TEST DATA AS AN AID IN PROVING PROGRAM CORRECTNESS

Matthew Geller
Department of Computer and Communication Sciences
The University of Michigan
Ann Arbor, Michigan

Abstract

Proofs of program correctness tend to be long and tedious whereas testing, though useful in detecting errors, usually does not guarantee correctness. This paper introduces a technique whereby test data can be used in proving program correctness. In addition to simplifying certification of correctness, this method simplifies the process of providing specifications for a program. The applicability of this technique to procedures, recursive programs, and modular programs is demonstrated.

Keywords and Phrases: program verification, program testing, modular programming, recursive programs, inductive assertions.

CR Categories: 4.22, 4.6, 5.24

1. Introduction

Traditionally, certification of programs has been attempted by "testing" programs on "representative" test data. The problem with this kind of testing was aptly pointed out by Dijkstra when he stated

> Program testing can be used to show the presence of bugs, but never to show their absence.†

Recently, several attempts have been made to provide techniques for choosing test data that would allow one to make meaningful conclusions about the absence of bugs in a program as a result of test runs.

Much research has been done in the area of choosing test data that exercises all the statements or paths of a program, cf. Howden [1974], Miller and Melton [1975]. The chief weakness in these techniques is that they do not match the choice of test data to the specifications for the program. Therefore, if a program is written which fails to examine a particular case, this error will not be detected using test data determined by the program as written. The type of error in which one completely forgets to handle some special kind of input occurs quite often in programming.

†Dahl, Dijkstra, and Hoare, Structured Programming, p. 6.

A new approach entails testing programs with symbolic as opposed to real values. Systems to symbolically execute programs have been implemented by King [1975] and Boyer, Elspas, and Levitt [1975]. These systems have proven very useful in providing information for debugging programs. However, it is not always clear how to use all this information to guarantee the "absence of bugs".

Recently, much research has been done in the area of providing formal proof of program correctness. London [1975] eloquently summarizes much of this work, and discusses many of the problems involved in providing a formal proof of correctness. A substantial bibliography of work in the field can be found in this paper. The chief difficulty of proving a program correct is the complexity of the process. Proving a program correct can be extremely tedious and difficult. It may be easier to convince oneself that a program works properly on the whole input domain on the basis of a few test cases. Information gained from running a program on sample test data is very easy to procure. The processor for the language automatically gives us values of given program variables resulting from the input of sample test data. A special case in which the use of test data will often save us work in proving a program correct occurs when we are proving a program correct using structural induction, cf. Burstall [1969]. The basis can always be proven by running the program on the basis "test data". Boyer and Moore [1975] also discuss how one can obtain valuable information about a LISP program by running the program.

The use of test data has an additional great advantage. Goodenough and Gerhart [1975] point out that a so-called "proof" of program correctness may not in fact guarantee that a program actually does what we want it to do. Of course, one may argue that in this case one did not think carefully enough about the specifications for the program's correctness. There are certain instances, however, where the specifications for a program are intuitively clear, yet where it is in fact quite easy to make a mistake in precisely stating these specifications. The specification problem is discussed in depth by Liskov and Zilles [1975]. In this paper, we use the example of the calendar. We are all quite familiar with how a calendar works, yet in formalizing these notions, it it quite easy to make errors. This is espe-

cially easy when specifying the behavior of a calendar at month boundaries.

One must realize at this point that "proving" a program correct is only worthwhile in that the specifications for a program somehow provide us additional assurance that a program is doing what we want it to do, above and beyond the program itself. One can always, in fact, say that a program is "correct" by using a program as its own specifications. In order that we assure ourselves that a program in fact behaves correctly at certain boundary points, or anomalies, it may be extremely helpful to include in our specifications the specific behavior that our program should obey at these points. One might then be able to specify how the program should behave on the whole input domain as a function of its specifications for this test data.

We wish to formalize these informal techniques whereby one generalizes from specific test cases to a large domain. In this paper, we will suggest a technique whereby information gained from testing a program can be used in helping to prove program correctness. Viewed conversely, we might view the process of choosing test data for a program as being guided by a pursuit of information useful in helping us prove our program correct.

We wish to provide techniques which will allow us to make statements about a large, possibly infinite set of inputs on the basis of some small set of inputs. Goodenough and Gerhart [1975] divide the input domain into a set of classes, and then prove that for each of these classes, if some single member of this class gives a correct answer on testing, then every member of that class will produce the correct answer. Sintzoff [1972] also uses valuation to verify certain properties of programs. Henderson [1975] validates programs by testing its components in an environment provided by a finite state machine.

In this paper, we will divide the input domain into classes in order to prove inductive assertions at given points in the program, rather than to attempt the more ambitious tasks of dividing the input domain into equivalence classes based simply on output specifications. We wish to execute sections of a program as they are being developed, rather than to go to great pains to find test data that will exercise all statements or paths of a program that has been already written.

Note that the techniques we shall discuss will provide a formal means of proving program correctness. They might be viewed as a special proof technique that may be useful in many circumstances. They will be shown to be of particular value in large, modularized programs, cf. Parnas [1972].

Before continuing, it seems necessary to discuss some of the things that testing in fact cannot accomplish.' It has often been stated that testing has an advantage over proving program correctness in that it detects run-time errors, such as out-of-bounds array references, which could not be detected using proof techniques. This is misleading since one can and should show that all array references are in bounds as part of a proof of program correctness. A further argument for testing as opposed to proving correctness goes as follows. In proving program correctness, one might forget entirely that an array reference may be out of bounds. Testing the program might detect this error. The weakness of this approach is that if we are lucky enough to choose appropriate test data, the error will show up during the test run. However, the test data that we choose may not detect this error. It may, in fact, require a great deal of computation to find the particular test data that will cause an error. If we have gathered enough information to calculate test data that will guarantee that all references to arrays are in bounds, then it might have been easier to formally prove this property.

The paper is divided into four sections. Section 2 will give the formalism that will be used. Section 3 illustrates the use of this model with examples. In Section 4, conclusions are drawn and suggestions are made for further research.

2. The Model

We shall begin this section with the model that we will use to prove programs correct using test data. We will then provide two theorems that will be useful in generalizing test data from specific data to larger classes of data.

In order to allow for the use of test data in proving program correctness, we must introduce the notion of distributed correctness. According to the model of Floyd [1967], in order to prove a program correct, one must show that for input that meets the input criteria, that after running the program, the output variables must satisfy the output predicates. We see that all the specifications for the program are provided at the point of termination of execution. In order to use test data to its fullest advantage, we must allow the prover to use as data the value of any variable in the program at any point in the execution of the program. In order to check that the value of this variable is correct at this point, we must be able to set criteria that the program should obey for the given test data at any point in the program.

The use of distributed correctness will buy us much more than allowing us to use test data to prove a program correct. It will in addition allow us to greatly reduce the size of assertions throughout the program.

Now, rather than carrying a complicated predicate through the programs, until it is used in the output predicate, we simply carry with us the fact that in some given respect, the program is correct. One need now only show that this correctness is maintained at any point at which we need to use it to prove further facts about the program.

Consider for example a compiler which is

driven by a precedence parser. Part of the program consists of initializing the tables of this parser. An output specification for the program would have to have as one of the disjuncts the specifications for each of the entries in the table. It would be necessary essentially to repeat the part of the program that initializes the table. Even if there are errors in the table, this process is useless, since the person proving the program is likely to make exactly the same error in the specifications as in the program itself. We see here that we have provided no additional assurance that our program is correct. Rather than stating as part of the output predicate the entries of the table, one would only specify as part of the output predicate that the tables were in fact correct. One could then certify that the table was correct at the point in the program immediately after the table had been created. One would then need only prove that the table was not changed after its creation, to assure that the correctness of the table was maintained to satisfy the output predicate.

We now describe the assertions that we shall use in proving a program correct using test data. These assertions will be assigned to arcs of the program flow graph, as in Floyd [1967]. Each assertion consists of three parts, namely

(1) Test data assertion
(2) Generalization assertion
and (3) Synthesized assertion.

The test data assertion gives the values that some given variable should have at a given arc of the flow graph based on a given value of that variable entering that arc. The generalization assertion will be some assertion which generalizes from the value produced by the program for some given test value to a larger domain. That is, it will be an assertion that states "If a program behaves a certain way on some given test data, perturbing the input data in a given fashion perturbs the output in a given fashion."

For a synthesized assertion to hold, we must show that both the test data and generalization assertions are satisfied and that the test data assertion and generalization assertion imply the synthesized assertion. Note that the proof that the test data assertion and generalization assertion imply the synthesized assertion can be done before testing the program.

We now give a very simple example of what we have in mind.

Example 2.1.. Consider the program

IF X>0 THEN Y:=1 ELSE Y:=0

Synthesized Assertion.

if x>0 then y=1
if x≤0 then y=0

Test Data Assertion

if x=1, y=1
if x=0, y=0

Verification of Test Data Assertion

This can be verified by running the program.

Generalization Assertion

Let f(x) be the value of y after executing the program with input variable x. For $x > 0$, $f(x) = f(1)$. For $x \leq 0$, $f(x) = f(0)$.

Proof. This follows immediately from the fact that $1 > 0$ and $0 \leq 0$, and the semantics of an IF-THEN-ELSE statement.

We now wish to prove our synthesized assertion using the test data assertion and the generalzation assertion.

Proof of Synthesized Assertion. Let $x > 0$. Then $f(x) = f(1) = 1$. Let $x \leq 0$. Then $f(x) = f(0) = 0$. □

This example is for illustrative purposes only. A simple proof that the synthesized assertion was satisfied could in fact be given without using test data.

We now provide two theorems that will be useful in proving a synthesized assertion, based on a test data assertion and a generalization assertion. These theorems are in no way meant to be a comprehensive set. They have been chosen in that they represent two different ways in which one can generalize from test data. The first theorem shows how one can generalize from test data when the output values can be divided by predicates into equivalence classes in which the values are identical.

Definition 2.1. Let $Q_1, Q_2, \ldots, Q_n$ be predicates on domain D, f a partial function on D. We say that f is totally determined by $Q_1, Q_2, \ldots, Q_n$ if

$$\begin{aligned} f(x) &= k_1 \text{ whenever } \sim Q_1(x), \sim Q_2(x), \ldots, \sim Q_{n-1}(x), \sim Q_n(x) \\ &= k_2 \text{ whenever } \sim Q_1(x), \sim Q_2(x), \ldots, \sim Q_{n-1}(x), Q_n(x) \\ &= k_3 \text{ whenever } \sim Q_1(x), \sim Q_2(x), \ldots, Q_{n-1}(x), \sim Q_n(x) \\ &\vdots \\ &= k_{2^n} \text{ whenever } Q_1(x), Q_2(x), \ldots, Q_{n-1}(x), Q_n(x). \end{aligned}$$

This is a binary ordering.

Example 2.2. Let D be the integers. Let Q_1 be the predicate, $x > 0$. Let Q_2 be the predicate, x is even. Define f as follows.

$f(x) = 0$ whenever $x > 0$, x is even.
$= 1$ whenever $x > 0$, x is odd.
$= 2$ whenever $x \leq 0$, x is even.
$= 3$ whenever $x \leq 0$, x is odd.

Then f is totally determined by $x > 0$, x is even.

The following theorem shows how we can pick representative values from our domain to determine the value of our function over the entire

domain.

Theorem 2.1. Let f and g be two partial functions on doamin D, both of which are totally determined by predicates $Q_1,Q_2,\ldots,Q_n$ on D. Let

$$\begin{aligned}
&\sim Q_1(x),\sim Q_2(x),\ldots,\sim Q_{n-1}(x),\sim Q_n(x) = P_1(x)\\
&\sim Q_1(x),\sim Q_2(x),\ldots,\sim Q_{n-1}(x), Q_n(x) = P_2(x)\\
&\qquad\vdots\\
&Q_1(x),Q_2(x),\ldots,Q_{n-1}(x),Q_n(x) = P_{2^n}(x)
\end{aligned}$$

Then $P_1,\ldots,P_{2^n}$ divide D into a set of equivalence classes $E_1,\ldots,E_{2^n}$. We now choose some set $\{x_1,\ldots,x_m\} = T$ which consists of some representative from each of the non-empty equivalence classes. Assume $f(x_i) = g(x_i)$ for $1 \leq i \leq m$. Then $f(x) = g(x)$ for all $x \in D$.

Proof. The proof of the theorem is immediate. □

We now apply the theorem to our example.

Example 2.3. Choose D, Q_1, Q_2, f as in Example 1. The non-empty equivalence classes are (x>0, x even), (x>0, x odd), (x≤0, x even) and (x≤0, x odd). Now we choose the set (4,3, -4,-3), elements from their respective equivalence classes. We see that we get (0,1,2,3) when f is applied to these elements.

Let g be any other partial function defined on D. Suppose we know that g(4) = 0, g(3) = 1, g(-4) = 2, g(-3) = 3 and that g is totally determined by Q_1 and Q_2. Then Theorem 1 tells us that f = g.

The theorem can be applied as follows. We let f be the function that is actually computed by some program P and g the specifications for P. We wish to prove that f = g. In order to do this we must show two things, namely

(1) the identical set of factors determines how the program behaves and how the program should behave, and
(2) the program behaves according to specifications on the same test data.

Once we have shown these two facts, it will follow immediately from our theorem that the program meets its specifications.

Theorem 2.1 is used in proving the synthesized assertion after line 4 in Example 3.2.

Our second theorem will allow us to generalize from test data by showing that on sample values the program meets its specifications, and that perturbing the test data in a given fashion has the same effect on the output of the program as on the value demanded by the specification. We first need a definition.

Definition 2.2. We say that a function $f(x_1,\ldots,x_n,y_1,\ldots,y_m)$ on domain D is additive in $x_1,\ldots,x_n$, subtractive in $y_1,\ldots,y_m$ if $f(x_1,\ldots,x_n,y_1,\ldots,y_m) = (x_1+\ldots+x_n) - (y_1+\ldots+y_m) + k$ where k is come constant.

Example 2.4. Let f(x,y) = x-y+1. Then f(x,y) is additive in x, subtractive in y.

We get the following theorem.

Theorem 2.2. Let f and g be functions on D that are additive in $x_1,\ldots,x_n$, subtractive in $y_1,\ldots,y_m$, such that for some $x_1^o,\ldots,x_n^o,y_1^o,\ldots,y_m^o$ we have $f(x_1^o,\ldots,x_n^o,y_1^o,\ldots,y_m^o) = g(x_1^o,\ldots,x_n^o, y_1^o,\ldots,y_m^o)$. Then for all $x_1,\ldots,x_n$, $y_1,\ldots,y_m$ we have $f(x_1,\ldots,x_n,y_1,\ldots,y_m) = g(x_1,\ldots,x_n, y_1,\ldots,y_m)$.

Proof. We have

$$f(x_1,\ldots,x_n,y_1,\ldots,y_m) = (x_1+\ldots+x_n) - (y_1+y_2+\ldots+y_m) + k ,$$

$$g(x_1,\ldots,x_n,y_1,\ldots,y_m) = (x_1+\ldots+x_n) - (y_1+y_2+\ldots+y_m) + \ell .$$

Since

$$f(x_1^o,\ldots,x_n^o,y_1^o,\ldots,y_m^o) = g(x_1^o,\ldots, x_n^o,y_1^o,\ldots,y_m^o) ,$$

we have $k = \ell$. Therefore,

$$f(x_1,\ldots,x_n,y_1,\ldots,y_m) = g(x_1,\ldots, x_n,y_1,\ldots,y_m) .$$ □

Example 2.5. Suppose in some program we have z = x - y + 1. Let g(x,y) be some function that is additive in x, subtractive in y. Assume g(0,0) = 1. Then by Theorem 2.2, f(x,y) = g(x,y) for all x, y.

Again, we let f represent the function computed by P, and g the specifications. In order to show that P meets its specifications, we must show that

(1) the function meets is specification on the sample test data, $\bar{x}$,
(2) the function computed by P, and the function defined by its specification are perturbed in the same fashion as the value $\bar{x}$ is perturbed.

Theorem 2.2 is used in proving the synthesized assertion after line 5 in Example 3.2.

3. Examples

In this section, three programs will be verified using the techniques discussed in the previous section. The programs that have been chosen can be verified using techniques that have been previously discussed in the literature. However, use of the techniques presented in this paper will in some cases simplify the proof, and

in most of the cases add to our confidence that the program is in fact accomplishing what it "should" accomplish.

Example 3.1. Our first example is the familiar "91" function, cf. Manna et. at. [1973], a recursive function often cited in the literature. The function is as follows:

$$F(X) = \underline{\text{IF}}\ x \geq 101\ \underline{\text{THEN}}\ x-10\ \underline{\text{ELSE}}\ F(F(x+11))$$

We wish to show that the function computes the following values.

for $x \geq 101$, $F(x) = x - 10$

for $x \leq 101$, $F(x) = 91$

where x is any integer.

A typical proof of this theorem is similar to one suggested in Manna et. al. [1973] as follows.

Proof of Example 1. We have three cases.

Case 1. $x \geq 101$. Clearly $F(x) = x - 10$.

Case 2. $101 \geq x \geq 91$. We wish to show that for $0 \leq x \leq 10$, that $F(101-x) = 91$.

Basis: $x = 0$. We have $F(101-x) = F(101) = 91$ from Case 1.

Induction Step: We assume that for some x, where $0 \leq x \leq 10$, we have $F(101-x) \leq 91$. Then $F(101 - (x+1)) = F(100-x)$. Since $0 \leq x$, we have $(100-x) < 101$. Therefore $F(100-x) = F(F(111-x))$. Now, since $X \leq 10$, we have $F(111-x) = F(111-x-10) = (101-x)$. Therefore $F(100-x) = F(101-x)$. But, by our induction hypotheses, $F(101-x) = 91$.

Case 3. $91 \geq x$. We wish to show that for x, $x \geq 0$, that $F(91-x) = 91$.

Basis: $x = 0$. We have $F(91-x) = F(91) = 91$ by Case 2.

Induction Step: We assume that for all x, $0 \leq x < X$, where $X > 0$, we have $F(91-x) = 91$. Now $F(91-(x+1)) = F(90-x)$. Since $x \geq 0$, we have $F(90-x) = F(F(101-x))$. If $0 \leq x \leq 10$, we have $F(101-x) = 91$ by Case 1. Therefore $F(F(101-x)) = F(91)$. By Case 2, $F(91) = 91$. If $x \geq 10$, we have $F(101-x) = F(91-(x-10))$, where $(x-10) \geq 0$. Therefore by our induction hypothesis, $F(91-(x-10)) = 91$. Thus $F(F(101-x)) = F(91) = 91$, by Case 2. □

We now give a proof of the correctness of this program using the techniques of Section 2.

Synthesized Assertion: For $x \geq 101$, $F(x) = x - 10$, for $x \leq 101$, $F(x) = 91$ where x is any integer.

Test Data Assertion. For all x such that $91 \leq x \leq 101$, we have $F(x) = 91$.

Verification of Test Data Assertion. We simply run our program with the values 91,92,..., 101 for x. We see that in each case, the program behavior meets the specifications of our test data assertion.

We can now generalize from these results to the domain of integers $\{I \mid I \leq 101\}$.

Generalization Assertion. For any i, $i \leq 10$, we have $f(i) = f^{(k)}(x)$ for some $k \geq 1$ and some x such that $91 \leq x \leq 101$.

Proof. By induction on (101 - i) DIV 11 (where DIV represents integer division of the first argument by the second).

Basis: (101-i) DIV 11 = 0. In this case $91 \leq i \leq 101$, and the result is clearly true.

Induction Step: We assume that our proved assertion is true for (101-i) DIV 11 = j, for some $j \geq 0$. Now choose ℓ such that $(101-\ell)$ DIV 11 = j + 1. Since $j \geq 0$, clearly $\ell < 91$. We have

$$(101 - (\ell+11) + 11) \text{ DIV } 11 = j+1$$

Therefore, $(101 - (\ell+11))$ DIV 11 = j. It follows from our induction hypothesis that

$$f(\ell+11) = f^{(k)}(x) \text{ for some } k \geq 1,\ 91 \leq x \leq 101.$$

But we know that $f(\ell) = f(f(\ell+11))$. Therefore $f(\ell) = f^{(k+1)}(x)$, and our Generalization Assertion holds.

We are now ready to prove our Synthesized Assertion.

Proof of Synthesized Assertion. It follows directly from the program that for $x \geq 101$, $F(x) = x - 10$. It follows directly from our test data assertion and generalization assertion that for $x \leq 101$, we have $F(x) = 91$. □

We see that our proof is substantially shorter than the first proof, and more intuitive.

Notice how the use of test data alone is insufficient to guarantee the correctness of the program. The technique suggested by some of breaking the input domain into classes on which the program gives the same value produces an infinite number of such classes, since a different result is produced for each x such that $x \geq 101$.

Example 3.2. For our second example, we choose a program which computes the number of days by which one date follows another date in some given calendar year. This particular program was chosen on several grounds. Although the notion of the number of days between two days is easily understood, axiomatization of this concept is quite difficult. Once we have provided an axiomatization for the correct numbers of days between two days, it is difficult to be sure that we have in fact chosen the correct axiomatization. It is particularly easy for the axiomatization to specify answers that are off by one or two days in certain cases.

Since the calendar program involves setting up a table, the notion of distributed correctness will allow us to greatly simplify our assertions,

by allowing us to merely specify that the table is correct.

Comment. The following procedure takes as input two dates in a given year, where a date is a pair (DAY,MONTH). We assume that both dates are legitimate dates for the year in question, and that the first date does not occur after the second date. The program computes the number of days by which the second date follows the first date.

```
     PROCEDURE CALENDAR(DAY1,DAY2,MONTH1,MONTH2,
        YEAR);
     BEGIN
(1)  IF MONTH2=MONTH1 THEN DAYS = DAY2 - DAY1
        COMMENT  IF THE DATES ARE IN THE SAME
        MONTH, WE CAN COMPUTE THE NUMBER OF DAYS
        BETWEEN THEM IMMEDIATELY;
     ELSE
        BEGIN
           DAYSIN(1) :=31; DAYSIN(3) :=31;
              DAYSIN(4) :=30;
           DAYSIN(5) :=31; DAYSIN(6) :=30;
              DAYSIN(7) :=31;
           DAYSIN(8) :=31; DAYSIN(9) :=30;
              DAYSIN(10) :=31;
(2)        DAYSIN(11) :=30; DAYSIN(12) :=31;
(3)        IF ((YEAR MOD 4) ≠ 0) OR ((YEAR MOD
              100) = 0 AND (YEAR MOD 400) ≠ 0)
           THEN DAYSIN(2) :=28
           ELSE DAYSIN(2) :=29;
           COMMENT  SET DAYSIN(2) ACCORDING TO
              WHETHER OR NOT YEAR IS A LEAP YEAR;
(4)        DAYS :=DAY2 + DAYSIN(MONTH1) - DAY1;
           COMMENT  THIS GIVES (THE CORRECT NUM-
           BER OF DAYS - DAYS IN COMPLETE INTER-
           VENING MONTHS);
(5)        FOR I = MONTH + 1 TO MONTH2 - 1 DO
              DAYS :=DAYSIN(I) + DAYS;
           COMMENT  ADD IN THE DAYS IN COMPLETE
              INTERVENING MONTHS;
        END
(6)  PRINT (DAYS)
     END
```

We now wish to verify the correctness of our procedure.

Line 1. We wish to prove the following synthesized assertion.

Synthesized Assertion. For any two dates, DAYS gives the correct number of days by which the second date follows the first date, assuming that MONTH1 = MONTH2.

It might at first appear that our use of test data in this instance is unnecessary. One might argue that our synthesized assertion could easily be argued to hold directly from the semantics of the statement DAYS :=DAY2 - DAY1.

Suppose, however, that we had really wished to compute the number of days between the two dates, including the first and last days. It would be very easy to err in the specifications for such a program, and on this basis argue that DAYS correctly computed this different quantity. This error, however, would be immediately detected by a choice of very simple test data.

We now use the formalism of our technique to show that in fact DAYS is correctly computed to meet the specification of this program. We begin by providing the test data assertion.

Test Data Assertion. For DAY1 = 15, DAY2 = 15, we should have DAYS = 0 after execution of this statement.

Our choice of the value 15 in particular was random. We could have chosen any other possible day of the month. However, it was important that DAY1 and DAY2 were chosen to be very close together. This allows us to easily state what the answer should be for the test data. Note that test data is useless unless it is simple to check whether or not the value produced by the program on the test data is correct. If, in fact, one has to run the program to check what the value of the test data should be, then we have gained nothing by testing the program.

Verification of Test Data Assertion. To verify the test data assertion, we now run the program from the point directly before where statement (1) lies, with DAY1 = 15, DAY2 = 15, and MONTH1 = MONTH2 = 1. After executing statement (1), we check the value of DAYS. In this case, since the program is correct, we find that we have the correct value of DAYS, namely 0. Notice that since this statement is only executed once, the value 0 will be produced for the given test data on every pass through this arc on the flow graph.

We now need to be able to generalize from this particular test data to any two particular days in the month. Our use of the generalization assertion will make it clear why we chose DAY1 = DAY2. Our generalization assertion states that increasing DAY2 increases DAYS in a certain fashion, increasing DAY1 decreases DAYS in a certain fashion.

Generalization Assertion. DAYS is additive in DAY2, subtractive in DAY1.

Verification of Generalization Assertion. We see in our program that DAYS = DAY2 - DAY1.

We now finally come to the point at which we prove the synthesized assertion. Note that the test data assertion and generalization assertion were verified, the test data assertion by use of test data, the generalizaed assertion logically. The synthesized assertion must be shown to logically follow from the test data assertion and generalization assertion.

Proof of Synthesized Assertion. The proof follows directly from the test data assertion, the generalization assertion, and Theorem 2.2. □

Note how we have made use of the notion of distributive correctness here, by concluding that DAYS is "correct" under certain conditions.

We now continue with out verification of the program.

Line 2.

This statement provides us with an example where the test data information provides all the information that is necessary to assure us that this "table" of the number of days in all of the months but February is correct. The program in a sense provides its own documentation in creating this table. It would certainly be valueless to add the values in this table as one of the disjuncts in the output assertion of this program. We therefore have

Test Data Assertion, Synthesized Assertion. DAYSIN(I) is correct for I = 1, $3 \leq I \leq 12$.

One might object and say that the value of DAYSIN(I) for some I might still be incorrect. This, of course, is true and the values inserted in the table should be carefully checked. However, the point still remains that the specifications of a program can never be guaranteed to correspond with what was "meant" to be done.

Line 3.

We wish to show that line (3) correctly computes DAYSIN(2) for any possible year. Our synthesized assertion is:

Synthesized Assertion. DAYSIN(2) is correct.

Again, this could be shown by comparing some predicate that specified what the program was supposed to compute, with the predicate actually computed by the program. Again, the problem here is that these two predicates will probably be identical, and both could well be wrong. However, if we allow ourselves the use of test data, we can check out the values produced on certain test data by looking them up, perhaps in an almanac. We will then be able to generalize from the specific years used as test data to all years.

Test Data Assertion. Our technique for choosing test data in this example is similar to the technique used in Goodenough and Gerhart [1975]. The major difference is that in this example, we are dividing the domain into classes locally. This is extremely crucial, since the number of test cases may grow exponentially in the number of factors we are considering. By definition, dividing our input into classes locally, the number of cases will be kept low.

Our model is the limited entry decision table. Notice that not all of the combinations of predicates can be satisfied. We choose one sample test data from each of the combinations of predicates that can be satisifed. Our particular choice of test data should be motivated chiefly by simplicity of certifying the correctness of the value of the result. Our choice of test data in this example is given in Table 3.1.

Table 3.1

Predicate				
YEAR MOD 4 ≠ 0	✓			
YEAR MOD 100 = 0		✓	✓	
YEAR MOD 400 ≠ 0	✓	✓		✓
Test Data	7	200	800	8
Correct Values	28	28	29	29

✓ indicates that the given predicate is satisfied.

Verification of Test Data Assertion. We now run the statement on this sample test data, and find that the correct answers are generated.

We now wish to generalize from our test data.

Generalization Assertion. DAYSIN is totally determined by the predicates YEAR MOD 4 = 0, YEAR MOD 100 = 0, and YEAR MOD 400 ≠ 0.

Verification of Generalization Assertion. From the program, it is clear that DAYSIN is totally determined by the predicates

YEAR MOD 4 ≠ 0, YEAR MOD 100 = 0, and
YEAR MOD 400 ≠ 0.

Notice that if we failed to consider some case, we could still end up with a correctness "proof" of a program that did not do what we wanted. We have, however, gained the following advantages over a traditional correctness proof. First of all, we will catch an error resulting from failing to combine the predicates correctly. Secondly, we are examining the predicates that must be considered at the point in the program where these are to be considered, rather than at the output arc.

Finally, we prove our synthesized assertion.

Proof of Synthesized Assertion. The proof follows directly from the test data assertion, the generalization assertion, and Theorem 2.1. □

Line 4: Synthesized Assertion.

DAYS = (# of days by which second date follows first date)
- (# of days in complete intervening months).

At this point, we wish to show that our program behaves properly over month boundaries. It is very simple here to make a mistake that will give answers that will be off by one or even two days, both in the program and in its specifications. We choose two days that are very close together, yet in different months so that the answer produced can easily be cerified. In this particular example, symbolic execution may be

very helpful. We wish to show that the program gives us a 1 if DAY1 = DAYSIN(I). The simplest way to do this is to symbolically execute the program with DAYSIN(I) as the symbolic value of DAY1. Otherwise, we would have to execute the program with all possible values of DAYSIN(I) for $1 \leq I \leq 11$. We have the following

Test Data Assertion. Let DAY1 = DAYSIN(I), MONTH(1) = I, DAY2 = 1. Then DAYS = 1.

Verification of Test Data Assertion. We run the statement on the given symbolic test data, and we find that DAYS = 1 after execution.

We now wish to generalize from this specific test data. Our generalization assertion is

Generalization Assertion. DAYS is additive in DAY2, subtractive in DAY1.

Verification of Generalization Assertion. The assertion follows from the fact that DAYS = DAY2 + DAYSIN(MONTH) - DAY1, and that DAYSIN(MONTH1) is independent of DAY2 and DAY1.

Finally, we must prove our synthesized assertion.

Proof of Synthesized Assertion. The proof follows immediately from the test data assertion, the generalization assertion, and Theorem 2.2. □

Line 5.

The fifth line computes the number of days in intervening months, and adds it to the value of DAYS that had been computed, giving us the number of days by which the second date follows the first.

Synthesized Assertion. DAYS = number of days by which the second date follows the first.

The proof of this assertion is greatly simplified by our use of distributed correctness.

Proof. The lower and upper bounds of the FOR loop are MONTH1+1 and MONTH1-1. These are clearly the first and last complete intervening months. By the correctness of DAYSIN(I) for $1 \leq I \leq 12$, proven after line 4, we know that this statement adds to DAYS the correct number of days in intervening months. Therefore after execution of line 5, we have

DAYS = number of days by which second date follows first date
- days in complete intervening month
+ days in complete intervening months.

Therefore

DAYS = number of days by which second date follows first date. □

Line 6.

Synthesized Assertion. The program prints the correct value of DAYS.

Proof. If MONTH1 = MONTH2, then the program prints the correct value of DAYS. If MONTH1 ≠ MONTH2, then the program prints the correct value of DAYS. □

Note that in this example, the proof follows directly from other assertions, and no test data information is really necessary or even desirable.

Example 3.3. Our third and final example due to Parnas is reproduced from Robinson and Levitt [1975]. Similar examples can be found in Parnas [1972] and Hoare [1972]. It is reproduced below in Table 3.2.

In this example, it is demonstrated that our testing techniques can be extremely useful in verifying large programs. The example we use employs Parnas modules, cf. Parnas [1972], to factor the program into pieces that can be verified. We shall show that our testing techniques can be used to show the correctness of the behavior of each of these modules.

Table 3.2

Register Module Specification

(Quotes signify value before application of function.)

integer V-Function: LENGTH
Comment: Returns the number of occupied positions in the register.
Initial value: LENGTH = 0
Exceptions: none

integer V-Function: CHAR (integer i)
Comment: Returns the value of the i^{th} element of the register.
Initial value: ∀i(CHAR(i) = undefined)
Exceptions: I_OUT_OF_BOUNDS: $i \leq 0 \vee i >$ LENGTH

0-Function: INSERT (integer i,j)
Comment: Inserts the value j after position i, moving subsequent values one position higher.
Exceptions: I_OUT_OF_BOUNDS: $i < 0 \vee i >$ LENGTH
J_OUT_OF_BOUNDS: $j < 0 \vee j > 255$
TOO_LONG: LENGTH ≥ 1000
Effects: LENGTH = 'LENGTH' + 1
∀k (1 ≤ k ≤ 'LENGTH' + 1)
[CHAR(k) = if k ≤ i then 'CHAR'(k)
else if k = i+1 then j
else 'CHAR'(k-1)]

0-Function: DELETE (integer i)
Comment: Deletes the i^{th} element of the register, moving the subsequent values to fill in the gap.
Exceptions: I_OUT_OF_BOUNDS: $i < 0 \vee i >$ LENGTH
UNDERFLOW: LENGTH = 0
Effects: LENGTH = 'LENGTH' - 1
∀k (1 ≤ k ≤ 'LENGTH' - 1)
[CHAR(k) = if k < i then 'CHAR'(k)
else 'CHAR'(k+1)]
CHAR('LENGTH') = undefined

Once the $\mathcal{O}$ and V functions have been written, testing is ideal for checking whether or not they behave properly with respect to exception conditions. However, for this example, we concern ourselves mainly with the behavior of the module as a whole.

Robinson and Levitt [1975] cite four invariants that can be proven about the register module, which they state are quite useful in shortening proof of programs that call the module. These invariants are

(1) $0 \leq \text{LENGTH} \leq 1000$
(2) $\forall i(0 < i \leq \text{LENGTH} \supset \text{DEFINED}(\text{CHAR}(i)))$.
(3) $\forall i(\text{DEFINED}(\text{CHAR}(i)) \supset 0 \leq \text{CHAR}(i) \leq 255)$.
(4) DEFINED(LENGTH).

We shall now sketch how testing can be used to verify the correctness of this module. For this example, we will present our arguments for correctness in a style that a programmer might wish to use in providing an informal argument for correctness of his program. Only two of the four are proven. In both of the proofs we shall present both test assertions and generalization assertions.

(1) $0 \leq \text{LENGTH} \leq 1000$.

Proof. Only the $\mathcal{O}$ functions can change the value of LENGTH. Therefore, we must prove that this invariant is preserved under the operations of INSERT and DELETE. We first check that LENGTH $\leq$ 1000 is an invariant. Since DELETE decreases LENGTH, DELETE will cause no problems. For INSERT, we have LENGTH = 'LENGTH' + 1. It is clear that if the assertion is maintained for LENGTH = 1000, the assertion will be maintained for any initial value of LENGTH. We run a test of INSERT with LENGTH = 1000, and we find that an exception condition is generated. Next, we check that LENGTH $\geq$ 0 is an invariant. Since INSERT increases LENGTH, INSERT will cause no problems. For DELETE, we have LENGTH = 'LENGTH' - 1. It is clear that if the assertion is maintained for LENGTH = 0, the assertion will be maintained for any initial value of LENGTH. We run a test of DELETE with LENGTH = 0, and find that an exception condition is generated. □

(3) We wish to show $\forall i(\text{DEFINED}(\text{CHAR}(i)) \supset 0 \leq \text{CHAR}(i) \leq 255)$. We examine the effect of INSERT and DELETE on this invariant. It is easy to show that DELETE maintains this invariant. To check if INSERT maintains the invariant, we execute calls INSERT(i,256) and INSERT(i,-1). We choose 256 and -1 since they lie immediately outside the boundaries of where CHAR(i) should be defined. Note that i could be any value that we choose. We could either pick any value of i, and show that if the invariant is preserved with some value of i, it is preserved with any value of i. If we had the facilities of a symbolic executor, we might wish to execute our program with the symbol 'i'. We see that INSERT (i,256) and INSERT(i,-1) both generate exception conditions. Clearly, j > 256 and j < -1 will generate exception conditions based on generalizing from our test data.

Notice how in both instances, the boundary values were crucial points at which errors would have been very easy to make. Thus, our consideration of test data results was crucial in arguing the invariance of these assertions.

4. Conclusions

We have shown how information gained by testing a program may be useful in helping to prove the program correct. Rather than being antithetical to proving correctness, proper use of test data may in fact suggest a simple strategy for proving programs correct. In the way in which we are using testing, the process in fact does not run counter to the ideas of structured programming. Rather, it becomes an integral part of the process. When developing a program, it is quite natural to conceive of the programmer considering how the program works on some specific examples, and trying to guarantee that in fact the program behaves in a similar fashion on the whole input domain. In the case of our calendar example, it was simpler to specify program behavior by specifying how it should behave on test data, and the use of test data simplified the verification.

The techniques described in this paper seem to formally model some of the informal techniques used by programmers in checking the correctness of their programs. For practical applications, the techniques specified in this paper might be short-cut. For example, one might use these techniques on larger chunks of code. By dealing with modules, we have shown how the technique is very flexible in terms of how large a chunk of code we deal with at a time.

A great deal of work remains to be done in the area of using test data to prove program correctness. Some of the areas that should be considered are:

(1) Many additional theorems could be proven, similar to the ones in Section 2, to be used in generalizing from specific test data to a larger domain.
(2) It would be useful to examine techniques for automatically generating test data. Techniques exist for automatically generating assertions, cf. Wegbreit [1974], Katz and Manna [1974]. Similar techniques might be used to automatically generate test data. This problem is probably very difficult.
(3) A system could be built for proving programs correct using test data. Such a potential might be built into current symbolic execution systems.
(4) The methods outlined in this paper should be tested against large-scale programs to examine their efficiency in either proving these programs as a whole, or in guaranteeing effectiveness of sections of these programs. The problem of testing interaction among modules of a large program should be examined further.

Acknowledgments

I would like to acknowledge Susan L. Graham and many graduate students of the Computer Science

Division at UC Berkeley, including Jeffrey Barth, Steven Glanville, Larry Ruzzo, Mark Wegman and Amiram Yehudai for their helpful suggestions.

Bibliography

1. Boyer, R.S., Elspas, B. and Levitt, K. SELECT - A formal system for testing and debugging programs by symbolic execution. Proceedings of International Conference on Reliable Software (1975) pp. 234-245.

2. Boyer, R.S. and Moore, J.L. Proving theorems about LISP functions. Journal of the Association for Computing Machinery. Vol. 22 (January 1975) pp. 129-144.

3. Burstall, R.M. Proving properties of programs by structural induction. Computer Journal, Vol. 12 (1969) pp. 41-48.

4. Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R. Structured Programming. Academic Press, London and New York, 1972.

5. Floyd, R.W. Assigning meanings to programs. Proceedings of a Symposium in Applied Mathematics, Vol. 19 (J.T. Schwartz, ed.), American Mathematics Society (1967) pp. 19-32.

6. Goodenough, J.B. and Gerhart, S.L. Toward a theory of test data selection. Proceedings of International Conference on Reliable Software (1975) pp. 493-510.

7. Henderson, P. Finite state modelling in program development. Proceedings of the International Conference on Reliable Software (1975) pp. 221-227.

8. Hoare, C.A.R. Proof of correctness of data representations. Acta Informatica, Vol. 1, Fasc. 4 (1972) pp. 271-281.

9. Howden, W.E. Methodology for the generation of program test data. University of California, Irvine, 1974.

10. Katz, S.M. and Manna, Z. Logical analysis of programs. The Weizmann Institute of Science, 1974.

11. King, J.C. A new approach to program testing. Proceedings of International Conference on Reliable Software (1975) pp. 228-233.

12. Liskov, B.H. and Zilles, S.N. Specification techniques for data abstractions. IEEE Transactions in Software Engineering, Vol. 1, No. 1, (March 1975) pp. 7-19.

13. London, R.L. A view of program verification. Proceedings of International Conference on Reliable Software (1975) pp. 534-545.

14. Manna, Z., Ness, S. and Vuillemin, J. Inductive methods for proving properties of programs. Communications of the ACM 16, 8 (August 1973) pp. 491-502.

15. Miller, E.F. Jr. and Melton, R.A. Automated generation of testcase datasets. Proceedings of International Conference on Reliable Software (1975) pp. 51-58.

16. Parnas, D.L. A technique for software module specification with examples. Communications of the ACM 15, 5 (May 1972) pp. 330-336.

17. Robinson, L. and Levitt, K. Proof techniques for hierarchically structured programs. Stanford Research Institute, Menlo Park, CA (1975).

18. Sintzoff, M. Calculating properties of programs by valuations on specific models. Proceedings of ACM Conference on Proving Assertions About Programs (January 1972) pp. 203-207.

19. Wegbreit, B. The synthesis of loop predicates. Communications of the ACM 16, 2 (February 1974) pp. 102-112.