, match;
    scanf("%d %d %d", &i, &j, &k);
    printf("%d %d %d\n", i, j, k);
    if (i <= 0 || j <= 0 || k <= 0) goto L500;
    match = 0;
    if (i != j) goto L10;
    match = match + 1;
    L10: if (i != k) goto L20;
        match = match + k;
    L20: if (j != k) goto L30;
        match = match + 3;
    L30: if (match != 0) goto L100;
        if (i+j <= k) goto L500;
        if (j+k <= i) goto L500;
        if (i+k <= j) goto L500;
        match = 1;
        goto L999;
    L100: if (match != 1) goto L200;
        if (i+j <= k) goto L500;
    L110: match = 2;
        goto L999;
    L200: if (match != k) goto L300;
        if (i+k <= j) goto L500;
        goto L110;
    L300: if (match != 3) goto L400;
        if (j+k <= i) goto L500;
        goto L110;
    L400: match = 3;
        goto L999;
    L500: match = 4;
    L999: printf("%d\n", match);
}
```

This program consists of 14 possible execution paths listed below:

<table>
<thead>
<tr>
<th>Index</th>
<th>Subdomain</th>
<th>Subfunction</th>
<th>Typical Input</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Program Analysis and Testing 2-27 © J. C. Huang 2004</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$P_i$</td>
<td>$S_i$</td>
<td>$(i, j, k)$</td>
</tr>
<tr>
<td>---</td>
<td>-----------------------------------------------------------------------</td>
<td>-------</td>
<td>----------------------</td>
</tr>
<tr>
<td>1</td>
<td>$(i &gt; 0)$ &amp;&amp; $(j &gt; 0)$ &amp;&amp; $(k &gt; 0)$</td>
<td>match = 1;</td>
<td>(4, 5, 6)</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i !\neq j)$ &amp;&amp; $(i !\neq k)$ &amp;&amp; $(j !\neq k)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i + j &gt; k)$ &amp;&amp; $(j + k &gt; i)$ &amp;&amp; $(i + k &gt; j)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$(i &gt; 0)$ &amp;&amp; $(j &gt; 0)$ &amp;&amp; $(k &gt; 0)$</td>
<td>match = 2;</td>
<td>(7, 7, 3)</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i == j)$ &amp;&amp; $(i !\neq k)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i + j &gt; k)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$(i &gt; 0)$ &amp;&amp; $(j &gt; 0)$ &amp;&amp; $(k &gt; 0)$</td>
<td>match = 2;</td>
<td>(1, 2, 1)</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i !\neq j)$ &amp;&amp; $(i == k)$ &amp;&amp; $(i == k)$ &amp;&amp; $(k == 1)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i + j &gt; k)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>(NOTE: Error: the program for this subdomain fails to check if $i+k&gt;j$)</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$(i &gt; 0)$ &amp;&amp; $(j &gt; 0)$ &amp;&amp; $(k &gt; 0)$</td>
<td>match = 2;</td>
<td>(4, 5, 4)</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i !\neq j)$ &amp;&amp; $(i == k)$ &amp;&amp; $(k !\neq 1)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i + k &gt; j)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$(i &gt; 0)$ &amp;&amp; $(j &gt; 0)$ &amp;&amp; $(k &gt; 0)$</td>
<td>match = 2;</td>
<td>(3, 2, 2)</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i !\neq j)$ &amp;&amp; $(j == k)$ &amp;&amp; $(k == 3)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(j + k &gt; i)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$(i &gt; 0)$ &amp;&amp; $(j &gt; 0)$ &amp;&amp; $(k &gt; 0)$</td>
<td>match = 2;</td>
<td>(7, 3, 3)</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i !\neq j)$ &amp;&amp; $(j == k)$ &amp;&amp; $(k == 3)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i + k &gt; j)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>(NOTE: Error: the program for this subdomain fails to check if $j+k&gt;i$)</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$(i &gt; 0)$ &amp;&amp; $(j &gt; 0)$ &amp;&amp; $(k &gt; 0)$</td>
<td>match = 3;</td>
<td>(8, 8, 8)</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i == j)$ &amp;&amp; $(i == k)$ &amp;&amp; $(j == k)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$(i &lt;= 0)$</td>
<td></td>
<td>$(j &lt;= 0)$</td>
</tr>
<tr>
<td>9</td>
<td>$(i &gt; 0)$ &amp;&amp; $(j &gt; 0)$ &amp;&amp; $(k &gt; 0)$</td>
<td>match = 4;</td>
<td>(2, 4, 9)</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i !\neq j)$ &amp;&amp; $(i !\neq k)$ &amp;&amp; $(j !\neq k)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i + j &lt;= k)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>$(i &gt; 0)$ &amp;&amp; $(j &gt; 0)$ &amp;&amp; $(k &gt; 0)$</td>
<td>match = 4;</td>
<td>(5, 2, 1)</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i !\neq j)$ &amp;&amp; $(i !\neq k)$ &amp;&amp; $(j !\neq k)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i + j &gt; k)$ &amp;&amp; $(j + k &lt;= i)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>$(i &gt; 0)$ &amp;&amp; $(j &gt; 0)$ &amp;&amp; $(k &gt; 0)$</td>
<td>match = 4;</td>
<td>(1, 6, 2)</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i !\neq j)$ &amp;&amp; $(i !\neq k)$ &amp;&amp; $(j !\neq k)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; $(i + j &gt; k)$ &amp;&amp; $(j + k &gt; i)$ &amp;&amp; $(i + k &lt;= j)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>$(i &gt; 0)$ &amp;&amp; $(j &gt; 0)$ &amp;&amp; $(k &gt; 0)$</td>
<td>match = 4;</td>
<td>(4, 4, 9)</td>
</tr>
</tbody>
</table>
&& (i == j) && (i != k) && (i + j <= k)

13 (i > 0) && (j > 0) && (k > 0) match = 4; (4, 9, 4) && (i != j) && (i == k) && (k != 1) && (i + k <= j)

14 (i > 0) && (j > 0) && (k > 0) match = 4; (5, 2, 2) && (i != j) && (j == k) && (k != 3) && (j + k <= i).

This program is written to implement the specification described below:

Write a program that takes three positive integers as input and determine if they represent three sides of a triangle, and if they do, indicate what type of triangle it is. To be more specific, it should read three integers and set a flag as follows:

- if they represent a scalene triangle then set it to 1,
- if they represent an isosceles triangle then set it to 2,
- if they represent an equilateral triangle then set it to 3,
- and if they do not represent a triangle then set it to 4.

There are two errors in this program: variable k in statement L200 and the one above L20 should be constant 2 instead. Although these errors are not of any of the types exemplified in the Goodenough and Gerhart's discussion on error classification quoted above, they are not improbable. This type of error frequently results from the use of “Replace All” command in editing the source code, especially when the source code is contained in a large file.

Thus P*, a possible correct program, would be

```c
main ()
{
    int i, j, k, match;
    scanf("%d %d %d", &i, &j, &k);
    printf("%d %d %d\n", i, j, k);
    if (i <= 0 || j <= 0 || k <= 0) goto L500;
    match = 0;
    if (i != j) goto L10;
    match = match + 1;
    L10: if (i != k) goto L20;
    match = match + 2;
    L20: if (j != k) goto L30;
    match = match + 3;
    L30: if (match != 0) goto L100;
    if (i+j <= k) goto L500;
    if (j+k <= i) goto L500;
    if (i+k <= j) goto L500;
    match = 1;
    goto L999;
```
L100: if (match != 1) goto L200;  
    if (i+j <= k) goto L500;  
L110: match = 2;  
    goto L999;  
L200: if (match != 2) goto L300;  
    if (i+k <= j) goto L500;  
    goto L110;  
L300: if (match != 3) goto L400;  
    if (j+k <= i) goto L500;  
    goto L110;  
L400: match = 3;  
    goto L999;  
L500: match = 4;  
L999: printf("%d\n", match);  
}

Program P* consists of 12 possible execution paths listed below:

<table>
<thead>
<tr>
<th>Index</th>
<th>Subdomain P_i*</th>
<th>Subfunction S*_i</th>
<th>Typical Input (i, j, k)</th>
</tr>
</thead>
</table>
| 1     | (i > 0) && (j > 0) && (k > 0)  
    && (i != j) && (i != k) && (j != k)  
    && (i + j > k) && (j + k > i) && (i + k > j) | match = 1; | (4, 5, 6) |
| 2     | (i > 0) && (j > 0) && (k > 0)  
    && (i == j) && (i != k)  
    && (i + j > k) | match = 2; | (7, 7, 3) |
| 3     | (i > 0) && (j > 0) && (k > 0)  
    && (i != j) && (i == k)  
    && (i + k > j) | match = 2; | (4, 5, 4) |
| 4     | (i > 0) && (j > 0) && (k > 0)  
    && (i != j) && (j == k)  
    && (j + k > i) | match = 2; | (3, 2, 2) |
| 5     | (i > 0) && (j > 0) && (k > 0)  
    && (i == j) && (i == k) && (j == k) | match = 3; | (8, 8, 8) |
| 6     | (i <= 0) || (j <= 0) || (k <= 0) | match = 4; | (0, 5, -3) |
| 7     | (i > 0) && (j > 0) && (k > 0)  
    && (i != j) && (i != k) && (j != k)  
    && (i + j <= k) && (i + k <= j) && (j + k <= i) | match = 4; | (2, 4, 9) |
| 8     | (i > 0) && (j > 0) && (k > 0)  
    && (i != j) && (i != k) && (j != k)  
    && (i + j <= k) && (i + k <= j) && (j + k <= i) | match = 4; | (5, 2, 1) |
Note that the correct program is obtained by replacing k in statement L200 as well as the one above L20 by 2. By White and Cohen's definition, it is apparently a domain boundary modification. To be more specific,

- $D_3^*$ is the result of change in $D_3$ or $D_4$,
- $D_4^*$ is the result of change in $D_5$ or $D_6$,
- $D_{11}^*$ is the result of change in $D_{13}$, and
- $D_{12}^*$ is the result of change in $D_{14}$.

There is no domain computation modification. Therefore, again, by White and Cohen's definition, the two k's in the program together constitute a domain error.

Next, with the correct $P^*$ given above, we see that Howden's error classification scheme cannot be applied. There is no isomorphism between the paths $P_i$ of $P$ and the paths $P_i^*$ of $P^*$. Thus we cannot say that $P$ has a path domain error or a path computation error. Furthermore, there is no isomorphism between the paths $P_i$ of $P$ and a subset of the paths $P_i^*$ of $P^*$ such that $C(P_i^*) = C(P_i)$ and $D(P_i) \supset D(P_i^*)$ for all paths $P_i$ in $P$. Thus we cannot say that $P$ has a subcase error.

Remark: Observe that there is an isomorphism between $\{1, 2, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14\}$ and $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ such that $C(P_i^*) = C(P_i)$ and $D(P_i) \supset D(P_i^*)$. But Howden’s definition requires that $D(P_i) \supset D(P_i^*)$, not $D(P_i) \subset D(P_i^*)$ as the case here.

**Exercise:** Based on the table of execution paths of $P$ given above, construct a correct program $P^*$ such that Howden's classification scheme becomes directly applicable.
By Goodenough and Gerhart's definition, the program has an inappropriate path selection error. As indicated on the path table, \( P_3 \) fails to check if \( i+k > j \), and \( P_6 \) fails to check if \( j+k > i \).

By the classification scheme given at the beginning of this section, this program has a domain error. To see why this is so, we analyze the specification to yield

\[
X = \{(a, b, c) \mid a, b, \text{and } c \text{ are integers}\}
\]
\[
Y = \{1, 2, 3, 4\}
\]
\[
f = \{f_1, f_2, f_3, f_4\}
\]

\[
\text{TRIANGLE} \equiv (a, b, c) \in X \land a>0 \land b>0 \land c>0 \land a+b>c \land b+c > a \land c+a > b
\]
\[
X_1 = \{(a, b, c) \mid \text{TRIANGLE} \land a \neq b \land b \neq c \land c \neq a\}
\]
\[
X_2 = \{(a, b, c) \mid \text{TRIANGLE} \land (a = b \land b \neq c \lor b = c \land c \neq a \lor c = a \land a \neq b)\}
\]
\[
X_3 = \{(a, b, c) \mid \text{TRIANGLE} \land a = b \land b = c\}
\]
\[
X_4 = \{(a, b, c) \mid \neg \text{TRIANGLE}\}
\]

\[
f_1(x) = \text{flag}=1'
\]
\[
f_2(x) = \text{flag}=2'
\]
\[
f_3(x) = \text{flag}=3'
\]
\[
f_4(x) = \text{flag}=4'
\]

and

\[
C_1 \equiv \text{TRIANGLE} \land a \neq b \land b \neq c \land c \neq a
\]
\[
C_2 \equiv \text{TRIANGLE} \land (a = b \land b \neq c \lor b = c \land c \neq a \lor c = a \land a \neq b)
\]
\[
C_3 \equiv \text{TRIANGLE} \land a = b \land b = c
\]
\[
C_4 \equiv \neg \text{TRIANGLE}
\]

Note that neither \( P_3 \) nor \( P_6 \) imply \( C_m \) for any \( 1 \leq m \leq 4 \). Hence the program contains a domain error.

In summary, the present error-classification method led us to the conclusion that the replacement of two occurrences of “2” in the source code by “k” together constitute a domain error. This error, therefore, is most likely to be detected by a test method that is effective in discovering domain errors, especially if the analysis or testing tools used also provide information about path domain definition, such as that shown in the table of execution paths given previously. An additional strength of this method is that it does not require the construction of a correct program.

Although in this case White and Cohen method also correctly determines the error to be of domain type, in general the result produced is dependent not only on the correct program used, but also dependent on the sequence in which the errors are corrected, if there are more than one. Thus it may not produce a clear result. For example, if the method is followed literally, one finds the program having domain as well as computational errors.
Howden’s method requires construction of a correct program as well. In this example, the most “naturally” or “intuitively” correct program to use should be the one given above, with two offending k’s removed. Unfortunately, the method cannot be applied unless a special correct program is constructed (which can be done based on the path table of the incorrect program).

Implicit in Howden’s and Goodenough and Gerhart’s methods is the assumption that an incorrect program always consists of no more execution paths than the correct one. As exemplified above, this assumption is invalid. In fact, most real programs contain more (possible) execution paths required to correctly implement the intended function. Programs can be made compact, i.e., to have as few execution paths as possible, but there is rarely an economical justification for doing that.

In summary, none of the error-classification methods discussed above classifies a programming error uniquely, and thus does not serve its intended purpose well. We will revisit this subject in a latter chapter.

### 2.4 PROGRAM MUTATION

A mutant of a program P is defined as a program P’ derived from P by making one of a set of carefully defined syntactic changes in P. Typical changes include replacing one arithmetic operator by another, one statement by another, and so forth [BLSD78, DELS78].

For example, consider the following C program:

```c
main() /* compute sine function */
{
    int i;
    float e, sum, term, x;

    scanf("%f %f" x, e);
    printf("x= %10.6f  e= %10.6f\n" x, e);
    term = x;
    for (i = 3; i <= 100 && term > e; i = i + 2)
    {
        term = term * x * x / (i * (i - 1));
        if (i % 2 == 0) sum = sum + term;
        else sum = sum - term;
    }
    printf("sin(x)= %8.6f\n" sum);
}
```

Listed below are some possible mutants of this program:

```c
/* a mutant obtained by changing variable x to a constant 0 */
main() /* compute sine function */
```

{  
  int i;
  float e, sum, term, x;

  scanf("%f %f", x, e);
  printf("x= %10.6f  e= %10.6f\n", x, e);
  term = 0;
  for (i = 3; i <= 100 && term > e; i = i + 2)
  {
    term = term * x * x / (i * (i - 1));
    if (i % 2 == 0) sum = sum + term;
    else sum = sum - term;
  }
  printf("sin(x)= %8.6f\n", sum);
}

/* a mutant obtained by changing a relational operator */
/* in i <= 100 to i >= 100 */
main()   /* compute sine function */
{
  int i;
  float e, sum, term, x;

  scanf("%f %f", x, e);
  printf("x= %10.6f  e= %10.6f\n", x, e);
  term = x;
  for (i = 3; i >= 100 && term > e; i = i + 2)
  {
    term = term * x * x / (i * (i - 1));
    if (i % 2 == 0) sum = sum + term;
    else sum = sum - term;
  }
  printf("sin(x)= %8.6f\n", sum);
}

/* a mutant obtained by changing constant 0 to 1 */
main()   /* compute sine function */
{
  int i;
  float e, sum, term, x;

  scanf("%f %f", x, e);
  printf("x= %10.6f  e= %10.6f\n", x, e);
  term = x;
  for (i = 3; i <= 100 && term > e; i = i + 2)
  {
    term = term * x * x / (i * (i - 1));
    if (i % 2 == 1) sum = sum + term;
    else sum = sum - term;
  }
  printf("sin(x)= %8.6f\n", sum);
}
Program mutation has been utilized by DeMillo, Lipton, and Sayward as the basis for an interactive program testing system [DELS78]. The basic idea involved can be explained as follows.

Let \( P' \) be a mutant of some program \( P \). A test case \( t \) is said to differentiate \( P' \) from \( P \) if an execution of \( P \) and \( P' \) with \( t \) produced different results.

If \( t \) failed to differentiate \( P' \) from \( P \), either \( P' \) is functionally equivalent to \( P \), or \( t \) is ineffective in revealing the changes (errors) introduced into \( P' \). Thus a test method can be formulated as follows.

Given a program \( P \), which is written to implement function \( f \),

\[ \text{Step 1: } \text{Generate } \Phi, \text{ a set of mutants of } P, \text{ by using a set of mutation operations.} \]

\[ \text{Step 2: } \text{Identify and delete all mutants in } \Phi \text{ which are equivalent to } P. \]

\[ \text{Step 3: } \text{Find } T, \text{ a set of test cases that as a whole differentiate } P \text{ from every mutant in } \Phi, \text{ to test-execute } P \text{ and elements of } \Phi. \]

These three steps constitute a \( \Phi \) mutant test. The test is successful if \( P(t) = f(t) \) for all \( t \in T \). A successful \( \Phi \) mutant test implies that the program is free of any errors introduced into \( P \) in the process of constructing \( \Phi \). If we can assume that \( P \) was written by a competent programmer who had a good understanding of the task to be performed and was not capable of making any mistakes other than those introduced in constructing \( \Phi \), we can conclude that \( P \) is correct. Further discussions on the significance of a successful \( \Phi \) mutant test can be found in BDLS80.

How can \( \Phi \) be constructed in practice? Budd et al. [DELS78] suggested that a set of syntactic operations can be used to construct the desired mutants systematically. The definition of such operations obviously would be language dependent. For Fortran programs, the mutation operations may include the following [DELS78]:

1. \( \text{Constant Replacement:} \) Replacing a constant, say, \( C \), with \( C+1 \) or \( C-1 \), e.g., statement \( A = 0 \) becomes \( A = 1 \) or \( A = -1 \).

2. \( \text{Scalar Replacement:} \) Replacing one scalar variable with another, e.g., statement \( A = B - 1 \) becomes \( A = D - 1 \).

3. \( \text{Scalar for Constant Replacement:} \) Replacing a constant with a scalar variable, e.g., statement \( A = 1 \) becomes \( A = B \).

4. \( \text{Constant for Scalar Replacement:} \) Replacing a scalar variable with a constant, e.g., statement \( A = B \) becomes \( A = 5 \).
(5) **Source Constant Replacement:** Replacing a constant in the program with another constant found in the same program, e.g., statement $A = 1$ becomes $A = 11$, where the constant 11 appears in some other statement.

(6) **Array Reference for Constant Replacement:** Replacing a constant with an array element, e.g., statement $A = 2$ becomes $A = B(2)$.

(7) **Array Reference for Scalar Replacement:** Replacing a scalar variable with an array element, e.g., statement $A = B + 1$ becomes $A = X(1) + 1$.

(8) **Comparable Array Name Replacement:** Replacing a subscripted variable with the corresponding element in another array of the same size and dimension, e.g., statement $A = B(2, 4)$ becomes $A = D(2, 4)$.

(9) **Constant for Array Reference Replacement:** Replacing an array element with a constant, e.g., statement $A = X(1)$ becomes $A = 5$.

(10) **Scalar for Array Reference Replacement:** Replacing a subscripted variable with a non-subscripted variable, e.g., statement $A = B(1) - 1$ becomes $A = X - 1$.

(11) **Array Reference for Array Reference Replacement:** Replacing a subscripted variable by another, e.g., statement $A = B(1) + 1$ becomes $A = D(4) + 1$.

(12) **Unary Operator Insertion:** Insertion of one of the unary operators such as - (negation) in front of any data reference, e.g., statement $A = X$ becomes $A = -X$.

(13) **Arithmetic Operator Replacement:** Replacing an arithmetic operator (i.e., +, -, *, /, **) with another, e.g., statement $A = B + C$ becomes $A = B - C$.

(14) **Relational Operator Replacement:** Replacing a relational operator (i.e., =, <>, <=, <, >=, >) with another, e.g., expression $X = Y$ becomes $X <> Y$.

(15) **Logical Connector Replacement:** Replacing a logical connector (i.e., .AND., .OR., .XOR.) with another, e.g., expression $A .AND. B$ becomes $A .OR. B$.

(16) **Unary Operator Removal:** Deleting any unary operator, e.g., statement $A = -B/C$ becomes $A = B/C$.

(17) **Statement Analysis:** Replacing a statement with a trap statement that causes the program execution to be aborted immediately, e.g., statement GOTO 10 becomes CALL TRAP.

(18) **Statement Deletion:** Deleting a statement from the program.

(19) **Return Statement:** Replacing a statement in a subprogram by a RETURN statement.
(20) **Goto Statement Replacement:** Replacing the statement label of a GOTO statement by another, e.g., statement GOTO 20 becomes GOTO 30.

(21) **DO Statement End Replacement:** Replacing the end label of a DO statement with some other label, e.g., statement DO 5 I = 2, 10 becomes DO 40 I = 2, 10.

(22) **Data Statement Alteration:** Changing the values of variables assigned by a DATA statement (in FORTRAN), e.g., statement DATA Y /22/ becomes DATA Y /31/.

**Exercise 2.4.1:** Give a set of syntactic operations similar to the ones listed above for generating mutants of a C program.

A mutant in \( \Phi \) is created by applying one mutation operation to one statement in the program. The set \( \Phi \) consists of all possible mutants constructed by applying every mutation operation to every applicable statement in the program.

In the second step of the mutant test method, after all possible mutants are generated, one needs to identify and to remove mutants that are functionally equivalent to the program. In general, determining the equivalency of two programs is a problem unsolvable in the sense that there does not exist a single effective algorithm for this purpose. Although a mutant differs from the original program only by one statement, determination of equivalency may become problematic in practice. This difficulty remains to be a major obstacle in making program mutation as a practical method for program testing.

Observe that a mutant of program P is created by altering a statement in P. A test case would not differentiate the mutant from P unless this particular statement is involved in the test-execution. Thus, to find a test case to differentiate a mutant is to find an input to P that causes the statement in question to be "exercised" during the test. Of course, causing the statement to be exercised is only a necessary condition. For some input, a non-equivalent mutant may produce an output fortuitously identical to that of P. A sufficient condition, therefore, is that the mutant and P do not produce the same output for that input.

For example, consider a C program that includes the following statement:

```c
while (fahrenheit <= upper) {
    celsius = (5.0 / 9.0) * (fahrenheit - 32.0);
    fahrenheit = fahrenheit + 10.0;
}
```

A mutant obtained by replacing constant 5.0 with 4.0, for instance, thus includes the following statement:

```c
while (fahrenheit <= upper) {
    celsius = (4.0 / 9.0) * (fahrenheit - 32.0);
    fahrenheit = fahrenheit + 10.0;
}
```
Obviously, any test case satisfying \( \text{fahrenheit} > \text{upper} \) will not be able to differentiate this mutant because the mutated statement in the loop body will not be executed at all. To differentiate this mutant, the test case must cause that statement to be exercised. In addition, the test case must cause the mutant to produce a different output. A test case that set \( \text{fahrenheit} = \text{upper} = 32.0 \) just before the loop will not do it (because the factor \( \text{fahrenheit} - 32.0 \) will become zero, and variable \( \text{celsius} \) will be set to zero regardless of the constant used there). Such a test case may satisfy the need of a statement-coverage test because it causes the statement in question to be executed, but not the need of this mutation test because it will cause the mutant to produce an output fortuitously identical to that of the original program.

A \( \Phi \) mutant test, therefore, is at least as thorough as a statement-coverage test (i.e., a test in which every statement in the program is exercised at least once). This is so because there is no program statement that can be made absolutely error-free, even if it is written by a competent programmer. That means \( \Phi \) should contain at least one mutant from every statement in the program, if the \( \Phi \) mutant test is to be effective. That in turn means that the set of test cases used should have every statement in the program exercised at least once, so that all mutants can be differentiated from the program.

A \( \Phi \) mutant test may be more thorough than a statement-coverage test because if a test case failed to differentiate a non-equivalent mutant in \( \Phi \), additional test cases must be employed. These additional test cases make it possible to detect errors of the type induced by the mutation operation used.

This added thoroughness is achieved with an enormous cost, however.

Suppose that a given program \( P \) has \( m \) mutants, and \( n \) test cases are used to differentiate all mutants. The number of \textit{mutant tests} (i.e., test-executions of mutant) needed depends on the number of mutants each test case is able to differentiate, and the order in which the test cases are used. In the best case, the first test case differentiates all but \( n-1 \) mutants with \( m \) test executions. The second test case differentiates one mutant with \( n-1 \) test executions, and so on and so forth. In general, the \( i \)-th test case differentiates one mutant with \( n-i+1 \) test executions (for all \( 1 < i \leq n \)). Thus the total number of mutant tests required will be

\[
m + (n - 1) + (n - 2) + ... + 1 = m + ((n - 1) + 1)/2 \times (n - 1) = m + n(n - 1)/2
\]

In the worst case, each of the first \( n-1 \) test cases differentiates only one mutant, and the last test case differentiates the remaining \( m-(n-1) \) mutants. The total number of mutant tests required will be

\[
m + (m - 1) + (m - 2) + ... + (m - (n - 1)) \\
= mn - (1 + 2 + ... + (n - 1)) \\
= mn - n(n - 1)/2
\]

These two figures represent two extreme cases. In average, the number of test executions required will be
\[
\frac{(m + n(n - 1)/2 + mn - n(n - 1)/2)}{2} = \frac{(m + mn)}{2} = \frac{m(n + 1)}{2}
\]

The number of test executions required for the 10 programs studied in [BDSL80] are tabulated below for reference.

<table>
<thead>
<tr>
<th>Program</th>
<th>Size (no. of statements)</th>
<th>No. of mutants</th>
<th>No. of test cases needed (minimum) (maximum) (average)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>900</td>
<td>4 906 3,594 2,250</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>773</td>
<td>7 794 5,390 3,092</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>383</td>
<td>7 404 2,660 1,532</td>
</tr>
<tr>
<td>4</td>
<td>62</td>
<td>5,033</td>
<td>34 5,529 170,626 88,078</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>3,348</td>
<td>13 3,426 43,446 23,436</td>
</tr>
<tr>
<td>6</td>
<td>57</td>
<td>8,026</td>
<td>17 8,162 136,306 72,234</td>
</tr>
<tr>
<td>7</td>
<td>43</td>
<td>1,028</td>
<td>40 1,808 40,340 21,074</td>
</tr>
<tr>
<td>8</td>
<td>55</td>
<td>6,317</td>
<td>5 6,327 31,575 18,951</td>
</tr>
<tr>
<td>9</td>
<td>34</td>
<td>945</td>
<td>9 981 8,469 4,725</td>
</tr>
<tr>
<td>10</td>
<td>19</td>
<td>567</td>
<td>12 633 6,738 3,686</td>
</tr>
</tbody>
</table>

Note that in other test methods, the number of test-executions is equal to the number of test cases needed to complete the test. In the mutant test, additional executions of mutants have to be carried out with the same test cases. The last three columns in the above table indicate the minimum, maximum, and average number of test-executions required.

Take program 8 in the above table as example. Only 5 test cases (and hence test-executions) are required to complete a statement-coverage test. For a mutant test, somewhere between 6,327 and 31,575 additional test-executions are required. Assuming that each test-execution can be completed in 10 seconds (including the time needed to analyze the test result), these additional test-executions will consume somewhere between 17.5 and 87.7 hours of time. It remains to be shown that a mutant test is cost effective in comparison with other test methods.

**Exercise 2.4.2:** Given a program and a set of its mutants, how to select test cases so that the number of test cases required to differentiate all mutants is minimum?

**Exercise 2.4.3:** Is the order in which the test cases are used affects the number of test-executions required? If so, how to order a set of given test cases so the number of test-executions required is minimum?