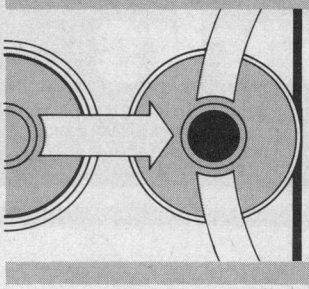
Hints on Test Data Selection: Help for the Practicing Programmer

Richard A. DeMillo
Georgia Institute of Technology

Richard J. Lipton and Frederick G. Sayward
Yale University



In many cases tests of a program that uncover simple errors are also effective in uncovering much more complex errors. This so-called coupling effect can be used to save work during the testing process.

Much of the technical literature in software reliability deals with tentative methodologies and underdeveloped techniques; hence it is not surprising that the programming staff responsible for debugging a large piece of software often feels ignored. It is an economic and political requirement in most production programming shops that programmers shall spend as little time as possible in testing. The programmer must therefore be content to test cleverly but cheaply; state-of-the-art methodologies always seem to be just beyond what can be afforded. We intend to convince the reader that much can be accomplished even under these constraints.

From the point of view of management, there is some justification for opposing a long-term view of the testing phase of the development cycle. Figure 1 shows the relative effect of testing on the remaining system bugs for several medium-scale systems developed by System Development Corporation.[1] Notice that in the last half of the test cycle, the average change in the known-error status of a system is 0.4 percent per unit of testing effort, while in the first half of the cycle, 1.54 percent of the errors are discovered per unit of testing effort. Since it is enormously difficult to be convincing in stating that the testing effort is complete, the apparently rapidly decreasing return per unit of effort invested becomes a dominating concern. The standard solution, of course, is to limit the amount of testing time to the most favorable part of the cycle.

Programmers have one great advantage that is almost never exploited: they create programs that are *close* to being correct!

How, then, should programmers cope? Their more sophisticated general methodologies are not likely to be applicable.[2] In addition, they have the burden of convincing managers that their software is indeed reliable.

The coupling effect

Programmers, however, have one great advantage that is almost never really exploited: they create programs that are *close* to being correct! Programmers do not create programs at random; competent programmers, in their many iterations through the design process, are constantly whittling away the distance between what their programs look like now and what they are intended to look like. Programmers also have at their disposal

- a rough idea of the kinds of errors most likely to occur;
- the ability and opportunity to examine their programs in detail.

Error classifications. In attempting to formulate a comprehensive theory of test data selection, Susan Gerhart and John Goodenough[3] have suggested that errors be classified as follows:

(1) failure to satisfy specifications due to implementation error;
(2) failure to write specifications that correctly represent a design;
(3) failure to understand a requirement;
(4) failure to satisfy a requirement.

But these are global concerns. Errors are always reflected in programs as

- missing control paths,
- inappropriate path selection, or
- inappropriate or missing actions.

0018-9162/78/0400-0034$00.75 © 1978 IEEE

We do not explicitly address classifications (2) and (3) in this article, except to point out that even here a programmer can do much without fancy theories. If we are right in our perception of programs as being close to correct, then these errors should be detectable as small deviations from the intended program. There is an amazing lack of published data on this subject, but we do have some idea of the most common errors. E. A. Youngs, in his PhD dissertation,[4] analyzed 1258 errors in Fortran, Cobol, PL/I, and Basic programs. The errors were distributed as shown in Table 1.

In addition to these errors, certain other errors were present in negligible quantities. There were, for instance, operating system interface errors, such as incorrect job identification and erroneous external I/O assignment. Also present were errors in comments, pseudo-ops, and no-ops which for various reasons created detectable error conditions.

Complex errors coupled. How, then, do the relatively simple error types discovered by Youngs connect with the Gerhart-Goodenough error classification? Well, the naive answer is that since arbitrarily pernicious errors may be responsible for a given failure, it must be that simple errors compound in more massive error conditions. For the practical treatment of test data, the Youngs error statistics, therefore, do not seem to help much at all. Fortunately though, the observation that programs are "close to correct" leads us to an assumption which makes the high frequency of simple errors very important:

> *The coupling effect:* Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors.

In other words, complex errors are *coupled* to simple errors. There is, of course, no hope of "proving" the coupling effect; it is an empirical principle. If the coupling effect can be observed in "real-world" programs, then it has dramatic implications for testing strategies in general and domain-specific, limited testing in particular. Rather than scamper after errors of undetermined character, the tester should attempt a systematic search for simple errors that will also uncover deeper errors via the coupling effect.

Path analysis. This point seems so obvious that it's not worth making: test to uncover errors. Yet it's a point that's often lost in the shuffle. In a common methodology known as *path analysis*, the point of the test data is to drive a program through all of its control paths. It is certainly hard to criticize such a goal, since a thoroughly tested program must have been exercised in this way. But unless one recognizes that the test data should also distinguish errors, he might be tempted to conclude, for example, that the program segment diagrammed in Figure 2 can be tested by exercising paths 1-2 and 1-3, even though one of the clauses P and Q may not have been affected at all! In general, the relative ordering of P and Q may be irrelevant or partially unknown and side effects may occur, so that actually the eight paths shown in Figure 3 are required to ensure that the statement has been adequately tested.

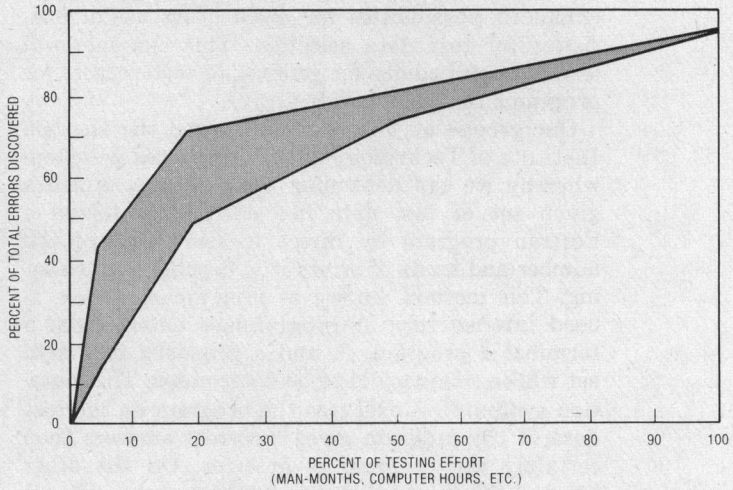


Figure 1. More programming errors are found in the early part of the test cycle then in the final part.

Table 1. Frequency of occurrence of 1258 errors in Fortran, Cobol, PL/I, and Basic programs.

Error Type	Relative Frequency of Occurrence
Error in assignment or computation	.27
Allocation error	.15
Other, unknown, or multiple errors	.11
Unsuccessful iteration	.09
Other I/O error	.07
I/O formatting error	.06
Error in brahching	
unconditional	.01
conditional	.05
Parameter or subscript violation	.05
Subprogram invocation error	.05
Misplaced delimiter	.04
Data error	.02
Error in location or marker	.02
Nonterminating subprogram	.01

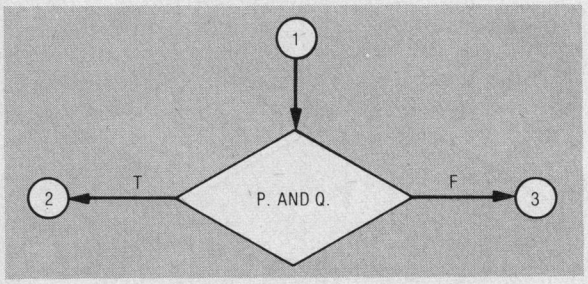


Figure 2. Sample program segment with two paths.

Two examples given below indicate that test data derived to uncover simple errors can, in fact, be vastly superior to, say, randomly chosen data or data generated for path analysis. A byproduct of the discussion will be some evidence for the coupling effect. A third example reveals another advantage of selecting test data with an eye on coupling: since it's a problem-specific activity, there are enhanced possibilities for discovering useful heuristics for test data selection. This example will lead to useful advice for generating test vectors for programs that manipulate arrays.

Our groups at Yale University and the Georgia Institute of Technology have constructed a system whereby we can determine the extent to which a given set of test data has adequately tested a Fortran program by direct measurement of the number and kinds of errors it is capable of uncovering. This method, known as *program mutation*, is used interactively: A programmer enters from a terminal a program, P, and a proposed test data set whose adequacy is to be determined. The mutation system first executes the program on the test data; if the program gives incorrect answers then certainly the program is in error. On the other hand, if the program gives correct answers, then it may be that the program is still in error, but the test data is not sensitive enough to distinguish that error: it is not adequate. The mutation system then creates a number of *mutations* of P that differ from P only in the occurrence of simple errors (for instance, where P contains the expression "B.LE.C" a mutation will contain "B.EQ.C"). Let us call these mutations $P_1, P_2, \ldots, P_k$.

Now, for the given set of test data there are only two possibilities:

(1) on that data P gives different results from the P_i mutations, or

(2) on that data P gives the same results as some P_j.

In case (1) P_i is said to be *dead*: the "error" that produced P_i from P was indeed distinguished by the test data. In case (2), the mutant P_j is said to be *live*; a mutant may be live for two reasons:

(1) the test data does not contain enough sensitivity to distinguish the error that gave rise to P_j, or
(2) P_j and P are actually equivalent programs and no test data will distinguish them (i.e., the "error" that gave rise to P_j was not an error at all).

Test data that leaves no live mutants or only live mutants that are equivalent to P is adequate in the following sense: Either the program P is correct or there is an unexpected error in P, which—by the coupling effect—we expect to happen seldom if the errors used to create the mutants are carefully chosen.

Now, it is not completely apparent that this process is computationally feasible. But, as we describe in more detail elsewhere, there is a very good choice of methodology for generating mutations to bring the procedure within attractive economic bounds.[5]

Apparently, the information returned by the mutation system can be effectively utilized by the programmer. The programmer looks at a negative response from the system as a "hard question" concerning his program (e.g., "The test data you've given me says it doesn't matter whether or not this test is for equality or inequality; why is that?") and is able to use his answers to the question as a guide in generating more sensitive test data.

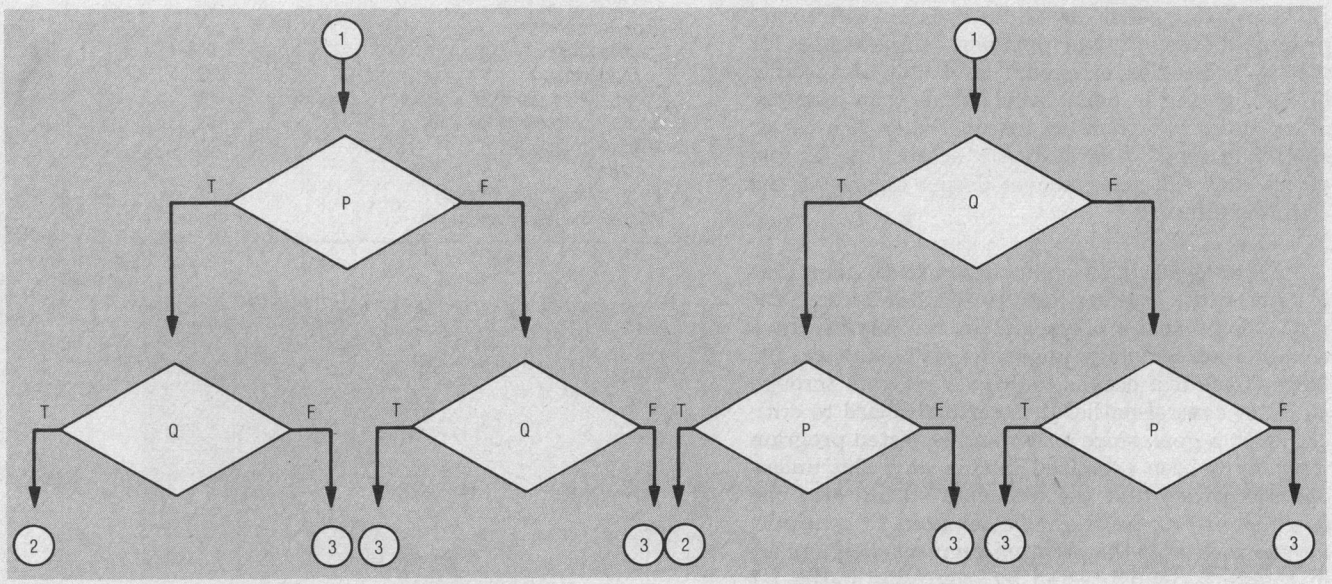


Figure 3. Eight paths may be required for an adequate test.

A simple example

Our first example is very simple; it involves the MAX algorithm used for other purposes by Peter Naur in the early 1960's. The task is to set a variable R to the *index* of the first occurrence of a maximum element in the vector A(1), . . ., A(N). For example, the following Fortran subroutine might be offered as an implementation of such an algorithm:

```
      SUBROUTINE MAX (A,N,R)
      INTEGER A(N),I,N,R
    1 R=1
    2 DO 3 I=2,N,1
    3 IF (A(I).GT.A(R))R=I
      RETURN
      END
```

We will choose for our initial set of test data three vectors (Table 2).

Table 2. Three vectors constitute the initial set of test data.

	A(1)	A(2)	A(3)
data 1	1	2	3
data 2	1	3	2
data 3	3	1	2

How sensitive is this data? By inspection, we notice that if an error had occurred in the relational operation of the IF statement, then either data 1, data 2, or data 3 would have distinguished those errors, except for one case. None of these data vectors distinguishes .GE. from .GT. in the IF statement. Similarly, these vectors distinguish all simple errors in constants except for starting the DO loop at "1" rather than "2." All simple errors in variables are likewise distinguished except for the errors in the IF statement which replace "A(I)" by "I" or by "A(R)."

That is, if we run the data set above in any of the following mutants of MAX, we get the same results.

```
      SUBROUTINE MAX (A,N,R)
      INTEGER A(N),I,N,R
    1 R=1
    2 DO 3 I=1,N,1
    3 IF(A(I).GT.A(R))R=1
      RETURN
      END

      SUBROUTINE MAX (A,N,R)
      INTEGER A(N),I,N,R
    1 R=1
    2 DO 3 I=2,N,1
    3 IF(I.GT.A(R))R=1
      RETURN
      END

      SUBROUTINE MAX (A,N,R)
      INTEGER A(N),I,N,R
    1 R=1
    2 DO 3 I=2,N,1
    3 IF(A(I).GE.A(R))R=1
      RETURN
      END

      SUBROUTINE MAX (A,N,R)
      INTEGER A(N),I,N,R
    1 R=1
    2 DO 3 I=2,N,1
    3 IF(A(R).GT.A(R))R=1
      RETURN
      END
```

Let us try to kill as many of these mutants as possible. In view of the first difficulty, we might guess that our data is not yet adequate because it does not contain repeated elements. So, let us add

	A(1)	A(2)	A(3)
data 4	2	2	1

Now, replacing .GT. by .GE. and running on data 4 gives erroneous results so that all mutants arising from simple relational errors are dead. Surprisingly, data 4 also distinguishes the two errors in A(I); so, we are left with only the last mutant arising from the "constant" error: variation in beginning the DO loop. But closer inspection of the program indicates that starting the DO loop at "1" rather than "2" has no effect on the program, other than to trivially increase its running time. So no choice of test data will distinguish this "error," since it results in a program equivalent to MAX. So we conclude that since the test data 1-4 leaves only live mutants that are equivalent to MAX, it is adequate.

Comparisons with path analysis

This example illustrates hidden paths in a program which should also be exercised by the test data. To illustrate what hidden paths are, consider the Fortran program—call it P—suggested by C. V. Ramamoorthy and his colleagues:[6]

```
      INTEGER A,B,C,D
      READ 10,A,B,C
   10 FORMAT(4I10)
    5 IF((A.GE.B) .AND.(B.GE.C) ) GOTO 100
      PRINT 50
   50 FORMAT(1H ,*LENGTH OF TRIANGLE NOT IN
     1ORDER*)
      STOP
  100 IF((A.EQ.B) .OR. (B.EQ.C)) GOTO 500
      A=A*A
      B=B*B
      C=C**2
      D=B+C
      IF (A.NE.D) GOTO 200
      PRINT 150
  150 FORMAT(1H ,*RIGHT ANGLED TRIANGLE*)
      STOP
  200 IF (A.LT.D). GOTO 300
      PRINT 250
  250 FORMAT(1H ,*OBTUSE ANGLED TRIANGLE*)
      STOP
  300 PRINT 350

  350 FORMAT(1H,*ACUTE ANGLED TRIANGLE*)
```

```
      STOP
500   IF ( (A.EQ.B) .AND. (A.EQ.C) ) GOTO 600
      PRINT 550
550   FORMAT(1H,*ISOCELES TRIANGLE*)
      STOP
600   PRINT 650
650   FORMAT(1H ,*EQUILATERAL TRIANGLE*)
      STOP
      END
```

The intent of this program is to categorize triangles, given the lengths of their sides. A typical path analysis system will derive test data—call it T—which exercises all paths of P (Table 3).

Table 3. Test data *T* to exercise the Fortran program *P*.

TEST CASE	A	B	C	TRIANGLE TYPE
1	2	12	27	ILLEGAL
2	5	4	3	RIGHT ANGLE
3	26	7	7	ISOSCELES
4	19	19	19	EQUILATERAL
5	14	6	4	OBTUSE
6	24	23	21	ACUTE

Now consider the following mutant program P':

```
      INTEGER, A,B,C,D
      READ 10,A,B,C
 10   FORMAT(4I10)
  5   IF( A.GE.B ) GOTO 100
      PRINT 50
 50   FORMAT(1H ,*LENGTH OF TRIANGLE NOT IN
     1ORDER*)
      STOP
100   IF( B.EQ.C ) GOTO 500
      A=A*A
      B=B*B
      C=C**2
      D=B+C
      IF (A.NE.D) GOTO 200
      PRINT 150
150   FORMAT(1H ,*RIGHT ANGLED TRIANGLE*)
      STOP
200   IF (A.LT.D) GOTO 300
      PRINT 250
250   FORMAT(1H ,*OBTUSE ANGLED TRIANGLE*)
      STOP
300   PRINT 350
350   FORMAT(1H ,*ACUTE ANGLED TRIANGLE*)
      STOP
500   IF ( (A.EQ.B) .AND. (A.EQ.C) ) GOTO 600
      PRINT 550
550   FORMAT(1H ,*ISOCELES TRIANGLE*)
      STOP
600   PRINT 650
650   FORMAT(1H ,*EQUILATERAL TRIANGLE*)
      STOP
      END
```

P' prints the same answers as P on T but P' is clearly incorrect since it categorizes the two test cases shown in Table 4 as acute angle triangles:

Table 4. Two test cases are acute angle triangles.

TEST CASE	A	B	C	TRIANGLE TYPE
7	7	5	6	ILLEGAL
8	26	26	7	ISOSCELES

P and P' differ only in the logical expressions found at statements 5 and 100.* The test data T does not sufficiently test the compound logical expressions of P; T only tests the single-clause logicals found in the corresponding statements of P'. Hence, T' is a stronger test of P than is T (i.e., for P we have more confidence in the adequacy of T' than in the adequacy of T). Note that the logical expression in statement 5 of P could be replaced by B.GE.C to yield a program P'' which produces correct answers on T'. The test case $A=5$, $B=7$, $C=6$ will remedy this and provide still a stronger test of P.

A more substantial example

Our last example involves the FIND program of C.A.R. Hoare.[7] FIND takes, as input, an integer array A, its size $N \geq 1$, and an array index F, $1 \leq F \leq N$. After execution of FIND, all elements to the left of $A(F)$ have values no larger than $A(F)$ and all elements to the right are no smaller. Clearly, this could be achieved by sorting A; indeed, FIND is an inner loop of a fast sorting algorithm, although FIND executes faster than any sorting program. The Fortran version of FIND, translated directly from the Algol version, is given below:

```
      SUBROUTINE FIND(A,N,F)
C
C     FORTRAN VERSION OF HOARE'S FIND
C     PROGRAM (DIRECT TRANSLATION OF
C     THE ALGOL 60 PROGRAM FOUND IN
C     HOARE'S "PROOF OF FIND" ARTICLE
C     IN CACM 1971).
      INTEGER A(N),N,F
      INTEGER M,NS,R,I,J,W
      M=1
      NS=N
 10   IF(M.GE.NS) GOTO 1000
      R=A(F)
      I=M
      J=NS
 20   IF(I.GT.J) GOTO 60
 30   IF(A(I).GE.R) GOTO 40
      I=I+1
      GOTO 30
 40   IF(R.GE.A(J)) GOTO 50
      J=J-1
      GOTO 40
 50   IF(I.GT.J) GOTO 20
C
C     COULD HAVE CODED GO TO 60 DIRECTLY
C     -DIDN'T BECAUSE THIS REDUNDANCY
C     IS PRESENT IN HOARE'S ALGOL
C     PROGRAM DUE TO THE SEMANTICS OF
C     THE WHILE STATEMENT.
C
      W=A(I)
      A(I)=A(J)
      A(J)=W
      I=I+1
      J=J-1
      GO TO 20
```

*The clause A.EQ.B in statement 500 is redundant.

```
60    IF(F.GT.J) GOTO 70
      NS=J
      GOTO 10
70    IF(I.GT.F) GOTO 1000
      M=I
      GOTO 10
1000  RETURN
      END
```

FIND is of particular interest for us because a subtle multiple-error mutant of FIND, called BUGGYFIND, has been extensively analyzed by SELECT, a system that generates test data by symbolic execution.[8] In FIND, the elements of A are interchanged depending on a conditional of the form

X .LE. A(F) .AND. A(F) .LE. Y

Since *A(F)* itself may be exchanged, the effect of this test is preserved by setting a temporary variable $R = A(F)$ and using the conditional

X .LE. R .AND. R .LE. Y

In BUGGYFIND, the temporary variable R is not used; rather, the first form of the conditional is used to determine whether the elements of A are to be exchanged. The SELECT system derived the test data $A = (3,2,0,1)$ and $F = 3$, on which BUGGYFIND fails. The authors of SELECT observed that BUGGYFIND fails on only 2 of the 24 permutations of (0,1,2,3), indicating that the error is very subtle.*

We will first describe a simple-error analysis of the mutants of FIND, beginning with initially naive guesses of test data and finishing with a surprisingly adequate set of 7 A vectors. This data will be called D_1. The detailed analysis needed to determine how many errors are distinguished by a data set were carried out on the Mutation system at Yale University.

We have asked several colleagues how they would test FIND, and they have nearly unanimously replied that they would use permutations. We first describe analysis which we have done using permutations of the array indices as data elements. In one case, we use all permutations of length 4 and in another case, we use *random* permutations of lengths 5 and 6. Surprisingly, the intuitively appealing choice of permutations as test data is a very poor one.

We then describe analysis in which another popular intuitive method is used: random data. We show that the adequacy of random data is very dependent on the interval from which the data is drawn (i.e., problem-specific information is needed to obtain good results).

Finally, we find evidence for the coupling effect (i.e., adequate simple-error data kills multiple-error mutants) in two ways. First, the multiple-error mutant BUGGYFIND fails on the test data D_1. Next, we describe the very favorable results of executing random multiple-error mutants of FIND on D_1.

We begin the analysis with the 24 permutations of (0,1,2,3) with F fixed at 3. The results are surprisingly poor, as 58 live mutants are left. That is, with these 24 vectors there are 58 possible changes that could have been made in FIND that would have yielded identical output. Eventually, by increasing the number of A vectors to 49, only 10 live mutants remain. Using a data reduction heuristic, the 49 A vectors can be reduced to a set of seven A vectors, leaving 14 live mutants. These vectors appear in Table 5.

Table 5. D_1—The simple-error adequate data for FIND.

TEST CASE	A	F
1	(−19,34,0,−4,22, 12,222,−57,17)	5
2	(7,9,7)	3
3	(2,3,1,0)	3
4	(−5,−5,−5,−5)	1
5	(1,3,2,0)	3
6	(0,2,3,1)	3
7	(0)	1

In constructing the initial data, after the 24 permutations, the 49 A vectors were chosen somewhat haphazardly at first. Later, A vectors were chosen specifically to eliminate a small subset of the remaining errors. There were some interesting observations concerning the 49 vectors:

(1) The average A vectors kills about 550 mutants.
(2) The "best" A vector kills 703 mutants (test case 1 of Table 5).
(3) The "worst" A vector kills only 70 mutants. This was the degenerate $A = (0)$.

The data reduction heuristic uses both the best and the worst A vectors to pare the 49 A vectors to seven.

The final step in showing that the data of Table 5 is indeed adequate is to show that the 14 remaining mutants are programs that are actually equivalent to FIND. That is, the 14 "errors" that could have been made are not really errors at all. One might be surprised at the large number of equivalent mutants (approximately 2 percent). This we attribute to FIND's long history (it was first published in 1961). Over the years, FIND has been "honed" to a very efficient state—so efficient that many slight variations result in equivalent but slower programs. For example, the conditional

I. GT. F

in the statement labeled 70 in the FIND can be replaced by any logically false conditional, or the IF statement can be replaced by a CONTINUE statement, to result in an equivalent but slower program. It is not likely that this phenomenon will occur in programs which haven't been "fine-tuned." We estimate that production programs have well under 1 percent equivalent mutants.

Let us now compare D_1 with exhaustive tests on permutations of (0,1,2,3) and then with tests on

*We found that BUGGYFIND failed on only the aforementioned permutation.

random permutations of (0,1,2,3,4) and (0,1,2,3,4,5). Table 6 describes the results for all permutations of (0,1,2,3).

Table 6. Results of all permutations of (1,2,3,4).

NUMBER OF TEST CASES	VALUES OF F	NUMBER OF LIVE MUTANTS
24	1	158
24	2	60
24	3	58
24	4	141
96	1,2,3,&4	38

In Table 7 the same information is provided for the case of random test data.

Table 7. Results of random permutations.

NUMBER OF RANDOM TEST CASES	SIZE OF A	VALUE OF F	NUMBER OF LIVE MUTANTS
10	UNIFORM FROM [5,6]	UNIFORM FROM 1 TO SIZE OF A	88
25	↓	↓	65
50	↓	↓	54
100	↓	↓	54
1000	↓	↓	53

As the data indicates, permutations give rather poor results compared with D_1.

Our analysis with random data can be divided into two cases: runs in which the vectors were drawn from poorly chosen intervals and runs in which the vectors were chosen from a good interval (−100,100). The results are described in Tables 8 and 9.

Table 8. Results of random data from poorly chosen intervals.

NUMBER OF RANDOM VECTORS	RANGE OVER WHICH VECTOR VALUES DRAWN	RANGE OVER WHICH SIZE OF A DRAWN	VALUE OF F	NUMBER OF LIVE MUTANTS
10	[100,200]	[1,20]	UNIFORM	28
10	[−200,−100]	[1,20]	FROM	28
10	[−100,−90]	[1,20]	SIZE OF VECTOR	25

Table 9. Results of random data drawn from [−100,100]; other parameters as in Table 8.

NUMBER OF RANDOM VECTORS	NUMBER OF LIVE MUTANTS
10	22
50	17
100	11
1000	10

Although the intervals in Table 8 are poor, one could conceive of worse intervals. For example, draw A from [1, size of A]. However, in view of the permutation results, such data will surely behave worse than that of Table 8.

Three points are in order. First, even with very bad data, D_1 is much better than simple permutations. Second, it took 1000 very good random vectors to perform as well as D_1. Third, using random vectors yields little insight. The insight gained in constructing D_1 was crucial to detecting the equivalent versions of FIND.

The coupling effect shows itself in two ways. First, BUGGYFIND fails on the adequate D_1; hence, we have a concrete example of the coupling effect. Although the second observation involves randomness, and thus is indirect, it is perhaps more convincing than the "one point" concrete BUGGYFIND example. We have randomly generated a large number of k-error mutants for $k > 1$ (called higher order mutants) and executed them on D_1.

Because the number of mutants produced by complex errors can grow combinatorially, it is hopeless to try the complete mutation analysis on complex mutants, but it is possible to select mutants at random for execution on D_1. Of more than 22,000 higher-order errors encountered, only 19 succeed on D_1. These 19 have been shown to be equivalent to FIND. Indeed, we have yet to produce an incorrect higher-order mutant which suceeds on D_1!

Conclusions

Our first conclusion is that systematically pursuing test data which distinguishes errors from a given class of errors also yields "advice" to be used in generating test data for similar programs. For instance, the examples above lead us to the following principles for creating random or nonrandom test data for Fortran-like programs which manipulate arrays (i.e., programs in which array values can also be used as array indices):

(1) Include cases in which array values are outside the size of the array.
(2) Include cases in which array values are negative.
(3) Include cases in which array values are repeated.
(4) Include such degenerate cases as D_1's $A = (0)$ and $A = (-5,-5,-5,-5)$.

Principle (4) was also noticed by Goodenough and Gerhart.[3]

It is important that a testing strategy be conducive to the formation of hypotheses about the way test data should be selected in future tasks. Information transferred between programming tasks provides a source of "virtual resources" to be used in subsequent work. Since the amount of available resources is limited by economic and political barriers, experience—which has the effect of expanding resources—takes on a special importance. It is,

Seemingly simple techniques can be quite sensitive via the coupling effect.

of course, helpful to have available such mechanical aids as the mutation system, but as we have shown even in the absence of the appropriate statistical information, a programmer can be reasonably confident that he is improving his test data selection strategy.

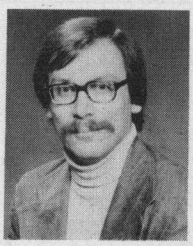
A second conclusion is that until more general strategies for systematic testing emerge, programmers are probably better off using the tools and insights they have in great abundance. Instead of guessing at deeply rooted sources of error, they should use their specialized knowledge about the most likely sources of error in their application. We have tried to illustrate that seemingly simple tests can be quite sensitive, via the coupling effect.

The techniques we advocate here are hardly ever general techniques. In a sense, they require one to deal directly in the details of both coding and the application—a notion that is certainly contrary to currently popular methodologies for validating software. But we believe there is ample evidence in man's intellectual history that he does not solve important problems by viewing them from a distance. In fact, there is an *Alice In Wonderland* quality to fields which claim they can solve other people's problems without knowing anything in particular about the problems.

So, there is certainly no need to apologize for applying ad hoc strategies in program testing. A programmer who considers his problems well and skillfully applies appropriate techniques to their solution—regardless of where the techniques arise—will succeed. ■

References

1. A. E. Tucker, "The Correlation of Computer Program Quality with Testing Effort," System Development Corporation, TM 2219/000/00, January 1965.
2. R. A. DeMillo, R. J. Lipton, A. J. Perlis, "Social Processes and Proofs of Programs and Theorems," *Proc. Fourth ACM Symposium on Principles of Programming Languages*, pp. 206-214. (To appear in *CACM*)
3. John B. Goodenough and Susan L. Gerhart, "Toward a Theory of Test Data Selection," *Proc. International Conference on Reliable Software*, SIGPLAN Notices, Vol. 10, No. 6, June 1975, pp. 493-510.
4. E. A. Youngs, *Error-Proneness in Programming*, PhD thesis, University of North Carolina, 1971.
5. T. A. Budd, R. A. DeMillo, R. J. Lipton, F. G. Sayward, "The Design of a Prototype Mutation System for Program Testing," *Proc., 1978 NCC.*
6. C. V. Ramamoorthy, S. F. Ho, and W. T. Chen, "On the Automated Generation of Program Test Data," *IEEE Trans. on Software Engineering*, Vol. SE-2, No. 4, December 1976, pp. 293-300.
7. C. A. R. Hoare, "Algorithms 65; FIND," *CACM*, Vol. 4, No. 1, April 1961, pp. 321.
8. R. S. Boyer, B. Elspas, K. N. Levitt, "SELECT—A System for Testing and Debugging Programs by Symbolic Execution," *Proc. International Conference on Reliable Software*, SIGPLAN Notices, Vol. 10, No. 6, June 1975, pp. 234-245.



Richard DeMillo has been an associate professor of computer science at the Georgia Institute of Technology since 1976. During the four years prior to that he was assistant professor of computer science at the University of Wisconsin-Milwaukee.

A technical consultant to several government and research agencies and to private industry, he is interested in the theory of computing, programming languages, and programming methodology.

DeMillo received the BA in mathematics from the College of St. Thomas, St. Paul, Minnesota, and the PhD in information and computer science from the Georgia Institute of Technology. He is a member of ACM, the American Mathematical Society, AAAS, and the Association for Symbolic Logic.

Richard J. Lipton is an associate professor of computer science at Yale University. A faculty member since 1973, he pursues research interests in computational complexity and in mathematical modeling of computer systems. He is also a technical consultant to several government agencies and to private industry.

Lipton received the BS in mathematics from Case Western Reserve University and the PhD from Carnegie-Mellon University.

Frederick G. Sayward is an assistant professor of computer science at Yale University, where he pursues research interests in semantical methods for programming languages, the theory of parallel computation as applied to operating systems, the development of programming test methods, and techniques for fault-tolerant computation. Earlier, he worked as a scientific and systems programmer at MIT Lincoln Laboratory.

A member of ACM, the American Mathematical Society, and Sigma Xi, Sayward received the BS in mathematics from Southeastern Massachusetts University, the MS in computer science from the University of Wisconsin-Madison, and the PhD in applied mathematics from Brown University.