

An Approach to Program Testing

J. C. HUANG

Department of Computer Science, University of Houston, Houston, Texas 77004

One of the practical methods commonly used to detect the presence of errors in a computer program is to test it for a set of test cases. The probability of discovering errors through testing can be increased by selecting test cases in such a way that each and every branch in the flowchart will be traversed at least once during the test. This tutorial describes the problems involved and the methods that can be used to satisfy the test requirement.

Keywords and phrases: program testing, program analysis, program instrumentation, test-case generation, path analysis, path predicate, directed graph

CR Categories: 3.64, 4.20, 4.42, 1.6, 5.25

CONTENTS

INTRODUCTION
1. TEST CRITERIA
2. TEST PROCEDURES
3. PATH PREDICATES
4. TEST-CASE GENERATION
CONCLUDING REMARKS
ACKNOWLEDGMENTS
REFERENCES

This paper appeared in ACM *Computing Surveys*, vol. 7, no. 3, September 1975. Copyright () 1976, Association for Computing Machinery, Inc. The privilege to post this paper on the web site for academic use was granted by permission of the Association for Computing Machinery.

INTRODUCTION

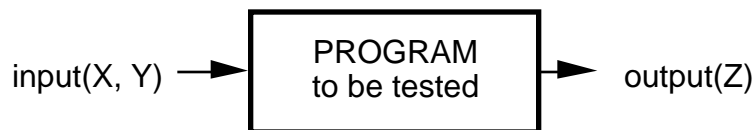
Given a computer program, how can we determine whether or not it will do exactly what it is designed to do? This question is not only intellectually challenging, but also of primary importance in practice.

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. As described in a recent survey by Elspas et al. [1], there have been considerable efforts to develop such techniques. 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 this paper. 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, as aptly pointed out by Dijkstra in [2], it is in general impractical for us to do this simply because of the number of possible input cases involved. The following is analogous to Dijkstra's explanation.

Suppose the program to be tested has two input variables and one output variable as depicted below:



If, for an assignment of values to the input variables X and Y , the output variable Z 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 X and Y , then we will be able to determine its correctness. The difficulty here is that, even for a program with only two input variables, the number of possible assignments will be prohibitively large. To see why this is so, let us assume that X and Y are integer variables. Furthermore, let us assume that the program is to be run on a computer with 32-bit registers. Then there are $2^{32} \times 2^{32} = 2^{64}$ possible assignments to the input pair (X, Y) . Now suppose this program is relatively small, and on the average it takes one millisecond to execute the program once. Then it will take more than 50 billion years for us to complete the test!

This example clearly indicates 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 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 (black-box view of the program) that all test cases (i.e., $(X, Y, f(X, Y))$) are equally important, although in fact they are not. When we *look inside the black box*, i.e., when we examine the structure 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 can be statistically significant.

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. 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 an obvious way to increase the probability of discovering errors through program testing is to have each and every statement in the program executed at least once during the test. A set of test cases that achieves this test goal, which can be constructed based on the program structure, is certainly of special significance. We shall explain this in detail in the next section.

1. TEST CRITERIA

A common 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 1-3, and then, at the end of this section, formalize the concepts involved.

Let us consider the program whose flowchart is given in Figure 1. 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 $e/2$.

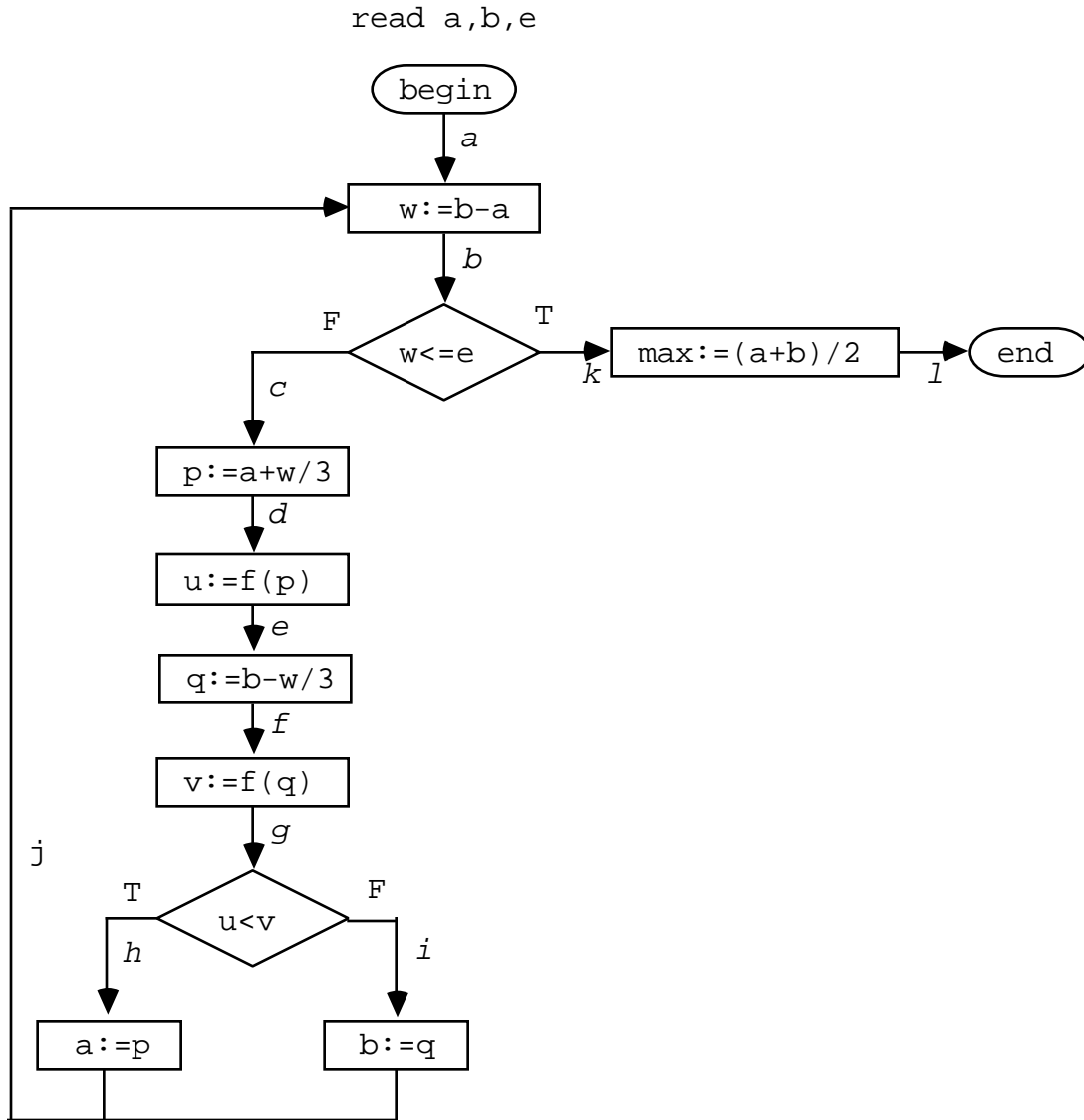


Figure 1. A program.

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. 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 \max for all three cases are found to be correct in the test. What can we say about the design of this test?

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.

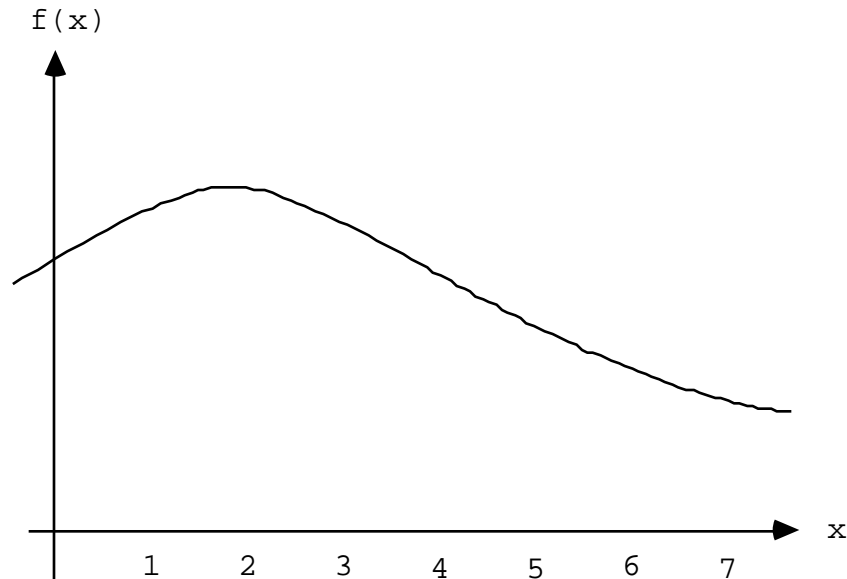


Figure 2. The function plot of $f(x)$.

The functional plot given in Figure 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.

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 3, where the flow of control is transferred to a wrong place as indicated by the dotted line. In this case the program produces correct results as long as the input data cause P 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 P 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 flowchart) 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.

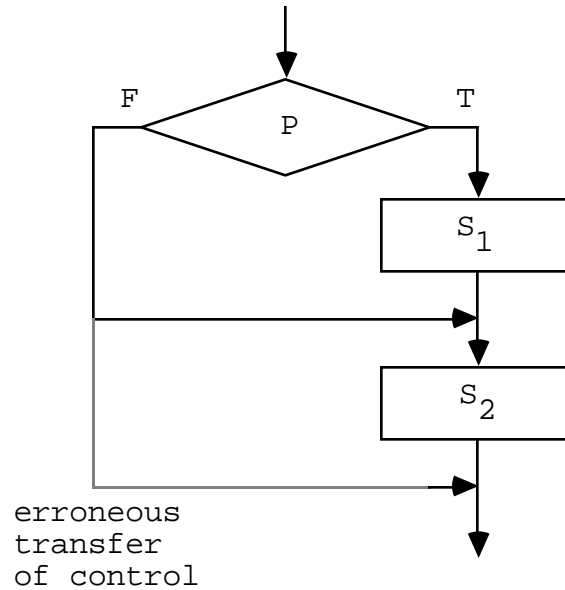


Figure 3. A type of programming error.

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 small 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 flowchart 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 P false, in addition to the one that satisfies P , in order to have every branch in Figure 3 traversed at least once. Hence our chances of discovering the error will be greatly improved, because the program will most likely produce all erroneous result for the test case that makes P 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 a flowchart is a connected graph in which each node is on some path from the entry to the exit (assuming that there is no inaccessible code in the program text). Since each branch emanates from a node and terminates at another, it is obvious that every statement has to be executed at least once in order to have every branch traversed at least once. 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 3.

It is interesting to see what this new test requirement 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 union of a set of partial functions, each defined on a subset of the intended input domain. Each partial function is associated with an executable path in such a way that the sequence of noncontrol 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* of that path. The path predicate essentially defines the membership of a subdomain in which the corresponding partial function is defined. Roughly speaking, to test a program by having each and every branch traversed at least once is to test the correctness of each and every

partial function for at least one point in the subdomain in which it is defined.

From the above analysis we see that 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 be checked for at least one point in the subdomain in which it is defined. However, we must remember that, for some input data, some programs may produce results that are fortuitously correct, as we have demonstrated 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.

Before we proceed to describe the techniques available for program testing, it may be helpful to recapitulate the ideas discussed thus far in a more precise manner:

- 1) A *program block* is a sequence of one or more statements having the property that, if a member statement is executed, all other statements in the sequence will also be executed.
- 2) A *program graph* is a directed graph in which each node is associated with a program block. Furthermore, there is an edge from node i to node j labeled by predicate P if and only if upon execution of the program block associated with node i the control will be transferred to the entry of the program block associated with node j , if predicate P is true. Note that an edge will be traversed in execution only if the associated predicate is true.
- 3) A *path* in a (program) graph is defined as usual [3, 4]. Each path in a program graph is associated with a predicate such that the path will be traversed only if the associated path predicate is true. A *minimal covering set* of path predicates is a set of predicates such that: a) each element in is associated with some path from the entry to the exit of the program graph; b) satisfaction of all predicates in entails that every edge in the program graph will be traversed at least once during execution; and c) no proper subset of has these properties.
- 4) Suppose the program is designed to compute the values of function $f: X \rightarrow Y$. This function usually is expressed in the program text as a union of functions $f_i: X_i \rightarrow Y$, where $X = X_1 \cup X_2 \cup \dots \cup X_n$ and f_i is f restricted to X_i for all $1 \leq i \leq n$. Furthermore, $X_i = \{x \mid x \text{ in } X \text{ and } p_i \text{ in } \text{ and } p_i(x)\}$

Here by $p_i(x)$ we mean x satisfies predicate p_i .

Based on the above discussion we may now define a measure of thoroughness of a test. We shall define a minimally thorough set of tests to be a set of pairs $\{(x_1, f(x_1)), (x_2, f(x_2)), \dots, (x_n, f(x_n))\}$ such that x_i satisfies the i th predicate p_i . In other words, a test is minimally thorough if each and every branch in the flowchart is traversed at least once during the test.

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. This can be attested to by the fact that many major software-testing tools reported so far have incorporated in them some provision to assist the programmers to achieve the test goal of having every branch in the flowchart traversed at least once [4 - 9]. The disagreement seems to be in how much more is needed beyond this basic requirement to entitle a test program to be considered adequate.

The objective of the remainder of this article is to develop methods for constructing a minimal covering set of path predicates and corresponding set of tests for a given program.

2. TEST PROCEDURES

A simple and practical way to determine the degree of thoroughness of a test is to calibrate the program to be tested by using a set of software counters (see, e.g., [7]). To do so, we first have to identify a (minimal) set of points on the flowchart such that, if we know the number of times each point is crossed, we can determine the number of times each branch is traversed. We then prepare the program for testing by inserting counters at these points (the counters can be removed later when the program is considered debugged, assuming no further performance data are required). After having the program tested for a number of test cases, we can determine the degree of thoroughness by examining the resulting counter values.

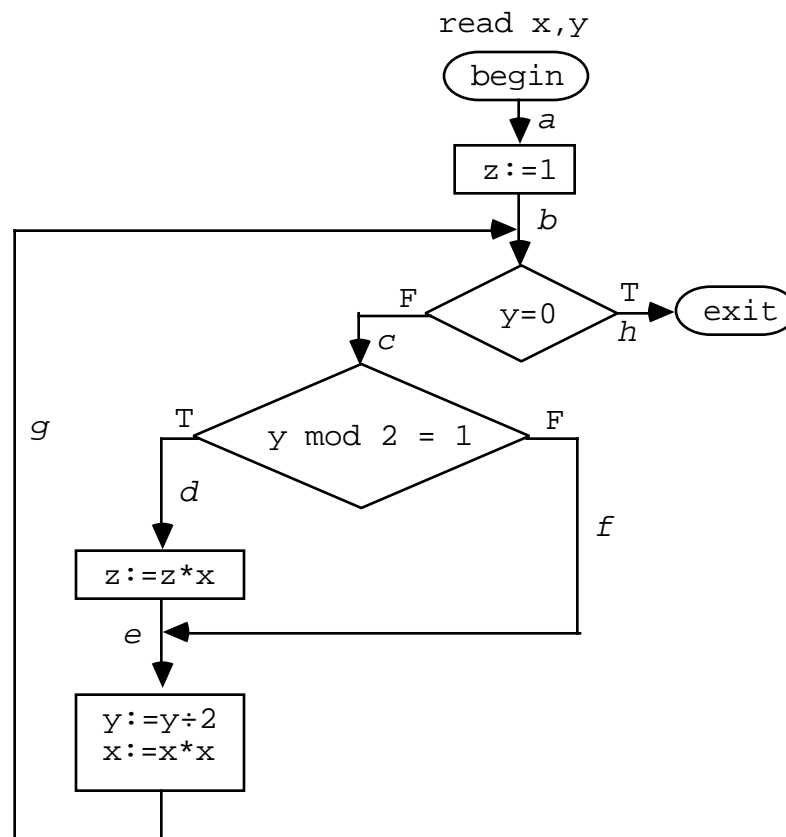


Figure 4. A program.

Where do we place the counters in order to determine whether or not every branch is traversed at least once? For the purpose of finding an answer to this question, it is convenient to introduce the concept of a decision-to-decision path [7]. A decision-to-decision path is defined to be a path in a flowchart such that a) its first constituent edge emanates either from an entry node or a decision box; b) its last constituent edge terminates either at a decision box or at an exit node; and c) there are no decision boxes on the path except those at both ends. For example, consider the flowchart shown in Figure 4. Here the paths *deg* and *abh* are decision-to-decision paths, but paths *abh* and *eg* are not. It should not be difficult to verify that in a flowchart a) every branch is on some

decision-to-decision path, and b) if the first branch of a decision-to-decision path is traversed during the execution, then every branch on that path will also be traversed. For convenience, let us call the first branch of a decision-to-decision path a reference branch. It is easy to identify the reference branches in a flowchart because, by definition of a decision-to-decision path, those and only those emanating from either an entry node or a decision box are the reference branches of that flowchart. Now the answer to our question is obvious. To determine whether or not every branch is traversed at least once, we need only to place a counter on each and every reference branch.

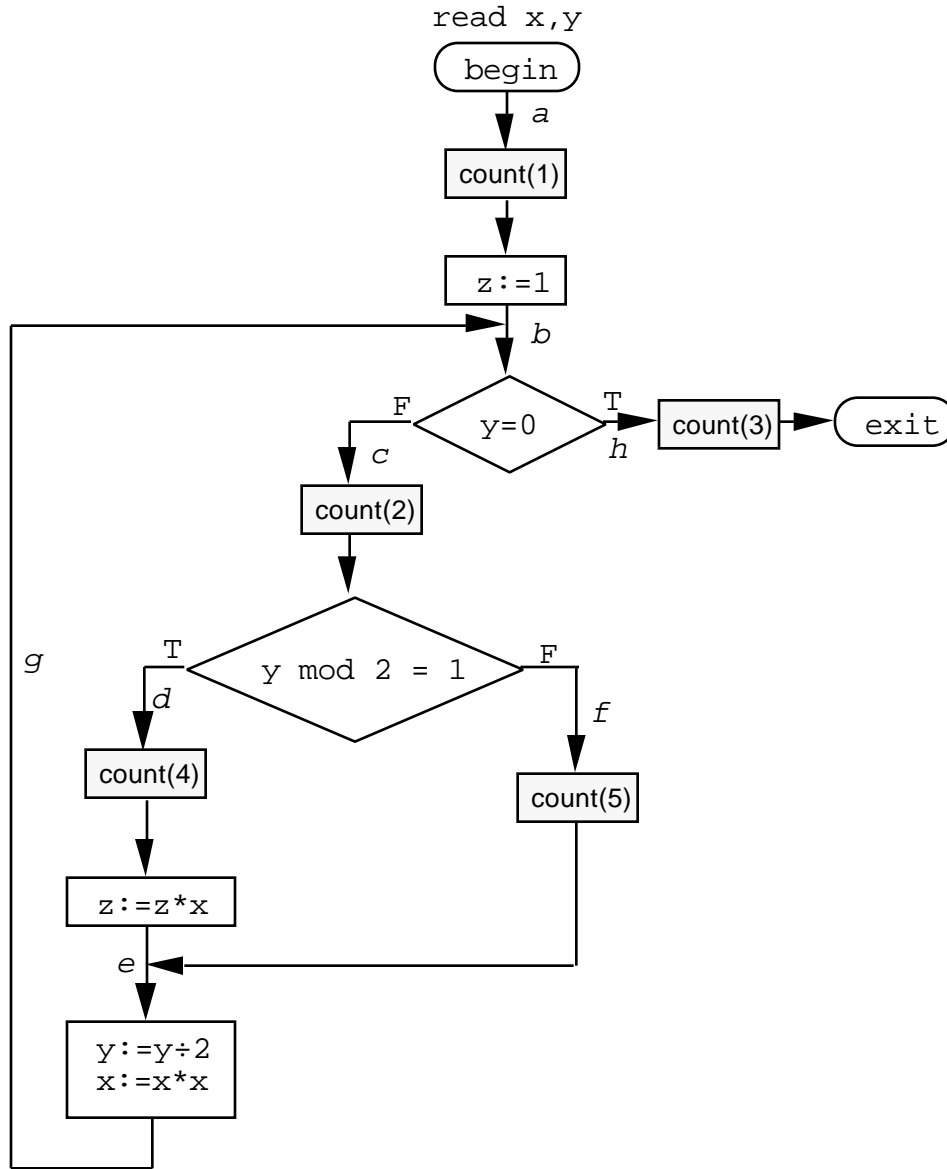


Figure 5. The program instrumented with counters.

The idea presented above can be illustrated by using the example program shown in Figure-4. For the purpose of discussion, let us assume that the software counters are to be implemented by using a procedure (subroutine) named `count(j)`, where `j` is a parameter. This procedure consists of an integer array `counter[1], counter[2], ..., counter[n]`; the value of each element is set to zero initially. A call of procedure `count(j)` will increase the value of `counter[j]` by 1. There are five reference branches in Figure 4, namely, a, c, h, d, and f. Thus we instrument the program by

placing the counters on these branches as shown in Figure 5. Now we are ready to test this program for a set of test cases. The test cases used, and the resulting counter values are listed in Figure 6. The fact that all resulting counter values are nonzero indicates that every path in the program has been traversed at least once in the test, and therefore the test is minimally thorough.

test cases		counter values				
x	y	counter[1]	counter[2]	counter[3]	counter[4]	counter[5]
10	0	1	0	1	0	0
20	1	1	1	1	1	0
5	2	1	2	1	1	1
40	4	1	3	1	1	2

Figure 6. Possible test cases and the corresponding counter values.

When a counter placed on a reference branch has a zero count, it indicates that the corresponding decision-to-decision path was not traversed at all during the test (and therefore the test is not minimally thorough). We can use this information to find additional test cases to satisfy the test requirement. To demonstrate how this can be done, let us suppose that only the first two test cases listed in Figure 6 are used in the test. Consequently, the value of counter[5] will remain zero, indicating that branch f in Figure 4 was not traversed at all. To make the test minimally thorough, we must find an additional test case that will force traversal of this branch. Note that branch f will be traversed if the execution proceeds along the path formed by branches a, b, c, f, and further down. By inspection we can tell that this path will be traversed if the input data satisfy the condition $y = 0$ and $y \bmod 2 = 1$. Thus we may use, say, $x = 24$ and $y = 2$ as the additional test case. After performing a test on this test case the value of counter[5] will be 1 as can be verified readily.

To recapitulate, we have described a method in which the program to be tested is instrumented by inserting counters at certain strategic points. The program is then tested for a set of test cases. By examining the resulting counter values we can tell whether the test is minimally thorough. If not, the information provided by the counters can be used to facilitate construction of additional test cases that are needed to make the test minimally thorough. Incidentally, the path usage information deducible from the counter values can also be used by the programmer to find "inner loops" for which the optimization payoff is greatest.

There is one thing that we did not make very clear in the above discussion. That is, where do we get the test cases when we say: "test the program for a set of test cases." It is possible that we know of a set of input data for which the corresponding correct output values are known; and therefore we would like to test the program against that set of input data. For instance, if the program is designed to compute the values of $\sin(x)$ then we certainly would like to see what the program is going to produce for x equal to, say, 0 , $\pi/6$, $\pi/3$, and $\pi/2$. In practice, however, the readily available set of test cases may be small compared to a set required to make the test minimally thorough. This is so particularly when the program to be tested is large. Of course we may enlarge the set of test cases by adding to it some randomly chosen input data. However, the chances are good that the test cases arbitrarily chosen will not add significantly to the degree of thoroughness. In order to satisfy the test requirement, we often have to use the information provided by the counters to find additional test cases.

This discussion leads us to consider another approach to program testing. That is, 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 computer time for program testing to a minimum.

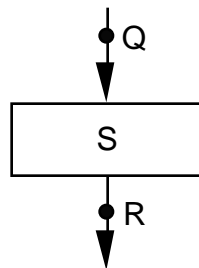
The question now is: how can we find for a given program a minimal set of test cases such that the test will be minimally thorough? The process of finding such a set is generally referred to as *test case generation*. Essentially, the desired set of test cases can be generated in three steps: 1) Find S , a minimal set of paths from the entries to the exits in the flowchart 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 1 (cf. Section 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 [3, 4, 10]. 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., $S = \{fabcdefghjkl, abcdefgijbkl\}$. Next, we need to find the path predicates for these two paths, i.e., the conditions assuring that these two paths will be traversed in execution. In general it is difficult to find the path predicates by inspection and we need a systematic method to derive it from the program text. We shall describe such a method in the next section.

3. PATH PREDICATES

Observe that a branch in the flowchart will be traversed in execution if some predicate, called the *branch predicate*, is satisfied at that stage of computation. The branch predicate for any branch can be found by examining the nature of the statement associated with the node from which this branch emanates. For instance, in Figure 4, $y = 0$ is the branch predicate of the branch labeled by h , while $\text{not}(y = 0)$ is that of the branch labeled by c . Obviously, if a branch is the only branch emanating from a node, then it is associated with the constant predicate T , which is always true.

Now consider two branches associated with predicates Q and R , respectively. Then the path formed by concatenating these two branches will be traversed if Q and R are both true. However, it should be noted that in a flowchart two branches can form a path with some statement S in the program as depicted below:



Insofar as the construction of path predicates is concerned, we need only consider the case when S is an assignment statement, because it changes the value of a variable upon execution. Let S be an assignment statement of the form $x := E$, where x is a variable, E is some expression, and $:=$ is the assignment operator. Obviously, if we want predicate R to be true after S is executed, then predicate $R_{E \ x}$ must be satisfied prior to the execution of S . Here by $R_{E \ x}$ we mean a predicate obtained by substituting expression E for each and every occurrence of x in predicate R . Thus the path predicate for the path so formed is seen to be:

$$Q \text{ and } R_{E \ x}.$$

The following examples should be helpful in clarifying the meaning of expression $R_{E \ x}$

R	$x := E$	$R_{E \ x}$
$x = 1$	$x := 1$	$1 = 1$
$a > 0$	$a := 10$	$10 > 0$
$b = 16$	$a := 4$	$b = 16$
$a < 10$	$a := a + 1$	$a < 9$
$a \quad b$	$a := a - b$	$a \quad 2b$

Expression $R_{E \ x}$ may be thought of as a predicate obtained by "dragging" predicate R backward along the path illustrated above. In general, an edge predicate can be dragged backward along a path until the first edge of the path is reached. Let Q be a predicate associated with an edge on path p, and let Q' be the predicate obtained by dragging Q backward along path p all the way to the first edge on p. It should be obvious that, if we want the program to be executed along path p, then Q' must be true when the path is entered. Thus if we want a specific path in the flowchart to be traversed during the execution, then the set of predicates obtained by dragging every edge predicate backward must be satisfied. A conjunction of the set of predicates so obtained is thus the path predicate of that path.

To illustrate the idea described above, we shall now construct the path predicate for path abcdefghjkl in Figure 1. We can ignore all branches associated with predicate T because it is a constant and is an identity under conjunction. The nontrivial branch predicates are:

- $c ::= \text{not}(w \quad e)$
- $h ::= u < v$
- $i ::= \text{not}(u < v)$
- $k ::= w \quad e.$

The process of dragging these three predicates backward toward the first branch of the path is shown step by step in the following. Note that in passing through an assignment statement the predicate remains unchanged unless the variable on the left-hand side of the assignment operator occurs in the predicate.

branch		predicates
k		$w \quad e$
b		$w \quad e$
j		$b - a \quad e$
h	$u < v$	$b - p \quad e$
g	$u < v$	$b - p \quad e$
f	$u < f(q)$	$b - p \quad e$

e		$u < f(b - w/3)$	$b - p \leq e$
d		$f(p) < f(b - w/3)$	$b - p \leq e$
c	$not(w \leq e)$	$f(a + w/3) < f(b - w/3)$	$b - (a + w/3) \leq e$
b	$not(w \leq e)$	$f(a + w/3) < f(b - w/3)$	$b - (a + w/3) \leq e$
a	$not(b - a \leq e)$	$f(a + (b - a)/3) < f(b - (b - a)/3)$	$b - a - (b - a)/3 \leq e$

Thus the path predicate for path abcdefghjkl is the conjunction of the three predicates obtained above, namely

$$not(b - a \leq e) \text{ and } f(a + (b - a)/3) < f(b - (b - a)/3) \text{ and } b - a - (b - a)/3 \leq e$$

which can be simplified to yield

$$not(b - a \leq e) \text{ and } f((b + 2a)/3) < f((a + 2b)/3) \text{ and } 2(b - a)/3 \leq e. \tag{3.1}$$

Similarly we obtain the path predicate for path abcdefgijbkl given below:

$$not(b - a \leq e) \text{ and } not(f((b + 2a)/3) < f((a + 2b)/3)) \text{ and } 2(b - a)/3 \leq e. \tag{3.2}$$

The path predicates constructed as described above are always expressed in terms of constants and input variables. To cause a path to be traversed during execution, all we need to do is to find an assignment of values to the input variables so that the path predicate is satisfied. We shall discuss how to find such an assignment in the next section.

4 TEST-CASE GENERATION

To find an assignment to satisfy a predicate is not difficult in principle, but it will be error-prone and rather time-consuming in practice. An ideal solution would be to delegate this task to a computer. Unfortunately, mechanization of this process is quite difficult. It essentially requires a program having almost the computational power of a theorem prover. There have been efforts to partially mechanize the process for the cases where the assignments can be found by using the techniques for solving algebraic equations [11], or those of linear programming [9]. It appears that a fully mechanized general inequality solver will not be available for some time to come. What we would like to do in the following is to describe a systematic method for finding the desired set of assignments. The method can then be mechanized and incorporated into an automated program-testing tool, or can be used by the programmer to find the assignments by hand.

In many practical programming languages a logical expression is an expression of the form:

$$E_1 R E_2 \tag{4.1}$$

where E_1 and E_2 are arithmetic expressions and R is one of the six relational operators: =, <, >, <=, >=, and \neq. For convenience, we shall call a logical expression of the form (4.1) an *atomic (logical)*

expression.

The path predicates obtained as described in the preceding section are constructed by using the atomic expressions and logical connectives such as *not*, *and*, and *or*. Furthermore, they are all expressed in terms of constants and input variables.

Ordinarily, it is a relatively simple matter for us to find an assignment that satisfies an atomic expression. It is also relatively easy to find an assignment that will satisfy a conjunction of atomic expressions if no variable that occurs in one atomic expression occurs in the other. But this is usually not the case in practice. In general, we will find the same variables occurring in more than one atomic expression in a path predicate. As a rule, the degree of difficulty in finding the desired assignments increases as the number of atomic expressions involved, and as the number of variables in common increases.

For instance, consider the following pair of logical expressions:

$$\begin{aligned} & \text{not}(b - a \leq 0.1) \\ & 2(b - a)/3 \leq 0.1. \end{aligned}$$

It is easy to find an assignment that will satisfy either member of this pair. But it will be more difficult to find one that satisfies both.

The central problem, then, is to devise a systematic method for finding an assignment that will satisfy a logical expression of the form:

$$P_1 \text{ and } P_2 \text{ and } \dots \text{ and } P_n, \tag{4.2}$$

where each P_i is an atomic expression, possibly prefixed by a *not* connective.

As the first step in the present method, we shall remove all *not* connectives involved. This can be accomplished as follows: If it is an expression of the form $E_1 R E_2$, then replace it with the equivalent expression $E_1 R' E_2$. The six relational operators and the corresponding R' are listed below:

R	R'
=	=
<	>
>	<

Thus, given a logical expression of the form (4.2) we can always rewrite it into an equivalent expression of the form:

$$Q_1 \text{ and } Q_2 \text{ and } \dots \text{ and } Q_n \tag{4.3}$$

where each Q_i is a non-negated atomic logical expression.

Next, we observe that to many of us it is easier to work with a set of equalities rather than

inequalities. For this reason we shall, as the second step in the present method, rewrite each Q_i in (4.3) into an expression in terms of = only. The notion that this can be done is based on the fact that inequalities can be stated in terms of equalities as follows:

- $a = b$ if and only if there exists an $x = 0$ such that $a + x = b$.
- $a < b$ if and only if there exists an $x > 0$ such that $a + x = b$.
- $a \leq b$ if and only if there exists an $x \geq 0$ such that $a + x = b$.
- $a > b$ if and only if there exists an $x < 0$ such that $a + x = b$.
- $a \geq b$ if and only if there exists an $x \leq 0$ such that $a + x = b$.

For the reasons that will become obvious in the ensuing discussion, we shall restate the above relations as follows: (Note: $(\exists x)_{>0}$ is to be read as "there exists an $x > 0$ such that ...")

$$\begin{aligned}
 a = b & \quad (\exists x)_{=0}(x = b - a) \\
 a < b & \quad (\exists x)_{>0}(x = b - a) \\
 a \leq b & \quad (\exists x)_{\geq 0}(x = b - a) \\
 a > b & \quad (\exists x)_{<0}(x = a - b) \\
 a \geq b & \quad (\exists x)_{\leq 0}(x = a - b)
 \end{aligned} \tag{4.4}$$

Now we shall explain by using examples how the reformulation of atomic expressions in terms of equality may help in finding the desired assignments. Suppose we wish to find an assignment that satisfies path predicate (3.2), which is restated below for convenience.

$$not(b - a \leq e) \text{ and } not f((b + 2a)/3) < f((a + 2b)/3) \text{ and } 2(b - a)/3 \leq e.$$

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 (cf. Section 1). Hence it is always true that $(b + 2a)/3 < (a + 2b)/3$. Now, from the function plot in Figure 2 we see that $not(f(x_1) < f(x_2))$ 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 $(b + 2a)/3 \geq 2$, or equivalently, $b + 2a \geq 6$. Thus, instead of predicate (3.2), we may work with the following predicate:

$$not(b - a \leq e) \text{ and } b + 2a \geq 6 \text{ and } 2(b - a)/3 \leq e.$$

By removing the not connective as explained before we obtain

$$(b - a > e) \text{ and } b + 2a \geq 6 \text{ and } 2(b - a)/3 \leq e.$$

Next, by (4.4), we can restate this predicate in terms of equalities as follows:

$$(\exists x)_{>0}(x = b - a - e) \text{ and } (\exists y)_{>0}(y = b + 2a - 6) \text{ and } (\exists z)_{>0}(z = e - 2(b - a)/3)$$

We use different variables (which are quantified) for each atomic expression because then the above logical expression can be readily rewritten into its so-called *prenex* normal form [12].

$$(\exists x)_{>0}(\exists y)_{>0}(\exists z)_{>0}(x = b - a - e \text{ and } y = b + 2a - 6 \text{ and } z = e - 2(b - a)/3). \tag{4.5}$$

The three equations in (4.5) are indeterminate because there are more than three variables involved. Therefore, we cannot directly obtain the desired assignment by solving the equations. However, we can combine these three equations to form a new equation in such a way that the number of variables involved in the new equation will be minimal. This can be accomplished by using the same techniques we use in solving simultaneous equations. In the present example, we can combine the three equations to yield

$$(x)_{>0} (y)_{>0} (z)_{>0} (3x - y + 3z = 6 - 3a). \quad (4.6)$$

As indicated in the above expression, the requirements on the assignments to x , y , and z are that $x > 0$, $y > 0$, and $z > 0$. So let us begin by making the following assignments:

$$x = 0.1, y = 0, z = 0.$$

Then (4.6) can be satisfied by letting

$$a = 1.9.$$

To satisfy the second atomic expression in (4.5) we must have $0 = b + 2 \times 1.9 - 6 = b - 2.2$, i.e., we have to make the assignment

$$b = 2.2.$$

Finally, the first and the third atomic expression can be satisfied by letting

$$e = 0.2.$$

In summary, predicate (3.2) can be satisfied by the following assignment:

$$a = 1.9, b = 2.2, e = 0.2. \quad (4.7)$$

Similarly we find that path predicate (3.1) can be satisfied by the following assignment:

$$a = 0, b = 0.5, e = 1/3. \quad (4.8)$$

Thus, the assignments (4.7) and (4.8) constitute a set of test cases that will test the program given in Figure 1 minimally thorough.

Having found the desired set of test cases the question now 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 1 by using test case (4.8). If the program is correct, we should obtain as the test result $\max = 0.5 - \epsilon$, where ϵ is the error less than or equal to $\epsilon/2 = 1/6$. If the program is in error, say, the assignment statement $p := a + w/3$ is somehow mistakenly written as $p := a - w/3$, 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 1) is erroneously written as $v < u$ instead of $u < v$, then variable \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 $\max = 0$, i.e., \max is within the distance $\epsilon = \epsilon/2 = 1/6$ from $a = 0$,

which clearly is 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 (4.7), because this statement will not be executed, assuming that $f(x)$ is monotonously decreasing from $x \geq 2$. Neither will the presence of this error be clearly reflected in the test result using test case (4.8). The reader should be able to verify that we still will obtain as the result $\max = 0.5 - \epsilon$ for some $\epsilon > 1/6$, although the magnitude of error ϵ could be smaller than that produced in the absence of this programming error.

CONCLUDING REMARKS

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 article it is deemed imperative for us to point out that the problems are actually much more complicated than we have described in the preceding sections. 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 paper. Here are some major factors that contribute to the complexity of test-case generation.

1) Number of paths involved in a large program.

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

$$\{abcdegh, abcfgh\}$$

and thus is a candidate minimal covering set for test-case construction. By an application of the method described previously we find that the path predicates for these two paths are:

$$\text{not}(y = 0) \text{ and } y \bmod 2 = 1 \text{ and } y / 2 = 0$$

and

$$\text{not}(y = 0) \text{ and not}(y \bmod 2 = 1) \text{ and } y / 2 = 0,$$

respectively. Some reflection will show that the first path predicate can be satisfied by letting $y = 1$, but the second predicate cannot be satisfied by any non-negative integer! This indicates that the second path *abcfgh* 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 *abcfgh* in the above example can be replaced by path *abcfgcdgdegh*, whose path predicate can be satisfied by letting $y = 2$.

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 procedure (subroutine) 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. Treatments of some such problems may be found in [13-18], in addition to the references cited previously. Reference [18] is particularly significant in that it contains many works that reflect the state of the art in program testing.

Manually carrying out the process of test-case generation is a very tedious, time consuming, and error-prone matter. Considering the fact that more than 50% of the man hours used in today's software industry are spent in program testing, it should be of great economic worth to develop computer aided or fully automated systems for test-case generation. In fact, a number of such systems are known to be in existence or in the process of being developed (see, e.g., [5, 6, 9, 19]). At least one such system [19] is already available on the market and the experience with that system should be of interest to the reader. Specifically, "Structurally based automatic program testing." by E. Miller, et al. [8] shows:

- Most real-life FORTRAN programs can be tested minimally thorough with a relatively small number of test cases. Very roughly, the number of tests is less than or equal to about 10% of the number of FORTRAN lines in the program.

- That in using the program instrumentation method [3], a test case conceived to execute a particular decision-to-decision path [4] frequently causes a large [5] number of other decision-to-decision paths to be traversed.
- The automated program testing system appears to work easily and well, and [6] contributes significantly to the improvement of software quality.

It is observed that, perhaps because of the enormous developmental cost involved, present interest in automated program-testing tools is found mainly among people with space and military applications. However, the use of such tools may in time become prevalent, particularly in applications where reliability is of primary concern.

It is true that program testing can only be used to detect the presence of bugs, but not to show their absence, as aptly observed by Dijkstra [2]. Nevertheless, it is a practical technique commonly used in the industry. In the absence of practical method that can be used to show a program as error-free, it should be worth the effort to improve our ability to discover bugs through program testing.

ACKNOWLEDGMENTS

The author is greatly indebted to Raymond T. Yeh and Peter J. Denning for their invaluable suggestions and encouragement. Peter Denning has devoted so much of his time to helping me revise several drafts of this article that he is virtually a co-author of this tutorial. During the summer of 1973 the author was fortunate to have the opportunity to work on the program-testing problems at NASA Johnson Space Center and had many fruitful discussions on the subject with Mary Ann Goodwin. This was made possible through the support of the NASA-ASEE Summer Faculty Research Fellowship Program.

REFERENCES

- [1] Elspas, B. et al., "An assessment of techniques for proving program correctness," *Computing Surveys*, 4, 2 (June 1972), 97-147.
- [2] Dahl, O.-J. et al., *Structured programming*, Academic Press, London and New York, 1972.
- [3] Berge, C., *Theory of graphs and its applications*, John Wiley & Sons, New York, 1962.
- [4] Harary, F., *Graph theory*, Addison-Wesley, Reading, Mass., 1969.
- [5] Krause, K. W., et al., "Optimal software test planning through automated network analysis," *Proc. 1973 IEEE Symposium on Computer Software Reliability*, New York, April-May 1973.
- [6] Howden, W. E., "Methodology for the generation of program test data," *IEEE Transactions on Computers*, C-24, 5 (May 1975), 554-560.
- [7] *Fortran automated verification system Level 1 — user's guide*, Program Validation Project, General Research Corp., October 1974.
- [8] Miller, E. F., et al. "Structurally based automatic program testing," presented at EASCON-74, Washington, D. C., October 1974.
- [9] Clarke, L., "A system to generate test data and symbolically execute programs," Tech. Report

#CU-CS-060-75, Dept. Computer Science, University of Colorado - Boulder, February 1975.

- [10] Sloane, N. J. A., "On finding the paths through a network," *Bell System Tech. J.*, 51, 2 (February 1972).
- [11] Hoffman, R. H., *Automated verification system user's guide*, TRW Note No. 72-FMT.
- [12] Copi, I. M., *Symbolic logic*, Macmillan, New York, 1965.
- [13] Miller, E. F., and Melton, R. A., "Automated generation of test-case data sets" *Proc. 1976 Internat'l. Conf. on Reliable Software*, Los Angeles, Calif., April 1975. IEEE Cat. No. 75CH0940-7CSR.
- [14] Hetzel, W. C., *Program test methods*, Prentice-Hall, Englewood Cliffs, N. J., 1972.
- [15] Ramamoorthy, C. V., et al., "Design and construction of an automated software evaluation system," *Proc. 1973 IEEE Symposium on Computer Software Reliability*, New York, April-May 1973.
- [16] Paige, M. R.; AND Balkovich, E. E., "On testing programs," *Proc. 1973 IEEE Symposium on Computer Software Reliability*, New York, April-May 1973.
- [17] Osterweil, L. J., and Fosdick, L. D. "Data flow analysis as an aide in documentation, assertion generation validation, and error detection," Tech. Report #CU-CS055-74, Dept. Computer Science, University of Colorado, Boulder, September 1974.
- [18] *Proc. 1975 Internat'l. Conf. on Reliable Software*, Los Angeles, Calif., April 1975, IEEE Cat. No. 75C-0940-7CSR.
- [19] *Fortran automated verification system Level 1 — System Summary*, Program Validation Project, (General Research Corp.), October 1974.