AD-A107 774 GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION A--ETC F/G 9/2
PAPERS ON PROGRAM TESTING,(U)
1979 R A DEMILLO, R J LIPTON, F G SAYWARD N00014-79-C-0231
UNCLASSIFIED GIT-ICS-79/04 NL

1 OF 3
AD
A107774

GIT-ICS-79/04

PAPERS ON PROGRAM TESTING

RICHARD A. DEMILLO*
RICHARD J. LIPTON**
FREDERICK G. SAYWARD***

SUMMER, 1979

* GEORGIA INSTITUTE OF TECHNOLOGY
** UNIVERSITY OF CALIFORNIA, BERKELEY
*** YALE UNIVERSITY

PAPERS ON PROGRAM TESTING[†]

[†]By the Program Mutation Groups at Georgia Tech, the University of California, Berkeley, and Yale University. The research reported herein was supported in part by AIRMICS through ARO grant no. DAAG29-78-G-0121 and by ONR grant no. N00014-79-C-0231.

Forward

Since late 1976, we have been involved in what we believe is a new approach to computer program testing, an approach called mutation analysis (and we shall forever be indebted to Jerome Feldman for suggesting the term). The main novelties of the mutation approach to program testing are its simplicity, its empirical basis, its ease of mechanical implementation, and its tractability for scientific analysis. Although much remains to be learned about mutation as a testing tool, there is a considerable body of written material which describes our initial experience with the technique.

Much of this material has appeared only in workshops or as memoranda, so we have been urged to collect it together for wider dissemination. The current collection is the result. The reader should note that the selections do not appear in chronological order; rather, they are organized so that a sufficiently patient reader may proceed from the conceptual basis of mutation analysis through implementation, application, and theoretical issues.

We expect to distill much of this material into a more formal treatment in the coming months; as always, comments and criticisms of all kinds will be appreciated.

Richard A. DeMillo
School of Information and Computer Science
Georgia Institute of Technology

Richard J. Lipton
Department of Electrical Engineering and
Computer Science
University of California, Berkeley

Frederick G. Sayward
Department of Computer Science
Yale University

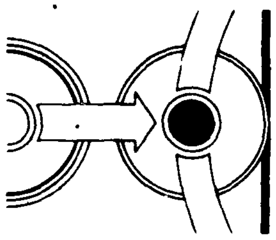
Summer, 1979

TABLE OF CONTENTS

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.

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.

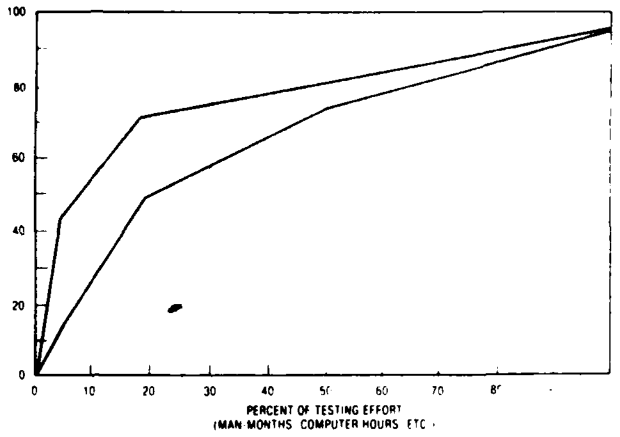


Figure 1. More programming errors are found in the early part of the test cycle then in the final part.

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 branching	
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

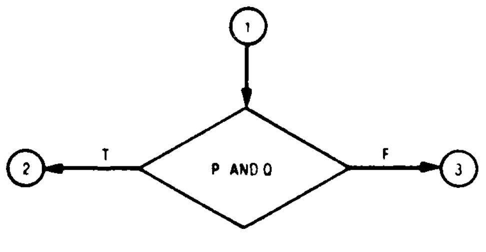


Figure 2. Sample program segment with two paths.

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, . . . , 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_i.

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_i 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_i, or

(2) P_i and *P* are actually equivalent programs and no test data will distinguish them (i.e., the "error" that gave rise to P_i 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.[4]

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.

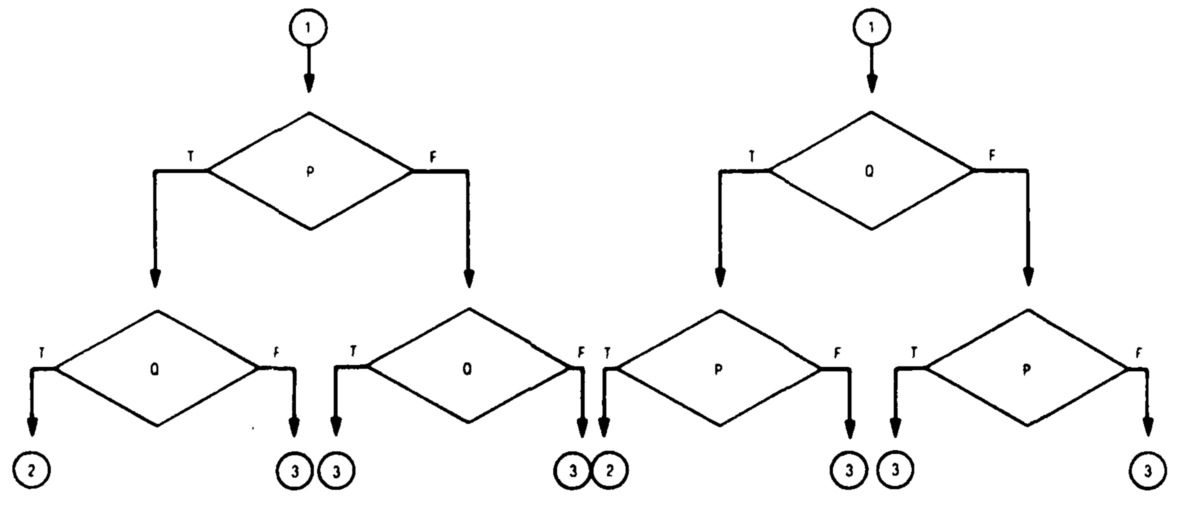


Figure 3. Eight paths may be required for an adequate test.

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=I
  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=I
  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=I
  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:[4]

```
    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.* 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.

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

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)	1
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 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 FROM SIZE OF VECTOR	28
10	[−200 −100]	[1,20]		28
10	[−100 −90]	[1,20]		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.

PROGRAM MUTATION: A NEW APPROACH TO PROGRAM TESTING

R A DeMillo

School of Information and Computer Science
Georgia Institute of Technology
Atlanta GA

R J Lipton

F G Sayward

Department of Computer Science
Yale University
New Haven CT

ACKNOWLEDGEMENT

We acknowledge the work of Tim Budd and Mike Lebowitz and the other members of the Yale University Testing Group for help in implementing and experimenting with the prototype mutation system.

PROGRAM MUTATION: A NEW APPROACH TO PROGRAM TESTING

ABSTRACT

Unlike contemporary software validation methods, where the goal is to establish absolute program correctness, program mutation is a testing method which has a less ambitious but quite useful goal: to establish that a program is either correct or is 'radically' incorrect. The basic concepts of program mutation are explained as well as how the method is applied in building interactive program mutation systems which aid users in establishing this goal. Also, the applications of program mutation as a software project management tool and as a tool for assessing the quality of procured software are overviewed. A prototype mutation system for a non-trivial subset of FORTRAN has been implemented and initial experience with this system is reported. A system for nearly full ANSI FORTRAN is about half implemented and is expected to be ready by early Fall 1978.

INTRODUCTION

Program testing is an inductive science which addresses the following fundamental question:

If a program is correct on a finite number of test cases, is it correct in general?

Finite test data which implies general correctness is called *adequate test data* (004) and since adequate test data cannot in general be derived algorithmically (003), program testing cannot in general be deductive. Recently, *path analysis* (001,002,005,006) and *symbolic execution* (007,008) have emerged as methods which allow one to gain confidence in one's test data's adequacy. Although as with any inductive science it is possible to make false inferences with path analysis, the basic idea is undeniable: test data which exercises all flowchart control paths of a program at least once must be better than test data which does not.

It has been said (020) that from a scientific point of view program testing can hardly be said to be in its infancy. The software engineering community, most notably the program verification school, continue to point out that program testing is insufficient to guarantee program correctness (see (019) for an argument against program verification). We agree. However, since program testing has been used in developing *all* software that has ever solved any *real* problems, we must ask the following rather obvious question:

Given that program testing, while not a perfect technique, has proved to be a very useful technique, how we can develop testing methodologies which have less than perfection (absolute program correctness) as their goals yet still yield substantial gains?

It is all too easy (and wrong) to take the popular viewpoint that program building is a purely logical deductive activity to which program testing is unsuitable. Our viewpoint is that program design and development is an empirical engineering activity for which an inferential formalism has not yet been developed. However, it seems clear that such a formalism is not entirely necessary if one is willing to accept that programming is a human, inductive activity which may never be subject to complete formalism.

In this paper we will describe an on-going research effort which is aimed at achieving gains from program testing while not ensuring perfection. We call our testing methodology *program mutation*. Besides discussing the method, we will overview a prototype system which implements the method and report our initial experiences with program mutation. Moreover, unlike the deductive approaches to software reliability such as program verification (009,010), program mutation provides quantitative information on the status of software development. We will explain how this information can be used effectively throughout a software project's management hierarchy. We will also explain how it can be used as a quality measure for procured software.

THE PROGRAM MUTATION METHODOLOGY

It has been observed (011,022) that the vast majority of errors that remain in software once it has been tested and put into production tend not to be *radical errors** but rather are interacting combinations of *simple errors*. Indeed, there are many 'horror' stories similar to the failure of an early Vangard missile launch because of a missing right parenthesis in a controlling program. So a reasonable goal of program testing is to rule out all combinations of simple errors. That is, design a program testing method with the goal being that if a program passes the test then either:

1 The program is correct.
2 The program is radically incorrect.

Even this seems too ambitious if one attacks directly. First, given a program we must be able to generate all of its simple errors. Assuming that this can be done, we next must eliminate the simple errors and the *complex errors* which emanate from their combinations. Clearly the number of complex errors will be a combinatorial explosion in the number of simple errors. While it may be feasible to eliminate all simple errors, explicit elimination of all complex errors appears intractable.

The goal of the program mutation testing methodology is to establish that a given program is either correct or radically incorrect. Let L be the programming language under consideration. A *mutant operator* is a simple program transformation, dependent on L which produces mutant programs of a given program P. The mutants are also programs in

* There are no agreed on technical definitions of error categories. We too will be informal. By radical we mean errors due to grossly misunderstanding the program specifications. Errors which are difficult if not impossible to capture by general algorithmic methods but which would easily be observed by almost any test or when the software is first put into production. An example would be forgetting to include an action sequence in a decision table program.

L. For example, if

$$I = I+1$$

is a statement in *P*, then

$$I = I-1$$
$$I = I+2$$
$$I = I+0 \text{ (i e a no-op)}$$

are all simple changes which lead to three mutants of *P*. The goal of the mutant operator is to introduce simple errors in *P*, thus producing mutants of *P*. Alternatively, if *P* is incorrect due to a single simple error, some mutant would be a correct program for the given task. There should be several mutant operators, each corresponding to different classes of simple errors that may occur in *L*. Let *M*(*P*) denote the set of all mutants of *P*. Ideally, *M*(*P*) should contain mutants corresponding to all and only the possible simple errors. However, this is too ambitious a goal for general purpose program transformations and we relax the requirement to be that *M*(*P*) covers all simple errors in the sense that *M*(*P*) may also contain mutants which are equivalent to *P*. We let *M**(*P*) denote all the mutants of *P* which come from multiple applications of mutant operators on *P*. These mutants are also programs in *L*.

Let *D* be the input domain of *P*. *P* is said to pass the *mutation test* with data *T* if there exists *T* a subset of *D* such that:

- *P* works as intended on *T*
- For each mutant *m* in *M*(*P*) either
  - *m* fails to work as intended on *T*, or
  - *m* is equivalent to *P*.

If *P* passes the mutant test then we are sure that *P* is free of simple errors. But what of complex errors? To this end we have observed a *coupling effect* which states:

Test data *T* which causes all the non-equivalent mutants of *M*(*P*) to fail is so sensitive that all the non-equivalent mutants of *M**(*P*) must also fail on *T*.

The justification of the coupling effect parallels the probabalistic argument for justifying the single fault methods used to test circuits (021). However, we have no theory to make it a hard-fast principle. Basically, if several simple errors (detectable by *T*) combine to make a complex error then it is extremely unlikely the simple errors will cancel to allow the successful execution on *T* of the mutant containing the complex error. The goal of program mutation theory is then to validate, depending on *L* either deductively or experimentally, the coupling effect for language *L* by establishing the following *metatheorem of program mutation*:

- If *P* passes the mutation test then either
  - *P* is correct, or
  - *P* is radically incorrect.

It is not hard to see that if the metatheorem holds for language *L* and if *P* is a non-radically incorrect program, then it is impossible for *P* to pass the mutation test.

In (017) the mutation metatheorem has been formally shown to hold where *L* is certain classes of decision tables and the mutant operator involves the reformulation of conditions and applied actions. Currently, programs which manipulate data structures are under investigation.

For general purpose programming languages such as FORTRAN, the task is more difficult. There is a noticeable lack of empirical studies on programming errors to draw on in formulating a complete set of mutant operators - a necessary requirement for program mutation to be deductive. Here, *complete* means that all simple errors will be captured in $M(P)$. Hence, at least for now, in the case of general purpose languages we can consider program mutation to be an inductive tool for gaining confidence that the metatheorem of program mutation holds for a particular program P. A prototype system for a subset of FORTRAN will be overviewed below. Some initial experience with it, the effectiveness of the implemented mutant operators and substantiations of the coupling effect can be found in (013). A mutation system for nearly full ANSI FORTRAN has been designed and is about half written. Several experiments to finding 'good' mutant operators and for evaluating the effectiveness of mutation testing are under consideration.

PROGRAM MUTATION APPLIED

We do not intend that program mutation can be effectively used by the novice programmer. Rather (unlike previous software reliability methods) in program mutation we are making and exploiting the following assumption:

> *Experienced programmers* write programs which are either correct or are 'almost' correct.

That is, in the mutation terminology:

> If a program is *not* correct, then it is a 'mutant' - it differs from a correct program by simple well-understood errors.

There is empirical evidence which supports this natural premise (011,022).

In order that it be feasible to perform the mutation test, the size of $M(P)$ and T must be small. Our view is that the mutation system should be interactive. The user specifies the program P and initial test data T_1 to the system whence the mutant operators are applied to P, thereby generating the mutants of P. The mutants are then executed on T_1. A list of mutants which fail and which succeed on T_1 is produced. If all mutants give incorrect results then, by the coupling effect and the experienced programmer assumption, it is very likely that P is correct. On the other hand, if some mutants are correct on T_1, then the user must then examine the results of the mutation run to determine:

- P contains a non-radical error
- Because mutants which should have failed did not, T_1 is inadequate and must be augmented to T_2 and the system re-run
- Some mutants are equivalent to P. Currently, this must be done manually but there is hope that symbolic execution techniques can partially automate this task.

This cycle can be viewed as a series of interactive sessions in which the user defends P and the current test data against a system adversary which asks questions of the form:

Why does your test data not distinguish this simple error?

Such an adversary forces the user of program mutation into a careful and detailed review of his program and the design decisions made in constructing it. The issues which the user must address include:

- Which mutant operators should be applied to the program?
- Are the program and its mutants correct on the given test data?
- Is a given mutant equivalent to the program?

In this view we hold hope that even radical errors can be uncovered by users of program mutation.

OTHER APPLICATIONS OF PROGRAM MUTATION

Several approaches to aid in the design, implementation and debugging of large-scale software have recently emerged. Examples are restricted modularization, structured programming, and program verification. However helpful they may be to programmers and low-level managers, the effects of these techniques cannot be utilized throughout the project management hierarchy since they are qualitative rather than quantitative; managers should not be expected to understand code and/or sophisticated mathematics.

Besides being a tool for determing adequate test data, program mutation also provides the type of information that managers need to monitor software development and personnel performance. By using an automated program mutation system with report generation capability, during software development managers may extract information such as:

- Mutant failure percentages for each module indicating how close the software is to being acceptable

- Who is responsible for classifying which mutants as equivalent

- Which mutants have yet to fail.

This quantitative information can be used in several ways at different levels in the management hierarchy. Among these are:

- Re-assignment of personnel to work on modules where the mutant failure rate is low

- Pinpointing responsibility for modules which fail after having been deemed acceptable

- Forced justification of why certain equivalent mutants exist

- Monitoring PERT chart adherence

- Rewarding personnel who achieve high mutant failure percentages.

See (023) for a description of how program mutation can be integrated with the chief programmer management concept.

Government agencies and profit making industries are currently finding that purchasing software from specialized software vendors is more economical than in-house development. The contracts generally consist of the specifications for the software and a date on which the software and test data on which the software meets the specifications are to be delivered. Occasionally, some test data is given with the specifications. Two problems for the contractor are apparent in this scheme:

- At any time during the contract period the purchaser has no indication as to how 'close' the software is to being ready
- Upon delivery, although the software works correctly on the supplied test data, there is no way to measure the quality of the purchased software.

We see program mutation as a partial solution to the first problem and as a definite solution to the second.

Since program testing is the final stage of software development, a contractor can specify that the vendor indicates at what point testing commences. Assuming that the vendor is using a mutation system, the contractor can monitor the final stage of development by having the vendor periodically report mutant elimination percentages.

To evaluate the delivered software, one can specify in contracts that the test data of modules must eliminate a certain percentage of the mutants with respect to 'standard' mutant operators. Here there are many options. Software not passing this quality test may be rejected or there could be a substantial financial penalty to the vendor. In this case it is not essential that the vendor uses a mutation system, only that the contractor has one available to evaluate the final product. Also, note that the contractor is not concerned with equivalent mutants; rather, a simple test (which can be entirely computerized) dependent solely on the mutant operators is used. Currently, we have little information on which mutant operators should be employed in this test, however, experiments to answer this question are in progress.

THE PROTOTYPE FORTRAN MUTATION SYSTEM

A prototype mutation system for a large subset of FORTRAN has been implemented as an interactive system on the PDP-10. See (018) for a more detailed description than the following. We chose FORTRAN as the source language in our first implementation of a mutation system since there is a large body of existing programs on which we can experiment. However, the methodology is language-independent - mutation systems for other languages are in the design stage.

The programs considered are FORTRAN subroutines with the following data types and statement types:

- Integer constants and variable
- One and two dimensional arrays
- GOTO statements
- CONTINUE statements
- ASSIGNMENT statements with general arithmetic expressions
- RETRURN statements
- Logical IF statements with general relational and logical expressions
- DO loops with one level of embedding.

The mutant operators which the system can apply to a program fall into four categories:

- *Declaration mutations.* There are mutant operators to insert default array limits and to permute the limits of two-dimensional arrays.

- *Data reference mutations.* Data references are instances of constants, scalar variables, and references to one and two-dimensional arrays in the statements of the program. There are mutant operators to replace any data reference by any other reference in the program as well as an operator to replace constants by other constants not necessarily appearing in the program. Also, there is an operator to permute the index expressions of references to two-dimensional arrays.

- *Operator evaluation mutations.* There are mutant operators to replace occurrences of arithmetic operators in the program by all the other arithmetic operators. There are mutant operators to do likewise for relational and logical operators.

- *Control mutations.* There are mutant operators to replace the label portion of GOTO statements by each of the other statement labels appearing in the program. Also, there are mutant operators to see if all control paths of the program are traversed at least once, to force DO loops to end on continue statements, to force DO loops not to end on continue statements and to replace each statement of the program by a return statement.

As discussed above, these operators are designed to capture simple errors and to assess the adequacy of the given test data to distinguish them. For example, in making an array have dimension, one checks whether the test data is causing the array to be accessed other than as a scalar.

The user specifies to the system his program, test data, and the mutant operators he wishes to be applied. The system then generates and executes the mutants on the test data and produces a report indicating which mutants are correct and which fail on the given test data. Various profiles and other useful information are also reported. An example of the report produced by the system is given in the appendix. The determination of mutant correctness or failure is done in one of two ways:

- By direct comparison of the mutant output with the program's output
- By a user supplied algorithm which examines the output of the mutant.

In both cases the system asks the user whether or not the program is acceptable on the test data. However, determination of mutant failure is done by the system.

Upon examining the report, the user may re-run the system and augment his test data in an attempt to make the remaining mutants fail. He may also specify that additional mutant operators be applied to the program. The system produces another report of the same nature as the first for the user to examine. This cycle continues until the user is satisfied that his current test data adequately tests his program or until an error in the program is discovered.

The prototype FORTRAN system heavily uses the PDP-10 file system to record transient information such as mutant correctness status and the current test data between runs. In spite of the fact that the program terminates on the test data, some mutants may actually be non-terminating. To handle this the system records the program's execution time for each test case and deems that a mutant has failed due to being non-terminating if the mutant has not terminated within a factor of the program's execution time.

Experience has shown that a factor of 10 is reasonable.

SOME INITIAL EXPERIENCES WITH PROGRAM MUTATION

The results of using the prototype FORTRAN mutation system on three programs are now described. The first is Hoare's FIND program (014) which, given an integer array *A*, of dimension *N* and an array index *F*, rearranges *A* such that *A*(1)-*A*(*F*-1) are no greater than *A*(*F*) and *A*(*F*+1)-*A*(*N*) are no less than *A*(*F*). The second is the Knuth, Morris, and Pratt PAT program (015,016) which, given two arrays of integers, decides whether the first array occurs in the second. The third, SCAN, is the scanner used in the prototype system itself.

For all three programs, the testing strategy was the following. We first constructed what we believed would be good test data for the programs independently of the mutation system. The program and this initial test data were input to the system with all implemented mutant operators in effect. The results of these initial runs are summarised in Figure 1.

PROGRAM	EXECUTABLE STATEMENTS	NUMBER OF TEST CASES	NUMBER OF MUTANTS	PERCENTAGE OF INCORRECT MUTANTS
FIND	34	24	758	92•2
PAT	42	9	1178	77•2
SCAN	104	19	8838	89•1

Figure 1: Initial mutation run

We then made mutation runs with augmented test data until all mutants either failed on some test case or were determined equivalent. The final results are shown in Figure 2.

PROGRAM	NUMBER OF RUNS	NUMBER OF TEST CASES	% OF INCORRECT MUTANTS
FIND	8	49	98•1
PAT	9	35	98•7
SCAN	7	35	97•9

Figure 2: Initial mutation run

In comparing the mutant elimination percentages of Figure 1 to Figure 2, we can demonstrate one reason why program testing as an art has been held in such low esteem:

> With all three programs, even after hard thought, our initial test data failed to distinguish a large number of incorrect mutants.

Although the initial mutant elimination percentages in Figure 1 seem adequate, correlated with Figure 2, we see that the initial test data failed to distinguish 44 incorrect mutants of FIND, 253 incorrect mutants of PAT, and 778 incorrect mutants of SCAN. The final mutant report for SCAN appears in the appendix.

The reason FIND did so well initially is due to our choosing all permutations of 1-4 with *F* fixed at 3 for the initial data. Permutations are a reasonable test of programs like FIND but they fail to distinguish all mutants (see (013)). Higher dimensioned permutations do no better. We have run 1000 uniformly drawn random permutations of sizes

5 and 6 as test data for FIND and they failed to distinguish 39 incorrect mutants of FIND. The reason is permutations of *A* are legal FORTRAN indices of *A* and not until negative data is used do these mutants of FIND fail. This is analogous to mixing pointers and the values pointed at which is a common programming blunder. The insight gained is that test data for pointer type programs should be constructed so that values pointed at are not legal pointers. See (013) for other such insights for test data selection that have been gained from program mutation as well as for a more detailed discussion on the pitfalls of using random test data.

Figures 1 and 2 suggest that 2% might be a good estimate for the expected number of equivalent mutants that a program will have, at least for the mutant operators implemented in the prototype FORTRAN mutation system. If one accepts this estimate, then eliminating better than (say) 97% of all mutants without trying to determine equivalent mutants allows one to gain high confidence in test data adequacy.

Another observation is that the number of mutants of a program appears bounded by cn^2 where n is the number of statements in the program and $c<1$. This compares favourably with other methodologies for achieving reliable software which all seem to have inherent exponential growth factors. In fact, the unclever prototype FORTRAN mutation system took 90 minutes of CPU time on the PDP-10 KA-10 processor to run the 8838 mutants of the 104 statement scanner program. The KA-10 is 5 times slower than the IBM 370/158 and 30 times slower than the CDC 7600. Because our system is CPU bound, the 90 minutes of CPU time scales down directly to 18 and 3 minutes on these faster machines.

We are currently working on a 300-line auditing program taken from a production environment. We see no reason why any FORTRAN module cannot be tested on a mutation system within acceptable cost-effective CPU times.

CONCLUSIONS

Program mutation is an engineering approach to program testing where the goal is to establish that a program is either correct or is radically incorrect. The method is based on the coupling effect: simple mutations are sufficient to distinguish complex mutations. Initial experience has suggested the validity of the coupling effect.

The effectiveness of program mutation depends on two factors:

- Human judgement
- The implemented mutant operators.

In the former case we have suggested how mutation systems can be designed to aid users in meeting the goals of program mutation. In the latter case we are currently running experiments to evaluate the mutant operators implemented in the prototype system and to develop 'good' mutant operators for future mutation systems.

REFERENCES

001 RAMAMOORTHY C V, HO S F and CHEN W T
On the automated generation of program test data
IEEE Trans on Software Eng vol 2 no 1 pp 293-300 (Dec 1976)

002 HOWDEN W E
Methodology for the generation of program test data
IEEE Trans on Computers vol 24 no 3 pp 554-560 (May 1975)

003 HOWDEN W E
Reliability of the path analysis testing strategy
IEEE Trans on Software Eng vol 2 no 3 pp 208-214 (Sept 1976)

004 GOODENOUGH J B and GERHART S L
Towards a theory of test data selection
IEEE Trans on Software Eng vol 1 no 2 pp 156-173 (June 1975)

005 HUANG J C
An approach to program testing
Comp Surv vol 7 no 3 pp 113-128 (Sept 1975)

006 MILLER E F and MELTON R A
Automated generation of test case datasets
Proc 1st Intl Conf on *Reliable software*

007 CLARKE L
A system to generate test data and symbolically execute programs
IEEE Trans on Software Eng vol 2 no 3 pp 215-222 (Sept 1976)

008 KING J
Symbolic execution and program testing
CACM vol 19 no 7 pp 385-394 (July 1976)

009 LONDON R
The current state of proving programs correct
Proc ACM Nat Conf New York (1972)

010 HANTZER S and KING J
An introduction to proving the correctness of programs
Comp Surv vol 8 no 3 pp 331-353 (Sept 1976)

011 YOUNGS E A
Human errors in programming
Intl J of Man Machine Studies no 6 pp 361-376 (1974)

012 BOEHM B
Software design and structuring
In *Practical strategies for developing large software systems* Horowitz (ed) Addison-Wesley (1975)

013 DeMILLO R, LIPTON R and SAYWARD F
Hints on test data selection
Computer (April 1978)

014 HOARE C
Algorithm 65: FIND
CACM vol 4 no 1 p 321 (April 1961)

015 MORRIS J and PRATT V
A linear pattern matching algorithm
Tech rep 40 Comp Centre Univ of California

Berkeley (1970)

016 KNUTH D and PRATT V
Automata theory can be useful
Stanford Univ Tech rep (1971)

017 BUDD T and LIPTON R
Mutation analysis of decision table programs
Conf on *Information sciences and systems* John Hopkins Univ (1978)

018 BUDD T, DeMILLO R, LIPTON R and SAYWARD F
The design of a prototype mutation system for program testing
NCC (1978)

019 DeMILLO R, LIPTON R and PERLIS A
Social processes and proofs of theorems and programs
4th ACM Symp on *Principles of programming languages* (1977)

020 GOODENOUGH J
A survey of program testing issues
In *Research directions in software technology* P Wegner (ed)
MIT Press (1978)

021 CHANG H Y *et al*
Fault diagnosis of digital systems
Wiley-Interscience (1970)

022 BOEHM B W
Software engineering
IEEE Trans on Computers vol 25 no 12 pp 1226-1241 (Dec 1976)

023 DeMILLO R, LIPTON R and SAYWARD F
Program mutation as a tool for managing large scale software development
ASQC Tech Conf (1978)

APPENDIX

MUTATION STATUS REPORT   SCAN.F4   23-Oct-77

LISTING OF THE PROGRAM BEING MUTATED

```
 1          SUBROUTINE SCAN(RECORD,N,TYPE,KIND,ID,EOL,IVAL,COLUMN)
 2  C
 3  C       1. SCAN THE RECORD, STARTING AT LOCATION N AND RETURN THE NEXT
 4  C               IDENTIFIER, CONSTANT OR SYMBOL.
 5  C
 6  C       2. RECORD IS AN 80 WORD ARRAY, ASSUMED 1 CHAR TO A WORD.  N IS
 7  C               LOCATION TO START SCAN, ON RETURN N POINTS TO LAST LOC
 8  C               SCANNED + 1.
 9  C
10  C       3. RETURNS INFORMATION IN COMMON/SCANNER/
11  C               TYPE - TYPE OF OBJECT FOUND (SEE SCANER.PAR)
12  C               KIND - SUBCLASS OF TYPE
13  C               ID(2)- CHARACTER FORM OF IDENTIFIER FOUND, PADDED WITH
14  C                       BLANKS
15  C               EOL  - SET TRUE IF END OF LINE WAS FOUND BEFORE A CHAR
16  C                       WAS
17  C               IVAL - INTEGER VALUE FOUND
18  C               COLUMN - COLUMN IN WHICH LOCATED OBJECT BEGAN
19  C
20  C          INTERNAL VARIABLES USED
21  C               CH   - CURRENT CHARACTER
22  C               K    - LOOP COUNTER
23  C               IDB  - UNPACKED ID BUFFER
24  C               LOG  - SYMBOLIC LOGICAL OPERATORS
25  C               LTYPE- TYPE FOR EACH OF THE ABOVE
26  C               LKIND- KIND FOR EACH OF THE ABOVE
27  C               SYMC - CHARACTER CODE FOR SYMBOLS TO BE RECOGNIZED
28  C               STYPE- TYPE FOR EACH OF THE ABOVE
29  C               SKIND- KIND FOR EACH OF THE ABOVE
30  C
31  C       4. NONE
32  C
33  C       5. NONE
34  C
35  C       6. TRAPON TO TRAP INTEGER OVERFLOWS.
36  C
37  C       7.  ENCODE TO PACK IDENTIFIER
38  C           LINE 32- ASSUMES CHAR NUMBER = CHAR/2**29
39  C
40  C       8. ASSUMES 5 CHARACTERS TO A WORD
41  C
42  C       9. NONE
43  C
44  C       10. TIM BUDD, JULY 29, 1977
45  C
46  C       11.
47  C
48  C       12.
```

```
C
C
C       INCLUDE 'SCANER.PAR/NOLIST'
C ****  TESTED BY MANUALLY CHANGING PARAMETERS IN BODY OF CODE
C
C
C       PARAMETER SPACE-1H, A=1HA,Z=1HZ,C0=1H0,C9=1H9,PERIOD=1H.,
C     *    STAR=1H*
C ****  ALSO TESTED AS ABOVE
C
C
        INTEGER N,RECORD(12)
C
C       INCLUDE 'SCANER.COM/NOLIST'
C ****  TESTED BY MAKING INTO PARAMETERS
        INTEGER TYPE,KIND,ID(2),EOL,IVAL,COLUMN
C
C
C
        INTEGER CH,K,IDB(10),SYMC(8),
      *    STYPE(8),SKIND(8)
C
C ****  DATA STATEMENTS SIMULATED BY ASSIGNMENTS
C       DATA SYMC/'(',')',',','=','+','-','/',''''/,STYPE/ LPARN,RPARN
C     *    COMMA,BECOMS,2*ADDOP,MULOP,APOST /,SKIND/ 4*NOKIND,PLUS,
C     *    MINUS,DIVIDE,NOKIND/
        SYMC(1) = 40
        SYMC(2) = 41
        SYMC(3) = 44
        SYMC(4) = 61
        SYMC(5) = 43
        SYMC(6) = 45
        SYMC(7) = 47
        SYMC(8) = 96
        STYPE(1) = 4
        STYPE(2) = 9
        STYPE(3) = 8
        STYPE(4) = 10
        STYPE(5) = 5
        STYPE(6) = 5
        STYPE(7) = 6
        STYPE(8) = 15
        SKIND(1) = 1
        SKIND(2) = 1
        SKIND(3) = 1
        SKIND(4)   1
        SKIND(5) = 2
        SKIND(6)   3
        SKIND(7) = 5
        SKIND(8) = 1
C
C-----------------------------------------------------------------------
        TYPE = 1
        KIND = 1
      ID(1) = 32
      ID(2) = 32
      EOL = 0
      IVAL = 0
C
C  SKIP OVER LEADING SPACES
C
   10 IF (N.GE.13) GOTO 110
      IF (RECORD(N).NE.32) GOTO 20
      N = N + 1
      GOTO 10
C
C  NOW HAVE NON-NULL CHAR
C
   20 CH = RECORD(N)
      COLUMN = N
C
C  READ AN IDENTIFIER---------------------------------------------
C
      IF ((CH.LT.65 ).OR.(CH.GT.90 ))  GOTO 30
      K = 0
   22 IF (K.GE.10) GOTO 23
      K = K + 1
      IDB(K) = CH
   23 N = N + 1
      IF (N.LT.13) GOTO 24
      CH = 64
      GOTO 25
   24 CH = RECORD(N)
   25 IF (((CH.GE.64 ).AND.(CH.LE.90 )).OR.
     *     ((CH.GE.48 ).AND.(CH.LE.57 )))
     *    GOTO 22
C    PAD WITH BLANKS
   26 IF (K.GE.10) GOTO 28
      K = K + 1
      IDB(K) = 64
      GOTO 26
C  28 ENCODE(10,999,ID) IDB
   28 DO 29 K=1,5
      ID(1) = ID(1) * 10 + IDB(K)
   29 ID(2) = ID(2) * 10 + IDB(K+5)
      TYPE = 2
      GOTO 90
C
C  READ A NUMBER--------------------------------------------------
C
   30 IF ((CH.LT.48 ).OR.(CH.GT.57 )) GOTO 40
   32 IVAL = IVAL * 10 + (CH - 48 )
      N = N + 1
      IF (N.EQ.13) GOTO 34
      CH = RECORD(N)
      IF ((CH.GE.48 ).AND.(CH.LE.57 )) GOTO 32
   34 TYPE = 3
      GOTO 90
```

```
C
C  READ A PERIOD (OR A LOGICAL EXPR)--------------------------------
C
   40   IF (CH.NE.46    ) GOTO 50
        TYPE = 16
        GOTO 90
C
C  READ A STAR-------------------------------------------------------
C
   50   IF (CH.NE.42  ) GOTO 60
        TYPE = 6
        KIND = 4
        N = N + 1
        IF (N.EQ.13) GOTO 90
   .    CH = RECORD(N)
        IF (CH.NE.42  ) GOTO 90
        N = N + 1
        TYPE = 7
        KIND = 1
        GOTO 90
C
C  READ ( ) , = + - / ' ---------------------------------------------
C
   60   DO 62 K=1,8
   62   IF (CH.EQ.SYMC(K)) GOTO 64
C  FALL THROUGH LOOP => SYMBOL ERROR
        GOTO 120
   64   TYPE = STYPE(K)
        KIND = SKIND(K)
        N = N + 1
C       GOTO 90
C
C       CORRECT EXIT POINT
C
   90   RETURN
C
C
C       ERRORS-
C  ERROR 110, END OF LINE
  110   EOL = 1
        COLUMN = 13
        GOTO 90
C
C  ERROR 120, INVALID SEQUENCE OF SYMBOLS
  120   GOTO 90
        END
```

```
CLASSIFICATION OF THE PROGRAM'S FORMAL PARAMETERS

STRICTLY OUTPUT PARAMETERS
TYPE       KIND       ID       EOL       IVAL       COLUMN

INPUT AND OUTPUT PARAMETERS
N

READ ONLY INPUT PARAMETERS
RECORD



THE METHOD OF DETERMINING MUTANT CORRECTNESS IS

BY COMPARISON TO THE PROGRAM
```

```
THE PIMS RUN TITLE

     BEFORE THIS RUN THERE WERE   6 PIMS RUNS ON THIS PROGRAM

       8838 MUTANTS WERE CREATED DURING THOSE RUNS

          0 NEW MUTANTS WERE CREATED DURING THIS RUN

     FOR A GRAND TOTAL OF   8838 MUTANTS
```

```
MUTANT'S STATUS BEFORE THIS RUN

   A TOTAL OF       29 TEST CASES
   A TOTAL OF     8838 MUTANTS
   OF THESE        206 ARE STILL ALIVE

   THERE ARE       104 PROGRAM STATEMENTS,
   GIVING        84.98 MUTANTS PER STATEMENT

   MUTANT PROFILE

   ARRAY LIMIT DEFAULT INSERTION                6              0
   SCALAR VARIABLE REPLACEMENT                539              2
   SCALAR VAR FOR CONSTANT REPLMT             872             34
   CONSTANT FOR SCALAR VAR REPLMT            1320              0
   COMPARABLE ARRAY NAME REPLMT               210              0
   CONST FOR ARRAY REF REPLACEMENT            360              0
   SCALAR VAR FOR ARR REF REPLMT              336              0
   ARRAY REF FOR CONST REPLACEMNT            2368            111
   ARR REF FOR SCALAR VAR REPLMT             2016             12
   ARITHMETIC OPERATOR REPLACEMNT              64              0
   RELATIONAL OPERATOR REPLACEMNT             105             15
   LOGICAL CONNECTOR REPLACEMENT                6              0
   GOTO LABEL REPLACEMENT                     427             21
   PATH ANALYSIS                              104              0
   CONTINUE STATEMENT INSERTION                 2              2
   RETURN STATEMENT INSERTION                 103              9

   THE PERCENTAGE OF ELIMINATED MUTANTS IS   97.67

   THE ELIMINATION PROFILE FOR ALL MUTANTS IS

   TYPE OF ELIMINATION           NUMBER OF ELIMINATED MUTANTS
   TIMED-OUT                                  310
   REFERENCED AN UNDEFINED VARIABLE          2034
   SUBSCRIPT RANGE ERROR                     1071
   DIVIDED BY ZERO                              0
   ARITHMETIC OVERFLOW OR UNDERFLOW            63
   WROTE A READ ONLY VARIABLE                  58
   EXECUTED A TRAP STATEMENT                  104
   PRODUCED WRONG ANSWERS                    4992
```

```
MUTANT STATUS AFTER THIS RUN

   A TOTAL OF      35 TEST CASES
   A TOTAL OF    8838 MUTANTS
   OF THESE       190 ARE STILL ALIVE

   THERE ARE      104 PROGRAM STATEMENTS,
   GIVING       84.98 MUTANTS PER STATEMENT

   MUTANT PROFILE

   ARRAY LIMIT DEFAULT INSERTION           6                 0
   SCALAR VARIABLE REPLACEMENT           539                 2
   SCALAR VAR FOR CONSTANT REPLMT        872                33
   CONSTANT FOR SCALAR VAR REPLMT       1320                 0
   COMPARABLE ARRAY NAME REPLMT          210                 0
   CONST FOR ARRAY REF REPLACEMNT        360                 0
   SCALAR VAR FOR ARR REF REPLMT         336                 0
   ARRAY REF FOR CONST REPLACEMNT       2368               111
   ARR REF FOR SCALAR VAR REPLMT        2016                12
   ARITHMETIC OPERATOR REPLACEMNT         64                 0
   RELATIONAL OPERATOR REPLACEMNT        105                 6
   LOGICAL CONNECTOR REPLACEMENT           6                 0
   GOTO LABEL REPLACEMENT                427                15
   PATH ANALYSIS                         104                 0
   CONTINUE STATEMENT INSERTION            2                 2
   RETURN STATEMENT INSERTION            103                 9

   THE PERCENTAGE OF ELIMINATED MUTANTS IS 97.85

   THE ELIMINATION PROFILE FOR ALL MUTANTS IS

   TYPE OF ELIMINATION           NUMBER OF ELIMINATED MUTANTS
   TIMED-OUT                             310
   REFERENCED AN UNDEFINED VARIABLE     2035
   SUBSCRIPT RANGE ERROR                1073
   DIVIDED BY ZERO                         0
   ARITHMETIC OVERFLOW OR UNDERFLOW       63
   WROTE A READ ONLY VARIABLE             58
   EXECUTED A TRAP STATEMENT             104
   PRODUCED WRONG ANSWERS               5005
```

Mutation Analysis

Timothy A, Budd, Richard J. Lipton,
Richard A. DeMillo, and Frederick G. Sayward

Research Report #155

April 1979

Supported in part by the Office of Naval Research under Grant N00014-75-C-0752, the Army Research Office under Grant DAAG 29-78-G-0121, and the National Science Foundation under Grant MCS-780-7291.

Mutation Analysis

Timothy A. Budd
Richard J. Lipton

Computer Science Division
University of California,
Berkeley, CA 94720

Richard A. DeMillo

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

Frederick G. Sayward

Computer Science Department
Yale University
New Haven, CT 06520

ABSTRACT

A New type of software test is introduced, called ***mutation analysis***. A method for applying mutation analysis is described, and the results of several experiments to determine its effectiveness are given. Finally it is shown how mutation analysis can subsume or augment many of the more traditional program testing techniques.

1. Introduction

Traditionally, program testing has been an ad hoc technique done by all programmers: the programmer creates test data which he intuitively feels captures the salient features of the program, observes the program in execution on the data, and if the program works on the data (i.e., passes his test) he then concludes the program is correct. Just as most programmers have tested programs in this manner, most programmers have also deemed to be correct programs which were indeed incorrect.

Modern testing techniques attempt to augment the programmer's intuition by providing quantitative information on how well a program is being tested by the given test data. Certainly the sheer number of test cases is not sufficient to significantly increase our confidence in the correct functioning of a program. If all the test cases exercise the program in roughly the same way then nothing has been gained over a smaller number of executions. The key idea of modern testing techniques is to exercise the program under a variety of ***different*** circumstances, thereby giving the programmer a greater confidence in the correct functioning of the software component.

Several popular testing techniques use an idea called *covering measure*. Examples of covering measures are: the number of statements executed, number of branch outcomes taken, or the number of paths traversed by the test cases. Test data with high coverage measures then exercise the program more throughly (according the the criterion) then ones with low measure.

In this paper we will discuss a new type of testing method, program mutation, which differs significantly from those previously mentioned. Numerous theoretical and empirical studies [1,2,4,5] indicate that data satisfying this test criterion often perform significantly better in discovering errors and validating programs then data satisfying .other criterion. In many cases, the new test will actually subsume the goals which have been earlier investigated.

2. Description of the Method

Mutation analysis starts with one important assumption which is surprisingly not often recognized:

> *experienced programmers* write programs which are either correct or are *almost* correct.

(one manifestation of this is the common programmers joke that the code is always "90%" finished.)

The mutation method can be explained as follows: Given a program P which performs correctly on some test data T, subject the program to a series of *mutant operators*, thereby producing mutant programs which differ from P in very simple ways. For example, if

```
I = I + 1
```

is a statement in P, then

```
I = I - 1
I = I + 2
I = J + 1
```

are all simple changes which lead to three mutants of P.

The mutant programs are then executed on T. If each mutant program produces an answer which differs from the original on at least one test case, then the mutation test for P is passed. If, as is more likely, some of the mutants produce the same answers as the original program on all the test cases submitted, then either

1) the mutant programs are equivalent to P
2) the test data T is inadequate for passing the mutation test and must be augmented.

In this case the original program must then be examined with the list of live mutants in order to derive test data on which some or all of the remaining mutants will fail. The degree of testing is then measured in terms of the number (or percentage) of mutants which have been eliminated by the test data.

As an intuitive aid one can think of the mutation system as proposing alternatives to the given program and asking the programmer for reasons, in the form of test cases, as to why the alteratives are not just as effective as the original program in solving the given task. This then insures that the program is correct relative to small perturbations in its structure.

At first glance, however, it would appear that a program and test data which passed this test might still contain some complex errors which are not explicitly mutations of P. To this end there is a *coupling effect* which states:

> *test data on which all simple mutants fail is so sensitive to changes in the program that it is highly likely that all complex mutants must also fail.*

By complex mutant we mean the transformation which takes the original incorrect program into the presumed correct version. Since therefore any such correct program will be differentiated from P, if P truly executed correctly on T there can be no complex mutants, hence P is correct.

Several experiments substantiating the coupling effect have been conducted[1,4]. Some of these will be described in the following sections. The DAVE group [15,16] at the university of Colorado have also observed that the ability to detect simple errors is often useful in insuring against quite complex errors. The types of simple errors considered in mutation analysis is, however, much more extensive then that considered by DAVE.

Constant Replacement (± 1)
Scalar for Constant Replacement
Source Constant Replacement
Array Reference for Constant Replacement
Scalar Variable Replacement
Constant for Scalar Replacement
Array Reference for Scalar Replacement
Comparable Array Name Replacement
Constant for Array Reference Replacement
Scalar for Array Reference Replacement
Array Reference for Array Reference Replacement
Arithmetic Operator Replacement
Relational Operator Replacement
Logical Connector Replacement
Unary Operator Removal
Unary Operator Replacement
Unary Operator Insertion
Statement Analysis
Statement Deletion
Return Statement Replacement
Goto Statement Replacement
Do Statement Replacement

figure 1

3. The System

A system has been constructed which performs mutation analysis on sets of subroutines written in ANSI FORTRAN. The system is interactive and iterative, so that the user presents the system with a program and an initial test set. After constructing and executing each mutant serially the system responds with summaries and reports on the number and type of mutants which remain (i.e. which produced the same result as the original program.) The user can then augment the test data set and reexecute the remaining mutants on the new test cases. This process can continue until the desired level of testing is attained.

The mutant operators used in the current system are shown in figure 1. The names are fairly self explanatory: for example, the three mutations given in section 2 are produced by arithmetic operator replacement, constant replacement, and scalar variable replacement, respectively.

Various versions of the mutation system have been in operation for about two years [2], and in that period numerous experiments have been qonducted investigating the coupling effect and the utility of the tool for program development and testing [5]. The next section details some experiments performed which substantiate the coupling effect.

4. The Coupling Effect

We have already reported on an experiment [4] involving Hoare's FIND program [9] that supplied empirical evidence for the coupling effect. The experiment went as follows:

(1) We derived a test data set T of 49 cases to pass the mutation test. (The large size of T was due to our inexperience.)

(2) For efficiency reasons, we reduced T heuristically to a test data set T' consisting of seven cases on which FIND also passed the mutant test.

(3) Random k-order mutants of FIND, k>1, were generated. (A k-order mutant comes from k applications of mutant operators on the program P.)

(4) The k-order mutants of FIND were then executed on T'.

The coupling effect says that the non-equivalent k-order mutants of FIND will fail on T'. Note that step 2 biases the experiment against the coupling effect since it removes the man-machine orientation of our approach to testing. We would have been quite happy to find a counterexample to the coupling effect for the mutation system, since it would have allowed us to improve the set of mutant operators. The results of the experiment, though, gave evidence that we had chosen a well coupled set of mutant operators for the pilot system:

K	*Number of k- order mutants*	*Number successful on T'*
2	21100	19
>2	1500	0

The 19 successful mutants were shown to be equivalent to FIND. We concentrated on the k=2 case since, intuitively, the more one mutates FIND the more likely one is to get a program that violates the competent programmer assumption.

The major criticism of the experiment concerns step 3. Since the first-order mutants that compose the k-order mutants are independently drawn, the resulting k-order mutant is likely to be very unstable and subject to quick failure, in contrast to the more desirable case where the k-order mutant contains subtly related changes that correspond to the subtle errors programmers find so hard to detect.

The current experiment on the coupling effect omits step 2 above and make the following important change to step 3:

(3) Randomly generate *correlated* k-order mutants of the program. By correlated we mean that each of the k applications of mutant operators will in some way be related to all of the others -- they could for instance effect the same statement of P, or the same variable name, or the same statement label, or the same constant.

Once again, if P passes the mutant test with test data T, the coupling effect says that the correlated k-order mutants of P will fail on T.

For this experiment three programs are being used: FIND, STKSIM and TRIANG. STKSIM is a program that maintains a stack and allows the standard operations of clear, push, pop, and top. TRIANG is a program that, given the lengths of the three legs of a triangle, categorizes the input as not representing a triangle or as representing a scalene, isoceles or equilateral triangle [3]. The following is a summary of the results of the experiment so far:

PROGRAM	K=2		K=3		K=4	
	number	successes	number	successes	number	successes
FIND	3000	2	3000	0	3000	0
STKSIM	3000	3	3000	0	3000	0
TRIANG	3000	1	3000	1	3000	0

In all cases, the successful correlated k-order mutants have been shown to be equivalent to the original program.

We have yet to find a non-trivial counterexample to the coupling effect for our FORTRAN systems. The one successful 3-order mutant of TRIANG deserves closer examination; indeed, we initially felt that it was a non-equivalent mutant. The mutant is

```
      SUBROUTINE TRIANG(I,J,K,MATCH)
C
      INTEGER I,J,K,MATCH
C
C      MATCH IS OUTPUT FROM THE ROUTINE
C        IF MATCH = 1 THE TRIANGLE IS SCALENE
C        IF MATCH = 2 THE TRIANGLE IS ISOSCELES
C        IF MATCH = 3 THE TRIANGLE IS EQUILATERAL
C        IF MATCH = 4 IT IS NOT A TRIANGLE
C
      IF (I .LE. 0 .OR. J .LE. 0 .OR. K .LE. 0) GOTO 500
      MATCH = 0
      IF (I .NE. J) GOTO 10
      MATCH = MATCH + 1
   10 IF (I .NE. K) GOTO 20
      MATCH = MATCH + 2
```

MO_1:change statement to MATCH = MATCH + K

```
   20 IF (J .NE. K) GOTO 30
      MATCH = MATCH + 3
   30 IF (MATCH .NE. 0) GOTO 100
      IF (I+J .LE. K) GOTO 500
      IF (J+K .LE. I) GOTO 500
      IF (I+K .LE. J) GOTO 500
      MATCH = 1
      RETURN
  100 IF (MATCH .NE. 1) GOTO 200
      IF (I+J .LE. K) GOTO 500
  110 MATCH = 2
      RETURN
  200 IF (MATCH .NE. 2) GOTO 300
```

MO_2: change statement to IF (MATCH .NE. K)

```
      IF (I+K .LE. J) GOTO 500
      GOTO 110
  300 IF (MATCH .NE. 3) GOTO 400
      IF (J+K .LE. I) GOTO 500
```

MO_3: change statement to IF (J+J .LE. I)

```
      GOTO 110
  400 MATCH = 3
      RETURN
  500 MATCH = 4
      RETURN
      END
```

Note that the correlation is with respect to the variable K. The mutant operators MO_1 and MO_2 produce incorrect mutants while MO_3 produces a mutant equivalent to TRIANG. Yet the 3-order correlated mutant is equivalent to TRIANG.

This makes a beautiful illustration of the part of the programming process that program mutation is trying to exploit. Using the constant 2 in the first two mutated statements is an arbitrary but coupled decision. Indeed, you can replace both instances of 2 by any positive constant (or any variable whose value doesn't change between the execution of the two statements) and you get an equivalent program -- replace only one instance and you get an incorrect program. In a sense, the constant 2 in those statements is what would be called in the terminology of formal logic a "bound variable.'

5. An Analysis of How Mutation Works

In this section we will go through a detailed analysis concerning how and why mutation analysis can be expected to uncover errors under a wide variety of situations.

5.1. Trivial Errors

If one of the mutants considered is indeed the correct program then of course the error will be discovered when an attempt is made to eliminate that particular mutant. Alternatively if the errors in the original program act in a reasonably independent manner and each error is individually captured by a single mutation then the errors will almost certainly be detected.

Given the vast folklore about large systems failing for extremely trivial reasons, the ability to detect such simple errors in indeed a good starting place. However many errors do not correspond exactly to the generated mutations, and multiple errors may interact in subtle fashions. This being the case we must demonstrate that mutation analysis possess *many more powerful capabilities.*

5.2. Statement Analysis

Many programming errors manifest themselves by sections of code being "dead", that is unexecutable, when they shouldn't be. Also many bugs are of such a serious nature that *any* data which executes the particular statement in error will cause the program to give incorrect results. These errors may persist for weeks or even years if the error occurs in a rarely executed section of code.

Accordingly a reasonable first goal for a set of test cases is that every statement in the program is to be executed at least once [12].

Various authors have presented methods to achieve this goal. Usually these methods involve the insertion of counters into the straight line segments of code. When all counters register non-zero values every statement in the program has been executed at least once.

In Mutation analysis we take a different approach to the same objective. If a statement is never executed then obviously any change we produce in it will not cause the altered program to produce test answers differing from the original. However as a means of directing the programmers attention to these errors in a more direct and unambiguous fashion a simpler approach is taken. Among the mutations generated are ones which replace the first statement of every basic block in turn with a call on a special routine which aborts whenever it is executed. Obviously these mutations are extremely unstable, since any data which executes the replaced statement will cause the mutant to produce an incorrect result, and hence to be eliminated. The reverse, however, is also true. That is, if any of these mutants survive, then the statement which the mutation altered has never been executed. Hence an accounting of the survival of this class of mutations gives important information about which sections of code have and have not been executed.

Mutation Analysis goes even one step further. Some authors have assumed that not executing a statement is equivalent to deleting it [8]. This is certainly not true. A statement can be executed but still not serve any *useful* purpose. In order to investigate this another class of mutants generated replaces every statement with a CONTINUE statement (a convenient FORTRAN NO-OP.) The survival or elimination of these mutations gives more information then merely whether the statement is executed or not, it indicates whether or not the statement is performing anything *useful*. If a statement can be replaced by a NO-OP with no effect then at best it indicates a waste of machine time and at worst it is probably indicative of much more serious errors.

Merely being able to execute every statement in the program is no guarantee that the code is correct [7,10]. Problems such as coincidental correctness or predicate errors may pass undetected even if the statement in error is executed repeatedly. In subsequent sections we will show how mutation analysis deals with these problems.

5.3. Branch Analysis

Some authors have pointed out [12] that an improvement over statement analysis can be achieved by insuring that every flowchart branch is executed at least once. For example the following program segment

```
A;
IF (expression)
 THEN B;
C;
```

has the flowchart shown in figure 2.

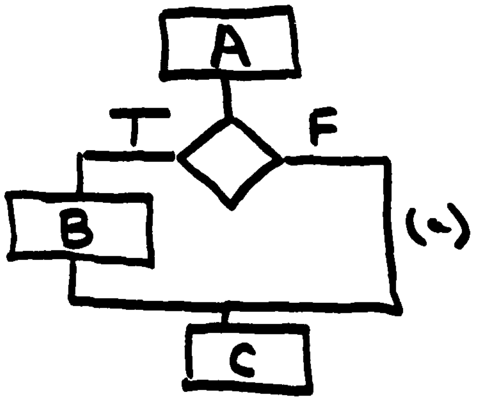


figure 2

All three statements A,B and C can be executed by a single test case. It is not true, however, that in this case all branches have been executed. For example in this case the empty else clause branch (a) has been ignored.

We can state the requirement that every branch be taken in an equivalent manner by requiring that every predicate expression must evaluate both TRUE and FALSE. It is this formalization which is used in mutation analysis.

Among the mutants generated are ones which replace each relational expression and each logical expression by the logical constants TRUE and FALSE. Of course, like the statement analysis mutations these are very unstable and easily eliminated by almost any data. But if they survive they point directly and unambiguously to a weakness in the test data which might shield a potential error.

By mutating each relation or logical expression independently we actually achieve a stronger goal than that achieved by usual branch analysis.

Consider the compound predicate

IF (A ≤ B AND C ≤ D) THEN

The usual branch analysis method would only require two test cases to test this predicate. If the test points were (A<B,C<D) and (A<B,C>D) this would have the effect of only testing the second clause, and not the first. This is because branch analysis fails to take into account the "hidden paths" [4], implicit in compound predicates. (see figure 3).

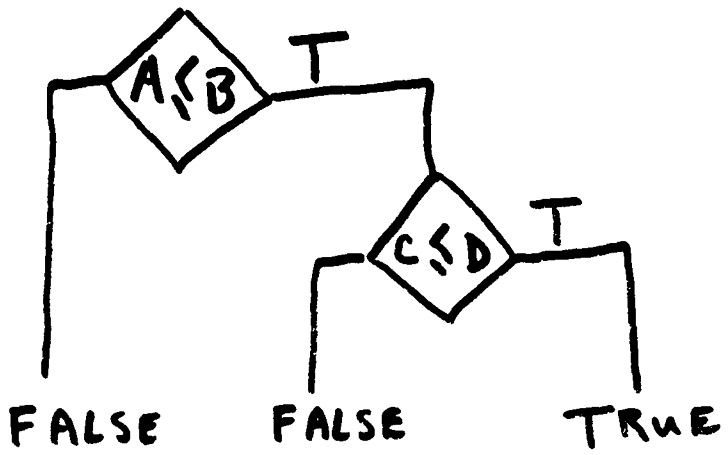


figure 3

In testing all the "hidden paths" mutation analysis would require at least three points to test this predicate,. The three points correspond to the branches (A > B,C > D), (A ≤ B,C > D), and (A ≤ B, C ≤ D).

As an example of this consider the program shown in figure 4, adapted from [6]. The program, which was also studied in [17], is intended to derive the number of days between two given days in a given year. The If statement which determines whether a year is a leap year or not is, however, incorrect in this version. Notice that if a year is divisible by 400 (year REM 400 = 0) it is necessarily divisible by 100 (year REM 100 = 0). Hence the logical expression formed by the conjunction of these two terms is equivalent to just the second term alone. Alternatively, the expression year REM 100 = 0 can be replaced by the logical constant TRUE and the resulting mutant will be equivalent to the original. Since this is obviously not what the programmer had in mind the error is discovered.

```
PROCEDURE calendar (INTEGER VALUE day1, month1, day2, month2, year);
 BEGIN
  INTEGER days;
  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
      INTEGER ARRAY daysin (1 .. 12);
      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;
      daysin(11):= 30;  daysin(12):= 31;
      IF ((year REM 4) = 0) OR
         ((year REM 100) = 0 AND (year REM 400) = 0)
        THEN daysin(2) := 28
        ELSE daysin(2) := 29;
      COMMENT set daysin(2) according to whether or not year
          is a leap year ;
      days := day2 + (daysin(month1) - day1);
      COMMENT this gives (the correct number of days - days
              in complete intervening months);
      FOR i := month1 + 1 UNTIL month2 -1 DO
        days := daysin(i) + days;
      COMMENT add in the days in complete intervening months;
    END;
   WRITE(days)
 END;
```

figure 4

5.4. DATA FLOW ANALYSIS

During execution a program may access a variable in one of three ways. A variable is *defined* if the result of a statement is to assign a value to the variable. A variable is *referenced* if the statement required the value of the variable to be accessed. Finally a variable is *undefined* if the semantics of the language do not explicitly give any other value to the variable. Examples of the latter are the values of local variables on invocation or procedure return, or DO loop indices in FORTRAN on normal do loop termination.

Fosdick and Osterweil [16] have defined three types of data flow anomalies which are often indicative of program errors. These anomalies are consecutive accesses to a variable of the forms:

1) undefined and then referenced

2) defined and then undefined
3) defined and then defined again

The first is almost always indicative of an error, even if it occurs only on a single path between the place where the variable becomes undefined and the reference place. The second and third, however, may not be indications of errors unless they occur on every path between the two statements.

Although the first type of anomaly is not attacked by mutations *per se* it is attacked by the *mutation system*, which is a large interpretive system for automatically generating and testing mutants. Whenever the value of a variable becomes undefined it is set to a unique constant *undefined*. Before every variable reference a check is performed to see if the variable has this value. If the variable does the error is reported to the user, who can take corrective action.

The second and third types of anomalies are attacked more directly. If a variable is defined and not used then usually the statement can be eliminated with no obvious change (by the CONTINUE insertion mutations described in the last section.) This may not be the case if, for example, in the course of defining the variable a function with side effects is invoked. In this case the definition can likely be mutated in any number of different ways which, while preserving the side effect, obviously result in the variable being given different values. An attempt to remove these mutations will almost certainly result in the anomaly being discovered.

5.5. Predicate Testing

Howden [10] has defined two broad categories of program errors under the names *domain error* and *computation errors*. The notions are not precise and it is difficult with many errors to decide which category they belong in. Informally, however, a domain error occurs when a specific input follows the wrong path due to an error in a control statement. A computation error occurs when an input follows the correct path but because of an error in computation statements the wrong function is computed for one or more of the output variables.

After Howden's study was published, some researchers examined the question of whether certain testing methodologies might reliably uncover errors in these or other classification schemes. One method proposed specifically directed to domain errors was the *domain strategy* of White, Cohen and Chandrasekaran [19].

The reader is referred to the references for a more complete presentation of the technical restrictions and applications of their method, but we can here give an informal description of how it works.

If a program contains N input variables (including parameters, array elements and I/O variables) then a predicate can be described by a surface in the N dimensional input space. Often the predicate is linear, in which case the surface is an N dimensional hyperplane. Let us consider a simple two dimensional case where we have input variables I and J and the predicate in question is

$$I+2J \leq -3$$

The Domain strategy would tell us that in order to test his predicate we need three test points, two on the line I+2J=-3 and one a small

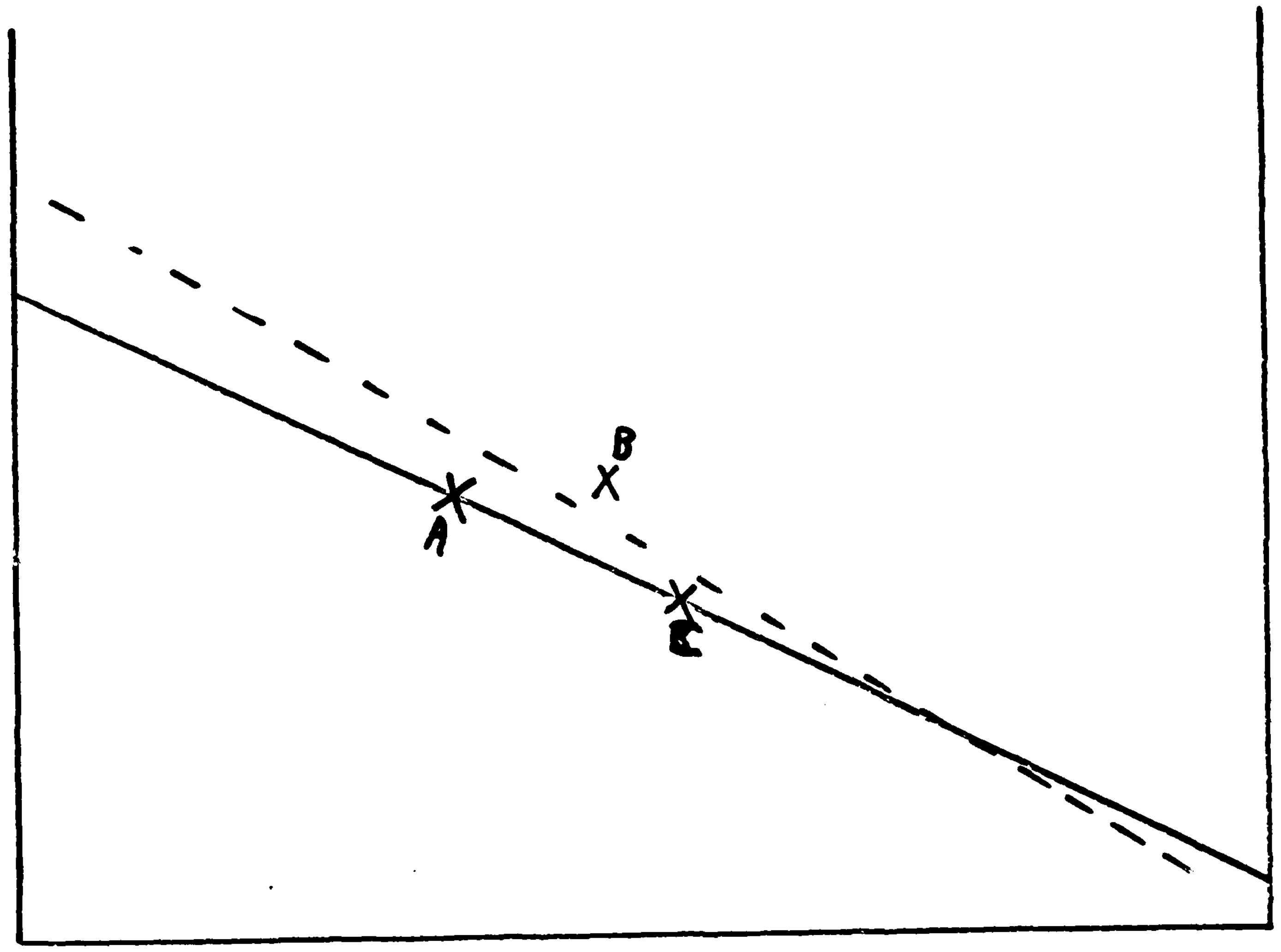


figure 5

distence ε from the line. (see figure 5.)

Assuming a correct outcome from these tests what have we discovered? We know the line of the predicate must cut the sections of the triangle AB and BC. Since ε is quite small the chances of the predicate being one of these alternatives is also small. Hence, although we don't have complete confidence that the predicate is correct, we do have a much larger degree of confidence then we could otherwise have attained.

To see how mutation analysis deals with the same problem we first observe that it really is not necessary to have both A and C be on the predicate line. If A is on the line and B and C are on *opposite sides* of the line the same result follows. We now described how mutations cause these three points to be generated.

As an intuitive aid one can think of mutation analysis as posing certain alternatives to the predicate in question, and requiring the tester to supply reasons, in the form of test data, why the alternative predicated would not be used just as well in place of the original. These alternatives are constructed in various ways.

A number of the alternatives are generated by changing relational operators. Changing an inequality operator to a strict inequality operator, or vice versa, generates a mutant which can only be eliminated by a test point which exactly satisfies the predicate. For example changing $I+2J\leq-3$ to $I+2J<-3$ requires the tester to exhibit a point for which $I+2J=-3$, hence which satisfies the first predicate but not the second.

A second class of alternatives involves the introduction of the unary operator "twiddle" (denoted ++ or --). Twiddle is an example of a non FORTRAN language construction used to facilitate the mutation process. For an integer expression a, ++a has the meaning a+1. For real expressions ++a means a + 1/100. --a has a similar meaning involving subtractions.

Graphically, the effect of introducing twiddle is to move the proposed constraint a small distance parallel to the original line (see figure 6). In order to eliminate these mutants a data point must be found which satisfies one constraint but not the other, hence is very close to the original constraint line.

Finally a third class of alternatives are constructed by changing each data reference into all other syntactically correct data references, and each operator into all other syntactically correct operators. The effects of these are related to the phenomenon of spoilers, which are described in section 5.8.

The total effect caused by so many alternatives is to increase the number of data points necessary for their elimination, hence by a process similar to that of Cohen et al[19] to increase our confidence that the predicate is indeed correct.

In order to more graphically illustrate the construction of these alternatives and demonstrate their utility we will go through a small example. The program in figure 7 was taken from [19]. No specifications were given, but the program can be compared against a presumably "correct" version. It was chosen here because it only involves two input variables, hence the alternatives can be easily illustrated in a graphical manner.

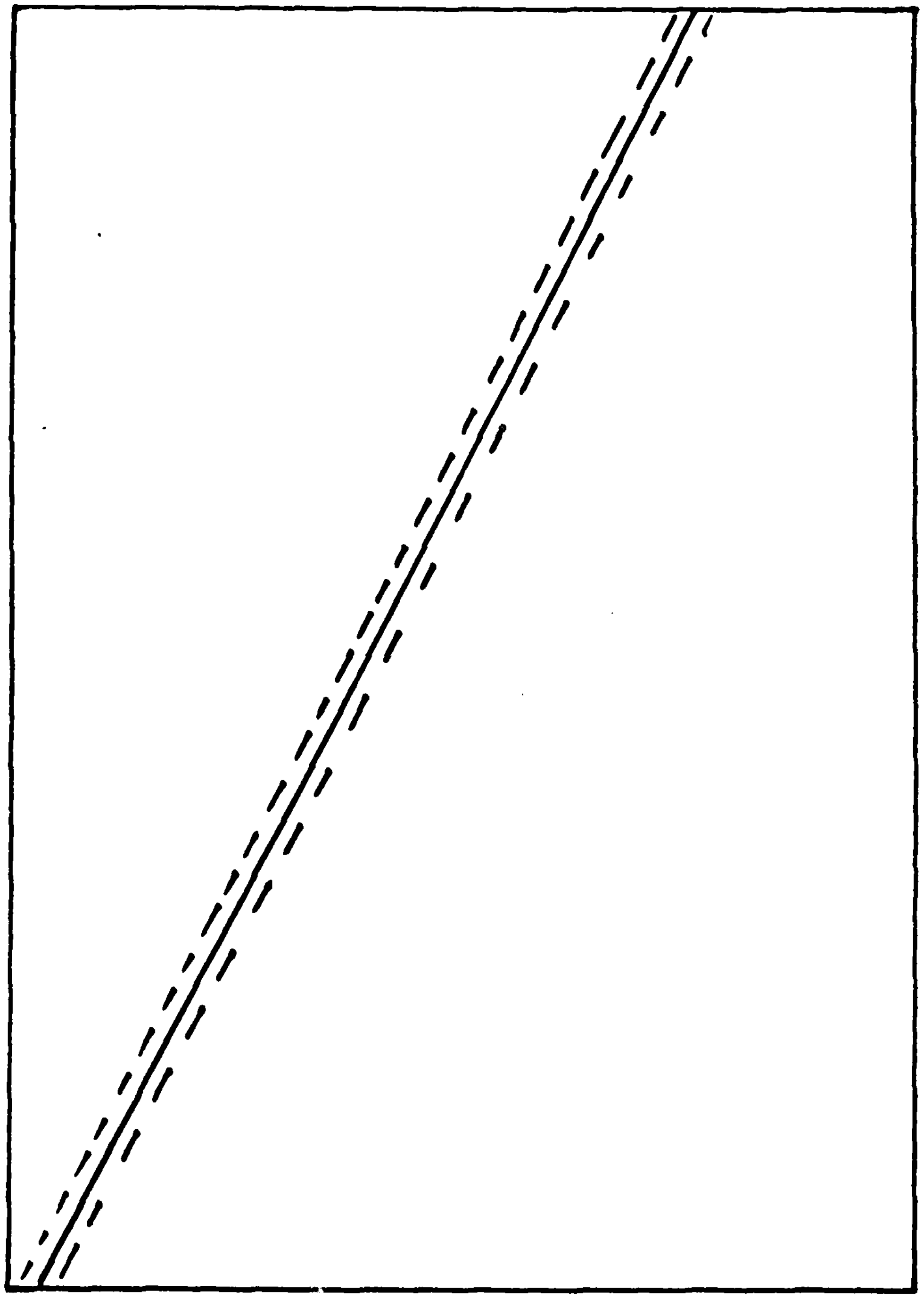


figure 6

```
READ I,J;

IF I ≦ J + 1
  THEN K = I + J - 1;
  ELSE K = 2*I + 1;

IF K ≧ I + 1
  THEN L = I + 1;
  ELSE L = J - 1;

IF I = 5
  THEN M = 2*L + K;
  ELSE M = L + 2*K -1;

WRITE M;
```

figure 7

```
1. IF (I ≦ J + 0)
2. IF (I ≦ J + 2)
3. IF (I ≦ J + I)
4. IF (I ≦ J + J)
5. IF (1 ≦ J + 1)
6. IF (2 ≦ J + 1)
7. IF (5 ≦ J + 1)
8. IF (I ≦ 1 + 1)
9. IF (I ≦ 2 + 1)
10. IF (I ≦ 5 + 1)
11. IF(I ≦  J + 5)
12. IF(-I ≦ J + 1)
13. IF( ++I ≦ J + 1)
14. IF(--I ≦ J + 1)
15. IF(I ≦ -J + 1)
16. IF(I ≦  ++J + 1)
17. IF(I ≦ --J + 1)
18. IF(I ≦ -(J + 1))
19. IF(I ≦  ++(J + 1))
20. IF(I ≦ --(J + 1))
21. IF(.NOT. I≦ J + 1)
22. IF(I ≦ J - 1)
23. IF(I ≦ MOD(J,1))
24. IF(I ≦ J/1)
25. IF(I ≦ J*1)
26. IF(I ≦ J**1)
27. IF(I ≦ J)
28. IF(I ≦ 1)
29. IF(I < J + 1)
30. IF(I = J + 1)
31. IF(I ¯ J + 1)
32. IF(I > J + 1)
33. IF (I ≧ J + 1)
```

figure 8

As you can see the program has three predicates: $I \leq J+1$, $K \geq I+1$ and $I=5$. We will illustrate only the effects of changing the first.

Figure 8 gives a listing of all the alternatives tried for the predicate $I \leq J+1$. Some of the choices are redundant, for example $++I \leq J+1$ and $I \leq --J + 1$. This is because the mutations are generated in an entirely mechanical way. It is our feeling that the processing time lost because of redundant mutations is much less then the time which would be required to eliminate them by preprocessing the alternatives.

The alternative predicates so introduced are illustrated in figure 9. The original predicate is the heavy line running from the lower left to the upper right.

. In the paper from which the example program was taken the authors hypothesize that the program contains the following four errors.

1) The predicate $K \geq I+1$ should be $K \geq I+2$.
2) The predicate $I=5$ should be $I=5-J$.
3) The statement $L=J-1$ should be $L=I-2$.
4) The statement $K=I+J-1$ should read

```
THEN IF (2*J < -5*I -40)
     THEN K = 3;
     ELSE K=I+J-1;
```

We leave it as an exercise to verify that the attempt to eliminate the alternative $K \geq I+2$ must necessarily end with the discovery of the first error. Note that his is not trivially the case since errors 1 and 4 can interact in a subtle fashion. In later sections we will show how the remaining three errors are dealt with.

5.6. Domain Pushing

One very important mutation which was mentioned in the last section is the introduction of unary operators into the program. These unary operators are introduced wherever they are syntactically correct according to the rules of FORTRAN expression construction. In addition to the operators ++ and -- discussed in the last section, the remaining unary operators are - (arithmetic negation) and a class of non FORTRAN operators ! (absolute value), -! (negative absolute value) and Z! (zero value). It is the last three which will be of most concern to us in this section.

Consider the statement

```
A = B + C
```

in order to eliminate the mutants

```
A = !B + C
A = B + !C
A = !(B + C)
```

we must generate a set of test points where B is negative (so that B+C will differ from !B+C), C is negative and the sum B+C is *negative*[1].

1) Notice that if it is impossible for B to be negative then this is an equivalent mutation, that is the altered program is equivalent to the original. In this case the proliferation of these alternative can either be a nuisance or an important documentation aid, depending upon the

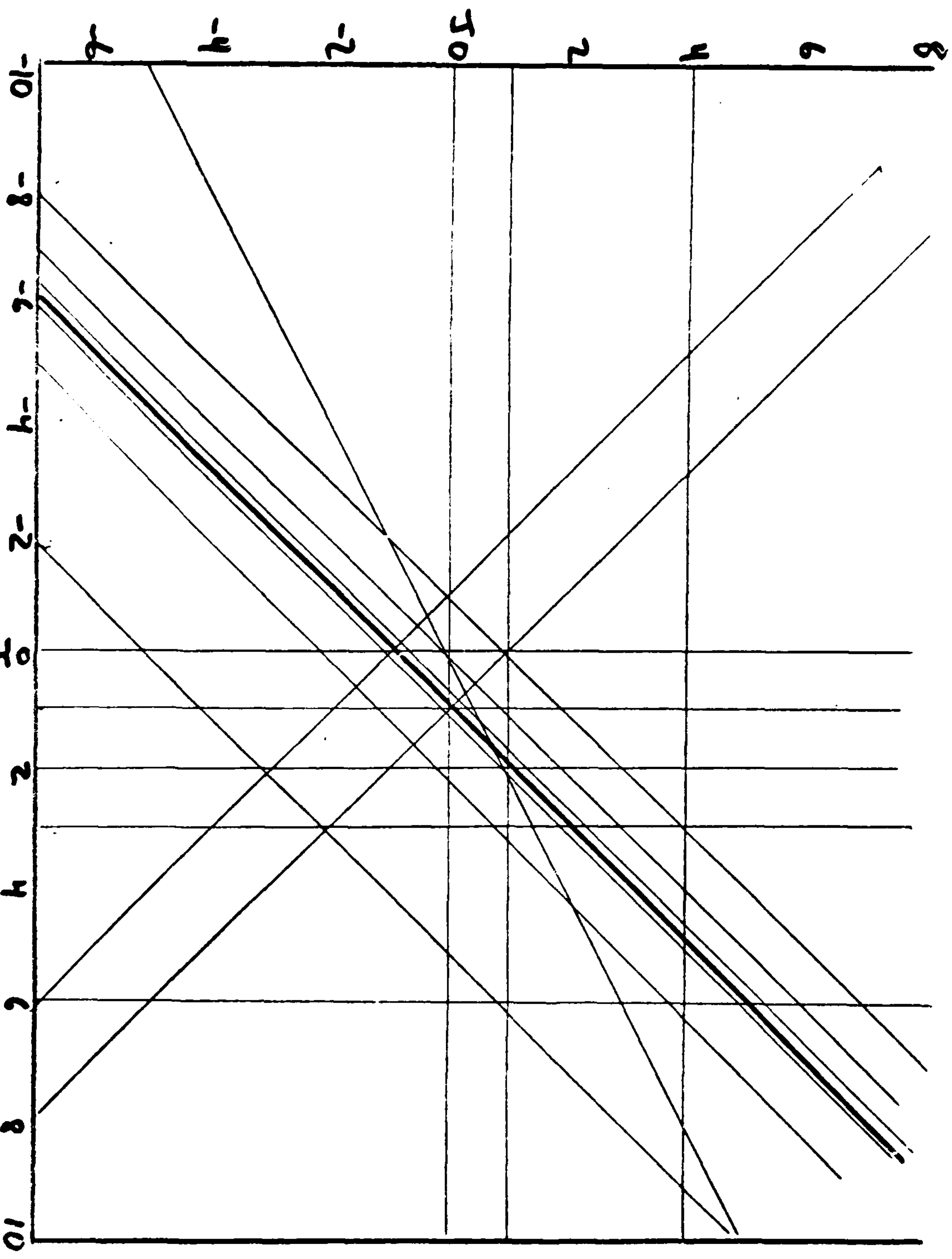


Figure 9

Similarly negative absolute value insertion forces the test data to be positive. We use the term *data pushing* for this process, meaning the mutations push the tester into producing test cases where the domains satisfy the given requirements.

Zero Value is an operator defined such that *Z! exp IS exp if the* value is non-zero, otherwise if the expression evaluates to zero the value is an arbitrarily chosen large positive constant. Hence the elimination of this mutant requires a test set where the expression has the value zero.

Multiply this process by every position where an absolute value sign can be inserted and you can see a scattering effect, where the tester is forced to include test cases acting in various conditions in numerous problem domains. Very often in the presence of an error this scattering effect will cause a test case to be generated which will demonstrate the error.

Consider again the example studied above. *Figure 10 gives a list of* mutants and the accompanying graph shows the domains they push into. As you can see even this simple example generates an extremely large number of requirements.

```
1. IF (!I > J + 1)
2. IF (I > !J + 1)
3. IF (I > !(J + 1))
4. K = (!I + J) - 1
5. K = (I + !J) - 1
6. K = !(I + J) - 1
7. K = !((I + J) - 1)
8. K = 2 * !I + 1
9. K = !(2 * I) + 1
10. K = !(2 * I + 1)
11. IF (!K < I + 1)
12. IF (K < !I + 1)
13. IF (K < !(I + 1))
14. L = !I + 1
15. L = !(I + 1)
16. L = !J - 1
17. L = !(J - 1)
18. IF (!I ¯ 5)
19. M = 2 * !L + K
20. M = !2 * L + K
21. M = 2 * L + !K
22. M = !(2 * L + K)
23. M = !L + 2 * K - 1
24. M = L + 2 * !K - 1
25. M = L + !2 * K - 1
26. M = !(L + 2 * K) - 1
27. M = !(L + 2 * K - 1)
```

figure 10

Recall again that one of the errors this program was presumed to contain was that the statement L=J-1 should have read L=I-2. One effect of this error is that *any* test point in the area bounded by I = J+1

testers point of view. The topic of equivalent mutants will be taken up in section 5.10.

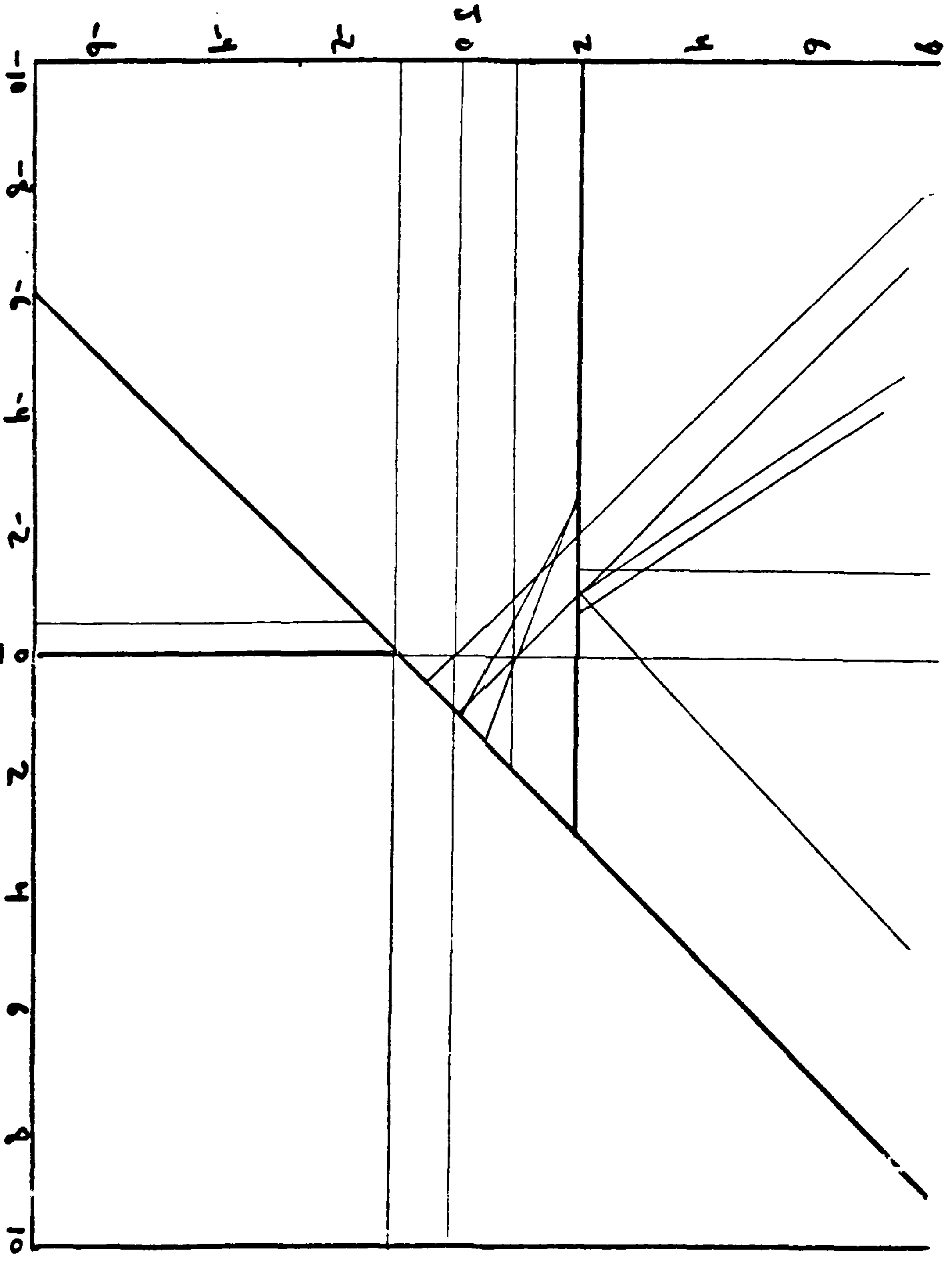


figure 11

and I = 1 will be computed incorrectly. But it is precisely this area that mutants 8, 9 and 10 push us into. This means that this error could not have gone undiscovered using mutation analysis.

This process of pushing the programmer into producing data satisfying some criterion is also often accomplished by other mutations. Consider the program in figure 12, which is based on a program by Naur[14], and has been previously studied in the literature [7].

```
alarm := FALSE
bufpos := 0;
fill := 0;
REPEAT
 i character(cw);
 I  cw = BL or cw = NL
 THEN
     IF fill + bufpos ≤ maxpos
     THEN BEGIN
        outcharacter(BL);
        END
     ELSE BEGIN
        outcharacter(NL);
        fill := 0 end;
        FOR k := 1 STEP 1 UNTIL bufpos DO
            outcharacter(buffer[k]);
        fill := fill + bufpos;
        bufpos := 0 END
   ELSE
     IF bufpos = maxpos
     THEN alarm := TRUE;
     ELSE BEGIN
       bufpos := bufpos + 1;
       buffer[bufpos] := cw END
 UNTIL alarm OR cw = ET
```

figure 12

Consider the mutant which replaces the first statement FILL:=0 with the statement FILL:=1. The effect of this mutation is to force a test case to be defined in which the first word is less then MAXPOS characters long. This test case then detects one of the five errors in the program [7]. The surprising thing is that the effect of this mutation seems to be totally unrelated to the statement in which the mutation takes place.

5.7. Special Values Testing

Another form of testing which has been introduced by Howden[11], is called *special values* testing. Special values testing is defined in terms of a number of "rules", for example

1. Every subexpression should be testing on at least one test case which forces the expression to be zero.
2. Every variable and subexpression should take on a distinct set of values in the test cases.

That the first rule is enforced by the zero values mutations has already been discussed in the last section on domain pushing.

That the second rule is important is undeniable. If two variables are always given the same value then they are not acting as "free variables" and a reference to one can be universally replaced with a reference to the second. In fact this is exactly what happens in this case, and the existence of these mutations enforces the goals of the distinct values rule.

A slightly more general method of enforcing this goal can be constructed as follows: A special array exactly as large as the number of subexpressions computed in the program is kept, with two additional tag bits for each entry in this array. Initially all tag bits are off, indicating the array is uninitialized. As each subexpression is encountered in turn the value at that point is recorded in the array and the first tag bit is set. Subsequently when the subexpression is again encountered if the second tag bit is still off the current value of the expression is compared against the recorded value. If they differ the second tag bit is set. Otherwise no change is made.

In this fashion by counting those expressions in which the second tag bit is OFF and the first ON one can infer which subexpression have not altered their value over the test case executions, and hence one can construct mutations to reveal this. This method is similar to one used in a compiler system by Hamlet[8].

5.8. Coincidental Correctness

We say the result of evaluating a given test point is coincidentally correct if the result matches the intended value in spite of the fact that the function used to compute the value is incorrect. For example if all our test data results in the variable I having the values 2 or 0, then the computation J = I*2 could be coincidentally correct if what was intended was J = I**2.

The problem of coincidental correctness is really central to program testing. Every programmer who tests an incorrect program, and deems it to be correct, has really encountered an incidence of coincidental correctness Yet with the exception of mutation analysis no testing methodology in the authors knowledge deals directly with this problem. Some researches even go so far as to state that the problems of coincidental correctness are intractable [19].

In mutation analysis coincidental correctness is attacked by the use of *spoilers*. Spoilers implicitly remove from consideration data points for which the results could obviously be coincidentally correct, in a sense "spoiling" those data points. For example by explicitly making the mutation J=I*2 => J=I**2 we spoil those test cases for which I = 0 or I = 2, and require that at least one test case have an alternative value.

Using again the example program introduced above, figures 13 and 14 show the spoilers and their effects associated with the statement M=L+2*K-1. Notice a single spoiler may be associated with up to four different lines depending upon the outcomes of the first two predicates in the program. Pictorially, the effects of spoilers are that within each data domain for each line there must be at least one test case which does *not* lie on the given line. In broad terms the effects of this are to require a large number of data points for which the possibilities of coincidental correctness are very slight.

```
1. M = (L + 1*K) - 1
2. M = (L + 3*K) - 1
3. M = (I + 2*K) - 1
4. M = (J + 2*K) - 1
5. M = (K + 2*K) - 1
6. M = (L + 2*I) - 1
7. M = (L + 2*J) - 1
8. M = (L + 2*L) - 1
9. M = (L + I*K) - 1
10. M = (L + J*K) - 1
11. M = (L + K*K) - 1
12. M = (L + L*K) - 1
13. M = (L + 2*K) - I
14. M = (L + 2*K) - J
15. M = (L + 2*K) - K
16. M = (L + 2*K) - L
17. M = (1 + 2*K) - 1
18. M = (2 + 2*K) - 1
19. M = (5 + 2*K) - 1
20. M = (L + 2*1) - 1
21. M = (L + 2*2) - 1
22. M = (L + 2*5) - 1
23. M = (L + 5*K) - 1
24. M = ( - L + 2*K) - 1
25. M = (L + - 2*K) - 1
26. M = (L + 2* - K) - 1
27. M = (L + 2* -- K) - 1
28. M = - (L + 2*K) - 1
29. M = - ((L + 2*K) - 1)
30. M = (L + 2 + K) - 1
31. M = (L + 2 - K) - 1
32. M = (L + MOD(2,K)) - 1
33. M = (L + 2/K) - 1
34. M = (L + 2**K) - 1
35. M = (L + 2) - 1
36. M = (L + K) - 1
37. M = L - 2*K - 1
38. M = (MOD(L,2*K)) - 1
39. M = L/2*K - 1
40. M = L*2*K - 1
41. M = L**(2*K) - 1
42. M = L - 1
43. M = (2*K) - 1
44. M = L + 2*K + 1
45. M = MOD(L + 2*K,1)
46. M = (L + 2*K)/1
47. M = (L + 2*K)*1
48. M = (L + 2*K)**1
49. M = (L + 2*K)
50. M = 1
```

figure 13

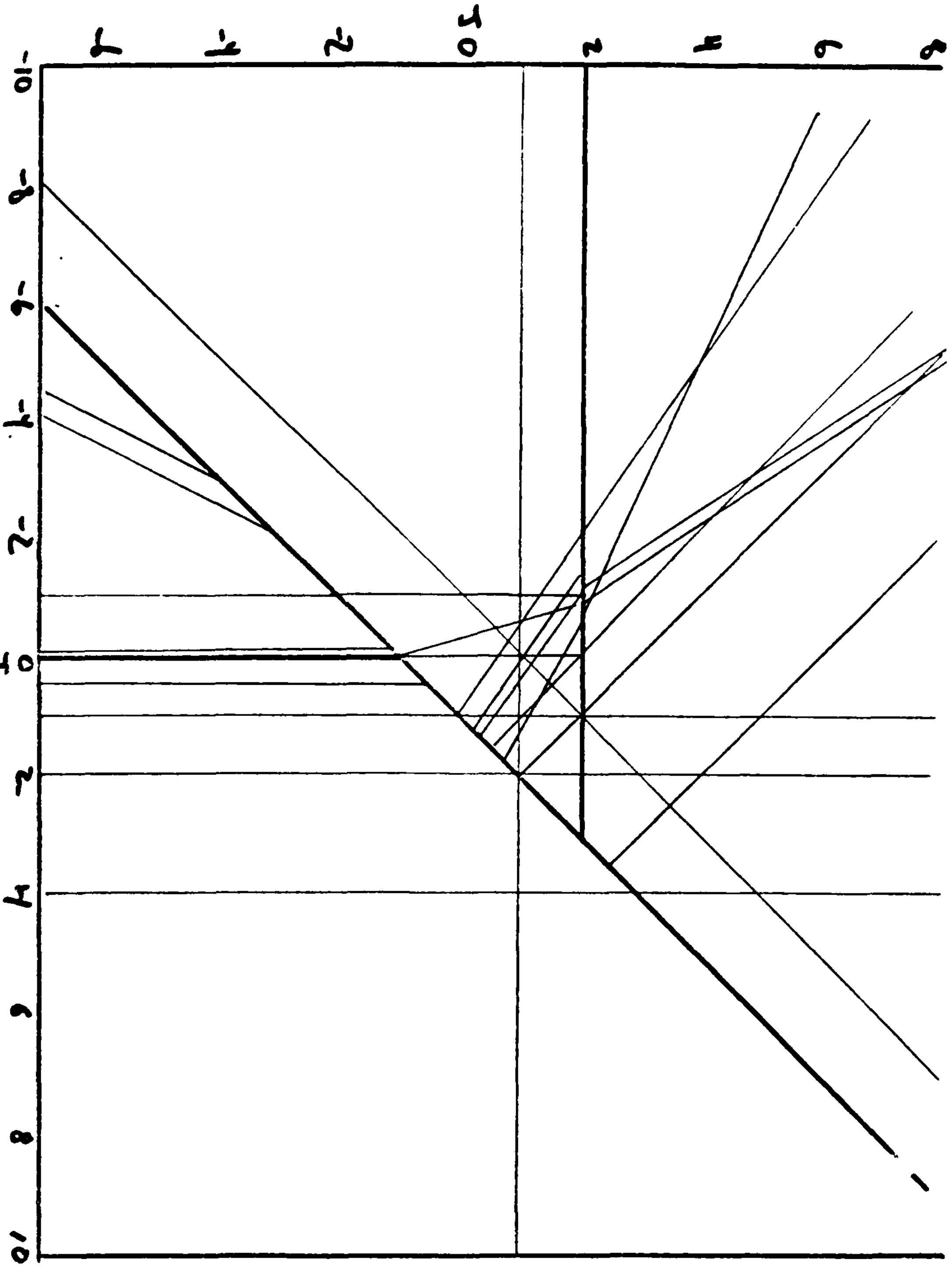


Figure 14

```
for R1 = 0 by 1 to N begin
  R0 <- a(R1)
  for R2 = R1+1 by 1 to N begin
    if a(R2) > R0 then begin
      R0 <- a(R2)
      R3 <- R2
    end
  end
  R2 <- a(R1)
  a(R1) <- R0
  a(R3) <- R2
end
```

figure 15

Often the fact that two expressions are coincidentally the same over the input data is an indication of program error or poor testing. For example the sorting program shown in figure 15, taken from a paper by Wirth[20], will perform correctly for a large number of input values. If, however, the statements following the IF statement are never executed for some loop iteration it is possible for R3 to be incorrectly set, and an incorrectly sorted array may be produced.

By constructing the mutant which replaces the statement a(R1) ← R0 with a(R1) ← a(R3) we point out that there are two ways of defining R0, only one of which is used in the test data. Therefore the error is uncovered.

5.9. Missing Path Errors

As identified by Howden [10], we can say a program contains a missing path error if a predicate is required which does not appear in the program under test, causing some data to computed by the same function when really different functions are called for. These missing predicates can really be the result of two different problems, however, so we might consider the following definitions.

A program contains a *specificational missing path error* if two cases which are treated differently in the specifications are incorrectly combined into a single function in the program. On the other hand a program contains a *computational missing path error* if within the domain of a single specification a path is missing which is required only because of the nature of the algorithm or data involved.

As example of the first type of path error is error number four from the example in section 5.5. Although this error might result from a specification, there is nothing in the code itself which would give any hint that the data in the range 2*j<-5*i-40 is to be handled any differently then given in the test program.

For an example of the second class of error consider the subroutine shown in figure 16, adapted from [13]. The inputs are a sorted table of numbers and an element which may or may not be in the table. The only specification is that upon return X(LOW) ≤ A ≤ X(HIGH), and HIGH <= LOW + 1. The problem arizes if the program is presented with a table of only one entry, in which case the program loops forever.

Nothing in the specifications state that a table with only one entry is to behave any differently from a table with multiple entries, it is only

```
        SUBROUTINE BIN(X,N,A,LOW,HIGH)
        INTEGER X(N),N,A,LOW,HIGH
        INTEGER MID
        LOW = 1
        HIGH = N
6       IF(HIGH - LOW - 1) 7,12,7
12      STOP
7       MID = (LOW + HIGH) / 2
        IF (A - X(MID)) 9,10,10
9       HIGH = MID
        GOTO 6
10      LOW = MID
        GOTO 6
        END
```

figure 16

because of the algorithm used that this must be treated as a special case.

Problems of the second type are usually caused by the necessity to treat certain values, for example negative numbers, differently from others. This being the case the process of data pushing and spoiling described in sections 5.6 and 5.8 will often lead to the detection of these errors. So it is in this case where an attempt to remove either of the following mutants will cause us to generate a test case with a single element.

```
IF (HIGH - LOW - 1) 12,12,7
MID = (LOW + HIGH) - 2
```

Since mutation analysis, like most other testing methodologies, deals only with the program under test (as opposed to dealing with the specifications of those programs), the problems of detecting specificational missing path errors are much more difficult. Since mutation analysis causes the tester to generate a number of data points which exercise the program in a multiplicity of ways our chances of stumbling into the area where the program misbehaves are high, but are by no means certain.

So it is with the missing path error from the example in section 5.5. It is possible to generate test data which passes our test criterion but which fails to detect the missing path error. We view this not as a failure of mutation analysis, however, but as a fundamental limitation in the testing process. In the authors view the only way that these sorts of problems have a hope of being eliminated is to start with a core of test cases generated from the specifications, independent of the program implementation. This core of test cases can then be augmented to achieve goals such as those presented by mutation analysis. Some methods of generating test data from specifications have been discussed elsewhere [7,17].

5.10. Equivalent Mutants

As was mentioned in a footnote in section 5.6, if a variable is constrained to being strictly positive (which is often the case) then inserting an absolute value sign before each reference to that variable will generate an alternative program which is in all respects functionally identical to the original. A mutation which produces such an equivalent program is called an *equivalent mutant.*

Almost any of the mutation types used in the current system can, under the right circumstances, produce an equivalent mutant. It has been observed empirically that with the exception of those mutations produced by inserting absolute value signs (which often vary widely) the number of equivalent mutants produced is usually 2-5% of the total number of mutants.

In the current system no attempt is made to remove equivalent mutants algorithmically, even though in a large number of cases it would be possible to do so. The reason for this decision is because even though equivalent mutants serve no purpose from the point of view of test data analysis, they serve a very important role in error detection.

No mutant is ever declared equivalent except by an explicit command from the tester. In order to determine equivalence the tester must often spend a considerable amount of time examining the code, and in the process obtain an intimate knowledge of the algorithm and how it works.

Often a number of mutants can be labeled equivalent on the strength of a single insight. Example are recognizing that a variable is by necessity positive during part of the program, or recognizing that in a binary search algorithm it doesn't matter how you choose the middle element as long as it is between the lower and upper bounds.

The fact is, however, that in attempting to remove equivalent mutants we are forcing the programmer into a very careful review of the program. How many errors are discovered in this manner is more of a question in psychology then in program testing, but our experience has been that often such a careful review will uncover very subtle errors which would be difficult to discover by other means.

As an example of this process, we must admit that no mutation in the current system would force the tester into discovering the second error in the program in section 5.5. (Notice that if J had been referenced in the section of code following the I=5 predicate then the process of data pushing would have revealed this error.) None the less the following mutants are equivalent for the given program. An examination of these would force the tester almost directly into a review of the area of code containing the bug. And the search would be intensified if the tester realized these changes would not be equivalent in the corrected program.

```
M = 2*!L + K
M = !2*L + K
M = 2*L + !K
M = !(2*L + K)
```

6. Discussion

After an extended exposition of the mechanics of mutation analysis we are now in a position to take a more global look into why this all works. It seems to us that there are two general arguments which can be put forth, summarized as follows:

1) With respect to error detection, it is not that the mutants themselves capture the errors which may be in the program, it is rather that the mutation task forces the tester into finding data which exercises the program in a multiplicity of ways, and this exercising is what is likely to uncover the errors.

2) The goal of mutation analysis is difficult to attain (this is confirmed by more then two years experience with this process), and by setting a difficult goal we force the programmer into a very careful review of the programs. Independent of all other claims made by this method, merely forcing the programmer to spend an extended period of time reviewing the coded product will often lead him into discovering errors in logic or design.

Of course we would hope that the first is the dominant reason for discovering errors in programs, and indeed the studies we have so far conducted indicate this. We mention the second, however, because it is often significant in real applications, and is a fact not usually noticed by automated tool designers.

As we saw in section 5.10, the mutations implemented in the current system are not sufficient to detect *all* programming errors. This we view not as a weakness in the methodology but in the mutation operators used. As we collect more and more examples of such errors we can look for patterns in the types of errors which can go undetected by our system. By observing these patterns we may find new mutant operators which will detect these errors. In this manner the system may be continually improved, and our understanding of the programming process itself increased.

We have also observed that as the complexity of programs increases, the number of "building blocks" from which mutations are constructed *grows*[2], and the chances for errors like those just described to go undetected actually diminishes. This is perhaps a novelty- a method which works better on complex programs then on simple ones !

Acknowledgements

We wish to thank Alan Acree, Jim Barns, Edie Martin, and Dan St. Andre for their contributions to the program mutation effort.

2) the number of mutants grows roughly proportional to the number of statements times the number of unique data references in the program.

[1] T.A. Budd and R.J. Lipton, "Mutation Analysis of Decision Table Programs", Proceedings of the 1978 Conference on Information Sciences and Systems, Johns Hopkins University, 1978.

[2] T.A. Budd, R.J. Lipton, F.G. Sayward and R.A. DeMillo, "The Design of a Prototype Mutation System for Program Testing", AFIPS 1978 NCC, pp 623-627.

[3] J.R. Brown and M. Lipow, "Testing for Software Reliability", Proceedings of the 1975 International Conference on Reliable Software.

[4] R.A. DeMillo, R.J. Lipton and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", COMPUTER, Vol. 11,4. April 1978.

[5] R.A. DeMillo, R.J. Lipton and F.G. Sayward, "Program Mutation as a Tool for Managing Large-Scale Software Development", ASQC Technical Conference Transactions- Chicago.

[6] M. Geller, "Test Data as an Aid in Proving Program Correctness", Comm. ACM Vol. 21,5 May 1978 , pp 368-375.

[7] J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions of Software Engineering, June 1975.

[8] R.G. Hamlet, "Testing Programs with the Aid of a Compiler", IEEE Transactions of Software Engineering, SE3-4, July 1977.

[9] C.A.R. Hoare, "Algorithm 65: FIND", Comm. ACM 4,1 (April 1961), pp. 321.

[10] W.E. Howden, "Reliability of the Path Analysis Testing Strategy", IEEE Transactions of Software Engineering, September 1976.

[11] W.E. Howden, "An Evaluation of the Effectiveness of Symbolic Testing", Software - Practice and Experience, Vol. 8,381-397(1978).

[12] J.C. Huang, "An Approach to Program Testing", ACM Computing Surveys, September 1975.

[13] B.W. Kernighan, and P.J. Plauger, *The Elements of Programming Style*, McGraw Hill, New York,N.Y., 1978 (2nd ed.)

[14] P. Naur, "Programming by Action Clusters", BIT, Vol. 9, pp 250-258, 1969.

[15] L.J. Osterweil and L.D. Fosdick, "Experience with DAVE- A Fortran Program Analyzer", Proc. 1976 AFIP NCC, Vol 45, PP. 909-915.

[16] L.J. Osterweil and L.D. Fosdick, "Data Flow Analysis as an Aide in Documentation, Assertion Generation, Validation, and Error Detection", Technical Report CU-CS-055-74, Department of Computer Science, University of Colorado, Boulder, September 1974.

[17] T.J. Ostrand, E.J. Weyuker, "Remarks on the Theory of Test Data Selection", Digest for the IEEE Workshop on Software Testing and Test Documentation, Ft. Lauderdale, Fl. 1978.

[18] R.J. Rubey, J.A. Dana, and P.W. Biche, "Quantitative Aspects of Software Validation", IEEE Transactions of Software Engineering, June 1975.

[19] L.J. White, E.I. Cohen and B. Chandrasekaran, "A Domain Strategy for Computer Program Testing", Ohio State University Technical Report OSU-CISRC-TR-78-4, 1978.

[20] N. Wirth, "PL360, a programming language for the 360 computer", JACM, 15, 37-74 (1968).

[21] E.A. Youngs, *Error Proneness in Programming*, PhD Thesis, University of North Carolina, 1971.

Discussion of "A Survey of Programming Testing Issues"

Timothy A. Budd*, Richard A. DeMillo+, Richard J. Lipton* and

Frederick G. Sayward*

In this paper Goodenough addresses a myriad of issues and goals encountered in testing computer programs. During his discussion of using testing to show program correctness it is explained that testing can be used to ensure the absence of program errors providing that one has successfully performed a test which is both reliable(1) and valid(1). The effectiveness of this approach lies in finding reliable and valid tests which, as observed by Goodenough, can be extremely difficult. Our main purpose in this note is to comment on these concepts, on their applicability to testing computer programs, and to suggest an alternative approach.

As in an earlier paper [5], reliability and validity are defined in terms of quite precise and formal properties of computer test data selection criteria. With these definitions a so-called "fundamental theorem" was proved in [5] which roughly states:

> If a test data selection criteria which is valid and reliable can be used to select test data for a given program, and if the program is correct on the selected test data, then the program is correct on any data.

(+) School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332

(*) Department of Computer Science, Yale University, New Haven, Connecticut 06520

(1) See Goodenough's paper for the formal definitions of these and other terms which we will drawn on in this discussion.

Several comments are in order on this approach. First, reliability and validity are defined as binary attributes; that is, either a test data selection criteria has or doesn't have one or both of these properties. However, intuition says we should expect that a program test is reliable and valid if it is useful in predicting the correctness of a program - not necessarily ensuring absolute correctness in the formal sense, but at least increasing our confidence that the program is indeed correct. Although Goodenough addresses this aspect indirectly in a footnote, where the idea of measuring a test data selection criteria's reliability and validity is discussed in passing, there is the danger that readers will follow him and not focus on this issue which we consider to be the most important issue of his approach to using program testing to ensure program correctness. Second, we feel that the fundamental theorem provides no useful information or guidelines for anyone who has to test real programs since it is aimed at showing absolute correctness. What Goodenough has done is to reiterate the conclusion of [5] - if you prove that your testing criteria is perfect in a fairly obvious sense, then your program is correct if it passes the test. Clearly, this is the expected deduction. He then says that research in this area of program testing should be directed toward finding reliable and valid testing methods, or at least establishing how "close" the methods are to being reliable and valid so that we can judge how "close" to perfect are programs which pass the test.

As an editorial note, while the fundamental theorem of [5] shows that validity and reliability are sufficient conditions for demonstrating program correctness by program testing, they certainly aren't necessary conditions. Yet Goodenough consistently says that program tests _must_ be valid and reliable if correctness is to be gotten from testing. Clearly, this is misleading and could adversly influence future program testing research efforts.

Goodenough ends on a pesimistic note in stating that, from a scientific point of view, testing research can hardly be said to be in its infancy. He, as others in the software engineering community, most notably the program verification school, continue to point out that program testing is insufficient to guarantee program correctness. We agree. However, since all software being used today, since all software that has ever been developed to solve any real problems, has been developed using testing, we must ask the following rather obvious question:

> Given that program testing, while not a perfect technique, has proved to be a very useful technique, how can we develop testing methodologies which have less than perfection (absolute program correctness) as their goals yet still yield substantial gains?

It is clear to us that the future direction of software engineering must not turn its back and risk not developing this very important research area the way it should be. It is all too easy, and wrong, to take the popular viewpoint that program building is a purely logical deductive activity to which program testing is unsuitable. Our viewpoint is that program design and development is an empirical engineering activity and when Goodenough says that program testing is not even in its infancy, we take it to mean that an inferential formalism for program testing has not yet been developed. However, it seems clear that such a formalism is not entirely necessary if one is willing to accept that programming is a human, inductive activity which may never be subject to complete formalism.

In the remainder of this discussion we will overview an on going research effort which is aimed at achieving gains from program testing while not ensuring perfection: namely, program mutation [4]. It has been observed [6] that the vast majority of errors that remain in software, once it has been tested and put into production, tend not to be radical errors(2) but rather are interacting combinations of simple errors. Indeed, there are many "horror" stories similar to the failure of an early Vangard missile launch because of a missing right parenthesis in a controlling program. So a resonable goal of program testing is to rule out all combinations of simple errors: that is, design a program testing method with the goal being that if a program passes the test then either

(1) the program is correct, or

(2) the program is radically incorrect.

But even this seems too ambitious if one attacks directly. First, given a program we must be able to generate all of its simple errors. Assuming that this can be done, we next must eliminate the simple errors and the complex errors which eminate from their combinations. Clearly the number of complex errors will be a combinatoric explosion in the number of simple errors. While it may be feasible to eliminate all simple errors, explicit elimination of all complex errors appears intractable.

The goal of the program mutation testing methodology is to establish that a given program is either correct or radically incorrect. Let L be the programming language under consideration. A mutant operator is a simple program transformation, dependent on L, which produces mutant programs of the a given program P. The mutants are also programs in L.

----

(2) There are no agreed on technical definitions of errors categories. We too will be informal. By radical we mean errors due to grossly misunderstanding the program specifications - errors which are difficult if not impossible to capture by general algorithmic methods but which would easily be observed by almost any test or when the software is first put into production.

The goal of the mutant operator is to introduce simple errors in P, thus producing mutants of P. Alternatively, if P is incorrect due to a single simple error, some mutant would be a correct program for the given task. There should be several mutant operators, each corresponding to different classes of simple errors that may occur in L. Let M(P) denote the set of all mutants of P. Ideally, M(P) should contain mutants corresponding to all and only the possible simple errors. However, this is too ambitious a goal for general purpose program transformations and we relax the requirement to be that M(P) covers all simple errors in the sense that M(P) may also contain mutants which are equivalent to P. We let $M^*(P)$ denote all the mutants of P which come from multiple applications of mutant operators on P. These mutants are also programs in L.

Let D be the input domain of P. P is said to pass the *mutation test* with data T if there exists T a subset of D such that

(1) P is OK(T), and

(2) for each mutant m in M(P) either

(a) m is not OK(T), or

(b) m is equivalent to P.

If P passes the mutant test then we are sure that P is free of simple errors. But what of complex errors? To this end we have observed a *coupling effect* which states:

Test data T which causes all the non-equivalent mutants of M(P) to fail is so sensitive that all the non-equivalent mutants of $M^*(P)$ must also fail on T.

The justification of the coupling effect parallels the probabalistic argument for justifying the single fault methods used to test circuits; however, we have no theory to make it a hard-fast principle. Basically, if several simple errors (detectable by T) combine to make a complex error then it is extremely unlikely the simple errors will cancel to allow the successful execution on T of the mutant containing the complex error. The goal of program mutation theory is then to validate, depending on L either deductively or experimentally, the coupling effect for language L by establishing the following metatheorem of program mutation:

If P passes the mutation test then either

(1) P is correct, or

(2) P is radically incorrect.

In [1] the mutation metatheorem has been formally shown to hold where L is certain classes of decision tables and the mutant operator involve the reformulation of conditions and applied actions. Currently, programs which manipulate data structures are under investigation.

For general purpose programming languages, such as FORTRAN, the task is more difficult. There is a noticable lack of empirical studies on programming errors to drawn on in formulating a complete set(3) of mutant operators - a necessary requirement for program mutation to be deductive. Hence, at at least for now, in the case of general purpose languages we can consider program mutation as an inductive tool for gaining confidence that the metatheorem of program mutation holds for a particular program P. A prototype system for a subset of FORTRAN has been implemented [2] and some initial experience with it and the effectiveness of the implemented mutant operators and substantiations of the coupling effect can be found in [3]. A mutation system for ANSI FORTRAN is in the design stage. Several experiments to finding "good" mutant operators and for evaluating the

----

(3) Here, complete means that all simple errors will be captured in M(P).

effectiveness of mutation testing are under consideration.

Some final commments on performing the mutation test are in order. Clearly the size of M(P) and T must be small. Our view is that the mutation system should be interactive. The user specifies the program P and initial test data T_1 to the system whence the mutants are generated and executed on T_1. A list of mutants which fail and which succeed on T_1 is produced. The user must then examine his results to decide

(1) P contains a non-radical error.

(2) Because mutants which should have failed didn't, T_1 must be augmented to T_2 and the system re-run.

(3) Some mutants are equivalent to P. There is hope here that symbolic execution techniques can partially automate this task.

This cycle can be viewed as a session in which the user defends P and the current test data against a system adversary which asks questions of the form, "Why doesn't your test data distinguish this simple error?" Such an adversary forces the user of program mutation into a careful and detailed review of his program and the design decisions made in constructing it. In this view we hold hope that even radical errors can be uncovered by users of program mutation.

REFERENCES

[1] T.Budd and R.Lipton, "Mutation Analysis of Decision Table Programs", to appear at the 1978 Conference on Information Sciences and Systems, Johns Hopkins University.

[2] T.Budd, R.DeMillo, R.Lipton and F.Sayward, "The Design of a Prototype Mutation System for Program Testing", to appear at the 1978 National Computer Conference.

[3] R.DeMillo, R.Lipton and F.Sayward, "Hints on Test Data Selection", to appear in *Computer*, April 1978.

[4] R.DeMillo, R.Lipton and F.Sayward, "PROGRAM MUTATION: A Method of Determining Test Data Adequacy", to appear, 1978.

[5] J.Goodenough and S.Gerhart, "Toward a Theory of Testing: Data Selection Criteria", IEEE Trans. on Soft. Eng. SE-1,2 (June 1975), pp. 156-173.

[6] E.A.Youngs, "Human Errors in Programming", International Journal of Man Machine Studies 6 (1974), pp. 361-376.

THE STATUS OF RESEARCH ON PROGRAM MUTATION

Richard J. Lipton(*) and Frederick G. Sayward(-)

December 1978

ABSTRACT

A status report on two new program mutation systems is given. The first is the EXPER system for testing programs written in ANSI FORTRAN and for experimenting on the concepts of program mutation. It has been designed, implemented, and is in its final debugging stages. The second is the CPMS systems for testing programs written in a COBOL subset. This system is in its final design stage.

Also, the results of a new experiment on substantiating the "coupling effect" of our FORTRAN systems are presented.

INTRODUCTION

Program mutation is a relatively new approach to program testing which, unlike traditional methods, attempts to exploit the fact that good programmers write code which is "close" to being correct. Traditionally, the fundamental question addressed in program testing has been:

> Given that a program P is known to work on test data T, can we conclude that P works in general?

As expected, the traditional question is theoretically unanswerable [8]. However, program testing researchers have made advances in providing definite answers for special cases [6,10] and, for the general case, have provided methods [3,9,11,12] for gaining confidence in a positive answer.

Program mutation, on the other hand, has striven to answer a weaker yet quite realistic question. The formulation of this weaker question is

---

(*) Department of Electrical Engineering, University of California at Berkeley, Berkeley, California 94720

(-) Department of Computer Science, Yale University, New Haven, Connecticut 06520

This research was supported in part by Georgia Institute of Technology subcontracts under the sponsorship of AIRMICS research grant DAAG 29-78-G-0121.

based on what we call the competent programmer assumption:

> A competent programmer, after several iterations and on deeming that his job of designing, coding, and testing is complete, has written a program that is either correct or "almost" correct in the sense that it differs from a correct program in only simple ways.

As a simple example, suppose we want a FORTRAN program that computes the distance from the origin to an N-dimensional vector X where the distance is defined to be the square root of the sum of the squares of the elements of X. We would accept the following incorrect program as being written by a competent programmer:

```
        PROGRAM P1
        SUM=1
        DO 1 I=1,N
        SUM=SUM+X(I)**2
      1 DIST=SQRT(SUM)
```

But we would question the competence of a programmer who produced

```
        PROGRAM P2
        DIST=X(1)
        DO 1 I=1,N
      1 DIST=MAX(X(I),DIST)
```

With the competent programmer assumption, the question addressed in program mutation becomes:

> Given that P is written by a competent programmer and that P is known to work on test data T, can we conclude that P works in general?

Note that the mutation question differs philosophically from the traditional testing questing in a very important way: traditionally, a program is treated as a random object, whereas in program mutation a program is assumed to be either correct or almost correct, a mutant of a correct program. Thus program P1 above is a mutant of the correct program P for the distance problem:

```
      PROGRAM P
      SUM=0.0
      DO 1 I=1,N
    1 SUM=SUM+X(I)**2
      DIST=SQRT(SUM)
```

P2, on the other hand, is not a mutant of P.

To apply program mutation, we choose the method of eliminating the alternatives -- developing a test set T on which the program P is correct but on which all mutants of P fail. In practice there are far too many mutants of P to consider. But by concentrating on the "first order" mutants of P the methodology becomes tractable. First order mutants of P come from a single application of a <u>mutant operator</u>, a simple syntactic or semantic program transformation such as (1) changing a particular instance of a relational operator to one of the five other relational operators, or (2) changing the label part of a particular GOTO statement to one of the other labels appearing in the program. We then rely on the <u>coupling effect</u>:

> Test data that causes all first-order mutants of a program to fail is so sensitive that all higher-order mutants of the program will also fail.

To illustrate, the following two programs are first-order mutants of program P above:

```
      PROGRAM M1                     PROGRAM M2
      SUM=1                          SUM=0.0
      DO 1 I=1,N                     DO 1 I=1,N
    1 SUM=SUM+X(I)**2                SUM=SUM+X(I)**2
      DIST=SQRT(SUM)               1 DIST=SQRT(SUM)
```

Program P1 is a mutant but not a first-order mutant of P. By the coupling effect, if P is correct on test data T while M1 and M2 fail, then P1 must also fail on T.

With this formulation, the effectiveness of program mutation now depends on the validity of two assumptions: the competent programmer assumption and the coupling effect. In practice, theoretical studies notwithstanding [2], it is not necessary to show formally that these assumptions hold in order for program mutation to be a useful tool for testing real programs written in real programming languages. We have found that in performing mutant tests on an incorrect program the user is forced into developing test data on which the program fails [1]. So we are interested in building interactive systems to aid programmers in performing mutant tests and in evaluating the effectiveness of the approach. We pick a real programming language L and, based on the literature and our personal experience, define an appropriate set of mutant operators for L. Then we build a man-machine mutation system that aids in performing mutant tests for L and the chosen mutant operators. Using the system and other aids, we then perform experiments to substantiate the competent programmer assumption and the coupling effect for L and the given mutant operators; we also check to see how effective the system is as a testing tool.

So far we have introduced program mutation [4] and built a pilot mutation system, called PIMS, for a subset of FORTRAN [1]. With PIMS we wanted to gain some initial experience with mutations and building mutation systems. The subset consisted of a single FORTRAN subroutine with DO, IF, GOTO, and assignment statements as the control structures. The data structures were integers with arrays of up to two dimensions. The mutant operators were four classes: declaration, data reference, operator evaluation, and control flow We discussed user methods of determining the correctness of a program on test data, automatic detection of mutant failure, mutants equivalent to a given program, non-terminating mutants, and managing the n^2 mutants generated by applying the mutant operators, n the number of executable statement in the program. We have also done experiments on the competent programmer assumption [1] and the coupling effect [5].

Now we would like to report on two new program mutation systems, one for nearly full ANSI FORTRAN, the other for a COBOL subset, and on a new, stronger experiment for substantiating the coupling effect for our FORTRAN systems.

THE EXPER SYSTEM

In working with PIMS, we observed that the test data most programmers intuitively feel is good as well as test data generated by automatic means, either randomly or by symbolic execution, do poorly with respect to the mutant test. Perhaps this gives evidence as to why program testing has traditionally been held in such low esteem. We admit that PIMS wasn't very flexible in its design and consequently we were able to perform only limited experiments with this system.

The EXPER system has as its language ANSI FORTRAN minus I/O and complex arithmetic. Its mutant operators are basically the same as in PIMS. The system was built at Yale on the DEC-20 and is nearly debugged. Recently, it has begun to be transported to the VAX computer at UC Berkeley. Among the goals of this system are (1) determining how program mutation can be integrated with the design, coding, and testing of multi-module programs, (2) determining whether the mutant operators of PIMS are sufficient for the additional data and control structures allowed in EXPER, (3) further experiments on the coupling effect, and (4) experiments on the effectiveness of the method.

Besides extending the language, there is another major difference between PIMS and EXPER. PIMS was designed solely as a user-oriented system; the user submits a program and a set of test data and selects which system-defined mutant operators are to be applied to the program. EXPER, on the other hand, is organized around the concept of an experiment which consists of a program, a set of test data, a subset of the system's mutant operators which may be applied to the program, and a further subset of this subset which will be applied to the program. The experimenter is easily able to generate slight differences in each of these elements and then monitor the progress of subjects using EXPER to perform the mutant test.

One current experiment involves the redundancy of mutant operators. Such redundancy could be counter-productive if time is spent constructing test data that don't significantly increase one's confidence in the correctness of a program. The aim of this experiment is to detect redundant mutant operators by statistical methods. The subjects are divided into two groups. Each group is given several programs and asked to develop test data by doing mutation analysis. Some of the programs contain bugs which the subjects are to try to find. The difference between groups is in what mutant operators they may apply to the programs. Group 1 is allowed to use all implemented operators while group 2 may use all but the operator(s) in question. The variables to be compared are the number of bugs located and the time used in locating them.

Several other experiments are currently being formulated, such as experiments to evaluate executing only some mutants versus executing all mutants on given test data. We plan to report on these experiments in a future paper.

THE CPMS SYSTEM

The design of a COBOL pilot mutation system, called CPMS, is in its final stages at Georgia Tech. The COBOL subset for this project consists of a single COBOL procedure with sentences of the MOVE, COMPUTE, IF, PERFORM, READ, WRITE type as its control structures, character and decimal scalar variables with the record feature as its data structures, and up to two sequential input and two sequential output files as its I/O structures. The mutant operators of CPMS will be similar to those of PIMS with major additions for data structures and I/O.

The CPMS design is based on PIMS: an interactive man-machine system to perform the mutant test for programs written in the COBOL subset. CPMS, however, will be more flexible than PIMS in its experimental capabilities.

Aside from applying program mutation to a new language, the major issue addressed in CPMS is the I/O problem, which was avoided in PIMS and EXPER. As in FORTRAN, a COBOL mutant may fail in one of three ways: it may have an execution fault, it may time out, or it may produce incorrect answers. Because the output of an average COBOL program tends to be much larger than that of an average FORTRAN program, it is not clear whether there is an efficient way to check for this third kind of failure. We intend to try the following scheme:

(1) A mutant fails if it tries to read a longer record than the program read.

(2) A mutant fails if it reads fewer (more) records than the program read.

(3) A mutant fails if it writes fewer (more) records than the program wrote.

(4) A mutant fails if it produces files that are unequal to the files that the program produced. Here the user specifies

whether a strong or a weak equality check is to be used. Two files are strongly equal if their records match character for character; they are weakly equal if the non-blank characters of their records match.

We hope that the vast majority of the COBOL mutants will fail before step 4 is involved. Of course, we are not certain whether a mutant that fails on some steps of the scheme should not be allowed to continue anyway. Part of the COBOL mutation project will be experimenting to find a realistic definition of mutant failure on I/O.

STRONGER SUBSTANTIATION OF THE COUPLING EFFECT

We have already reported on an experiment [5] involving Hoare's FIND program [7] that supplied empirical evidence for the coupling effect. The experiment went as follows:

(1) We derived a test data set T of 49 cases to pass the mutant test. (The large size of T was due to our inexperience.)

(2) For efficiency reasons, we reduced T heuristically to a test data set T′ consisting of seven cases on which FIND also passed the mutant test.

(3) Random k-order mutants of FIND, k>1, were generated. (A k-order mutant comes from k applications of mutant operators on the program P.)

(4) The k-order mutants of FIND were then executed on T′.

The coupling effect says that the non-equivalent k-order mutants of FIND will fail on T′. Note that step 2 biases the experiment against the coupling effect since it removes the man-machine orientation of our approach to testing. We would have been quite happy to find a counterexample to the coupling effect for PIMS, since it would have allowed us to improve the set of mutant operators. The results of the experiment, though, gave evidence that we had chosen a well coupled set of mutant operators for PIMS:

k	Number of k-order mutants	Number successful on T'
2	21100	19
>2	1500	0

The 19 successful mutants were shown to be equivalent to FIND. We concentrated on the k=2 case since, intuitively, the more one mutates FIND the more likely one is to get a program that violates the competent programmer assumption.

The major criticism of the experiment concerns step 3. Since the first-order mutants that compose the k-order mutants are indepentently drawn, the resulting k-order mutant is likely to be very unstable and subject to quick failure, in contrast to the more desirable case where the k-order mutant contains subtly related changes that correspond to the subtle errors programmers find so hard to detect.

The current experiment on the coupling effect, which uses EXPER rather than PIMS, omits step 2 above and makes the following important change to step 3:

(3) Randomly generate <u>correlated</u> k-order mutants of the program. By correlated we mean that each of the k applications of mutant operators will in some way be related to all of the others -- they could for instance effect the same statement of P, or the same variable name, or the same statement label, or the same constant. Once again, if P passes the mutant test with test data T, the coupling effect says that the correlated k-order mutants of P will fail on T.

For this experiment three programs are being used: FIND, STKSIM, and

TRIANG. STKSIM is a program that maintains a stack and allows the standard operations of clear, push, pop, and top. TRIANG is a program that, given the lengths of the three legs of a triangle, categorizes the input as not representing a triangle or as representing a scalene, isosoleses or equilateral triangle [3]. The following is a summary of the results of the experiment so far:

PROGRAM	k=2		k=3		k=4	
	number	successes	number	successes	number	successes
FIND	3000	2	3000	0	3000	0
STKSIM	3000	3	3000	0	3000	0
TRIANG	3000	1	3000	1	3000	0

In all cases, the successful correlated k-order mutants have been shown to be equivalent to the original program. The detailed results of the experiment on TRIANG are listed in the appendix.

We have yet to find a non-trivial counterexample to the coupling effect for our FORTRAN systems. The one successful 3-order mutant of TRIANG deserves closer examination; indeed, we initially felt that it was a non-equivalent mutant. The mutant is

```
                    SUBROUTINE TRIANG(I,J,K,MATCH)
            C
                    INTEGER I,J,K,MATCH
            C
            C       MATCH IS OUTPUT FROM THE ROUTINE:
            C               MATCH = 1 IF TRIANGLE IS SCALENE
            C               MATCH = 2 IF TRIANGLE IS ISOSCELES
            C               MATCH = 3 IF TRIANGLE IS EQUILATERAL
            C               MATCH = 4 IF NOT TRIANGLE
            C
 1   2              IF (I.LE.0.OR.J.LE.0.OR.K.LE.0) GOTO 500
 3                  MATCH=0
 4   5              IF (I.NE.J) GOTO 10
 6                  MATCH=MATCH+1
 7   8         10   IF(I.NE.K) GOTO 20
 9                  MATCH=MATCH+2

MO.: change statement 9 to MATCH=MATCH+K

10 11          20   IF(J.NE.K) GOTO 30
12                  MATCH=MATCH+3
```

```
13 14       30  IF(MATCH.NE.0) GOTO 100
15 16           IF(I+J.LE.K) GOTO 500
17 18           IF(J+K.LE.I) GOTO 500
19 20           IF(I+K.LE.J) GOTO 500
21              MATCH=1
22              RETURN
23 24      100  IF(MATCH.NE.1) GOTO 200
25 26           IF(I+J.LE.K) GOTO 500
27         110  MATCH=2
28              RETURN
29 30      200  IF(MATCH.NE.2) GOTO 300
```

MO_2: change statement 29 to IF(MATCH.NE.K)

```
31 32           IF(I+K.LE.J) GOTO 500
33              GOTO 110
34 35      300  IF(MATCH.NE.3) GOTO 400
36 37           IF(J+K.LE.I) GOTO 500
```

MO_3: change statement 36 to IF(J+J.LE.I)

```
38              GOTO 110
39         400  MATCH=3
40              RETURN
41         500  MATCH=4
42              RETURN
                END
```

Note that the correlation is with respect to variable K. The mutant operators MO_1 and MO_2 produce incorrect mutants while MO_3 produces a mutant equivalent to TRIANG. Yet the 3-order correlated mutant is equivalent to TRIANG.

This makes a beautiful illustration of the part of the programming process that program mutation is trying to exploit. Using the constant 2 in statements 9 and 29 is an arbitrary but coupled decision. Indeed, you can replace both instances of 2 by any positive constant (or any variable whose value doesn't change between the execution of statements 9 and 29) and you get an equivalent program -- replace only one instance and you get an incorrect program. In a sense, the constant 2 in statements 9 and 29 is what would be called in the terminology of formal logic a "bound variable."

ACKNOWLEDGEMENTS

We wish to thank Alan Acree, Tim Budd, Jim Burns, Rich DeMillo, Edie Martin, and Dan St. Andre for their contributions to the program mutation effort and also to thank Mary-Claire Van Leunen for the careful editing and stylistic mutations she has made to our initial draft.

REFERENCES

[1] T.A.Budd, R.A.DeMillo, R.J.Lipton, and F.G.Sayward, "The Design of a Prototype Mutation System for Program Testing", Proceedings of the 1978 National Computer Conference, pp. 623-627.

[2] T.A.Budd and R.J.Lipton, "Mutation Analysis of Decision Table Programs", Proceedings of the 1978 Conference on Information Sciences and Systems, pp. 346-349.

[3] L.A.Clarke and J.L.Woods, "Program Testing Using Symbolic Execution", presented at the Navy Laboratory Computing Committee Symposium on Software Specification and Testing Technology, April 1978.

[4] R.A.DeMillo, R.J.Lipton, and F.G.Sayward, "PROGRAM MUTATION: A New Approach to Program Testing", presented at the Navy Laboratory Computing Committee Symposium on Software Specification and Testing Technology, April 1978.

[5] R.A.DeMillo, R.J.Lipton, and F.G.Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", Computer 11,4 (April 1978), pp. 34-41.

[6] M.Geller, "Test Data as an Aid in Proving Program Correctness", in Proc. of the Third ACM Symp. on the Principles of Programming Languages (1975), pp.209-218.

[7] C.A.R.Hoare, "Algorithm 65: FIND", Comm. of the ACM 4,1 (April 1961), pp. 321.

[8] W.E.Howden, "Reliability of the Path Analysis Testing Strategy", IEEE Trans. on Soft. Eng. SE-2,3 (Sept. 1976), pp. 208-214.

[9] W.E.Howden, "Methodology for the Generation of Program Test Data", IEEE Trans. on Computers C-24,5 (May 1975), pp.554-560.

[10] A.Lew and D.Tamanaha, "Decision Table Programming and Reliability", in Proc. of the Second International Conf. on Reliable Software (1976), pp. 345-349.

[11] L.J.Osterweil and L.D.Fosdick, "Some Experiences with DAVE - A Fortran Program Analyzer", AFIPS Conference Proceedings 45 (1976), pp. 909-915.

[12] C.V.Ramamoorthy, S.F.Ho, and W.T.Chen, "On the Automated Generation of Program Test Data", IEEE Trans. on Soft. Eng. SE-2,4 (Dec. 1976), pp.293-300.

APPENDIX

This appendix lists the output generated by EXPER and an associated experimental subsystem for performing the correlated k-order mutation experiment on program TRIANG.

LISTING OF THE PROGRAM BEING MUTATED

```
      SUBROUTINE TRIANG(I,J,K,MATCH)
      IF(I .LE. 0 .OR. J .LE. 0 .OR. K .LE. 0) GOTO 500               1  2
      MATCH = 0                                                          3
      IF(I .NE. J) GOTO 10                                            4  5
      MATCH = MATCH + 1                                                  6
 10   IF(I .NE. K) GOTO 20                                            7  8
      MATCH = MATCH + 2                                                  9
 20   IF(J .NE. K) GOTO 30                                           10 11
      MATCH = MATCH + 3                                                 12
 30   IF(MATCH .NE. 0) GOTO 100                                      13 14
      IF(I + J .LE. K) GOTO 500                                      15 16
      IF(J + K .LE. I) GOTO 500                                      17 18
      IF(I + K .LE. J) GOTO 500                                      19 20
      MATCH = 1                                                         21
      RETURN                                                            22
 100  IF(MATCH .NE. 1) GOTO 200                                      23 24
      IF(I + J .LE. K) GOTO 500                                      25 26
 110  MATCH = 2                                                         27
      RETURN                                                            28
 200  IF(MATCH .NE. 2) GOTO 300                                      29 30
      IF(I + K .LE. J) GOTO 500                                      31 32
      GOTO 110                                                          33
 300  IF(MATCH .NE. 3) GOTO 400                                      34 35
      IF(J + K .LE. I) GOTO 500                                      36 37
      GOTO 110                                                          38
 400  MATCH = 3                                                         39
      RETURN                                                            40
 500  MATCH = 4                                                         41
      RETURN                                                            42
      END
```

LISTING OF THE TEST CASES ON WHICH TRIANG PASSES THE MUTANT TEST

TEST CASE NUMBER	LEG LENGTHS I	J	K	TRIANGLE TYPE (N=NOT A TRIANGLE, S=SCALENE, I=ISOSOLESE, E=EQUILATERAL)
1	0	0	0	N
2	3	4	8	N
3	1	1	1	E
4	1	2	2	I
5	3	4	6	S
6	8	4	3	N
7	3	8	4	N
8	2	2	5	N
9	2	5	2	N
10	2	3	2	I
11	5	2	2	N
12	0	1	1	N
13	1	0	1	N
14	1	1	0	N
15	5	9	9	I
16	9	5	9	I
17	9	9	5	I
18	-1	1	1	N
19	1	-1	1	N
20	1	1	-1	N
21	4	5	9	N
22	9	4	5	N
23	4	9	5	N
24	4	4	8	N
25	4	8	4	N
26	8	4	4	N
27	9	5	6	S
28	5	9	6	S
29	5	2	6	S
30	3	9	5	N
31	3	9	7	S
32	10	10	13	I
33	10	13	10	I
34	13	10	10	I
35	6	7	2	S
36	7	2	6	S
37	10	5	6	S

```
FINAL STATISTICS FOR PASSING THE MUTANT TEST ON TRIANG



MUTANT STATE FOR ALL PROGRAM UNITS

    FOR EXPERIMENT "EIGHT.EXP " THIS IS RUN     3

    NUMBER OF TEST CASES =  37

    NUMBER OF MUTANTS =   1026
    NUMBER OF DEAD MUTANTS =     955 ( 93.1%)
    NUMBER OF LIVE MUTANTS =       0 (  0.0%)
    NUMBER OF EQUIV MUTANTS =     71 (  6.9%)

    NUMBER OF MUTATABLE STATEMENTS =      42
    GIVING A MUTANTS/STATEMENT RATIO OF       24.43



MUTANT ELIMINATION PROFILE FOR ALL PROGRAMS

MUTANT TYPE                  TOTAL      DEAD            LIVE            EQUIV

CONSTANT REPLACEMENT            30        30 100.0%       0    0.0%        0    0.0%
SCALAR VARIABLE REPLACEME      126       120  95.2%       0    0.0%        6    4.8%
SCALAR FOR CONSTANT REP.        60        60 100.0%       0    0.0%        0    0.0%
CONSTANT FOR SCALAR REP.       170       168  98.8%       0    0.0%        2    1.2%
SOURCE CONSTANT REPLACEME       36        36 100.0%       0    0.0%        0    0.0%
UNIARY OPERATOR INSERION       205       149  72.7%       0    0.0%       56   27.3%
ARITHMETIC OPERATOR REPLA       63        61  96.8%       0    0.0%        2    3.2%
RELATIONAL OPERATOR REPLA       80        76  95.0%       0    0.0%        4    5.0%
LOGICAL CONNECTOR REPLACE        6         6 100.0%       0    0.0%        0    0.0%
STATEMENT ANALYSIS              42        42 100.0%       0    0.0%        0    0.0%
STATEMENT DELETION              42        42 100.0%       0    0.0%        0    0.0%
RETURN STATEMENT REPLACEM       38        37  97.4%       0    0.0%        1    2.6%
GOTO STATEMENT REPLACEMEN      128       128 100.0%       0    0.0%        0    0.0%
```

```
               RESULTS FOR THE GENERATION OF 2-ORDER MUTANTS OF TRIANG

**********************************************************************
**********************************************************************

 THE FOLLOWING 2-ORDER MUTANT OF TRIANG SUCCEEDED

 MUTANT PHYSICAL RECORD IS    428
    STATEMENT      1 CHANGED FROM
       IF(I .LE. 0 .OR. J .LE. 0 .OR. K .LE. 0) GOTO 500
    TO
       IF(I .LE. -0 .OR. J .LE. 0 .OR. K .LE. 0) GOTO 500

 MUTANT PHYSICAL RECORD IS    726
    STATEMENT     13 CHANGED FROM
 30    IF(MATCH .NE. 0) GOTO 100
    TO
 30    IF(MATCH .GT. 0) GOTO 100

**********************************************************************
**********************************************************************


  WITH THE ORDER AT    2
  THE NUMBER OF CORRELATED MUTANTS OF TRIANG DRAWN WAS  3000
  OF THOSE THE NUMBER OF LIVE DRAWS WERE       1


       PROFILE OF EQUIVALENT COMPONENTS
  NO. EQU MTS      NUMBER DRAWN     NO. SUCCESSFUL

        0                2819                  0
        1                 180                  0
        2                   1                  1


       PROFILE ON METHOD OF 2-ORDER MUTANT FAILURE

   2518  TERMINATED BUT PRODUCED WRONG ANSWERS
      0  HAD AN ARITHMETIC FAULT
      0  HAD AN ARRAY INDEXING ERROR
      0  EXECUTED A TRAP STATEMENT
    137  REFERENCED AN UNDEFINED VARIABLE
      0  ATTEMPTED TO DIVIDE BY ZERO
    152  EXCEEDED THE TIME LIMIT
      0  ATTEMPTED ILLEGAL DATA COERSION
    192  ATTEMPTED TO ALTER A READ ONLY VARIABLE
```

```
        RESULTS FOR THE GENERATION OF 3-ORDER MUTANTS OF TRIANG

 ***********************************************************************
 ***********************************************************************

  THE FOLLOWING 3-ORDER MUTANT OF TRIANG SUCCEEDED

MUTANT PHYSICAL RECORD IS    204
   STATEMENT     29 CHANGED FROM
200   IF(MATCH .NE. 2) GOTO 300
   TO
200   IF(MATCH .NE. K) GOTO 300

MUTANT PHYSICAL RECORD IS    147
   STATEMENT     36 CHANGED FROM
      IF(J + K .LE. I) GOTO 500
   TO
      IF(J + J .LE. I) GOTO 500

MUTANT PHYSICAL RECORD IS    180
   STATEMENT      9 CHANGED FROM
      MATCH = MATCH + 2
   TO
      MATCH = MATCH + K

***********************************************************************
***********************************************************************



   WITH THE ORDER AT    3
   THE NUMBER OF CORRELATED MUTANTS OF TRIANG DRAWN WAS    3000
   OF THOSE THE NUMBER OF LIVE DRAWS WERE      1


        PROFILE OF EQUIVALENT COMPONENTS
   NO. EQU MTS     NUMBER DRAWN     NO. SUCCESSFUL

         0              2743               0
         1               249               1
         2                 8               0
         3                 0               0


        PROFILE ON METHOD OF 3-ORDER MUTANT FAILURE

    2419  TERMINATED BUT PRODUCED WRONG ANSWERS
       0  HAD AN ARITHMETIC FAULT
       0  HAD AN ARRAY INDEXING ERROR
       0  EXECUTED A TRAP STATEMENT
     202  REFERENCED AN UNDEFINED VARIABLE
       0  ATTEMPTED TO DIVIDE BY ZERO
      57  EXCEEDED THE TIME LIMIT
       0  ATTEMPTED ILLEGAL DATA COERSION
     322  ATTEMPTED TO ALTER A READ ONLY VARIABLE
```

```
        RESULTS FOR THE GENERATION OF 4-ORDER MUTANTS OF TRIANG

WITH THE ORDER AT    4
THE NUMBER OF CORRELATED MUTANTS OF TRIANG DRAWN WAS    3000
OF THOSE THE NUMBER OF LIVE DRAWS WERE      0


     PROFILE OF EQUIVALENT COMPONENTS
NO. EQU MTS      NUMBER DRAWN       NO. SUCCESSFUL

      0               2644                    0
      1                338                    0
      2                 18                    0
      3                  0                    0
      4                  0                    0


     PROFILE ON METHOD OF 4-ORDER MUTANT FAILURE

 2308  TERMINATED BUT PRODUCED WRONG ANSWERS
    0  HAD AN ARITHMETIC FAULT
    0  HAD AN ARRAY INDEXING ERROR
    0  EXECUTED A TRAP STATEMENT
  274  REFERENCED AN UNDEFINED VARIABLE
    0  ATTEMPTED TO DIVIDE BY ZERO
   13  EXCEEDED THE TIME LIMIT
    0  ATTEMPTED ILLEGAL DATA COERSION
  405  ATTEMPTED TO ALTER A READ ONLY VARIABLE
```

1978 ASQC TECHNICAL CONFERENCE TRANSACTIONS—CHICAGO

PROGRAM MUTATION AS A TOOL FOR MANAGING LARGE-SCALE SOFTWARE DEVELOPMENT

Richard DeMillo
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

Richard Lipton and Frederick Sayward
Department of Computer Science
Yale University
New Haven, Connecticut 06520

INTRODUCTION

Several approaches to aid in the design, implementation and debugging of large-scale software have recently emerged. Examples are restricted modularization (14), structured programming (14), and program verification (9,10). However helpful they may be to programmers and low-level managers, the effects of these techniques cannot be utilized throughout a software project management hierarchy since they are qualitative rather than quantitative: managers should not be expected to understand code and/or sophisticated mathematics.

In this paper we explain how an important phase of software development, testing, can be managed effectively by use of the new program testing approach known as *program mutation* (15). Program mutation provides as a side effect the qualitative type of information that managers need to monitor software development and personnel performance. The basic idea is: given a program module and its test data, program mutation provides a measure, in terms of a percentage, of how "well" the data actually tests the module. The higher the percentage, the more adequately the program has been tested. A program mutation system produces the percentage and users increase the measure by either augmenting the data in a controlled fashion or by answering "hard" questions about the module which are posed by the system. This process iterates until a satisfactory testing percentage is obtained. Meanwhile, the program mutation system records all the involved information in a data base which can be querried at any time by members at all levels of the project hierarchy to obtain reports containing relevant information on the project's testing status. For example, the project manager may wish to know only the testing percentages of all program modules while a programmer may wish to review in detail some or all of the questions and answers previously recorded for a given module.

In section 2 we detail the theory of program mutation as a program testing tool. Section 3 explains what types of information various members of the project hierarchy would draw from the mutation system and how that information would be used as a management tool. These concepts are illustrated in terms of a hypothetical compiler construction project. Finally, in section 4 we present another application of program mutation: monitoring software procurment.

THE PROGRAM MUTATION METHODOLOGY

Program testing is an inductive science which addresses the following fundamental question:

"If a program is correct on a finite number of test cases,
is it correct in general?"

Finite test data which implies general correctness is called *adequate test data* and since adequate test data cannot in general be derived algorithmically (4) program testing cannot be deductive. Recently, *path analysis* (1,2,5,6) and *symbolic execution* (7,8) have emerged as methods which allow one to gain confidence in one's test data's adequacy. Although as with any inductive science, it is possible to make false inferences with path analysis (3), the basic idea is undeniable: test data which exercises all flowchart control paths of a program at least once must be better than test data which doesn't. Symbolic execution is associated to path analysis since, among other things, it attempts to derive test data which exercises all paths of a program.

Unlike previous software reliability methods, in program mutation we make the following assumption:

> *Experienced programmers* write programs which are either correct or are "almost" correct.

That is, in the mutation terminology,

> If a program is *not* correct, then it is a "mutant" - it differs from a correct program by simple well-understood errors.

There is empirical evidence which supports this natural premise (11).

Boehm has found (12) that errors fall into three categories: clerical, logical, and misunderstanding of specifications. In the above assumption we do not explicitly mention errors due to programmers misunderstanding specifications; rather, it appears we are dealing exclusively with clerical errors. While a system which would solve the clerical error problem would be quite useful, program mutation does even more: indeed, below we explain how the use of the program mutation methodology can lead to the detection of all three error types.

With the "experienced programmer assumption", the mutation method is: take a program P which is correct on some test data T and subject it to a series of *mutant operators*, thereby producing mutant programs which differ from P in very simple ways. For example, if

```
I = I+1
```

is a statement in P, then

```
I = I-1
I = I+2
I = I+0  (i.e., a no-op)
```

are all simple changes which lead to three mutants of P. The mutant programs are then executed on T. If all mutants give incorrect results then it is very likely that P is correct (i.e., we can infer with high confidence that T is adequate). On the other hand, if some mutants are correct on T then we can infer that either:

(1) The mutants are equivalent to P,
(2) The test data T is inadequate, or
(3) The program P is incorrect.

If it cannot be determined that P is incorrect from this information, then T must be augmented and the mutation method re-applied in an attempt to make the non-equivalent mutants which are correct on T subsequently fail. This augmentation process forces the programmer to examine P in detail with respect to the mutants.

At first glance it would appear that if T is determined adequate by mutation analysis, then P might still contain some complex errors which are not explicitly mutants of P. To this end there is a *coupling effect* which states:

> Test data on which all simple mutants fail is so sensitive that it is highly likely that all complex mutants must also fail.

That is, if a program passes tests for all possible simple errors then it has been implicitly tested for all possible complex errors. It is in this effect that the power of program mutation to detect the so-called logical errors of Boehm (12) is revealed. Experiments which substantiate the coupling effect are reported in (13).

Using program mutation as a tool for obtaining reliable software is a highly interactive process whose success depends in part on human judgement. Due to the following issues, the programmer must re-examine in critical detail both his program and its specifications and why he made the decisions that led to the construction of his program. The crucial issues which must be addressed by the users include:

(1) Which mutant operators should be applied to the program?
(2) Are the program and its mutants correct on the given test data?
(3) Is a given mutant equivalent to the program?

It is here that specifications errors are discovered. Note that it is possible for a mutation system to provide the users with information which greatly facilitates resolving these issues: indeed, a mutation system can even resolve them automatically in some cases.

In using a program mutation system, a programmer specifies to the system his program, test data, and the mutant operators he wishes to be applied. The system then generates and executes the mutants on the test data and produces a report indicating which mutants are correct on the given test data. The determination of mutant correctness is done in one of two ways: (1) by direct comparison of the mutant output with the program's output, or (2) by a user-supplied algorithm which examines the output of the mutant. In both cases the system asks the user whether or not the program is acceptable on the test data. However, determination of mutant failure is done by the system.

Upon examining the report, the user may re-run the system and augment his test data in an attempt to make the remaining mutants fail. He may also specify that additional mutant operators be applied to the program. The system produces another report of the same nature as the first for the user to examine. This cycle continues until the user is satisfied that his current test data adequately tests his program.

MANAGEMENT ASPECTS OF PROGRAM MUTATION

Successful large-scale programming projects rely on a hierarchical flow of information and decisions. A fragment of such a project structuring is represented in figure 1. In addition, there is a recognizable time-ordering of events for gathering

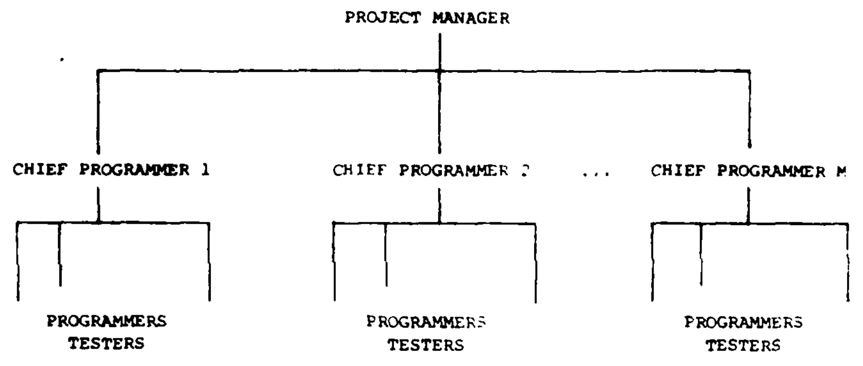


Figure 1. Hierarchical Management Organization

information and making decisions which correlates with the hierarchical management structure. Events such as "decide input file structure", "gather documentation from the submodules of module M1", "begin testing module M5" provide transformations of the programming task, replacing the as yet incomplete project with the next stage as determined by the most current information. The management hierarchy generally parallels the modular decomposition of the programming task. This can be seen directly in figure 2 where we illustrate a decomposition of a multiple pass compiler.

During the test phase of the project the mutation system records a wealth of information in its data base and this data is used to produce reports which directly influence decision-making throughout the project hierarchy. The type of information drawn from the mutation system and its uses vary depending on the project hierarchy level of the querrier. In this section we sketch some possibilities for the three levels illustrated in figures 1 and 2. Additional possibilities can readily be imagined. The general idea is: the higher the querrier is in the project structure, the less programming oriented is the gathered information.

Project Manager's Report

The project manager periodically meets with the chief programmers to evaluate the project's testing status. Also, the assignment of personnel and the evaluation of personnel performance are done at this level. The project manager's report would contain information such as:

(1) The name of each module.

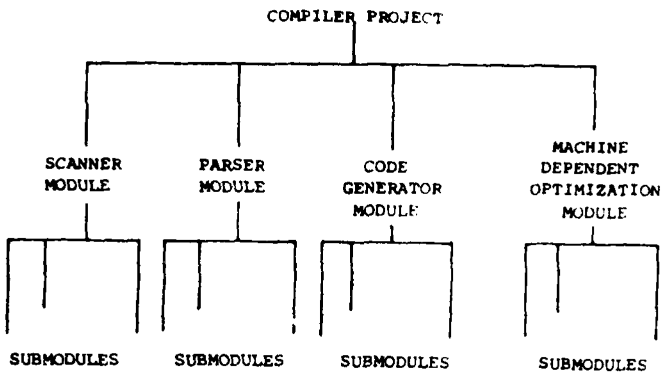


Figure 2. Modular Decomposition of a Multiple Pass Compiler

(2) The chief programmer responsible for each module.
(3) Plots of the mutant elimination percentage vs. time for each submodule.
(4) For each submodule, (a) the number of mutants, (b) the number and the percentage of eliminated mutants, (c) the number and percentage of mutants deemed equivalent, and (d) the number and percentage of non-eliminated mutants.
(5) For each module, the number and type of assigned personnel.
(6) For each submodule, the number and type of assigned personnel.

This information can be used by the project manager to help do the following:

(1) Monitor adherence to the project's testing pert-chart.
(2) Decide whether an acceptable level of testing has been obtained for a given module or submodule.
(3) Re-assignment of personnel to work on modules where the mutant elimination percentage is low.
(4) Rewarding personnel who achieve high mutant elimination percentages.
(5) Pinpointing responsibility for modules which fail after having been judged acceptable.

Chief Programmer's Report

A chief programmer should be familiar with the program code of all the submodules but he doesn't necessarily do any of the programming himself. He meets daily with his subordinate personnel. The type of information contained in a chief programmer's report would include:

(1) The names and program code for each submodule of his module.
(2) The personnel assigned to each submodule.
(3) Plots of the mutation elimination percentage vs. time for each submodule.
(4) The mutant operators being applied to each submodule.
(5) For each submodule, (a) the number of mutants, (b) the number and the percentage of eliminated mutants, (c) the number and percentage of mutants deemed equivalent, and (d) the number and percentage of non-eliminated mutants.
(6) Listings, in coded forms, of mutants determined equivalent.
(7) Personnel responsible for classifying mutants as equivalent.

This information can be used by the chief programmer to do the following:

(1) Suggest to the programmers additional mutant operators for a given submodule.
(2) Ask a programmer to justify his judgement of mutants as equivalent. The chief programmer may want to know, for instance, why it does not matter if a certain variable can be mutated without changing the effect of the submodule. That is, why is his submodule so insensitive to that mutation?

(3) Determine that a given submodule has been acceptably tested and prepare evidence on this decision for presentation to the project manager.

Programmer's and Tester's Report

These personnel are concerned mainly with the details of program code and data and thus their reports will be the most lengthy. The type of information would include:

(1) A listing of the submodule code.
(2) The current test data for the submodule.
(3) The mutant operators currently being applied to the submodule.
(4) For the submodule, (a) the number of mutants, (b) the number and the percentage of eliminated mutants, (c) the number and percentage of mutants deemed equivalent, and (d) the number and percentage of non-eliminated mutants.
(5) Profiles of the information in (4) with respect to the mutant operators currently being applied.
(6) Listings, in coded form, of the non-eliminated mutants.
(7) Listings, in coded form, of the mutants determined equivalent.

This information could be used by programmers and testers to do the following:

(1) Augment the current test data so as to eliminate mutants on the next mutation run.
(2) Augment the set of applied mutant operators for the next mutation run.
(3) Classify non-eliminated mutants as equivalent.
(4) Determine that the submodule has been adequately tested and prepare evidence of this for presentation to the chief programmer.

SOFTWARE PROCURMENT ASPECTS OF PROGRAM MUTATION

Government agencies and profit making industries are currently finding that purchasing software from specialized software vendors is more economical than in-house development. The contracts generally consist of the specifications for the software and a date on which the software and test data on which the software meets the specifications are to be delivered. Occasionally, some test data is given with the specifications.

Two problems for the contractor are apparent in the above scheme: (1) at any time during the contract period the puchaser has no indication as to how "close" the software is to being ready, and (2) upon delivery, although the software works correctly on the supplied test data, there is no way to measure the quality of the purchased software. We see program mutation as a partial solution to the first problem and as a definite solution to the second.

Since program testing is the final stage of software development, a contractor can specify that the vendor indicate at what point testing commences. Assuming that the vendor is using a mutation system, the contractor can monitor the final stage of development by having the vendor periodically report mutant elimination percentages.

To evaluate the delivered software, one can specify in contracts that the test data of modules must eliminate a certain percentage of the mutants with respect to "standard" mutant operators. Here there are many options. Software not passing this quality test may be rejected or there could be a substantial financial penalty to the vendor. In this case it is not essential that the vendor use a mutation system, only that the contractor have one available to evaluate the final product. Also, note that the contractor is not concerned with equivalent mutants; rather, a simple test (which can be entirely computerized) dependent solely on the mutant operators is used. Currently, we have little information on which mutant operators should be employed in this test; however, experiments to answer this question are underway. We have observed empirically (13,15) that the percentage of equivalent mutants tends to be about two.

SUMMARY

Program mutation is an important new tool in the field of program testing which has applications in other fields. Above it has been explained how, unlike other current programming methodologies, a program mutation system can provide quantitative information which can be used throughout the management hierarchy of a large programming project. Furthermore, program mutation has an important application in that it can be incorporated into contracts for software procurment. It provides purchasers of software with a means of measuring the quality of the delivered product.

ACKNOWLEDGEMENT

We acknowledge the work of Tim Budd and the other members of the Yale University Testing Group for help in implementing and experimenting with the prototype mutation system developed at Yale University.

REFERENCES

1. C. V. Ramamoorthy. S. F. Ho, and W. T. Chen, "On the Automated Generation of Program Test Data," *IEEE Transactions on Software Engineering* SE-2,4 (December 1976), pp. 293-300.

2. W. E. Howden, "Methodology for the Generation of Program Test Data," *IEEE Transactions on Computers* C-24,5 (May 1975), pp. 554-560.

3. W. E. Howden, "Reliability of the Path Analysis Testing Strategy," *IEEE Transactions on Software Engineering* SE-2,3 (September 1976), pp. 208-214.

4. J. B. Goodenough and S. L. Gerhart, "Towards a Theory of Test Data Selection," *IEEE Transactions on Software Engineering* SE-1,2 (June 1975), pp. 156-173.

5. J. C. Huang, "An Approach to Program Testing," *Computing Surveys* 7,3 (September 1975), pp. 113-128.

6. E. F. Miller and R. A. Melton, "Automated Generation of Testcase Datasets," in Proceedings of the First International Conference on Reliable Software, *SIGPLAN Notices* 10,6 (June 1975), pp. 51-58.

7. L. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Transactions on Software Engineering* SE-2,3 (September 1976), pp. 215-222.

8. J. King, "Symbolic Execution and Program Testing," *Communications of the ACM* 19,7 (July 1976), pp. 385-394.

9. R. London, "The Current State of Proving Programs Correct," in *Proceedings of the ACM National Conference*, 1972, ACM, New York, pp. 39-46.

10. S. Hantler and J. King, "An Introduction to Proving the Correctness of Programs," *Computing Surveys* 8,3 (September 1976), pp. 331-353.

11. E. A. Youngs, "Human Errors in Programming," *International Journal of Man Machine Studies* 6 (1974), pp. 361-376.

12. B. Boehm, "Software Design and Structuring," in *Practical Strategies for Developing Large Software Systems*, Horowitz (Editor), Addison-Wesley, 1975, pp. 103-128.

13. R. DeMillo, R. Lipton, and F. Sayward, "Hints on Test Data Selection," to appear in *Computer*, April 1978.

14. Special Issue: Programming, *ACM Computing Surveys* 6,4 (December 1974), pp. 209-319.

15. R. DeMillo, R. Lipton, and F. Sayward, "PROGRAM MUTATION: A Method of Determining Test Data Adequacy," submitted to the Third Int. Conf. on Soft. Eng. (1978).

AD-A107 774   GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION A--ETC F/G 9/2
PAPERS ON PROGRAM TESTING,(U)
1979   R A DEMILLO, R J LIPTON, F G SAYWARD   N00014-79-C-0231
UNCLASSIFIED GIT-ICS-79/04   NL

2 OF 3
AD
A107774

STABILITY OF TEST DATA FROM PROGRAM MUTATION

James E. Burns
School of Information and Computer Science
GEORGIA INSTITUTE OF TECHNOLOGY
Atlanta, GA 30332

1. INTRODUCTION

Program testing is an expensive part of program development. A significant portion of this cost may go into the creation of high quality test data. In an active environment, it is rare for a program to go unmodified over a long period. Considerable effort can be saved if test data created for earlier program versions can be shown to be satisfactory for testing new versions.

The second section of this paper briefly introduces a promising new tool for program testing, program mutation. Program mutation has the attractive qualities that it aids in finding good test data sets and also provides a quantitative measure of how good they are. Section 3 describes the experiment performed to test the hypothesis that test data produced by program mutation tends to be stable. The final two sections give the results of the experiment and draw conclusions.

2. PROGRAM MUTATION

Program mutation is a recently developed technique for creating high quality test data. A brief description of the technique is given here, but the reader is referred to references [1-4] for a more complete explanation, especially regarding motivation.

This work was supported in part by U.S. Army Research Office Grant #DAAG29-78-G-0121.

The central idea of program mutation is the construction of a set of "mutants" of the target program. A mutant is a copy of the target program which differs only by a single "mutation". A mutation is a transformation of a program statement in a way which simulates typical program errors. For example, one mutation is to modify the value of a literal constant - the FORTRAN statement "I = I+3" might be changed to "I= I+2". Some mutants may turn out to be equivalent, functionally, to the target program. The remainder should be distinguished from the target program by sufficiently powerful test data. Test data which is able to distinguish all non-equivalent mutants of a target program must thoroughly exercise the program and, hence, provide strong evidence of the program's correctness.

Let P be a program and M(P) be the set of mutants of P. (Note: M(P) depends on the language of P and the set of mutations chosen. We assume a fixed language and a fixed set of mutations for purposes of discussion.) Let Q(P) be the subset of M(P) that are functionally equivalent to P. For a given set of test data, T, an element $m \in M(P)$ is said to be *eliminated* by T if and only if there is at least one element of T which distinguishes m from P, otherwise, m is said to be *live*. Now we may define the *adequacy of T for P* by

$$A(T,P) = 100\% \times \frac{M(P) - L(T,P) - Q(P)}{M(P) - Q(P)}$$

where

$$L(T,P) = \{ m \in M(P) \mid m \text{ is live for P under T} \}$$

When A(T,P) = 100%, we say that T is *adequate* for P. If no element of T can be removed without reducing the adequacy of T, then T is said to be *reduced*.

Adequacy provides a quantitative measure of the thoroughness with which a set of test data exercises the target program. Unfortunately, this measure may be difficult to compute since Q(P) is usually difficult to determine. However, the following approximation to A(T,P) is usually sufficiently accurate since, (empiricially), Q(P) is rarely greater than 5% of M(P):

$$A'(T,P) = 100\% \times \frac{M(P) - L(T,P)}{M(P)}$$

Note that A'(T,P) always approximates A(T,P) from below. Also, if part of Q(P) can be easily determined, the approximation can be improved.

3. STABILITY OF TEST DATA

Intuitively, the stability of a set of test data refers to how powerful it is in testing programs which are slightly modified versions of the program for which the test data was developed. The adequacy measure gives a means of quantifying this concept with the following definition.

Let $\mathcal{P}$ be a set of closely related programs, $\{P_1, P_2, \ldots, P_n\}$. Assume that all the programs in $\mathcal{P}$ will correctly accept the same set of test data, T. Then the *stability of T relative to* $\mathcal{P}$ is given by

$$S(T,\mathcal{P}) = \min_{1 \leq i \leq n} A(T,P_i)$$

We also define an approximation to this measure by

$$S'(T,\mathcal{P}) = \min_{1 \leq i \leq n} A'(T,P_i)$$

The stability of T is, of course, highly dependent on P. We wish to determine whether or not test data produced by program mutation is relatively stable for P chosen to be sufficiently "similar" to the program for which the test data was created. For this experiment, ten sorting algorithms were chosen, (see Appendix A). Since these programs are functionally identical, they are certainly highly similar. There is some motivation for using functionally identical programs in this study since one type of program modification is the replacement of an algorithm with a functionally identical but more efficient one. If test data produced by the program mutation method is stable, high values of the stability measure would be expected. More complex algorithms would be expected to have higher values of the measure since they would tend to require stronger test data.

Each of the algorithms listed in Appendix A was coded into the subset of FORTRAN accepted by the PIlot Mutation System (PIMS) developed at Yale University. PIMS automatically generates the set of mutants which are to be eliminated by test data. Test cases may be entered interactively and tested by the PIMS interpreter against the mutants. The living mutants may be examined through the system to aid in selecting additional test data.

Test data was developed for each program in the set independently. An attempt was made in each case to eliminate all mutants which could not be identified as being functionally equivalent to the original program. (In all but three cases, all non-equivalent mutants were eliminated.) The resulting test set were then reduced to remove any inessential test cases. Finally, the test set for each program was run against each of the other programs to determine the number of mutants eliminated in each case.

4. RESULTS

The raw results of the experiment described above are presented in Table 1. The programs are ordered by the number of mutants produced by PIMS, which is a rough measure of t..eir complexity. Table 2 gives the number of living mutants left by each test set with all mutants which could be determined to be equivalent removed. The adequacy measures, A'(T,P) and A(T,P), and stability measures, S'(T,P) and S(T,P), are given in Tables 3 and 4.

Two sets of test data, (I and J, produced by Quicksort and the Natural Two-way Merge Sort), provided very strong test data with stability measures of over 98%. In fact, the test data from Quicksort was able to eliminate more mutants of the Merge Exchange (H) and the Natural Two-way Merge Sort (J) than the data created using the PIMS system directly. This may result in part from the large number of test cases (14) required by Quicksort.

The remaining test sets did not produce impressive stability measures over this set of programs, although Heapsort (F) did achieve a respectable 88.6%. However, if the two most complex programs (I & J) are removed from the set used to compute the measure, all of the test sets have stability measures near 90%.

5. CONCLUSIONS

If all of the test sets produced in this experiment had proven to have had very high stability measures, we could conclude that there was evidence that test data from program mutation was stable. This

TABLE 1

Number of Live Mutants

	Program / # of Mutants									
	A	B	C	D	E	F	G	H	I	J
Test Set (# cases)	187	240	266	274	282	830	1094	1233	1838	2292
A (4)	8	11	9	12	15	128	78	168	567	1104
B (6)	8	8	6	10	15	127	71	159	547	1087
C (5)	8	10	6	9	10	98	71	154	421	1094
D (5)	8	11	7	9	8	52	72	142	548	555
E (4)	10	15	12	13	8	56	81	166	788	563
F (8)	8	9	6	9	8	23	69	75	247	300
G (6)	8	9	7	10	8	79	50	152	409	1084
H (5)	8	8	6	9	8	25	68	42	246	517
I (14)	9	8	6	9	8	24	50	32	132	54
J (6)	10	12	7	10	8	25	55	43	193	74

TABLE 2

Number of (Live - Equivalent) Mutants

	Program									
Test Set	A	B	C	D	E	F	G	H	I	J
A	0	3	3	3	7	105	28	138	439	1061
B	0	0	0	1	7	104	21	129	419	1044
C	0	2	0	0	2	75	21	124	293	1054
D	0	3	1	0	0	29	22	112	420	512
E	2	7	6	4	0	33	31	136	660	520
F	0	1	0	0	0	0	14	45	119	257
G	0	1	1	1	0	56	0	122	281	1041
H	0	0	0	0	0	2	18	12	118	474
I	1	0	0	0	0	1	0	2	4	11
J	2	4	1	1	0	2	5	13	65	31

TABLE 3

Approximation of Adequacy : % A'(T,P)

	Program										
Test Set	A	B	C	D	E	F	G	H	I	J	S'(T,P)
A	95.7	95.4	96.6	95.6	94.7	84.6	92.9	86.4	69.2	51.8	51.8
B	95.7	99.2	97.7	96.4	94.7	84.7	93.5	87.1	70.2	52.6	52.6
C	95.7	95.8	97.7	96.7	96.5	88.2	93.5	87.5	77.1	52.3	52.3
D	95.7	95.4	97.4	96.7	97.2	93.7	93.4	88.5	70.2	75.8	70.2
E	94.7	93.8	95.5	95.3	97.2	93.3	92.6	86.5	57.1	75.4	57.1
F	95.7	96.3	97.7	96.7	97.2	97.2	93.7	93.9	86.6	86.9	86.6
G	95.7	96.3	97.4	96.4	97.2	90.1	95.4	87.7	77.7	52.7	52.7
H	95.7	99.2	97.7	96.7	97.2	97.0	93.8	96.6	86.6	77.4	77.4
I	94.7	99.2	97.7	96.7	97.2	97.1	95.4	97.4	92.8	97.6	92.8
J	94.7	95.0	97.4	96.4	97.2	97.0	95.0	96.5	89.5	96.8	89.5

TABLE 4

Adequacy : % A(T,P)

	Program										
Test Set	A	B	C	D	E	F	G	H	I	J	S(T,P)
A	100	98.7	98.8	98.9	97.4	87.0	97.3	88.5	74.3	52.8	52.8
B	100	100	100	99.6	97.4	87.0	98.0	89.3	75.5	53.6	53.6
C	100	99.2	100	100	99.3	90.7	98.0	89.7	82.9	53.1	53.1
D	100	98.7	99.6	100	100	96.4	97.8	90.7	75.4	77.2	75.4
E	98.9	97.0	97.6	98.5	100	95.9	97.0	88.7	61.4	76.9	61.4
F	100	99.6	100	100	100	100	98.7	96.3	93.0	88.6	88.6
G	100	99.6	99.6	99.6	100	93.1	100	89.9	83.6	53.7	53.7
H	100	100	100	100	100	99.7	98.3	99.0	93.1	78.9	78.9
I	99.4	100	100	100	100	99.9	100	99.8	99.8	99.5	99.4
J	98.9	98.3	99.6	99.6	100	99.7	99.5	98.9	96.4	98.6	98.3

would imply that it would not be necessary to re-analyze a program every time a small change was made to it. The negative result implies instead that the test data may not be stable, especially if the change made to a program increases it complexity. Thus, it is prudent to perform mutation analysis on revisions to programs; however, it is likely that previously derived test data will provide a good starting point for mutant elimination. In many cases it will be unnecessary to generate any new test cases at all.

References

[1] Budd, T. A. and R. J. Lipton, "Mutation Analysis of Decision Table Programs," Proc. of the 1978 Johns Hopkins Conf. on Information Systems and Sciences, p. 346-349.

[2] Budd, T. A., R. J. Lipton, F. G. Sayward and R. A. DeMillo, "The Design of a Prototype Mutation System for Program Testing," Proc., 1978 NCC, p. 623-628.

[3] DeMillo, R. A., R. J. Lipton and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," Computer 11(4), (April 1978), p. 34-41.

[4] DeMillo, R. A., R. J. Lipton and F. G. Sayward, "PROGRAM MUTATION: A New Approach to Program Testing," Infotech/SRA State of the Art Report : Program Testing (to appear 1978.)

Volume 7, number 4 INFORMATION PROCESSING LETTERS June 1978

A PROBABILISTIC REMARK ON ALGEBRAIC PROGRAM TESTING

Richard A. DEMILLO
School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, USA

Richard J. LIPTON
Computer Science Department, Yale University, New Haven, CT 06520, USA

Received 8 August 1977; revised version received 27 March 1978

Software reliability, program testing

Until very recently, research in software reliability has divided quite neatly into two – usually warring – camps: methodologies with a mathematical basis and methodologies without such a basis. In the former view, "reliability" is identified with "correctness" and the principle tool has been formal and informal verification [1]. In the latter view, "reliability" is taken to mean the ability to meet overall functional goals to within some predefined limits [2,3]. We have argued in [4] that the latter view holds a great deal of promise for further development at both the practical and analytical levels. Howden [5] proposes a first step in this direction by describing a method for "testing" a certain restricted class of programs whose behavior can – in a sense Howden makes precise – be ***algebraicized.*** In this way, "testing" a program is reduced to an equivalence test, the major components of which become

(i) a combinatorial identification of "equivalent" structures;

(ii) an algebraic test

$$f_1 \equiv f_2 ,$$

where f_i, $i = 1, 2$ is a multivariable polynomial (multinomial) of degree specified by the program being considered.

In arriving at a method for exact solution of (ii), Howden derives an algorithm, that requires evaluation of multinomials $f(x_1, ..., x_m)$ of maximal degree d at $O[(d + 1)^m]$ points. For large values of m (a typical case for realistic examples), this method becomes prohibitively expensive.

Since, however, a test for reliability rather than a certification of correctness is desired, a natural question is whether or not Howden's method can be improved by settling for less than an exact solution to (ii).

We are inspired by Rabin [6] and, less directly, by the many successes of Erdös and Spencer [7] to attempt a *probabilistic* solution to (ii). Using these methods, we show that (ii) can be tested with probability of error ϵ * with only $O(g(\epsilon))$ evaluations of multinomials, where g is a slowly growing function of only ϵ. In particular, 30 or so evaluations should give sufficiently small probability of error for most practical situations. The remainder of this note is devoted to proving this result.

Let us denote by $P_{\neq 0}(m, d)$ the class of multinomials

$$f(x_1, ..., x_m) \not\equiv 0$$

(over some arbitrary but fixed integral domain) whose degree does not exceed $d > 0$. We define

$$P(m, d, r) = \min \text{Prob}\{1 \leqslant x_1 \leqslant r, f(x_1, ..., x_m) \neq 0\}$$
$$f \in P_{\neq 0}(m, d) .$$

* See Rabin's account of algorithms that may err with fixed probability [6].

We think of $P(m, d, r)$ as the minimal relative frequency with which witnesses to the non-nullity of a multinomial of the appropriate kind can occur in the chosen interval. We will derive a lower bound p for $P(m, d, r)$. Then $(1 - p)$ is an upper bound on the error in selecting a random point from the m-cube. We then iterate the procedure by t independent random selections to obtain a small probability of error $(1 - p)^t$. Notice, in particular, that since a polynomial of degree d has at most d roots (ignoring multiplicity), the largest probability of finding a root must be at least the probability of finding a root by randomly sampling in the interval $1 \leqslant x_1 \leqslant r$; thus

$$P(1, d, r) \geqslant 1 - d/r .$$

Now, consider some

$$f(x_1, ..., x_m, y) \not\equiv 0$$

of degree at most d. But there are then multinomials $\{g_i\}_{i \leqslant d}$, not all $\not\equiv 0$, such that

$$f(x_1, ..., x_m, y) = \sum_{i=0}^{d} g_i(x_1, ..., x_m) y^i .$$

Let us suppose that $g_k \in P_{\neq 0}(m, d)$. Thus

$$\begin{aligned} &\mathrm{Prob}\{1 \leqslant x_i \leqslant r, f(x_1, ..., x_m, y) \neq 0\} \\ &\quad\geqslant \mathrm{Prob}\{g_k(x_1, ..., x_m) \neq 0 \text{ and } y \text{ is not a root}\} \\ &\quad\geqslant P(m, d, r)(1 - d/r) . \end{aligned}$$

Continuing inductively, we obtain a lower bound in $P(m, d, r)$ as follows:

$$P(m, d, r) \geqslant (1 - d/r)^m . \tag{1}$$

But

$$\lim_{m\to\infty} (1 - d/r)^m = \lim_{m\to\infty} \left[1 + \frac{1}{m}\left(\frac{-dm}{r}\right)\right]^m$$

$$= \exp(-dm/r) \tag{2}$$

Combining (1) and (2), we have for large m, $r = dm$,

$$P(m, d, dm) \geqslant e^{-1} .$$

Thus, with t evaluations of f for independent choices of points from the m-cube with sides $r = dm$, the probability of missing a witness to the non-nullity of $f(x_1, ..., x_m)$ is at most

$$(1 - e^{-1})^t .$$

Table 1 shows the probable error in testing $f \equiv 0$ by t evaluations of f at randomly chosen points for some typical values of d, m, r, t. Notice that for $dm = r$, $t = 30$, this is already $<10^{-5}$.

References

[1] Z. Manna, Mathematical Theory of Computation (McGraw-Hill, New York, 1974).

Table 1
Probable error in testing $f(x_1, ..., x_m) \equiv 0$ (degree $\leqslant d$) by t random evaluations in $\{1, ..., r\}$

$[1 - P(m,d,r)]^t$						
dm	r	$t + 10$	$t = 20$	$t = 30$	$t = 50$	$t = 100$
10	10	10×10^{-3}	106×10^{-6}	1×10^{-6}	109×10^{-12}	12×10^{-21}
20	10	233×10^{-3}	54×10^{-3}	13×10^{-3}	695×10^{-6}	483×10^{-9}
50	10	935×10^{-3}	873×10^{-3}	816×10^{-3}	713×10^{-3}	509×10^{-3}
10^2	10	~1	~1	~1	~1	~1
10	10^2	61×10^{-12}	$<10^{-20}$	$<10^{-20}$	$<10^{-20}$	~0
20	10^2	38×10^{-9}	1×10^{-15}	$<10^{-20}$	$<10^{-20}$	~0
50	10^2	88×10^{-6}	8×10^{-9}	704×10^{-15}	$<10^{-20}$	$<10^{-20}$
10^3	10^2	~1	~1	~1	~1	~1
10	10^3	$<10^{-20}$	$<10^{-20}$	$<10^{-20}$	~0	~0
20	10^3	9×10^{-18}	$<10^{-20}$	$<10^{-20}$	~0	~0
50	10^3	76×10^{-15}	$<10^{-20}$	$<10^{-20}$	~0	~0

[2] J.R. Brown, M. Lipow, Testing for software reliability, Intern. Conf. in Reliable Software, SIGPLAN Notices, 10, 6, (June 1975) 518–527.

[3] A.I. Llewelyn, R.F. Wilkins, The testing of computer software, 1969 Conf. on Software Engineering, 189–199.

[4] R.A. DeMillo, R.J. Lipton, A.J. Perlis, Social processes and proofs of theorems and programs, Fourth ACM Symposium in Principles of Programming Languages (to appear in CACM).

[5] W.E. Howden, Algebraic program testing, Computer Science Technical Report No. 14 (November 1976), UC-San Diego, La Jolla, CA.

[6] M.O. Rabin, Probabilistic algorithms, in: J. Traub, ed., Algorithms and Complexity (Academic Press, New York, 1976) 21–40.

[7] P. Erdos, J. Spencer, Probabilistic Methods in Combinatorics (Academic Press, New York, 1974).

MUTATION ANALYSIS OF DECISION TABLE PROGRAMS

Timothy A. Budd
Richard J. Lipton

Department of Computer Science
Yale University
New Haven, Connecticut 06520

I. INTRODUCTION

For years computer programmers have been testing programs on small sets of test data in order to infer correctness, on the assumption that if the program works correctly on a certain set of "hard" test data, it will probably work correctly on any data. Of course, most programmers have little more than an intuitive idea of what represents "hard" data. Expressed in such vague terms, such faith is obviously not well founded.

Recently interest has increased in formalizing the theoretical aspects of program testing [4,7]. As in this earlier work, we can give a formal interpretation to the ideas of program testing as follows: We can view a program R as being a function from an input domain to an output domain. For every program R we can assume there exists a function F which R was intended to compute. The correctness question can then be phrased as "is R a realization of the function F ?"

Previous work has been directed toward finding a predicate P over the space of programs and input domains such that, for a given function F and program R , if a set of test cases T satisfies P(T,R) and R correctly computes F on T , then we can infer that R is a realization of F . The results of [4] show that such a predicate must always exist. However, that does not imply that testing is easy: it need not be the case that T is finite or that P be decidable.

To give a more concrete example: For any logical expression we can construct a program which computes the value of that expression over a set of boolean inputs. Suppose we have such a program and wish to assert that it is the constant function FALSE. Any predicate which will satisfy the above will imply that either 1) T must in some cases be exponential in the size of the program, or 2) P must solve an NP-hard problem [1].

Given then the difficulty of the task of finding test data that will without doubt show that a program is correct, the goals of mutation analysis are much simpler. In mutation analysis we define a predicate P as before, and in addition we have some measure M of the "syntactic distance" between programs. The theorem we then hope to prove is: Given a function F and program R then if a set of test cases T satisfies P(T,R) and R correctly computes F on T then either 1) R is a correct realization of F , or 2) R is (in terms of the measure) very far away from ALL programs that correctly relaize F .

Informally, the latter condition can be stated by saying if R is incorrect, it is incorrect in a very radical fashion. Of course, the truth and/or utility of such theorems depends very strongly on the predicates and measures chosen. Previous papers on mutation analysis have demonstrated a large body of empirical evidence showing that for a very realistic problem domain (FORTRAN programs) there is a relatively easy to verify predicate and a natural measure for which that theorem seems to be true [2]. The present paper presents for the first time analytical results for a simpler problem domain.

This paper proves analytically a theorem similar to the one described above, but for the problem domain of decision table programs. In section II we formally define decision table programs. In section III we define a measure on the space of decision table programs, and introduce the concept of mutation. Section IV contains the main results of the paper, leading up to a formal theorem similar to the one given above. In section V we comment on the complexity of mutant analysis, as opposed to explicit enumeration, and conclude the paper with some open problems and directions for future research.

II. DECISION TABLE PROGRAMS

Decision tables are a method of organizing rules that specify the conditions under which certain actions are to be performed. Decision tables are chiefly used in business and data processing applications [5,6], although in [4] they are used as a means of organizing test data selection predicates.

We can abstract the notion of a Decision Table Program as follows [5,6]. We have first a set of n Conditions and a set of p Actions. The conditions are a set of predicates in some language, say English or FORTRAN. The actions are given in the same language, and are assumed to be independent; that is, the results of executing any subset of the actions are independent of the order in which they are executed. (Alternatively, we could merely define an ordering on the actions.) The actions are also assumed to be detectable; that is, given input and output data, it is possible to tell precisely what actions were executed on the input to produce the output.

The decision table itself is then composed of two matrices: an n by m condition matrix and a p by m action matrix. We say the program contains m RULES where each rule corresponds to a cross section along columns of the condition and action matrices.

Elements of the condition matrix contain one of three values: Y, N or * (read YES, NO and DON'T CARE). Elements of the action matrix contain one of two values: X or blank.

To execute the program on a selected data point we proceed as follows: first we evaluate each of the conditions on the data, forming a vector of size n containing YES/NO values. We then consider each of the M rules (in some unspecified order). If any rule is SATISFIED in the sense that for each position in the rule that contains a Y the data satisfies the indicated condition, and fails to satisfy the conditions associated with a N, then the actions part of the rule is executed.

If for each possible data item there is at least one rule that can be satisfied we say the decision table is COMPLETE. We say it is CONSISTENT if there is at most one rule. There are mechanical methods to determine whether an arbitrary decision table program is complete and or consistent [5].

We can assume that no two rules specify exactly the same set of actions. We can do this without loss of generality since two rules that specify the same actions can be combined with at worst the addition of one new condition row.

III. ERRORS AND MUTATIONS

We will say a program is correct if it correctly realizes the function it was intended to. A program is incorrect if it is not correct; that is, there is at least one point at which the program and the function compute differing results.

Given a decision table program P , let S be the set of all decision table programs having the same conditions, actions and number of rules as P . The definition implies that each program in S can differ from P only in the entries it contains in its matrices.

We will say a program P is <u>radically incorrect</u> if not only does it not correctly compute the function it was intended to, but no program in S computes this function either. A radically incorrect program cannot just have a few table entries wrong, but must be wrong in its conditions, actions, or in the number of rules it contains.

Define a subset M of S (in [2] these are called the <u>mutants</u> of P) to be the set of programs formed by making changes to a <u>single</u> entry in the tables representing P . We can classify these changes into four types as follows:

TYPE 1 CHANGE:	A Y or N entry is changed to a *.
TYPE 2 CHANGE:	A Y is changed to a N or vice versa.
TYPE 3 CHANGE:	A * is changed to a Y or N.
TYPE 4 CHANGE:	An X is changed to a blank or vice versa.

Notice that even if P is complete and/or consistent, members of M need not share this property.

It may happen that certain members of M will be equivalent to P , that is, will evaluate identically to P on all inputs. An equivalent mutant cannot be produced by a type 2 change, since if a type 2 change is made the set of values that satisfy the conditions of the original rule is totally disjoint from that which satisfy the altered rule. Notice also that, by the assumption of detectability, no type 4 change can produce an equivalent mutant. If any equivalent mutant can be produced by a type 1 change, let P be this mutant and repeat the previous procedure. After a bounded number of such iterations, we then have a program P equivalent to the original such that the only equivalent mutants of P are produced by a type 3 change.

Having accomplished the procedure described in the previous paragraph, we construct a test set T such that every mutant that is not equivalent to P differs on at least one data point d in T . We shall have more to say about the construction of T in section IV. Such a test set is called <u>adequate</u> in [2].

We will now consider what the construction of such a test set tells us about P .

IV. CORRECTNESS RESULTS

In this section we will denote the decision table program under consideration by P . We denote by P_i the *ith* rule, that is, the cross-section of the condition and action matrices taken along the *ith* column.

For brevity we will state the following conditions once. They are assumed to hold in all theorems given.

1) P is consistent, although it need not be complete.
2) The only mutants equivalent to P are produced by a type 3 change.
3) Each element d of the test set T satisfies a rule, and the results of P on d are correct.

LEMMA 1: For each rule P_i in P , there exists a data point d in T satisfying the conditions of P_i .

PROOF: If we assume to the contrary that no data item satisfies some rule, then the action portion of that rule can be mutated in any fashion with no perceptible change, contradicting the assumption concerning the construction of T .

THEOREM 1: No program P' in S that differs from P by at least one type 4 change can evaluate correctly on each data point in T .

PROOF: The proof is a simple pigeon-hole argument. By Lemma 1, there is at least one data item that executes every rule. By assumption, each rule's actions are unique and detectable, hence any program that evaluates correctly on T must contain at least the m action parts of P . But no program in S can contain more rules, hence the result follows.

Theorem 1 implies a one-to-one correspondence between rules in P and rules in any other program in S that evaluates correctly on T . We will use this fact implicitly in what follows.

LEMMA 2: Any program P' in S that is not equivalent to P but that evaluates correctly on T must contain at least one change that, by itself, would produce a non-equivalent mutant.

PROOF: Assume that P' and P differ by changes that, by themselves, construct equivalent mutants. By construction, we see that P' and P differ by type 3 changes. But the fact that these are equivalent changes implies that within each rule the conjunction of the conditions P and P' have in common is sufficient to imply the conditions at each of the disputed points, hence P must be equivalent to P' .

The remainder of this section is devoted to showing by cases that there cannot be any program P' in S that is not equivalent to P but that evaluates correctly on T .

Assume we have a program P' in S that is not equivalent to P but that evaluates correctly on T . By Lemma 2, there must be at least one change between P and P' that if made to P would produce a non-equivalent mutant. Let P* be the mutant so formed, with P_1 (P^*_1) indicating the single rule that has been altered.

Theorem 1 tells us that the change could not have been of type 4; the next three theorems show us it cannot have been of types 1, 2 or 3 either.

THEOREM 2: The difference between P and P* cannot be a type 3 change.

PROOF: Assume we have a program for which the hypothesis holds but deny the conclusion. Since P* is not equivalent to P there exists an element d in T such that P(d) and P*(d) differ. But this can only happen if d satisfies the conditions associated with P_1 but not those of P^*_1 . But one can see then that no other change that can be made to P^*_1 will allow d to satisfy its conditions. Since P(d) executed the correct actions and no other rule have the same actions, a contradiction is obtained.

THEOREM 3: The difference between P and P* cannot be a type 2 change.

PROOF: Assume as before we have a program for which the hypothesis is true but deny the conclusion. By Lemma 1, there exists some element d in T that satisfies the conditions in the original rule. Since d cannot possibly satisfy any rule that includes the change under consideration, the fact that we were satisfied with the actions of P on d and no other rules can have these same actions gives us a contradiction.

THEOREM 4: The difference between P and P* cannot be a type 1 change.

PROOF: Assume we have a program for which the hypothesis is true but deny the conclusion. By construction, there exists an element d in T such that P and P* differ, but this can only mean that d satisfies P^*_1 and not P_1 . Since we were satisfied with P(d) there must exist at least one more change of type 2 or 3 between P_1 and the associated rule in P' which allows us to reject d in P' . Furthermore, this change cannot produce an equivalent mutant. But using Theorems 2 and 3 this gives us a contradiction.

Combining Theorems 1-4 then gives us the main result of this paper.

THEOREM 5: If P evaluates correctly on T then either P is correct or it is radically incorrect.

V. THE COMPLEXITY OF MUTANT ANALYSIS

If we think of each of the n conditions as dividing the space of possible test cases in two, there are then possibly 2^n potential categories a test case could fall into. A test procedure that operated by explicitly constructing a representative from each of these categories might then require an exponential number of test cases (in the size of the matrix). We shall see that mutation analysis requires significantly fewer test cases.

It is not difficult to see that if it is possible to differentiate a program P from a mutant P* , then it is possible to differentiate it with a single test case. Since there are at most 2nm+np mutants, we have the following theorem.

THEOREM 6: If for a given program P there exists an adequate test set T with respect to the mutant operations, then there exists a test set T' with no more than 2nm+np elements that is also adequate.

The theorem is strengthened by the fact that the constructive method of mutant testing, that is, choosing a mutant and finding a test case to eliminate it, results in a test set of no more than the indicated size. Furthermore, it is probable that the test set will be much smaller, since empirical evidence suggests that a single test case may eliminate a large number of mutants [3].

VI. CONCLUSIONS

In this paper it has been shown that by using mutation analysis a relatively small set of test cases (linear in the size of the decision table, versus exponential for explicit enumeration) can be used to infer a very strong conclusion concerning the correctness of a program. We can show, using our methods, that if a program satisfies these test cases then if it is incorrect, it is incorrect in a very dramatic fashion, and it may be possible, using other methods (say specification), to insure that this is not the case.

This result is in striking contrast to the usual view, which holds that testing is of almost no help in showing a program correct. In view of the complex problems associated with program proving, we fell it makes good economic sense to investigate the capabilities of testing performed in a systematic and rational fashion.

These results suggest a paradigm for research in other models of programming. Possibilities for future work are linear recurrences (LISP type functions), or partial recursive functions.

On the other hand, in [2] a number of empirical observations of FORTRAN programs are given that fortify the hope that a theory along the lines of the one presented here might be developed for that problem domain.

Taken all together, this data suggests that in the future mutation analysis may become an important new tool in the field of program testing.

ACKNOWLEDGEMENTS: We would like to thank Fred Sayward for a careful reading of an earlier draft, and Richard Ladner for helping to simplify some of the theorems.

[1] A. Aho, J. Hopcroft and J. Ullman.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, Reading, Mass., 1976.

[2] R. DeMillo, R. Lipton and F. Sayward.
PROGRAM MUTATION: A Method of Determining Test Data Adequacy.
To be presented at the ONR and Navy Computing Labs workshop on Software Technology Transfer (April 1978).

[3] R. DeMillo, R. Lipton and F. Sayward.
"Hints on Test Data Selection."
To appear in *Computer*, Vol. LL(4), April 1978.

[4] J.B. Goodenough and S.L. Gerhart.
"Towards a Theory of Test Data Selection."
IEEE Tran. on Software Engineering, SE-1,2 (June 1975).

[5] M. Montalbano.
Decision Tables.
Science Research Associates, Chicago, 1974.

[6] S.L. Pollack, H.T. Hicks and W.J. Harrison.
Decision Tables: Theory and Practice.
John Wiley Sons, New York, 1971.

[7] R.T. Yeh (ed.).
Current Trends in Programming Methodology, Vol. II.
Prentice-Hall, Englewood Cliffs, N.J., 1977.

This work was partially supported by ONR Grant N00014-75-C-0752 and NSF Grant MCS-76-81486.

PROVING LISP PROGRAMS USING TEST DATA

Timothy A. Budd

Richard J. Lipton

Yale University

Department of Computer Science

New Haven, Ct.

and

University of California at Berkeley

Computer Science Division

Berkeley, Calif.

1. INTRODUCTION

An idea proposed in [1] is the concept of proving individual programs correct with respect to some larger class of programs. That is, instead of proving a program correct we prove that either a) the program is correct, OR b) no program in this class realizes the intended function. It is assumed that most programmers at least know if the function they are trying to compute can be realized in some large class of programs, and therefore from a theoretical point of view the introduction of this disjunction may make the task of validating programs vastly easier.

A previous paper has analysed programs written in a decision table format [4]. In this paper we will be concerned with lisp programs composed of CAR, CDR and CONS with lisp predicates composed of CAR, CDR and ATOM. Similar classes of programs have been studied in [5,6,7].

Associated with each S-Expression X we can construct a binary tree as follows: Consider the infinite binary tree where each left arc is marked CAR and each right arc CDR (call this the complete CAR/CDR tree.) Starting with X at the root of the tree, travel down each arc in turn taking the appropriate CAR or CDR. Prune the complete tree each time you reach an atom. The resulting finite binary tree will be called the projection of X (or PROJ[X]). An example is shown in figure 1. Notice the PROJ[X] is a representation of the structure of X, and in invariant under the renamings of the atoms of X.

We can define a relation $\leq$ as follows. Given two S-expressions X and Y we will say $X \leq Y$ if PROJ[x] is the intersection of PROJ[X] and PROJ[Y]. Using this relation one can show the set of lisp structures form a lattice. (The proofs can be adapted from Summers[7], although he defines the projection slightly differently.)

We will make the convention that all S-Expressions (we will use the less clumsy expression point) have unique atoms. Certainly if two programs agree on all such points they must agree on all inputs. Hence we can do this without loss of generality.

We will call a lisp program a Selector program if it is composed of just CAR and CDR. We will call it a Straight line program if it is a selector program or is formed by CONS on either selectors or other straight line programs. We will call it a Predicate program if it has the following form

$$\text{COND(ATOM}(G_i(X)) \rightarrow P_1(X)$$
$$T \rightarrow P_2(X)\ )$$

Where the G's are selectors and the P's are straight line programs or other predicate programs.

Assume we have a function F which we know can be computed by a program in some schemata class S.

We have a program P in S which we wish to show computes F. We assume we have some method of verifying that P(X)=F(X) on a finite number of test cases (say by hand calculation.) We wish to show that there exists a finite set of test cases T such that if P correctly computes F on every element of T then either 1) P correctly computes F for all inputs, or 2) no program in the schemata class S correctly computes F. This goal is similar to that of mutation analysis [1-4].

Call such a test set Adequate.

We then wish to discover conditions under which we can construct adequate test data.

2. STRAIGHT LINE PROGRAMS

We will say a program P(X) is Well formed if for every occurrence of the construction CONS(A,B) it is the case that A and B do not share an immediate parent in X. The intuitive idea of the definition should be clear: a program is well formed if it is not doing any more work then it needs to. Notice that being well formed is an observable property of programs, independent of testing.

We can define a measure of the complexity of straight line programs by their CONS-depth, where

CONS-depth is defined as follows:

1) The CONS-depth of selector function is zero.

2) The CONS-depth of a straight line program $P(X) = CONS(P_1(X), P_2(X))$ is $1+MAX($ CONS-depth$(P_1(X))$, CONS-depth$(P_2)))$.

LEMMA 1: If any two selector programs compute identically on any point X, they must compute identically on all points.

PROOF: The only power of a selector program is to choose a subtree out of its input and return it. We can view this process a selecting a position in the complete CAR/CDR tree and returning the subtree rooted at that position. Since there is a unique path from the root to this position, there is a unique predicate which selects it out. Since atoms are unique by merely observing the output we can infer the subtree which was selected. The result then follows.

LEMMA 2: If two well formed programs compute identically on any point then they must have the same CONS-depth.

PROOF: Assume we have two programs P_1 and P_2 and a point X such that $P_1(X) = P_2(X)$ yet the CONS-depth(P_1) < CONS-depth(P_2). This then implies that there is at least one subtree in the structure of P_2 which was produced by CONSing two straight line programs while the same subtree in $P_1(X)$ was produced by a selector. But then the objects P_2 CONSed must have an immediate ancestor in X, contradicting the fact that P_2 is well formed.

THEOREM 1: If two well formed straight line programs agree on any point X then they must agree on all points.

PROOF: The proof will be by induction on the CONS-depth. By lemma 2 any two programs which agree at X must have the same CONS-depth. By lemma 1 the theorem is true for programs of CONS-depth zero. Hence we will assume it is true for programs of CONS-depth n and show the case for n+1.

If program P_1 has CONS-depth n+1 then it must be of the form CONS(P_{11},P_{12}) where P_{11} and P_{12} have CONS-depth no greater then n. Assume we have two programs P_1 and P_2 in this fashion. Then for all Y:

$$P_1(Y) = P_2(Y) \text{ IFF}$$

$$\text{CONS}(P_{11}(Y),P_{12}(Y)) = \text{CONS}(P_{21}(Y),P_{22}(Y)) \text{ IFF}$$

$$P_{11}(Y) = P_{21}(Y) \text{ and } P_{12}(Y) = P_{22}(Y)$$

Hence by the induction hypothesis P_1 and P_2

must agree for all Y.

We define a test point to Generic if by itself it constitutes an adequate test set as defined in the introduction.

Corollary: For any well formed straight line lisp program, and unique atomic point for which the function is defined is generic.

3. PREDICATE PROGRAMS

We can view the structure of a predicate program as a binary tree. Associated with each interior node is a predicate and associated with each leaf is a straight line program (see figure.)

We will call a predicate program Well formed if

1) each of the straight line programs associated with each leaf are well formed, and

2) for each leaf on the space of all possible inputs there is at least one item which passes all conditions leading to that leaf and causes the associated straight line program to be executed.

Notice that whether a program is well formed or not is an observable fact independent of testing.

For notation we will denote the leaves going from left to right by l_i $i=1,..n$. Let e_i $i=1,..n$ be the set of straight line programs associated with the leaves. We will assume that for no i,j $i \neq j$ is it the case that e_i is equivalent to e_j. Notice again theorem 1 gives us an effective method to test this.

Given a well formed predicate program P is S we construct a set of n data points d_1, ... d_n such that d_i follows the path to leaf l_i and executes the program e_i correctly. Call this set T_1. There is an obvious effective procedure to generate such a test set.

LEMMA 3: Given any well formed program P' in S which evaluates correctly on each element of T, at least one data point d_i in T must exercise every straight line leaf program in P'.

PROOF: Assume we have a program P' satisfying the hypothesis but for which the conclusion is false. By the pigeon hole principle there must be at least two points d_i and d_j which were evaluated by different leaves in P but which are evaluated by the same leaf in P'. Let f denote the straight line program which evaluates these points in P'. Since the d points are generic this implies that e_i is equivalent to f. But also e_j is equivalent to f. Hence e_i must be equivalent to e_j which is a contradiction.

Corollary: Given any well formed program P' in S which evaluates correctly on each element of T, the leaf programs of P' are simply a permutation of those of P.

It might seem that exercising all the paths of P' is sufficient to show it is equivalent to P. But this is not the case. We might simply have consistently chosen the right path for the wrong reason. To rule out this possibility requires a more stringent set of test cases. We construct this test set in the following manner.

For each leaf l_i and for each element d_j in T_1 construct a point d_{ij} in the following way. Consider the infinite CAR/CDR tree. color each point RED which is tested and found to be atomic on the path leading to the leaf l_i. Color the points which are tested and found to be non atomic BLUE. As long as it is not contained in a subtree rooted at a red point and does not contain a blue point in its subtree, color a point red if it is atomic in d_j. AS long as it is not contained in a subtree rooted at a red point, color a point blue if it is not atomic in d_j. d_{ij} is then the smallest unique atomic point where the red colored vertexes are atomic and the blue vertexes non atomic.

Denote by T the set T_1 augmented with these points.

THEOREM 2: Any well formed program P' in S which agrees with P on T must agree with P on all points.

PROOF: Assume we have a program P which satisfies the hypothesis, yet there is a point X such that P(X) and P'(X) differ.

The point X must be evaluated by some leaf l_i in P, hence it must satisfy all the constraints associated with that leaf.

This point is also evaluated by a leaf program e_k in P'. By lemma 6 some data item d_j in T also executes this leaf program. This implies that no matter what the constraints are on this path in P' (and we make no assumptions about what they might be) they cannot interfere with the constraints along the path leading the l_i.

But this then necessarily implies that point d_{ij} would be evaluated by e_i in P and e_k in P' where $k \neq$ to i. Since d_{ij} is also generic using the earlier theorems a contradiction is obtained.

Corollary: There is an effective procedure to construct an adequate test set for predicate programs.

4. RECURSIVE PROGRAMS

We will define a class of programs (δ_n) as follows:

The input to the program shall consist of two sets of variables: Selector variables , denoted x_1, ... x_m and Constructor variables , denoted y_1, ...y_p.

a program will consist of two parts, a program body and a recurser.

A program body consists of n statements, each statement composed of two parts. The first part is a Predicate of the form ATOM($t(x_i)$) where $t(x_i)$ is a selector function and x_i a selector variable. The second part is a straight line output function over the selector and constructor variables.

A recurser is divided into two parts. The constructor part is composed of p assignment statements for each of the p constructor variables where y_i is assigned a straight line function of the selector variables and y_i. The selector part is composed of m assignment statements for the m selector variables so that x_i is assigned a selector function of itself. The following diagram should give a more intuitive picture of this class of programs.

Program $P(x_1,\ldots x_m,y_1,\ldots y_p)$ =

$p_1(x_{i1}) \rightarrow f_1(x_1,\ldots x_m,y_1,\ldots y_p)$

$p_2(x_{i2}) \to f_2(x_1,\ldots x_m, y_1,\ldots y_p)$

...

$p_n(x_{in}) \to f_n(x_1,\ldots x_m, y_1,\ldots y_p)$

$y_1 \leftarrow g_1(y_1, x_1,\ldots x_m)$

...

$y_p \leftarrow g_p(y_p, x_1,\ldots x_m)$

$x_1 \leftarrow h_1(x_1)$

...

$x_m \leftarrow h_m(x_m)$

Given such a program, execution proceeds as follows: Each predicate is evaluated in turn. If any predicate is undefined so is the result of the execution, otherwise if any predicate is TRUE the result of execution is the associated output function. Otherwise if no predicate evaluates true then the assignment statements in the recurser and constructor are performed and execution continues with these new values.

We will say a variable is a <u>predicate variable</u> if it is tested by at least one predicate. Similarly it is an <u>output variable</u> if it is used in at least one output function. A variable can be both a predicate and an output variable.

We will make the following restrictions on the programs we will consider:

1) every recursion selector and every constructor must be

non trivial.

2) every variable is either a predicate or an output variable.

3) there is at least one output variable

4) (freedom) for and $1 \leq k \leq n$ and $l \geq 0$ there exists a set of inputs which cause the program to recurse l times before correctly exiting by output function k.

5) each output function is unique.

6) every constructor variable appears totally in at least one output function.

Given a program P in δ_n, let δ be the union of δ_i for i=1,n.

Let us assume we know, on independent grounds, that a correct program P^* exists in δ, furthermore that no predicate, output function, selector or constructor in P^* has a depth greater then some constant $u>3$.

GOAL: We wish to construct a set of test inputs with the property that any program P in δ which executes correctly on these values must then be equivalent to P^*. The existence of such a test set would then imply (under the assumption that at least one correct program exists in δ) that P is correct.

We will use capital letters from the end of the alphabet (X, Y and Z) to represent vectors of inputs.

Hence we can refer to $P(X)$ rather then $P(x_1,\ldots,x_m,y_1,\ldots,y_p)$. Similarly we can abbreviate the simultaneous application of constructor functions by $C(X)$ and recursion selectors by $S(X)$.

We will use the initial greek letters to represent positions in a variable, where a position is defined by a finite CAR-CDR path from the root. When no confusion can arize we will frequently refer to "position α in X" whereby we mean position α in some x_i in X.

We can form a lattice on the space of inputs by saying $X \leq Y$ if and only if for all selector variables x_i in X are smaller then their respective variables in Y, and similarly the constructor variables.

We can define the notion of "Pruning X at position α" as follows: We will say Y is X "pruned" at position α if Y is the largest input $\leq X$ where α is atomic. This process can be viewed as simply taking the subtree in X rooted at α and replacing it by a unique atom.

If a position α (relative to the original input) is tested by some predicate we will say that the position in question has been touched.

The assumption of freedom asserts only the existence of inputs X which will cause us to recurse a specific number of times and exit by a specific output function.

Our first lemma shows that this can be made constructive.

LEMMA 1. Given $l \geq 0$ and $1 \leq i \leq n$ we can construct an input X such that P(X) is defined and while executing X P recurses l times before exiting by output function i.

PROOF: Consider m+p infinite trees corresponding to the m+p input variables. Mark in BLUE every position which is touched by a predicate function and found to be non-atomic in order for P to recurse l times and reach the i th predicate. Then mark in RED the point touched by the i th predicate after recursing l times.

The assumption of freedom implies that no blue vertex can appear in the infinite subtree rooted at the red vertex, and that the red vertex can not also be marked blue.

Now mark in YELLOW all points which are touched by constructor functions in recursing l times, and each position touched by the i th output function after recursing l times. The assumption of freedom again tells us that no yellow vertex can appear in the infinite subtree rooted at the red vertex. The red vertex may, however, also be colored yellow, as may the blue vertexes. It is a simple matter to then construct an input X such that

1) all BLUE vertices are non atomic in X,

2) The RED vertex is atomic, and

3) all YELLOW vertexes are contained in X (they may be

atomic)

It is trivial to verify that such an X satisifies our requirements. $\triangle$

Notice that the procedure given in the proof of lemma 1 allows us to find the smallest X such that the indicated conditions hold. If α is the position touched by the i^{th} predicate after recursing l times call this point the minimal α point, or X_α.

Freedom implies no point can be twice touched, hence the minimal α point is a well defined concept.

Given an input X such that P(X) is defined, let $F_X(Z)$ be the straight line function such that $F_X(X) = P(X)$. Note that by the property of being generic, F_X is defined by this single point.

LEMMA 2: For any X for which P(X) is defined, we can construct an input Y with the properties that P(Y) is defined, $Y \geq X$ and $F_X \neq F_Y$.

PROOF: There exist some constants l and i such that on input X P recursed l times before exiting by output function i. Let the predicate P_i test variable x_j and let s_j be the recursion selector for this variable.

There are two cases, depending upon whether the output function f_i is constant or not. If f_i is not a constant then since X is bounded there must be a minimal $k >$ 1 such that the predicate $p_i(s^k(x_j))$ is undefined.

By lemma 1 we can find an input Z which causes P to recurse k times before exiting by output function i. Let Y = X union Z. Since $Y \geq Z$ P must recurse at least as much on Y as it did on Z. Since the final point tested is still atomic P(Y) will recurse k times before exiting by output function i.

It is simple to verify the fact that $F_X \neq F_Y$.

The second case arises when f_i is a constant function. By assumption 6 there is at least one output function which is not a constant function. Let f_i be this function. Let the predicate p_i test variable x_j. The same argument as before goes through with the exception that is may happen by chance the P(Y) = P(X) (i.e. P(Y) returns the constant value.) In this case we increment k by 1 and perform the same process and it cannot happen that P(Y) = P(X). $\triangle$

LEMMA 3: If P touched a location α, then we can construct two inputs X and Y such that P(X) and P(Y) are defined, and for any P' in δ, if P(X) = P'(X) and P(Y) = P (Y) then P' must touch α.

PROOF: Let Z be the minimal α point. By lemma 2 we can construct an input X such that P(X) is defined, $X \geq Z$ and $F_X \neq F_Z$. Let Y be X pruned at α.

We first assert that P(Y) is defined and $F_Y = F_Z$. To see this we note that every point which was tested by P is computing P(Z) and found to be non atomic is also non atomic in Y. α is atomic in both, and if the output function was defined on Z then it must be defined on Y which is strictly larger.

Now suppose there existed some program P' such that P'(X) and P (Y) were computed correctly but P' did not touch α. We see immediately that this cannot happen since _all_ other positions are either the same in X and in Y or they exist in X but not in Y. Hence if P'(Y) is defined it would imply $F_X = F_Y$, a contradiction. $\underline{\triangle}$

Define the positions which P touches without going into recursion to be the _primary positions_ of P.

Given a program P to test our first task is then to construct a set of test inputs using theorem 1 which demonstrate that each of the primary positions must be touched.

Observe that this set contains at most 2n elements.

We will say a selector function f _factors_ a selector function g if g is equivalent to f composed with itself some number of times. For example CADR factors CADADADR. We will say that f is a _simple factor_ of g if f factors g and no function factors f, other then f itself.

Let us denote by σ_i $i=1,\ldots,m$ the simple factors of each of the m recursion selectors. That is, for each i there is a constant l_i such that the recursion selector $s_i = \sigma_i^{l_i}$.

Let $q = GCD(l_i\ i=1,\ldots,m)$.

Let S be the simultaneous recursion selector where the i^{th} term is $\sigma_i^{l_i/q}$. Hence the recursion selectors of P can be written as S^q.

We now construct a second set of data points in the following fashion:

For each selector variable x_i:

1) x_i is an output variable used in output function f_j. Let α be the position first tested by p_j after P(X) has recursed to a depth of at least u^2. Then we generate the minimal α point.

2) x_i is not an output variable, but is a predicate variable. Let α be the first time a position with depth greater then u^2 is touched in x_i. First generate the minimal α point, then using lemma 3 generate two inputs which demonstrate that position α must be touched.

Notice that we have added no more then 3m points.

THEOREM 1: If P' is in δ and P computes correctly on all data points computed so far, then the recursion selectors of P' must be powers of σ_i.

PROOF: Observe the fact that if x_i is an output variable in P, it must appear as a result in at least one input X in our test data space, hence if P'(X) is correct x_i must be an output variable for P' also.

The proof of theorem 1 will then rest on the following two cases.

Case 1. If x_i is an output variable. By construction there exists some X in our test data space such that P(X) recurses to a depth of at least 3u ($<U^2$) before exiting by the j^{th} output function, where x_i is an output variable in f_j.

Assume that the i^{th} recursion selector in P' is not a power of σ_i. Then somewhere before the i^{th} variable has recursed to a depth of u their paths must diverge.

Once the i^{th} variable steps past the points where the paths in the two programs diverge it can never have access to the subtrees used in P by f_j in its output. Hence P' on X must halt before the i^{th} variable has recursed to a depth of u.

But if that is the case then its output functions cannot access subtrees rooted any deeper then 2u. By construction the correct output requires trees which can only be accessed by going at least 3u deep, hence a contradiction is obtained.

Case 2: If x_i is not used as an output variable.

Assume the recursion selector of x_i in P is not a power of σ_i. Then once the variables x_i have recursed past the depth u they will be in totally different subtrees of their input (see figure 3.)

By construction it is required that P' touch a point whose depth is at least 3u. P' must therefore touch this point before the ith variable diverges from the path taken by P, hence before it has has reached a depth of u. But by definition P' cannot touch any points deeper then 2u in this region, hence a contradiction is obtained. $\triangle$

Theorem 1 gives us a way to demonstrate that a program Q must have the same recursion selectors, up to a power, as does P. We now wish to derive a slightly stronger result. We will show that there exists a constant r such that the recursion selectors of P' are exactly s^r.

Note that by definition we know that $|S^r|$ (that is, the maximum depth of any function in S^r) is less then u.

THEOREM 2: If P' is in δ computes correctly on all the points we have so far computed, then there exists a constant r such that the recursion selectors of P are exactly S^r.

PROOF: We know by theorem 2 that the recursion selectors of P must be powers of σ_i. For each $1 \leq i \leq m$

construct the ratio of the power of σ_i in P' to that of P. Let x_i be the variable with the smallest such ratio and x_j be the variable with the largest. From the fact that these ratios are different we will obtain a contradiction.

Case 1: x_i is an output variable. By construction there is an input X such that P'(X) must recurse on X to a depth of at least u^2 before outputing by a output function which uses x_i . This implies that P' must recurse at least u times. Since in comparison to the program P the variable x_j is gaining at least one level each recursion we have that either 1) P'(X) is undefined because x_j ran off the end of its input, or 2) P'(X) must halt before it has recursed to a depth of u(u-1) in x_i in which case it cannot have produced the correct output.

The argument in the case where x_i is a predicate variable, but not an output variable is almost the same and is here omitted. $\triangle$

By lemma 3 we know that if P touches a location α, then we can construct a pair of inputs with the property that any program P' in Φ which executes correctly on these two inputs must also touch α. We now present the converse lemma.

LEMMA 4: If P works correctly on the test data so far constructed, and does not touch a location α, then we can construct two inputs X and Y with the property that any P' in $\mathcal{S}$ which executes correctly on all this data must also not touch the position α.

PROOF: Let x_i be the variable containing α. Let v the maximum depth any variable has obtained just after the ith recursion selector passes the depth of α. Let X be a set of complete trees of depth v+2u, pruned at α.

There are two cases, depending upon whether P(X) is defined or not.

Case 1: P(X) is not defined. Assume P' touches α. Let Z be the minimal α point in P' (we need not be able to construct this point.) We see that $Z<X$. But this then implies that P'(X) must be defined, a contradiction.

Case 2: P(X) is defined. By lemma 1 we can construct an input $Z>X$ so that $F_X \neq F_Z$. Let Y be Z pruned at α.

Assume P(X)=P'(X) and P(Y)=P'(Y) and P' touches α. If P(Y) is undefined we are done, since P'(Y) must be defined. So assume P(Y) is defined. In this case, since P does not touch α, $F_Y=F_Z\neq F_X$. But if P touched α, then since $X\leq Y$ we would have $F_X=F_Y$, a contradiction. $\triangle$

Next we show that the primary positions of P' must be exactly those of P.

Let $p_1, \ldots, p_n$ be an ordering of the primary positions of P such that the depth of the position tested by p_i is less then or equal to the depth of that tested by

ρ_{i+1}.

We know the recursion selectors of P' are S^r where $|S^r| < u$. This gives us at most u possibilities. For each possibility we proceed in turn as follows:

Assume position ρ_i (i = 1,.. n) is not primary in P'. We can construct a point which is then tested by P' earlier then ρ_i by imagining the root input was actually the result of one recursion, and then looking at the position ρ_i in relation to the earlier root (see figure 4.)

Now one of two cases arises. Either

1) the new position is not touched by P, or

2) the new position corresponds to a position ρ_j j<i.

In the first case we can construct two inputs which demonstrate the position in question must not be touched. The second case immediately rules out S^r as the recursion selector, since by induction ρ_j is primary to P and hence P' would not by an element of $\mathfrak{d}$.

Notice we have increased our test case size by no more then 2nu elements. The resulting test case then gives us the following theorem.

THEOREM 3: If P'(X) = P(X) for X in our test set, then the primary positions of P' are exactly those of P.

Notice also that by the generic property that this also implies the following corollary:

THEOREM 4: The output functions of P' are exactly those of P.

Once we have that the primary positions of P' are exactly those of P, we can now return to the problem of showing that the selector functions of P' must be S^q. Consider each of the alternative possibilities for S^r (no more then U of them.) Since the rates of recursion of P and P' differ, one of three cases must arise. Either

1) P' touches the same point twice (which means P' is not in ϑ and is out of the running.)

2) P' touches a point which P fails to touch, or

3) P touches a point which P' fails to touch.

Since we only need to test for the last two conditions we need augment out test case with no more then 2u points.

we then have the following theorem:

THEOREM 5: The recursion selectors of P' must be exactly those of P.

Pushing onward we next want to consider the recursion constructors. Once we have the other elements fixed, however, the constructors are almost given free. All we need do is to construct p data points so that the i^{th} data point causes the program P to recurse once and exit using an output function which uses the i^{th} constructor variable. By the generic property and the fact that the entire i^{th} constructor variable is then open to inspection we have the the next theorem.

THEOREM 6: The recursion constructors of P' must be exactly those of P.

What remains? Well the order in which the primary positions are tested is the only thing we have not nailed down. For each primary position α add X_α to our test data. We leave it to the reader to verify:

THEOREM 7: The order of predicate evaluation in P' is exactly that of P.

Counting the size of our test set, we see now that it contains no more then 3(n+m)+2(p+u+nu) points. Combining all the theorems proved in this section we then have our main result, which states:

THEOREM: Given a program P in δ, there exists a set of no more then 3(n+m)+2(p+u+nu) elements such that if P' is any program in δ which computes the same results on this set as P does, then P must be equivalent to P.

COROLLARY: Either P is correct or no program in δ

realizes the intended function.

5. AN EXAMPLE

The following example, taken from [6], will be used to illustrate some of the ideas here presented.

The program is given by [6] as follows:

```
(REVDBL
   (LAMBDA (ARG1)
      (COND
        ((NULL ARG1) NIL)
        (T (APPEND (REVDBL (CDR ARG1))
               (LIST (CAR ARG1) (CAR ARG1])
```

We will translate it into the following form.

```
REVDBL(X,Y) = ATOM(X) -> Y

              Y <- CONS(CAR(X),CONS(CAR(X),Y)))

              X <- CDR(X)
```

Using the formula given in the main theorem, we see that a test set exists for this program containing no more then 20 points. However, if one follows the arguments given in this paper, one finds that actually the three points given in figure 5 suffice. This illustrates the point that we have actually been rather liberal in our counting, and usually a much smaller test set can be found then the limit stated in our main result.

[1] R.A. DeMillo, R.J. Lipton and F.G. Sayward, "PROGRAM MUTATION: A new Approach to Program Testing", presented at the Navy Laboratory Computing Committee Symposium on Software Specificationa and Testing Technology, April 1978.

[2] T.A. Budd, R.A. DeMillo, R.J. Lipton and F.G. Sayward, "The Design of a Prototype Mutation System for Program Testing", Proceedings of the 1978 National Computer Conference, pp 623-627.

[3] R.A. DeMillo, R.J.Lipton and F.G.Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", Computer, 11,4 (April 1978), pp34-41.

[4] T.A.Budd and R.J.Lipton, "Mutation Analysis of Decision Table Programs", Proceedings of the 1978 Conference on Information Sciences and Systems", pp. 346-349.

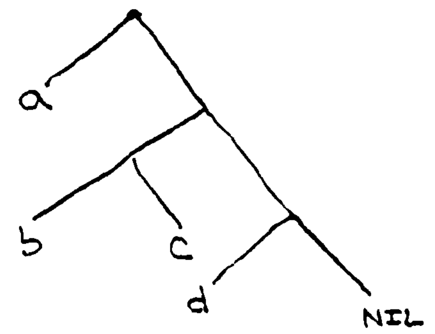
[5] S. Hardy, "Synthesis of LISP functions from Examples", Proceedings of the Fourth International Joint Conference on Artificial Intelligence.

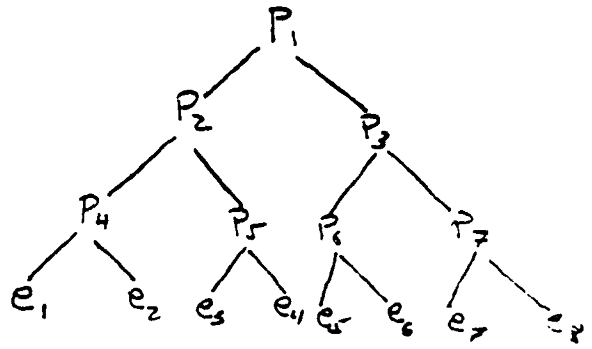
[6] D.E.Shaw, W.K.Swartout, and C.C.Green, "Inferring LISP programs from Examples", Proceedings of the Fourth Internaional Joint Conference on Artificial Intelligence.

[7] P.D.Summers, "program Construction from Examples", Ph.D. Thesis, Department of Computer Science, Yale University, New Haven, Ct., 1975.

$X = (a\ (b.c)\ d)$

roj[x] =



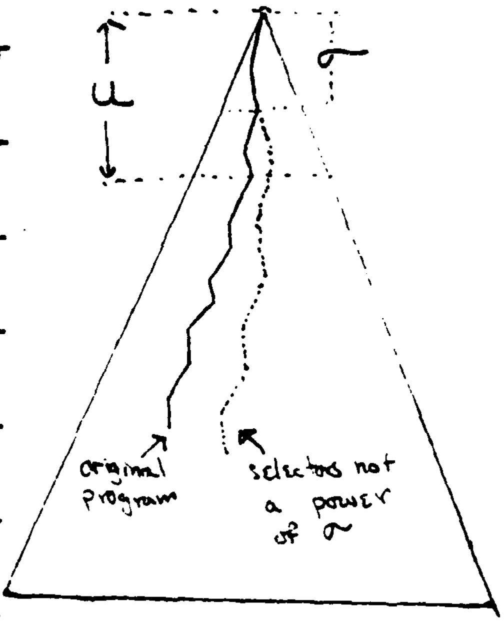


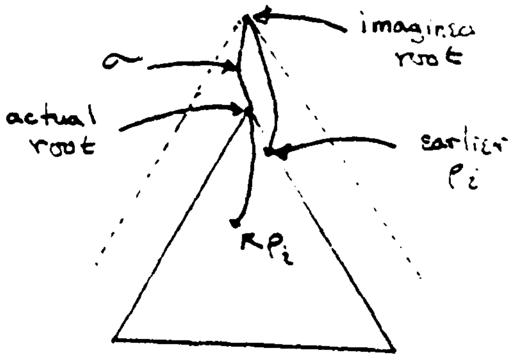
$P_1, \ldots P_7$ are Predicates

$e_1, \ldots e_8$ are straight line programs

figure 1

figure 2





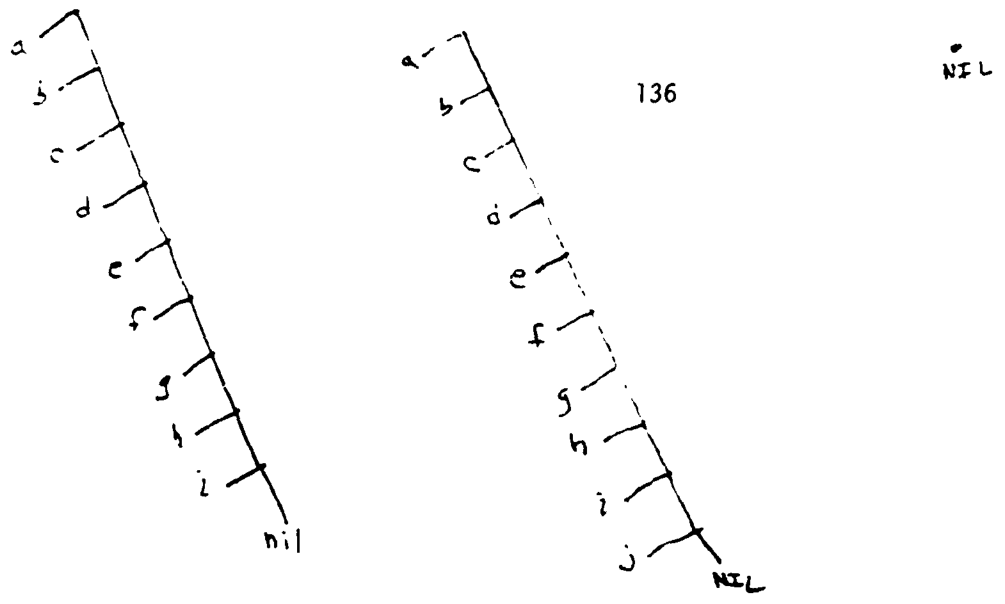


figure 5

The design of a prototype mutation system for program testing*

by TIMOTHY A. BUDD, RICHARD J. LIPTON and FREDERICK G. SAYWARD
Yale University
New Haven, Connecticut

and

RICHARD A. DEMILLO
Georgia Institute of Technology
Atlanta, Georgia

INTRODUCTION

When testing software the major question which must always be addressed is "If a program is correct for a finite number of test cases, can we assume it is correct in general." Test data which possess this property is called Adequate test data, and, although adequate test data cannot in general be derived algorithmically,[1] several methods have recently emerged which allow one to gain confidence in one's test data's adequacy.

Program mutation is a radically new approach to determining test data adequacy which holds promise of being a major breakthrough in the field of software testing. The concepts and philosophy of program mutation have been given elsewhere,[2] the following will merely present a brief introduction to the ideas underlying the system.

Unlike previous work, program mutation assumes that competent programmers will produce programs which, if they are not correct, are "almost" correct. That is, if a program is not correct it is a "mutant"—it differs from a correct program by simple errors. Assuming this natural premise, a program P which is correct on test data T is subjected to a series of mutant operators to produce mutant programs which differ from P in very simple ways. The mutants are then executed on T. If all mutants give incorrect results then it is very likely that P is correct (i.e., T is adequate). On the other hand, if some mutants are correct on T then either: (1) the mutants are equivalent to P, or (2) the test data T is inadequate. In the latter case, T must be augmented by examining the non-equivalent mutants which are correct on T: a procedure which forces close examination of P with respect to the mutants.

At first glance it would appear that if T is determined adequate by mutation analysis, then P might still contain some complex errors which are not explicitly mutants of P. To this end there is a *COUPLING EFFECT* which states that test data on which all simple mutants fail is so sensitive that it is highly likely that all complex mutants must also fail.

Readers wishing a further exposition of the ideas of mutation and substantiation of the assumptions made are referred to References 2 and 10.

THE SYSTEM

A pilot system has been built to implement mutation analysis on programs written in a subset of FORTRAN. The key features of this system are summarized in Figure 1. The system itself consists of 10,000 lines of FORTRAN code, and required six man months to design, implement and debug.

Notice we claim the system is *man/machine interactive*. In general an attempt is made to assign tasks to both the user and the machine processors which are best suited to using their particular capabilities. One way to see this is to view the system as a sort of "Devils Advocate", which when confronted with a program asks very difficult questions about the motivation behind it ("why did you use this type of statement here, when an alternative statement works just as well?"). The job of the human is then to provide justification (in the form of test data), which will give an answer to such questions.

An overview of the structure of the system is given in Figure 2. We point out that the language FORTRAN was chosen for the first implementation merely as a matter of convenience since it is in common use and there is a large body of software in existence to experiment on. The heart of the system (roughly that shown within the dotted box) is however, language independent, and given a sufficiently general internal form to implement a new language one would merely write a new input/output interface. Projects are currently under way to implement mutation analysis on

* This research was supported in part by NSF Grant MCS76-81486 and U.S. Army Research Grant DAAG-29-76-G-0338.

INTERACTIVE
MACHINE INDEPENDENT
LANGUAGE INDEPENDENT STRUCTURE
MODULAR DESIGN
INTENSIVE MAN/MACHINE INTERACTION

Figure 1—Key features of the pilot mutation system

COBOL and C (an ALGOL like language) using the structure represented by the box contained in the dotted lines.

An attempt was also made to keep the structure of the system largely machine independent. The system was originally programmed to run on a PDP-10 at Yale University. Currently we are in the process of transferring it to a CDC 7600 at the Georgia Institute of Technology.

A single run of a mutation system divides naturally into three phases the RUN PREPARATION phase, in which the necessary variables to send to the mutation executor are defined, the MUTATION phase, in which the actual mutations are produced and executed, and the POST RUN phase, in which results are analyzed and reports are generated. In the following we will describe in more detail the structure and effects of each phase.

The role of the run preparation phase is to initialize the various files and data buffer areas used by the mutation executor. It is characterized by a very interactive nature. The first object the user is requested to supply is the name of the file on which the FORTRAN subroutine resides. Then depending on whether PIMS has been run previously on this routine (in which case the internal form is stored on one of the many files PIMS constructs, see below) the subroutine is parsed into a concise internal format which is subsequently interpreted to simulate execution of the program. A

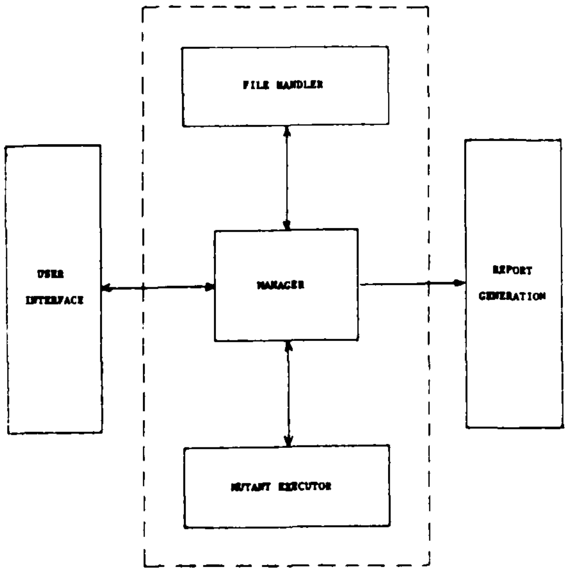


Figure 2

```
IF (A .LT. X(2)) P = 1


[SCALAR. A]
[ARRAY1. X]
[CONSTANT.2]
[AOP.SUBSCRIPT]
[ROP.LT]
[TRF.O]
[SCALAR. p]
[CONSTANT.1]
[ASSIGN.O]
```

Figure 3

fragment of the internal code generated for a given statement is shown in Figure 3.

The user is then interactively prompted for the test data on which the program and mutants are to be tested. After each test case has been specified the original program is executed on the test case and the results displayed so that the user may satisfy himself that the results produced are indeed correct.

After the test data has been entered the user is prompted for a listing of which mutant operators he wishes to enable. At present there are 25 mutant operators. These range from very simple low level ones, such as replacing each data occurrence (where a data occurrence is a scalar, constant or array reference) with all other syntactically correct data occurrences, to very high level mutations, such as deleting statements or altering the control structure of the program. A more detailed description of the mutations performed can be found in Reference 3.

Instead of constructing multiple copies of the program, for each mutant a short (four word) description of the mutation to be performed is kept. Each time the mutant is to be run the original program is then mutated according to the contents of this descriptor.

After the user has specified to the system his program, test data and the mutant operators he wishes applied, the system then enters the MUTATION phase. During this phase there is no user interaction. Mutation descriptor records are read in, one by one, and the mutation is produced. The mutant program is then executed on the test data and marked either "dead," meaning it produced results differing from the original program on at least one test case, or "living." A dynamic record is kept of the number and percentage of living mutants of each mutation type.

When all the mutant programs have been tested the post run phase is entered. In this phase statistics are displayed indicating the results of the mutation run. In addition the user can interactively view descriptions of those mutations which have survived. He can also specify that certain re-

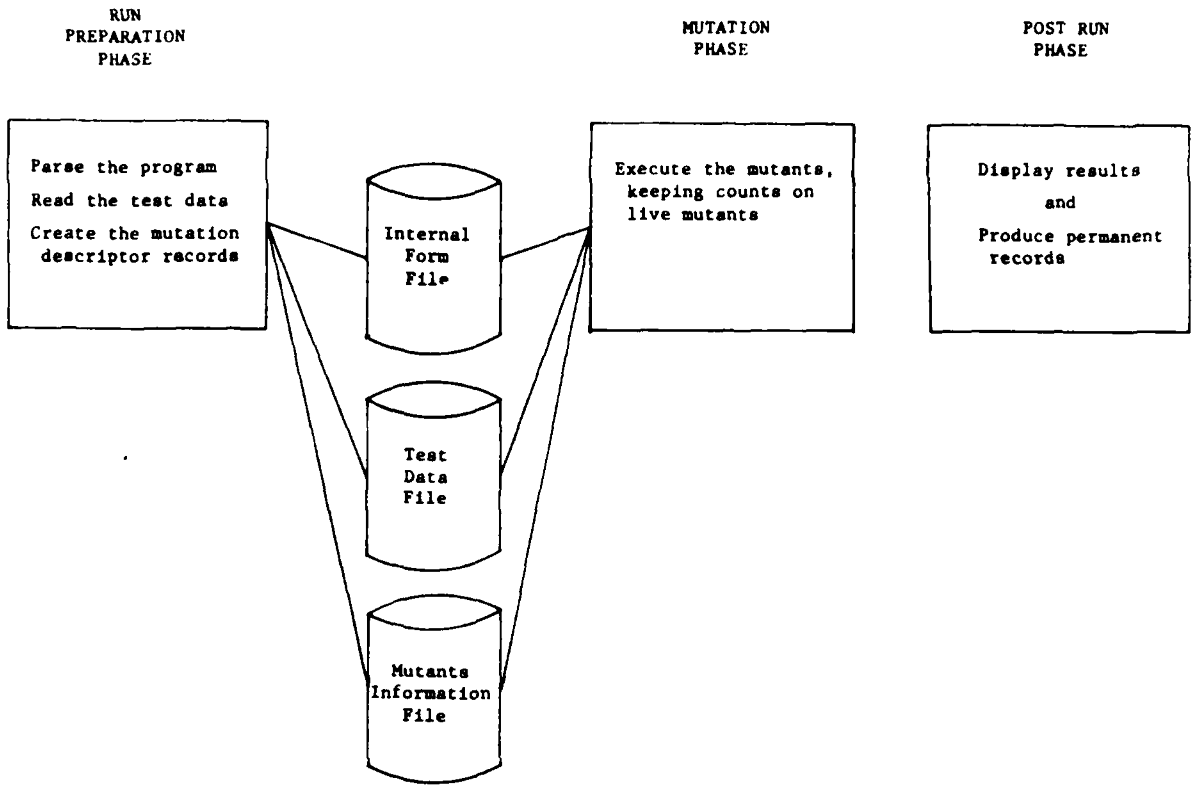


Figure 4

ports be generated in order to provide a detailed permanent record of the mutation run.

At this point, or at a later date, the user can re-run the system and augment his test data in an attempt to make the remaining mutants fail. He may also specify that additional mutant operators be applied to the program. This cycle can continue until the user is satisfied that the current test data adequately tests his program.

There are several files the system produces in order to store information from one run to the next. These are shown in Figure 4, which outlines the major functions of each phase. The internal form file stores the parsed version of the program. The test data file stores for each test case the test data input and the results of execution of that test data. The mutants information file keeps the mutant descriptor records plus various other counts on what types of mutants have been produced.

A COMPARISON OF PIMS TO OTHER DATA TESTING SYSTEMS

Various systems have been discussed in the literature for increasing confidence in the adequacy of test data, as the PIMS system does, or automatically constructing test data which meets some criterion. In this section we will report on experiments which show that the PIMS system is an improvement in this area over other systems which have been proposed.

The most widely known method of constructing test data automatically are those systems which utilize path analysis.[4-7] Essentially, these procedures attempt to construct data which force each statement to be executed at least once, and furthermore which transverse each feasible flow path through the code at least once. In some cases, such as loops, only an approximation to this can be made as the number of flow paths may be infinite. Here it is usual to just construct data which cause the loop to be executed at least twice.

These same objectives are met with mutant analysis in a number of ways, some directly by mutant operators, others indirectly by the coupling effect. There are mutants which cause each statement in the original program to be replaced by a TRAP statement, a special type of statement which if ever executed causes the program to immediately abort. Obviously, then if there is some statement in the program which is never executed, changing that statement to a TRAP statement will not alter the output of the program and hence will easily be detected.

Checking that every decision path is taken is essentially

```
        DO 10 I=1.J
               .
               .
               .
        10 CONTINUE
       ===============
        DO 10 I=1.1
               .
               .
               .
        10 CONTINUE

  A LIVE MUTATION IF THE LOOP IS
  ALWAYS EXECUTED ONLY ONCE.
```

Figure 5

the same as checking that every predicate in the program evaluates at least once to both true and false. If this is not the case, say the predicate always evaluated to TRUE, then we can mutate the predicate in any way we desire as long as it retains this property of always remaining TRUE. These types of mutations are also usually quite obvious and easily detectable.

Mutation analysis can also insure that each loop is traversed at least twice. The only way a loop can be traversed only once (and all loops must be traversed at least once to pass the TRAP statement mutations) is if the terminating condition is the same as the starting condition. But in this case the mutant which replaces the terminating condition by the starting condition will survive (see Figure 5). This is once more easily detected.

With this, mutant analysis possesses all the capabilities of path analysis systems which have been discussed in the literature.

```
       SUBROUTINE BSERCH(X.Y.N.A.IHIGH.LOW.ERR)
       INTEGER X(N).Y(N).N.A.IHIGH.LOW.EPR.MID
C      BINARY SEARCH PROCEDURE. IF X CONTAINS A ON RETURN
C      X(IHIGH) = A.IHIGH=LOW. IF NOT X(LOW) < A < X(IHIGH).
C      IF A IS OUT OF RANGE ON RETURN ERR CONTAINS 1
       ERR = 0
       IF ((X(1)-A).GT.0) GOTO 11
       IF ((A-X(N).LE.0) GOTO 5
  11   ERR = 1
       RETURN
  5    LOW = 1
       IHIGH = N
  6    IF ((IHIGH-LOW-1).NE.0) GOTO 7
       RETURN
  7    MID = (LOW+IHIGH)/2
       IF ((A-X(MID)).GT.0) GOTO 10
       IHIGH = MID
       GOTO 6
  10   LOW = MID
       GOTO 6
       END
```

Figure 6

```
Replace  IF ((IHIGH-LOW-1).NE.0) GOTO
by       IF ((IHIGH-LOW-1).GT.0) GOTO 7

Replace  MID = (LOW+IHIGH)/2
by       MID = (LOW+IHIGH)-2
```

Figure 7

Another class of systems for which extensive claims have been made are those which detect uninitialized variables and dead code.[8] Uninitialized variables are caught as a consequence of the interpretation process in the mutant system. Dead code is easily caught since an assignment made to a dead variable can be mutated in any way whatsoever and the program will remain the same.

A third class of systems for which there has recently been much discussion involves symbolic execution of the program. In one study[9] Howden analyzed 12 programs containing a total of 22 errors. He found that symbolic execution would catch 13 of those errors, while path analysis would discover only nine. In a similar study we estimated that mutation analysis, using only the mutant operators in the present PIMS system, would uncover 18 of the 22 errors. Of the remaining four, three would probably be discovered if we added two new mutant operators which the authors simply had not thought about. Hence, mutation analysis is in certain cases an improvement over symbolic execution.

As an example of the very subtle errors which mutation analysis can discover consider the program to perform binary search shown in Figure 6. If it happens that N = 1 when the subroutine is called (i.e., the vector to be searched contains only a single element) then it is not difficult to see that the program will loop indefinitely. It is not clear that either symbolic execution or path testing would be sufficient to discover this error.

When mutant analysis is applied to this program there are two mutants generated (shown in Figure 7) which can only be eliminated by a test case consisting of one element. Hence the error is easily detected using mutant analysis. (There is a second error in this program which is also uncovered by mutant analysis. The discovery of that second error is left to the reader).

FUTURE WORK

There are several directions in which work is currently being pursued with respect to mutation analysis and the pilot mutation system. The most obvious is to show h[illegible] a similar system might be built around another language, and research is under way to construct systems for COBOL and for C.

Another area of study is the design of an easy to use language for the description of test cases which allows for a variety of features. Test datasets can often be quite lengthy, yet two test cases can be very similar. Also, a user often wishes just to construct a number of random test cases following some specification. (Some of the pitfalls of using random data to test programs are discussed in Reference 10 where it is seen that mutation analysis can help in deriving "good" random test data.) Finding an easy yet powerful method of solving this problem is the goal of one area of research.

Finding a method to detect equivalent mutants is another area currently being pursued. It is often the case that a mutation will not produce a significantly different program (replacing the sequence I=1 J=1 with the sequence I=1 J=I is a trivial example). We have observed that programs tested have between one and two percent equivalent mutants. A method to automatically detect and remove equivalent mutants would allow us to provide even more significant measures of the adequacy of a test data set.

We point out that as a consequence of the modular design of the pilot system either of the above two major extensions can be added without a significant reprogramming effort.

A final area of current interest is the study of mutant operators. Certain operators seem to have a much greater ability to detect errors then others. Analysis of data along these lines would allow us to discover an order of application of mutant operators which would maximize the cost/benefit ratio.

CONCLUSIONS

It has been shown that the ideas of program mutation can be quickly and easily implemented as an interactive system for program testing. The resulting system represents a cost effective engineering approach to testing real world software. Large subroutines (over a hundred statements long) have been analyzed by our system with relative ease.

Mutation is a method of program testing which will significantly raise the level of reliability in both new and existing software, and is a major advance in the area of software testing.

REFERENCES

1. Goodenough, J. B. and S. L. Gerhart, "Towards a Theory of Test Data Selection," *IEEE Tran. Soft. Eng.*, SE-1,2, June 1975, pp. 156-173.
2. DeMillo, R., R. J. Lipton and F. Sayward, "PROGRAM MUTATION—A Method of Determining Test Data Adequacy," in preparation.
3. Budd, T. and F. Sayward, "Users guide to the Pilot Mutation System," Yale University Tech. Rep 114, 1977.
4. Ramamoorthy, C. V., S. F. Ho, and W. T. Chen, "On the Automated Generation of Program Test Data," *IEEE Trans. on Soft. Eng.*, SE-2,4, Dec. 1976, pp. 293-300.
5. Howden, W. E., "Methodology for the Generation of Program Test Data," *IEEE Trans. on Comp.*, C-24,5, May 1975, pp. 554-560.
6. Huang, J. C., "An Approach to Program Testing," *Computing Surveys*, 7,3, Sept. 1975, pp. 113-128.
7. Miller, E. F. and R. A. Melton, "Automated Generation of Testcase Datasets," Proc. 1st Int. Conf. on Reliable Software, *SIGPLAN Notices* 10,6, June 1975, pp. 51-58.
8. Osterweil, L. J. and L. D. Fosdick, "Some Experience with DAVE—A Fortran Program Analyzer," *AFIPS Conference Proceedings*, Vol. 45, 1976, pp. 909-915.
9. Howden, W. E., "Symbolic Testing and the DISSECT Symbolic Evaluation System," *IEEE Trans. on Soft. Eng.*, SE-3,4, pp. 266-278
10. DeMillo, R., J. Lipton and F. Sayward, "Hints on Test Data Selection," to appear in *Computer*, April 1978.

HEURISTICS FOR DETERMINING EQUIVALENCE OF PROGRAM MUTATIONS

Douglas Baldwin

and

Frederick Sayward

Department of Computer Science
Yale University
New Haven, Connecticut 06520

ABSTRACT

A mutant of a program P is a program M which is derived from P by making some well-defined simple change in P. Some initial investigations in the area of automatically detecting equivalent mutants of a program are presented. The idea is based on the observation that compiler optimization can be considered a process of altering a program to an equivalent but more efficient mutant of the program. Thus, the inverse of compiler optimization techniques can be seen as, in essence, equivalent mutatuion detectors.

1.0 INTRODUCTION

A mutant of a program P is defined as a program P′ derived from P by making one of a set of carefully defined syntactic changes in P. Typical changes include replacing one arithmetic operator by another, one statement by another, and so forth. Program mutation has been used by DeMillo, Lipton and Sayward as the basis for an interactive program testing system [2]. The theory behind this system is that a set of test data T adequately tests a program P if all mutants of P are distinguished from P by either failing to produce any result or producing a different result for some element of T. On the other hand, if a mutant performs identically to P then either T does not fully test the program and further cases must be developed, or the mutant is equivalent to P. Obviously it is impossible to develop test data that distinguish between equivalent forms of the same program, and thus it is desirable that equivalent mutants be excluded from the testing process. Unfortunately, user recognition of equivalent mutants has proven to be a difficult and tedious task. Thus it is important that the system aid the user by either automatically detecting equivalent mutants or by posing questions which provide insights on how to do so.

Our goal is to develop heuristics by which equivalent mutants can be recognized. The heuristics are primarily derived from techniques used to optimize compiler code, since the process of optimizing compiler code can be thought of as producing a series of mutants which are equivalent to the original program. It is thus expected that some of the tests developed to determine when an optimization is equivalence preserving can be applied to determine when a mutation is equivalence preserving.

Once a body of heuristics has been developed to detect equivalence of mutants it will be possible to develop a program to actually recognize them in a program testing system. This system will probably be very similar to the optimization phase of a compiler. It will generate some representation of each mutant which can be easily manipulated and apply the heuristics described below to determine if it is equivalent to the original. If so then the mutant will be flagged as equivalent and will be excluded from future testing runs.

2.0 PROGRAM MUTATION

As defined above a mutant of a program is a second program derived from the first through carefully defined syntactic transformations. Program mutation is the process of forming mutants from an input program.

The work described here is intended to find ways of determining equivalence of mutants derived as part of a process for testing FORTRAN programs on the EXPER [4] testing system. The mutations made by EXPER are chosen so as to duplicate as closely as possible the mistakes which a good programmer might make in coding a FORTRAN program. Thus many of the mutants involved, such as DO-loop end replacement, are specific to FORTRAN. The mutations of interest are described below:

1. Constant Replacement: Replacement of a constant, C, with C+1 or C-1.
   Ex: A=1 becomes A=0.

2. Scalar Replacement: Replacement of one scalar by another.
   Ex: A=B becomes A=C.

3. Scalar for Constant Replacement: Replacement of a constant with some scalar variable
   Ex: A=2 becomes A=B.

4. Constant for Scalar Replacement: Replacement of some scalar variable with a constant.
   Ex: A=B becomes A=2.

5. Source Constant Replacement: Replacement of one constant in the program with some other constant found in the program.
   Ex: A=3 becomes A=1 where the constant 1 appears in some other statement.

6. Array Reference for Constant Replacement: Replacement of a constant with an array reference.
   Ex: A=1 becomes A=B(1).

7. Array Reference for Scalar Replacement: Replacement of a scalar reference with an array reference.
   Ex: A=B becomes A=C(1).

8. Comparable Array Name Replacement: Replacement of a reference to one array with a reference to the same element of another array of the same size and shape.
   Ex: A=B(1,3) becomes A=X(1,3).

9. Constant for Array Reference Replacement: Replacement of an array reference with a constant.
   Ex: A=B(1) becomes A=3.

10. Scalar for Array Reference Replacement: Replacement of an array reference with a refereance to a scalar.
    Ex: A=B(1) becomes A=C.

11. Array Reference for Array Reference Replacement: Replacement of one array reference by another.
Ex: A=B(1) becomes A=C(2).

12. Unary Operator Insertion: Insertion of one of the unary operators ! (absolute value), - (negation), ++ (increment by 1) or -- (decrement by 1) in front of any data reference.
Ex: A=B becomes A=-B.

13. Arithmetic Operator Replacement: Replacement of one arithmetic operator (+,-,*,/,**) with another.
Ex: A=B+C becomes A=B-C.

14. Relational Operator Replacement: Replacement of one relational operator (.EQ.,.LE.,.GE.,.LT.,.GT.,.NE.) with another.
Ex: IF(A.EQ.B) GOTO 1 becomes IF(A.NE.B) GOTO 1.

15. Logical Connector Replacement: Replacement of one logical connector (.AND.,.OR.) with the other.
Ex: A.AND.B becomes A.OR.B.

16. Unary Operator Removal: Deletion of any unary operator.
Ex: A=!B becomes A=B.

17. Statement Analysis: Replacement of any statment with a trap statement whose execution causes immediate failure of the program.
Ex: GOTO 2 becomes CALL TRAP.

18. Statement Deletion: Removal of any statement.
Ex: GOTO 2 is removed, i.e. becomes CONTINUE.

19. Return Statement Replacement: Replacement of any statement by a RETURN statement.
Ex: A=0 becomes RETURN.

20. Goto Statement Replacement: Replacement of any GOTO statement with a GOTO to a different label.
Ex: GOTO 1 becomes GOTO 3.

21. DO Statement End Replacement: Replacement of the end label in a DO statement with some other label.
Ex: DO 2 I=1,10 becomes DO 1 I=1,10.

22. Data Statement Alteration: Changing the values assigned by a DATA statement.
Ex: DATA A /2/ becomes DATA A /1/.

23. Unary Operator Replacement: Replacement of one unary operator by another.
Ex: A=!B becomes A=++B.

Obviously some of the mutations described above can produce mutants which are equivalent to the original program. For instance, replacing A=0 with A=10 does not change a program. It might be hoped that detection of equivalent mutants would be easy, since the mutations involved are so simple and well defined. Unfortunately this is not the case. It is easily shown that the general problem of determining the equivalence of two primitive recursive functions is undecidable [1]. If we let P1 and P2 be FORTRAN routines corresponding to two arbitrary primitive recursive functions we can show that the equivalence of mutants is undecidable. Consider the following program to which the mutation "GOTO Statement Replacement" has been applied:

```
              GOTO 1
        1     P1
              STOP
        2     P2
              STOP
```

The resulting mutant looks like:

```
              GOTO 2
        1     P1
              STOP
        2     P2
              STOP
```

Plainly these programs are equivalent if and only if P1 and P2 are equivalent. Since the equivalence of P1 and P2 is undecidable, the equivalence of the mutant and original programs must also be undecidable.

The easiest way to show that two programs are not equivalent is to find some input on which they produce different outputs. This is the basic function of EXPER as a program testing tool, and thus many mutants do not need to be tested for equivalence. At any given stage those mutants which produce the same output as the original program on all test data are called live mutants. Obviously it is only the live mutants to which sophisticated equivalence tests must be applied at all. Since the equivalence problem for program mutants is undecidable, any equivalence testing process will not always be able to detect all equivalent mutants. Thus the final decision about whether a mutant is equivalent to the original program might have to be left to the user. The goal of the testing process should be to make one of three decisions about any mutant:

1. It is definitely equivalent to the original program.
2. It might be equivalent to the original program, but the information needed to make this determination is not completely available. The system should identify the needed information and ask the user to supply it.
3. None of the known tests are able to determine whether the mutant and the original are equivalent. The system is unable to help the user at all.

3.0 OPTIMIZATION TECHNIQUES

Almost all of the techniques used in optimizing compiler code can be applied in some way to decide whether a mutant is equivalent to the original program. Some are useful only in very limited sets of situations, whereas others can be applied to many types of mutation. All the techniques discussed below can be applied widely enough that it would be worthwhile to implement them in an actual equivalence tester.

The easiest way to implement these techniques is in conjunction with a flow graph of the program being mutated. A flow graph is a directed graph in which each node represents a statement or group of statements through which program control flows linearly (basic blocks). Thus any node in the flow graph represents a fragment of code which is entered only at the first statement of the block and exited only from the last. Furthermore there are no loops or branches within the node. The edges of the flow graph represent branches within the program from one basic block to another. Efficient algorithms exist for generating flow graphs from programs, for instance the process outlined by Schaefer ([5], pages 12-20). Thus it is reasonable to expect such a representation to be available to the equivalence tester. Furthermore, since mutants are so similar to the program from which they are derived, it will be easy to derive the flow graph of the mutant directly from the flow graph of the original in most cases. In the discussion below it is assumed that the equivalence tester can examine programs at the statement and token level; whether these entities are individual nodes in the flow graph or packed many per node is irrelevant.

The various optimization techniques which seem applicable to testing mutant equivalence are listed below.

3.1 Constant Propagation

Constant propagation involves replacing expressions involving constants with other constants to eliminate run-time evaluation. Generally the compiler keeps track as far as possible of the value of each variable throughout the program. At any point where an expression involves only variables whose values are known the result of the expression can be computed at compile time and placed in the program as a new constant. Thus this optimization applied to the code fragment

```
A=1
B=2
C=A+B
```

would produce the equivalent code

```
A=1
B=2
C=3
```

An elegant scheme for global program analysis is given by Kildall [3]. This scheme associates with each statement of the program a pool of data which are being propagated through the program. Such data pools can be used for constant propagation by letting the elements of the pool be ordered pairs whose first element represents a variable and whose second element represents a value. Other applications of this approach to program analysis are discussed below. This scheme is ideally suited to the needs of an equivalence tester.

3.2 Invariant Propagation

Invariant propagation is similar to constant propagation in that it involves associating with each statement of the program a set of invariant relationships between data elements. For instance, invariant propagation will note such things about a program as "X<0" or "B=1". As indicated by the last example constant propagation is a special case of invariant propagation. This technique is of limited use in compilers, but is very powerful for detecting equivalent mutants.

Invariant propagation can be implemented using Kildall's scheme for constant propagation by replacing the variable and value pairs with triples of the form <object>, <relation>, <object>. Each <object> represents either a variable or constant, and <relation> is one of the algebraic relations $<$, $>$, $=$, $\leq$, $\geq$, or $<>$. The only difficulty is that an invariant propagation algorithm should be able to replace a strong relationship with a weaker one (i.e. replace "A=1" with "A≥1"). The propagation algorithm should also be able to apply transitivity to deduce relationships such as "A<0" from the relationships "A<B" and "B<0".

3.3 Common Subexpression Elimination

One of the optimizations frequently performed by compilers is to recognize subexpressions which occur many times but only need to be evaluated once. For instance, in the code fragment

```
A=X+Y
B=X+Y+Z
```

The expression "X+Y" is evaluated two times. The common subexpression can be eliminated by evaluating it once and assigning the result to a temporary variable T, yielding:

```
T=X+Y
A=T
B=T+Z
```

Kildall [3] demonstrates how his scheme for global analysis can be applied to common subexpression elimination. In this application the data pools are sets of expressions which are partitioned into equivalence classes such that all expressions in equivalence class E have the same value. Thus the example above might have sets as shown below, where "|" divides equivalence classes: (Note the addition of a CONTINUE statement to show the set after the assignment to B.)

```
A=X+Y       { }
B=X+Y+Z     {A,X+Y}
CONTINUE    {A,X+Y | B,X+Y+Z,A+Z}
```

Note that the algorithm described by Kildall generates equivalent expressions which are not used in the program, such as A+Z in the same partition as X+Y+Z above. This feature allows the widest possible range of equivalent expressions to be recognized.

3.4 Recognition of Loop Invariants

A common optimizing technique removes code from inside loops if the execution of that code does not depend on the iteration of the loop. Thus a loop of the form

```
              DO 1 I=1, 10
              A(I)=0
   1          B=0
```

would be replaced by

```
              DO 1 I=1, 10
   1          A(I)=0
              B=0
```

Since many of EXPER's mutations change the boundaries of loops, techniques for recognizing when code can be removed from a loop can be useful in detecting equivalences. Conditions for detecting operations which can be removed from loops are given by Schaefer ([5], pages 122-134).

3.5 Hoisting and Sinking

Hoisting and sinking are related to removal of code from loops in that they involve moving code which would be repeated several times tc a place where it will only be executed once. Thus the code fragment

```
              IF(A.EQ.0) GOTO 1
              C=0
              B=2
              GOTO 2
   1          C=1
              B=2
   2            etc.
```

could be replaced by

```
              B=2
              IF(A.EQ.0) GOTO 1
              C=0
              GOTO 2
      1       C=1
      2         etc.
```

Here the assignment B=2 has been hoisted to a position before the conditionally executed part of the program. Similarly sinking involves moving code to a position after some set of blocks. Mathematical rules for detecting the feasibility of hoisting or sinking are given on pages 115-119 of Schaefer [5].

3.6 Dead Code Detection

Dead code detection involves the identification of sections of a program which will either never be executed or whose execution is irrelevant. An example of typical dead code is the fragment below, in which the second assignment to A kills the first:

```
      A=B+C
      A=0
```

Schaefer [5] discusses rules for detecting dead code of this form on pages 156-161.

Another example of dead code is the case in which one or more basic blocks of a program are not connected to the rest of the flow graph. Then, as long as there is only one entrance to the program some section is never executed and can be removed entirely. This case is not expected to arise very often in programs written by humans, but mutations may easily make a large part of a program inaccessible from the entry node. For example, consider the following mutant of a program:

```
A=1
RETURN
B=A+2
   etc
```

Here the insertion of the RETURN statement has made everything between it and the next label which is referenced in a GOTO inaccesible. This type of dead code is easily detected by examining the flow graph of the program in question.

4.0 APPLICATIONS

Each of the above optimization techniques can be applied to detect equivalent mutants arising from one or more of the mutations applied by EXPER. Each is discussed below.

4.1 Constant Propagation

Constant propagation is most useful for detecting cases in which a mutant is not equivalent to the original program. Any mutant which could affect the known value of a variable can be detected in this fashion. The mutants most easily checked using this scheme are those involving replacement of one data reference with another (Constant Replacement, Scalar Replacement, Scalar for Constant Replacement, Constant for Scalar Replacement, Source Constant Replacement, Array Reference for Constant Replacement, Array Reference for Scalar Replacement, Array Name Replacement, Constant for Array Reference Replacement, Scalar for Array Reference Replacement, Array Reference for Array Reference Replacement, and Data Statement Alteration). Equivalences which may be detected, but with lower probability, are those involving changes to expressions (Arithmetic Operator Replacement, Unary Operator Removal, Unary Operator

Insertion, and Unary Operator Replacement). It is possible that equivalences involving actual changes to the program flow could be detected, but it should be much easier to detect these by comparing the flow graphs.

The mechanism for testing equivalence of mutants using constant propagation is as follows: At all points subsequent to the mutation compare the constant pools of the original program and the mutant. If they differ it is likely (though not certain) that the mutant is not equivalent to the original program. The following example demonstrates this form of detection:

Original Program		Mutant Program	
Code	Constants	Code	Constants
A=1		A=2	
B=A+2	(A,1)	B=A+2	(A,2)
etc	(A,1),(B,3)	etc	(A,2),(B,4)

Here a mutation has replaced the assignment of 1 to A with an assignment of 2. The change in the program is reflected in the changed constant pools following the mutation. Unless the assignments to A and B are dead it is reasonable to assume that the mutation is not equivalent to the original, and to try to develop test data which substantiate this assumption.

A firm test of non-equivalence can be made if one of the output variables appears in the constant pool for a RETURN statement. Then if the known value of this variable differs between the mutant and original programs we know that they are not equivalent, since they return different values on identical inputs. Obviously this test is valid only if some path exists from the entry node of the program being tested to the exit in question. This question can be resolved through dead code detection.

4.2 Invariant Propagation

As shown above invariant propagation is really a super-set of constant propagation, and thus it can be used to test all the sorts of mutants discussed under constant propagation. However since a great deal more information is carried by invariant relationships than by equality to a constant, this technique is far more powerful than constant propagation. It is particularly useful for testing the equivalence of mutants involving unary operators (i.e. Unary Operator Removal, Unary Operator Insertion, and Unary Operator Replacement). In many cases these operators only affect an expression if it has a certain relationship to 0. For example, taking the absolute value of an expression only changes the program if that expression evaluates to a value less than zero; negating an expression does not change anything if that expression always evaluates to 0, and so forth. These facts can be used as shown in the following example:

```
  Original Program                 Mutant Program
Code             Invariants      Code             Invariants
IF(A.LT.0) GOTO 1                IF(A.LT.0) GOTO 1
B=A              A≥0             B=!A             A≥0
```

In this case the conditional allows us to determine an invariant ($A \geq 0$), which in turn allows us to determine that the mutant program is equivalent to the original, since taking the absolute value of a positive quantity is a no-op.

The power of invariant propagation is vastly increased if the propagation and testing algorithms can take advantage of transitivity and replacement of one condition by a weaker one. Both of these features are demonstrated below:

```
            Original Program
      Code               Invariants
      A=0
      CONTINUE           A=0
    1 B=A                A≥0,A<5
      C=!B               A≥0,A<5,B=A
      A=A+1              A≥0,A<5,B=A
      IF(A.LT.5)         A≥0,A<5,B=A
          GOTO 1         A≥0,A<5,B=A

            Mutant Program
      Code               Invariants
      A=0
      CONTINUE           A=0
    1 B=A                A≥0,A<5
      C=B                A≥0,A<5,B=A,C=B
      A=A+1              A≥0,A<5,B=A,C=B
      IF(A.LT.5)         A≥0,A<5,B=A,C=B
            GOTO 1       A≥0,A<5,B=A,C=B
```

Note that the algorithm for generating invariant pools recognizes the loop in this program and is thus able to determine an upper bound on A. Obviously the invariants shown assume that no other branches to label 1 exist. The relation A=0 is replaced with the weaker A≥0 when the statement A=A+1 is detected at the end of the loop. Applying transitivity to the mutated pair C=!B and C=B allows us to decide that the mutant is equivalent to the original since B=A and A≥0.

There is one important feature of EXPER which is useful in generating invariant pools: EXPER can perform run-time checks of array bounds. Thus the following statements generate the invariant pool shown:

```
      Code               Invariants
    DIMENSION A(5)
    A(J)=0               J≥1,J≤5
```

Because EXPER checks array bounds any program aborts if J is less than 1 or greater than 5 in the assignment to A(J). Thus any program or mutant for which the given invariants did not hold prior to executing the assignment would have failed, and thus would obviously not be a correct program.

4.3 Common Subexpression Elimination

Kildall's equivalence partitions[3] provide an excellent way to handle mutations in assignment statements. Changing an arithmetic operator changes the expression placed in the equivalence class of the variable to which the assignment was made. Similarly, mutations which change an operand or destination in an assignment will produce changes in the equivalence classes following the assignment. Thus comparing equivalence classes can show that the mutant and original differ. As an example, consider the program and mutant shown below:

```
     Original Program
Code             Equivalence Classes
A=B+C
   etc.          {A,B+C}

     Mutant Program
Code             Equivalence Classes
A=B-C
   etc.          {A,B-C}
```

Comparing the two sets of equivalence classes shows that A has a different value in the two programs. As with constant propagation, we can assume that the mutant is not equivalent to the original program, and that test data should be developed to verify this assumption.

Common subexpression detection can also be used to show that a mutant is equivalent to the original program. If the mutation has changed part of an expression E to E', but E and E' are in the same equivalence class, then the mutant is equivalent to the original program. The example below demonstrates this situation:

```
   Original Program
Code               Equivalence Classes
A=B+C
D=B+C              {A,B+C}
X(A+E)=0           {A,B+C,D}

   Mutant Program
Code               Equivalence Classes
A=B+C
D=B+C              {A,B+C}
X(D+E)=0           {A,B+C,D}
```

Since A and D are in the same equivalence class we can conclude that the mutation (replacing A with D in the subscript) did not change the program. Note that since the equality of A and D is determined through assignment of a common expression this equivalence would be hard to detect using a simpler heuristic such as invariant propagation.

4.4 Recognition of Loop Invariants

Many mutations change the size of loops. The most obvious of these is the DO-loop End Replacement operator, although the GOTO Replacement operator can also alter loops. In cases where a loop has been changed to include more or less code than in the original, recognition of loop invariants can be used to decide whether or not the change is significant. Examination of the flow graphs should make cases in which loops have changed fairly easy to detect; thus it is easy to decide when to apply these tests. The basic application simply involves deciding whether or not the excess code (that is, the code which does not appear in both loops) is loop invariant. If it is then the expansion (or contraction) of the loop has not changed the outputs of the program. As an example, consider the following code:

```
   Original Program                      Mutant Program
     DO 1 I=1,10                         DO 2 I=1,10
     A(I)=0                              A(I)=0
   1 CONTINUE                          1 CONTINUE
   2 B=0                               2 B=0
```

The mutation above expanded the DO-loop to include the assignment of 0 to B. Since this assignment is loop-invariant it does not matter whether it is done 10 times inside the loop or 1 time outside it. Thus the original and mutant programs are equivalent.

4.5 Hoisting and Sinking

These tests are used in situations similar to those in which testing of loop-invariants is used, except that they apply to cases in which the code skipped or included by a branch is changed. Candidates for this sort of change include GOTO Replacement and Statement Deletion. In these cases the mutant and original programs are equivalent if the code added to or removed from a basic block can be hoisted or sunk out of that block. Consider the following example:

```
    Original Program                    Mutant Program
   IF(A.EQ.0) GOTO 1                  IF(A.EQ.0) GOTO 2
   A=A+1                              A=A+1
 2 B=0                              2 B=0
   GOTO 3                             GOTO 3
 1 B=0                              1 B=0
 3   etc                            3   etc
```

In this case B is set to zero regardless of whether we do it at line 2 or line 1. A more compact form is produced by hoisting the assignment to B, namely

```
     B=0
     IF(A.EQ.0) GOTO 3
     A=A+1
   3    etc
```

Because this hoisting is possible the mutant is equivalent to the original program.

Because the code skipped by the statement "GOTO 3" can be hoisted the branch is unnecessary. Thus the hoisting test will also show that the mutant derived by deleting this branch is equivalent to the original program.

4.6 Dead Code Detection

As mentioned above this test is very important in guaranteeing the reliability of tests based on invariant propagation (including constant propagation). It can also be used to test the equivalence of some mutants in its own right. The equivalences which are most likely to be detected by this technique are those arising from mutations that alter the flow graph in some way. Such mutants include Statement Analysis (since this mutant replaces any statement with an abnormal exit), Statement Deletion (if GOTO or RETURN statements are deleted), Return Statement Replacement, and GOTO Replacement.

The best way to use dead code detection to test mutants of this form is to examine the flow graphs of the two programs. If any node appears in the mutant which is not connected to the rest of the graph it is reasonable to expect that the mutant is not equivalent to the original. (The only exception being the case in which the disconnected node consists only of dead assignments. This situation is discussed in general below). An example involving Return Statement Replacement is shown below:

```
    Original Program             Mutant Program
Code            Flow Graph    Code            Flow Graph
                ---------                     ---------
A=1             |       |     A=1             |       |
                |       |                     ---------
B=2             |       |     RETURN
                |       |                     ---------
C=3             |       |     C=3             |       |
                ---------                     ---------
```

The RETURN statement has broken the original single node into 2 nodes with no connection between them. Thus one can conclude that since code which is executed in the original program (assuming the node is accessible in the first place) is not executed in the mutant, the two are different.

A slightly different application of dead code detection involves making sure that mutated code is not inaccessible or dead in the first place. If it is then the mutant must be equivalent to the original program. This application is identical to the application in compiler optimization where code is identified as dead and excluded from the final output. It applies to all mutant operators. An example of this sort of analysis in testing equivalence is shown below:

```
Original Program          Mutant Program
    A=1                       A=2
    A=B+C                     A=B+C
```

Here the first assignment to A is killed by the second assignment, and thus any change to its right-hand side is insignificant. A more drastic example shows inaccessible code. Again, the mutant to code which can never be executed is unimportant.

```
Original Program          Mutant Program
   GOTO 1                    GOTO 1
   A=2                       A=-2
  1    etc                  1    etc
```

Some cases in which a mutation has killed a block of code can be detected by using invariant propagation. The program fragment shown below shows how this can happen:

```
                Original Program
            Code                  Invariants
        IF(A.GT.B) GOTO 1
        FLAG1=.TRUE.             A≤B
        IF(A.LT.B) GOTO 2        A≤B
        FLAG2=.TRUE.             A=B
      2     etc                  A≤B

                Mutant Program
            Code                  Invariants
        IF(A.GT.B) GOTO 1
        FLAG1=.TRUE.             A≤B
        IF(A.LE.B) GOTO 2        A≤B
        FLAG2=.TRUE.
      2     etc                  A≤B
```

Here the mutation has replaced the test A<B with the test A≤B. However, the invariant pool tells us that A is always less than or equal to B, and thus the branch will always be taken, and the assignment to FLAG2 is dead. Note that without knowing the relationship between A and B it is impossible to determine that this assignment is dead.

5.0 AN EQUIVALENCE TESTING POST-PROCESSOR FOR EXPER

The above ideas for determining equivalence can be applied in a post-processor to EXPER in order to reduce the time spent by the user dealing with equivalent mutants. This processor should be run after the mutants have been executed on the test data, since experience shows that as many as 90 per cent of the mutants can be eliminated on the first testing run. Of the remaining mutants, those which are found by the post-processor to be equivalent are flagged as such and the user need not consider them further. Only those which are not found to be equivalent are analyzed by the user to improve his test data. At any point the user can manually over-ride the post-processor by declaring a live mutant to be equivalent to the original program or by declaring one that was thought to be equivalent to be live again.

The analysis proceeds much as it would in a compiler, with a few exceptions which arise due to the fact that we do not necessarliy want to produce efficiently optimized code. For instance, it is not important that we worry about compiler-generated constants, since they can never be mutated.

The first step is to express the original program as a flow graph, as discussed above. This step may be done as part of EXPER's parsing or other processing of the program. As each live mutant is tested for equivalence to the original program a flow graph is generated for it. In many cases this flow graph will be isomorphic to the original so that only the contents of one node need to be modified. In more complex cases, where the shape of the flow graph is changed, the mutant's flow graph can still be derived from the original. EXPER represents mutants as a descriptor record describing the change made to the original program. These records fully describe the mutant, and thus allow the mutant's flow graph to be derived without re-generating it from a source program.

Just as it is expected that mutant flow graphs can be efficiently derived from the original flow graph, it is also expected that the invariant and common expression pools described above will not have to be computed for each mutant. Instead, the pools for the original can be computed at parse time and the mutant's pools derived from them. As suggested above, many mutations cause a relation to change, move an expression from one equivalence class to another, or make similarly limited changes in the pools. These changes can be easily detected using the descriptor record of the mutant, and can be made as local modifications to the pools. Obviously, care will have to be taken that any side effects of these local changes are detected, but doing so should be significantly less expensive than regenerating the entire pool.

The invariant and common expression pools described above can be combined into a single pool by replacing the individual variables or constants involved in invariant relationships with the equivalence class sets used to recognize common expressions. Note that using this scheme the relationships "equal to" and "not equal to" do not need to be explicitly represented, since if two objects are in the same set they must be equal, whereas if they are not in the same set they must be unequal. If the entire structure of sets and relationships is represented as a directed graph whose nodes correspond to sets and whose edges to relationships (obviously the edges must be labelled as to what relationship) then the problem of applying transitivity becomes one of simply following either edges labelled '$>$' and '$\geq$' or edges labelled '$<$' and '$\leq$' until either the desired relationship is derived or no edges with the appropriate labels remain. Note that no cycles can occur which involve such paths. Assume such a cycle did exist, for instance a path using only edges marked '$<$' or '$\leq$' from node A to node B and back to node A. Since a path from A to B exists, transitivity implies that for any X in A and Y in B, $X \leq Y$. However, because a path from B to A exists we also have the statement $Y \leq X$. Because X and Y are in different sets we know that X is not equal to Y, and thus the derived relationships are contradictory.

Representing the pools in this manner allows a great deal of flexibility in testing equivalences. The following example shows how this can happen:

```
                Original Program
             Code                  Invariant & Expression Pool
          A=B+C
          D=E+F                    {A,B+C}
          IF(B+C.LE.D) GOTO 1      {A,B+C},{D,E+F}
          X(A+G)=0                 {A,B+C}>{D,E+F}
             etc.

                Mutant Program
             Code                  Invariant & Expression Pool
          A=B+C
          D=E+F                    {A,B+C}
          IF(B+C.LE.D) GOTO 1      {A,B+C},{D,E+F}
          X(D+G)=0                 {A,B+C}<{D,E+F}
             etc.
```

In this example the conditional branch allows a relationship between B+C and D to be deduced. Because the relationship is then applied to all elements equal to either B+C or D we can conclude that replacing A with D in the subscript yields a mutant subscript which is always greater than the original subscript. This fact suggests that the mutant is not equivalent to the original.

Once the modified invariant pool described above is formed it is used to aid the detection and removal of dead code. Once dead code has been removed the mutant and original are compared to see if they are obviously equivalent. If so, the mutant is placed in the equivalent mutants pool and not procesed further.

Since dead code is irrelevant to the state of the program, removing it will not make the invariant pools incorrect. However, it may be possible that removing dead code enables invariant conditions to be strengthened. The following example shows how this can happen:

```
    Original Program             Mutant Program
        A=0                          A=0
        IF(C.GT.D) GOTO 2            IF(C.GT.D) GOTO 2
        IF(C.LT.D) GOTO 1            IF(C.LE.D) GOTO 1
        A=A+1                        A=A+1
      1      etc                   1      etc.
```

The mutation above is a case in which changing a conditional (C.LT.D became C.LE.D) kills a block of code. The section of code killed is the increment of A. Because of this increment the strongest statement that can be made about A at label 1 is $A \geq 0$. Because the increment of A is dead in the mutant this invariant can be tightened to A=0, assuming no other branches to label 1 exist.

Those mutants which have not been eliminated by manipulation of the flow graphs are then tested for equivalence based on loop invariants or the possibility of hoisting. Any equivalences thus found are placed in the equivalent mutants pool. Again, it is often possible to apply these tests to the original program at parse time and deduce their results on a mutant from the mutant's descriptor record. Only rarely will it be necessary to actually test the mutant.

The final phase of the post-processor applies the invariant pools generated in the first phase to actual detection of equivalent mutants. In this phase many mutants may be automatically eliminated, especially those involving unary operators. This is also a convenient place to provide user interaction in the equivalence determining process. The processor would be driven by a set of rules describing sufficient conditions for equivalence of a mutant to the original. For instance, there might be a rule concerning absolute values which can be conceptualized as "Insertion of absolute value preserves equivalence if its argument is greater than or equal to 0". When the processor is unable to decide whether a rule is applicable by itself, it turns to the user for help. This help is requested by forming a question from the rule and posing this question to the user. For example, if an absolute value operation has been inserted in front of a variable which does not appear in the invariant pool for that statement the processor could prompt "Is X always greater than or equal to 0?". If the user replies in the affirmative the mutant is flagged as equivalent.

6.0 REMARKS

It has been shown above how many techniques from compiler optimization can be applied to detect equivalent mutants of a program. Several areas remain to be explored however.

In the EXPER system only first order mutations are considered (i.e. mutants coming from one program change), but conceivably some higher order mutants may be worthy of consideration. In many cases the heuristics described here can be extended very easily to detect equivalent mutants of higher order. It is also true that in many cases equivalence can be tested transitively, i.e. if program P is equivalent to P′ and P′ is equivalent to P′′ then P is equivalent to P′′. However, it is often true that a high-order mutant can be equivalent to some program without having intermediate mutants equivalent to either. For instance the following program fragments are equivalent:

```
        IF(I.EQ.1) GOTO 1
```

and

```
        IF(--I.EQ.0) GOTO 1
```

However, neither is necessarily equivalent to either of the intermediate mutants

```
        IF(I.EQ.0) GOTO 1
```

or

```
        IF(--I.EQ.1) GOTO 1
```

Fortunately the problem of equivalence of high order mutants is not a serious problem because of the Coupling Effect: Test data that screens out all first order mutants will screen out all higher order mutants [2]. Thus only first order mutants need to be considered in evaluating test data

A more interesting problem involves the detection of equivalences which are very dependent on the form in which the programmer has chosen to express his algorithm. As an example consider the fragment below which tests whether or not a number N is prime.

```
        IF(N.LE.2) GOTO 3
        L=N-1
        DO 1 I=2,L
        IF(N.EQ.(N/I)*I) GOTO 2
      1 CONTINUE
      3 PRIME=.TRUE.
        RETURN
      2 PRIME=.FALSE.
        RETURN
```

It is really only necessary to let the DO loop run from 2 to INT(SQRT(N)). The test N.LE.2 means that only N greater than or equal to 3 will be used as upper limits for the loop. Since INT(SQRT(3))=1, INT(SQRT(N))≤N-2. Thus the mutation which replaces L with --L in this loop is equivalent to the original. Because the equivalence of this mutant is so closely related to the conceptual nature of the program it seems very difficult to automatically prove it. This problem might be solved through the interactive part of the post-processor. Specifically, it is easy to find out where the mutant occurred, and the processor could simply ask "Is it acceptable for this loop be executed from 2 to L-1?".

Several techniques for detecting equivalent mutants have been described. These techniques should be capable of finding a significant number of cases in which a mutant is equivalent to the original program, since experience indicates that most equivalences are very simple ones. Often they involve the insertion of the absolute value operator, a case that is particularly easy to detect using invariant propagation. More complex equivalences can be tested interactively with the user. The questions thus posed should help the user decide whether or not to manually declare a mutant equivalent to the original program.

Several questions concerning equivalence detection remain open. At several points in the above discussion it is asserted that the data needed to determine equivalence (e.g. flow graphs, invariant pools, etc.) can be derived efficiently from the corresponding data for the original program and the mutant's descriptor record. While these assertions are undoubtedly true in many cases, exactly how often remains unknown. Further experimentation is required in this area, particularly with regard for the following questions:

1. In what fraction of the cases is it necessary to generate a flow graph for a mutant from scratch?
2. In what fraction of the cases is it necessary to regenerate the invariant pools for a mutant?

3. It is unlikely that a change to an invariant pool will affect only that pool. On the average, how many pools will be affected? How does the cost of determining all affects compare to the cost of re-computing the invariant pools?

REFERENCES

1. Davis, Martin *Computability and Unsolvability* (McGraw-Hill Co., New York, New York: 1958).

2. DeMillo, Richard A.; Lipton, Richard J.; and Sayward, Frederick G. "Hints on Test Data Selection: Help for the Practicing Programmer" reprinted from *Computer* 11, 4 (April 1978), pp. 34-43.

3. Kildall, Gary A. "A Unified Approach to Global Program Optimization" in *Conference Record of ACM Symposium on Programming Languages*, pp. 194-205, 1973.

4. Lipton, Richard J. and Sayward, Frederick G. "The Status of Research on Program Mutation", reprinted from *Digest for the Workshop on Software Testing and Test Documentation*, Dec. 1978, pp. 355-373.

5. Schaefer, Marvin. *A Mathematical Theory of Global Program Optimization* (Prentice Hall, Englewood Cliffs, N.J., 1973)

CPMS USERS GUIDE

Allen Acree

July 1, 1979

Document CPMS_1.1

INTRODUCTION

The COBOL Pilot Mutation System (CPMS) is being developed at the Georgia Institute of Technology by Allen Acree, Rich DeMillo, Jeanne Hanks, and Fred Sayward. It is based in part on the Pilot Mutation System (PIMS) for FORTRAN designed at Yale University, and implemented at Yale University, Georgia Institute of Technology, and the University of California, Berkely.

Program mutation is a methodology for program testing. The underlying assumption is that programmers produce programs that are, in some sense, nearly correct. The goal of the mutation system is to aid in the selection of good test data by taking advantage of this fact. A _mutation_ of a program P is a program P' that differs from P in only a single minor change, such as substituting one variable for another in an assignment or changing a + to a - in an arithmetic expression. Usually the number of simple mutants of P grows quadratically with the size of P. Naturally, some of these mutations will produce mutant programs that are functionally equivalent to the original, but for the others we should be able to find test data that will distinguish between the original program and any mutant.

CPMS is designed to take as input a fixed program P, and to automatically produce mutants of it according to a set of mutant operators. The system will then accept test cases from the user, run the original program and all its mutants on it, and tell the user how many mutants have been "killed". (A mutant is killed when it fails by program fault or produces a different output than the original program.) The aim, of course, is to kill all the mutants, or at least to kill enough so that the user is reasonably certain that those remaining are functionally equivalent to the original and could never be killed. At this point the user has a set of test data that is sufficiently powerful to distinguish between the original program and all its simple (nonequivalent) mutants. According to the _coupling hypothesis_ this test data will also be sufficiently powerful to distinguish between the original program and any other program "close" to it. (Multiple mutations.) This hypothesis has been proved for certain classes of programs and for certain definitions of "close", and theoretical work continues in this area. Rececnt experiments with higher order mutants of FORTRAN routines also support this hypothesis.

Thus the user can, with the aid of CPMS, produce test data that will distinguish between the program used as input and any program "close" to it. Since we assume that the program used as input is close to a correct program, the test data will be sufficient to distinguish between the input program and the correct program, if they are not equivalent. So the test data will be sufficient to demonstrate program correctness, to a high degree of certainty.

IMPLEMENTATION NOTES

The user of CPMS provides the name of the file containing the source program. This program should be in the subset of the COBOL language specified elsewhere. CPMS parses this source program into an internal form suitable for interpretive execution. This internal form is also suitable for "decompilation", and the user is provided with a decompiled version of his program. This "source listing" may not be textually identical to the original source, but it should be equivalent.

The system then produces an internal file of all mutations of the original program. These are stored, not as complete programs, but rather as short descriptions of how a mutant is to be created. The user is then asked to provide a file or files of test data for his program. These files may be created outside CPMS using the editor, or they may be created "on the fly" in CPMS, with editing capability being restricted to backspace and line delete. However the user choses to provide the input files, CPMS interpretively executes the source program on this test data, saving the output. The user may examine the output and decide whether or not to accept it. If he does, then the test data is run against all enabled mutants, and the results of each are compared to the results of the source. A mutant producing a different result is marked "killed". The user is then presented with a statistical summary. If he wishes, he may also examine more detailed information about the mutants still living. Then the cycle repeats until either an error is uncovered in the original program, or the user is satisfied that all remeining mutants are equivalent to the original. A CPMS experiment may be interrupted and continued later, with the system saving all information necessary for the resumption of the run.

In response to the experience of trying to transfer PIMS from one environment to another, we have decided to try to do as much as possible to isolate machine dependencies. At the risk of possible inefficiencies, we will concentrate references to file access techniques, character storage, word length, and such machine- and operating system-dependent features in a few small routines. For example, PIMS contained 22 random access calls in the DEC FORTRAN dialect. Each of these had to be rewritten as a PRIMOS call during the transfer procedure. In CPMS, all random access will be through the routines REARAN and WRTRAN. Those two (small) routines are all that need to be modified to interface CPMS with a different operating system. Some machine dependency is tolerated in the interpretive execution phase of CPMS, since this is the most time-consuming phase of the mutation process. However, this dependency is kept to a minimum even here. The buffers used in interpretively executing programs are integer arrays of one or two dimensions. The sizes of the arrays are parameters. We assume in designing these arrays that a single integer consists of at least 16 bits. (i.e. integers are restricted, wherever possible, to a range of +/- 32783.)

NOTES ON THE COBOL PILOT MUTATION SYSTEM

1. We limit ourselves to a simple subset of the language.

2. We limit ourselves to two nonrewindable sequential input files and two nonrewindable sequential output files. This should be sufficient for such common applications as making sorted transactions against a sorted master file and producing a transaction report and an updated master file.

3. Rather than providing for a "predicate subroutine" as in PIMS we will simply check mutant output against original program output to determine whether they have read the same number of input records and produced identical output files. It is believed that just checking record counts on input and output will eliminate many mutants without more detail comparisons.

4. Mutations to be performed:

1	DECIMAL ALTERATION - Move implied decimal in numeric items one place to the left or right, if possible.
2	REVERSE TWO-LEVEL TABLE DIMENSIONS
3	OCCURS CLAUSE ALTERATION - Add or subtract one from an OCCURS clause.
4	INSERT FILLER - of length one between two items in a record.
5	FILLER SIZE ALTERATION - Add or subtract one from length.

6 ELEMENTARY ITEM REVERSAL

7 FILE REFERENCE ALTERATION

8 STATEMENT DELETION - Replace by null operation.

9 GO TO --> PERFORM

10 PERFORM --> GO TO

11 THEN - ELSE REVERSAL - Negate condition.

12 STOP STATEMENT SUBSTITUTION

13 THRU CLAUSE EXTENSION

14 TRAP STATEMENT REPLACEMENT

15 SUBSTITUTE ARITHMETIC VERB

16 SUBSTITUTE OPERATOR IN COMPUTE

17 PARENTHESIS ALTERATION - Move one parenthesis one place to the left or right

18 ROUNDED ALTERATION - Change ROUNDED to truncation, and vice versa.

19 MOVE REVERSAL - reverse direction of move in simple MOVE A TO B, if the result would be legal in COBOL.

20 LOGICAL OPERATOR REPLACEMENT

21 SCALAR FOR SCALAR REPLACEMENT - Substitute one (non-table) item reference for another, where the result would be legal.

22 CONSTANT FOR CONSTANT REPLACEMENT

23 CONSTANT FOR SCALAR REPLACEMENT

24 SCALAR FOR CONSTANT REPLACEMENT

25 NUMERIC CONSTANT ADJUSTMENT

COBOL SUBSET ACCEPTED BY CPMS

```
IDENTIFICATION DIVISION.

PROGRAM-ID.  program-name.

[AUTHOR.  comment-entry.]

[INSTALLATION.  comment-entry.]

[DATE-WRITTEN.  comment-entry.]

[DATE-COMPILED.  comment-entry.]

[SECURITY.  comment-entry.]

[REMARKS.  comment-entry.]


ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

[SOURCE-COMPUTER.  comment-entry.]

[OBJECT-COMPUTER.  comment-entry.]

[SPECIAL-NAMES.  [ [C01 IS mnemonic-name]

[INPUT-OUTPUT SECTION.

FILE-CONTROL.

    SELECT file-name ASSIGN TO {INPUT1 | INPUT2 | OUTPUT1 |
OUTPUT2}...]


DATA DIVISION.

FILE SECTION.

FD file-name RECORD CONTAINS integer CHARACTERS
    [LABEL RECORDS ARE {STANDARD | OMITTED}]
    DATA RECORD IS data-name.

    level-number {data-name | FILLER}
       [REDEFINES data-name-2]
       [{PICTURE | PIC} IS character-string]
       [OCCURS integer TIMES]
       [VALUE IS literal]
    ...
```

```
...

[WORKING-STORAGE SECTION.

    [77 level entries.]

    [record entries .]...]


PROCEDURE DIVISION.

[paragraph-name.]

     ADD {ident-1 | lit-1} [ident-2 | lit-2]...  {TO |
     GIVING} ident-m
    [ROUNDED] [ON SIZE ERROR imperative-statement] .

     CLOSE filename-1 [filename-2] ...   .

     COMPUTE identifier [ROUNDED] = arithmetic-expression
     [ON SIZE ERROR imperative]

     DIVIDE {ident-1 | lit-1} {INTO | BY} {ident-2 | lit-2}
    [GIVING ident-3] [ROUNDED] [ON SIZE ERROR imperative] .

     EXIT.

     GO TO procedure-name-1 [ [procedure-name-2] ...
    DEPENDING ON identifier] .

     IF condition {statement-1 | NEXT STATEMENT}
    [ELSE {statement-2 | NEXT STATEMENT} ] .

     MOVE ident-1 TO ident-2 [ident-3] ...   .

     MULTIPLY {ident-1 | lit-1} BY {ident-2 | lit-2}
    [GIVING ident-3] [ROUNDED] [ON SIZE ERROR imperative] .

     OPEN [INPUT filename-1 [filename-2] ]
    [OUTPUT filename-3 [filename-4] ]

     PERFORM procedure-name-1 [THRU procedure-name-2]
    [{ident-1 | integer-1} TIMES] .

     READ filename RECORD [INTO identifier]
    AT END imperative

     STOP RUN.

     SUBTRACT {ident-1 | lit-1} [ident-2 | lit-2] ...  FROM
     {ident-m | lit-m}
    [GIVING ident-n] [ROUNDED] [ON SIZE ERROR imperative] .

     WRITE record-name [FROM identifier-1]
    [AFTER ADVANCING {ident-2 | integer | mnemonic} LINES] .
```

THE CPMS RUN

The four phases of the CPMS run are the ENTRY phase, the PRE-RUN phase, the MUTATION phase, and th POST-RUN phase. The ENTRY phase is executed only when the user first enters the system. Thereafter the PRE-RUN, MUTATION, and POST-RUN phases are exected cyclically.

I. The entry phase.

The session will begin when the user enters the system by logging in and typing seg cpms (In either upper or lower case.) If all is well, the system will respond:
WELCOME TO THE COBOL PILOT MUTATION SYSTEM
followed by:
PLEASE ENTER THE NAME OF THE COBOL PROGRAM FILE:
The user should do just that. CPMS creates several working files of its own, whose names are variations of the source file name formed by adding suffixes to it. The system checks to see if those working files already exist. If they do, the user can either continue the previous run on that source file where he left off, or he can start over from scratch. Therefore, if the working files already exist, the system asks:
DO YOU WANT TO PURGE WORKING FILES FOR A FRESH RUN ?
If a new run is needed the system begins with the message
PARSING PROGRAM
A syntax error in the source program automatically aborts the CPMS run. The user must correct the error and re-enter the system. Errors are reported to the user as a source program line number and the probable cause.

The system the issues the messages
SAVING INTERNAL FORM
CREATING MUTANT DESCRIPTOR RECORDS

II. The pre-run phase.

In this phase the user supplies test data and turns on mutants. The system asks
DO YOU WANT TO SUBMIT A TEST CASE ?
and the user should respond YES or NO. The system will ask
WHERE IS INPUT1 ?
(if there is a SELECT statement for INPUT1)
to which the user should respond HERE or <filename>
If it is HERE, the user enters the input data directly, ending with the control-C for end of file.
The system then goes through the same procedure for INPUT2, if it has been named in a SELECT statement.

At this point the system will execute the program interpretively on the test input. After finishing, the output on files OUTPUT1 and OUTPUT2 will be displayed . The user is asked:

IS THIS TEST CASE ACCEPTABLE ?
To which the user should respond YES or NO.
If YES, the test case (input and output, along with the time used, and counts of records read) are catalogued for later use with mutant programs. If NO, the test case is purged from memory.

This process of entering test cases iterates until all input cases for this pass have been entered.
At this time the system will ask
WHAT NEW MUTANT TYPES ARE TO BE CONSIDERED ?
The user should respond ALL or NONE or SELECT or should give the numbers of the mutant types to be used next. SELECT causes the system to list each type that has not yes been considered, and then ask for types.

The list of numbers should be terminated with the command STOP.

III. The Mutation Phase

At this time the test cases will be run against the mutant program. The time that this takes depends on the number of test cases presented, the length and "density" of the program, and the types of mutants currently being considered.
After all the test cases have been executed for each mutant still alive, the system will display the statistics of the run, indicating the number of mutants created and the number still alive of each type that has been considered, as well as summary counts of how the dead mutants died.

IV. The Post-Run Phase

Now the user has a chance to view the mutants still remaining (either all of them, or selected types, or one randomly selected mutant of each type), or he can send information about the run to an output file for later printing. To end the post-run phase the user types either HALT, ending the session, but saving the temporary files for future resumption, or LOOP sending the system back in a loop to the pre-run phase to enter more test data and/or consider new mutant types.
The user may terminate the session at any time the a command is requested by typing KILL.
The user can receive an explanation of his options at many points in the cycle by typing HELP.

OTHER WORK IN PROGRESS

Mutation analysis depends on our ability to restrict our attention to single mutations, to avoid a combinatorial explosion in the number of mutations performed. This is justified by the coupling hypothesis that says that any test data that is strong enough to distinguish between a program and all its nonequivalent single mutants is also strong enough to distinguish the original program from more complex mutants. A version of CPMS has been prepared to test this hypothesis by reandomly sampling higher order mutants, executing them on test data that has been found sufficient for first order mutants, and reporting any higher order mutants that are not eliminated, along with statistics on how all of the other mutants were eliminated. It is hoped that this will provide us with an estimate of the likelihood that a more compilcated error would escape detection in the mutation process.

PILOT MUTATION SYSTEM (PIMS)
USERS' MANUAL

by

Donald M. St. Andre

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

A familiarity with the PRIMOS operating system and the file management system, as applied to the PRIME-400 computer, and a familiarity with the Software Tools Subsystem [17] is assumed. Detailed discussion of the respective command syntax will be avoided except where required for clarity or completeness.

This paper is a discussion of the operational processes and the implementation of a pilot system for performing program mutations. Since this is an operational discussion I will not attempt a detailed theoretical study. Such studies are available in [14,15].

I will also describe the operation of a pilot mutation system for performing mutations on FORTRAN subroutines as a means for testing program correctness. The system will accept an input file which is assumed to contain a FORTRAN subroutine valid in the language subset (see Appendix B). Mutations are generated according to operator commands and each mutant is checked for correctness.

The system is divided into three operational phases: a pre-run phase, a mutation phase, and a post-run phase. In the present implementation, the entire system is resident in approximately 67K words of virtual memory. The phases are independent enough procedurally that they may be overlayed for use on a smaller memory configuration.

In the pre-run phase files are opened or created and instructions are accepted for the processing in other phases. If an "internal-form" file does not exist, a run is called an **initial** run. Throughout all phases of system operation, activities are different for initial and subsequent runs.

In the mutation phase, mutants are created and tested. It is the mutation phase that is the central part of the system. During the post-run phase, statistics are displayed about the mutations tested thus far in the processing. In addition, files are closed for use in subsequent runs. One feature of the system design is the ability to execute the three phases in sequence and then loop back to the pre-run phase for additional processing. In this manner, a user can operate the system and gain insight which is then used in tailoring the responses in the next pass. This repetitive refinement of the test data contributes greatly to the rapid convergence on a set of acceptable test data[14].

CHAPTER II

THEORETICAL OVERVIEW

An increasing productivity burden on software developers has contributed to the increased use of several aids for the design, implementation and debugging of large-scale software products. However, these aids are intended for the actual programmers and first-level management. They provide qualitative descriptions rather than quantitative information that may be used throughout a management hierarchy. The typical manager will ask questions like, "How close is the project to something that the users will find acceptable as a first release?" and, "How well has the program been tested?" The techniques of modularization[14], structured programming[14], and program verification[9,10] do not seem to answer these and similar questions. This lack of answers appears understandable because management should not be expected to understand programming languages and/or sophisticated mathematics. In this chapter, we will explain how a new testing approach known as program mutation[15] can be used to manage software effectively.

A statement of the program testing problem, as seen by management, might be: Given a program module and its associated test data, how well does the data test the module--in quantitative terms? To solve this problem, program mutation provides a quantitative measure of the "goodness" of the test data. We make the assumption that the better the test data (i.e., the more complete) the more thorough a program has been tested. And in a somewhat simplistic fashion, the more thorough the testing, the more confidence can be placed in a program's correctness.

A pilot system which performs program mutation[15] produces a "score" which indicates the adequacy of the test data. The users attempt to improve upon this "score" by either augmenting the test data or by answering "questions" about the program being tested. These questions address the essence of the program by forcing the user to compare alternate forms of a given statement and to make a decision whether the many forms are equivalent. This process of supplying test data, answering questions and interpreting results continues interactively until the user is satisfied with the quality of the test data. Meanwhile, all of the data remains available at each iteration for management review so that a quantitative answer to "how well a program has been tested" can be obtained.

Mutation Methodology

Program testing cannot be deductive. We know this since program testing attempts to derive finite test data which implies general correctness. Test data of [illegible]

is known as "adequate test data." And, since adequate test data cannot in general be derived algorithmically[4], program testing is not deductive. For this inductive process, we are therefore trying to answer a fundamental question, "If a program is correct on some finite number of test cases, is it correct in general?" Several methods have emerged which allow one to gain confidence in test data adequacy. These methods include path analysis[1,2,5,6] and an associated technique, symbolic execution[7,8]. The basic idea of path analysis is to exercise all control paths within a program. Symbolic execution attempts to derive the test data necessary to do this. Test data known to exercise each flowchart path at least once is better than test data that does not. It should be apparent that the possibility of faulty analysis is very real[3].

Let us approach the problem of testing from a different viewpoint and assume that experienced programmers write programs which are either correct or are "almost" correct. Stated more formally:

> If a program is not correct, then it is a "mutant"--that is, it differs from a correct program by simple, well understood errors [11].

Errors have been found to be caused by one of three broad categories[12]. First, the specifications may be misunderstood. Second, the specifications may be implemented incorrectly these are the so-called "logical errors." Third, the errors may be of a purely clerical nature. The program mutation methodology can lead to the detection of all three error types[16].

CHAPTER III

PIMS USER'S GUIDE

This document, in conjunction with the appendices, describes how to use a terminal to operate PIMS on the PRIME-400 computer. All communications to PIMS must be in capital letters. Lower case letters are treated as errors by PIMS.

PIMS consists of three sequentially executed phases which are called the "Pre-Run Phase," the "Mutation Phase," and the "Post-Run Phase." Throughout these phases errors may occur; and the types of errors detected by PIMS, the error messages, and PIMS' reaction to errors are described in "PIMS Error Messages" (see Appendix A). In this chapter it is assumed that no errors take place.

In the Pre-Run Phase the user tells PIMS what program is to undergo mutation analysis, describes those aspects of the program and the test data needed by PIMS to execute mutations, describes the types of mutations he wants done, and partially describes the contents of his output file. The user may also request that certain status information be displayed on the terminal. During the Pre-Run Phase the user may terminate his run, leaving his transient files unchanged, by issuing a KILL response as a reply to any PIMS prompt for input.

In the Mutation Phase PIMS creates and executes mutants. There is no user interaction during this phase.

In the Post-Run Phase the user completes his description of his output file. He may also request that certain status information be displayed on the terminal.

Running PIMS

In the explanations and examples that follow, "space" characters are significant and should be used exactly as shown. In addition, any response to a terminal question should be terminated by a "carriage-return" or "newline" character as appropriate for the user's specific terminal. To begin the execution of PIMS the user types the following command:

```
        OK, SEG RUN>PIMS (see note below)
```

PIMS responds, as soon as it has been loaded, by displaying the following message

```
        PRE-RUN PHASE
        ALL INPUT MUST BE IN UPPER CASE
```

NOTE:
Non-casual users of PIMS should consult with the PIMS staff for details about login and file integrity.

The Pre-Run Phase

The Pre-Run Phase consists of six sequentially executed parts, some optional depending on whether or not PIMS is being run for the first time on the given program. PIMS requests the name of the Raw Program File by displaying the following message:

```
ENTER THE RAW PROGRAM FILE NAME.
```

Raw Program files are created using a text editor prior to entering PIMS. A short tutorial on using this editor is available as a login option (see also Appendix F). The user types, on the next line, exactly six characters which tell PIMS the file in which the raw program resides. This also sets the file nomenclature convention.

PIMS File-Name Convention

The raw program file name is exactly six characters. We represent this six character name by the symbol <name>. Then the PIMS system files are created and may be accessed with the following suffixes.

```
<name>.I.....Internal Form File
<name>.T.....Test Data File
<name>.C.....Correctness Descriptor File
<name>.M.....Mutant Information File
<name>.O.....Report Output File
<name>.D.....New Test Data File
<name>.N.....New Mutant Information File
<name>.P.....Predicate-subroutine internal file
```

NOTE: The user is referred to Appendix F for the details of creating, editing, and maintaining the raw program files which will be processed by PIMS.

PIMS determines the run type, either an initial run on the program or a subsequent run, by searching for a file with the six character name entered and a suffix of ".I" If this file is found, the run is considered to be a subsequent run; if it is not found, the run is considered initial. Once the system determines that a run is subsequent, the user is given the opportunity to discard all previous files and start over.

During an initial run, PIMS accepts instructions about the routine being tested and any associated test cases. These instructions consist of the sub-parts as described below.

(1) Classification of the Formal Parameters

PIMS requests that the user categorize each formal parameter (for illustrative purposes let the variable be named X) by successively displaying, until all parameters are categorized, the following message:

```
          CATEGORIZE FORMAL PARAMETER X
```

The user then types the keyword corresponding to one of the categories INPUT, OUTPUT, or INPUT/OUTPUT, or types HELP if he has forgotten details and wants PIMS to display the command keywords.

(2) Mutant Correctness Option

To determine whether mutant correctness is determined by the "predicate subroutine" method or the "same as the program" method, PIMS displays the following message:

```
          IS MUTANT CORRECTNESS DEPENDENT ON A PREDICATE
          SUBROUTINE?

          TYPE YES OR NO
```

The user types in the appropriate reply. If YES is entered, PIMS displays the predicate subroutine statement it has found in the Predicate Subroutine File. The user creates this file prior to any initial runs with the appropriate file name as described under file name conventions.

```
PREDICATE SUBROUTINE STATEMENT
(the predicate subroutine name and formal
parameters)
```

(3) Creation of the Test Data File

At this point PIMS is ready to receive the test data from the user and signifies this by displaying the following message:

```
HOW MANY TEST CASES ARE TO BE SPECIFIED?
```

The user enters an appropriate count. For each test case, PIMS prompts the user to enter values for the input formal parameters of the program. First PIMS requests the values of the scalar parameters, then the one dimensional array formal parameters, and finally the two dimensional arrays, all in a manner to be described below. Requests for a specific test case are signalled by PIMS displaying the following message:

```
SPECIFY TEST CASE i:
```

The values of the scalars are requested by PIMS, five at a time, until all scalars are satisfied, by iterating the message

```
ENTER VALUES FOR V1 V2 V3 V4 V5
```

The user then inputs the numbers. Should the user have a large volume of test data, he may enter the keyword FILE at this point. The system will ask for a file name and read the test case data from that file. Single dimensioned arrays are input one at a time by PIMS requesting values for specific array elements until all values have been entered. For example, let the array be named A and its dimension be 7. (NOTE: PIMS gets this dimension from the program's DIMENSION statement--it must be either a constant or an input formal parameter scalar variable.) The session would be as follows: PIMS displays

```
ENTER VALUES FOR A(1) A(2) A(3) A(4) A(5)
```

The user would enter five numbers. PIMS then displays

```
ENTER VALUES FOR A(6) A(7)
```

The user would enter the final two numbers. PIMS then repeats this process for another one dimensional array or goes on to request values for the two dimensional arrays.

In the case where a user wants to input array partially defined, he enters UND rather than a number for the undefined array elements. Only numeric data of the type, INTEGER, may be processed. PIMS requests the values for two dimensioned array elements in a manner similar to that for single dimensioned arrays. The values are requested in row-major order, five at a time. For example, if A is of dimension (2,7) PIMS will make the following four prompts

```
ENTER  VALUES  FOR  A(1,1)  A(1,2)  A(1,3)  A(1,4)
A(1,5)
ENTER VALUES FOR A(1,6) A(1,7)
ENTER  VALUES  FOR  A(2,1)  A(2,2)  A(2,3)  A(2,4)
A(2,5)
ENTER VALUES FOR A(2,6) A(2,7)
```

(4) Additional Test Cases

When additional test cases can be added to the test case file of the given program, PIMS displays the following message:

```
HOW MANY NEW TEST CASES FOR THIS RUN?
```

The user then enters the appropriate count. PIMS then prompts the user to specify the new test cases in the same manner as described above in the "Creation of the Test Data File" subsection. The result is to extend the Test Data File. Test cases cannot be deleted.

(5) Addition of and Status of Mutant Types Considered

To see if new types of mutants are to be considered for this run, PIMS displays the following message:

WHAT NEW TYPES OF MUTANTS ARE TO BE CONSIDERED:

At this point the user has several options. He may type in any of the following replies

NONE - Part (5) terminates.

HELP ----- PIMS displays all the code names of the mutant types as described in Appendix C. Part (5) is then re-executed by PIMS.

ALL ----- Every type of mutation will be considered. Part (5) terminates.

T1 T2 ... Tn ----- T1 ... Tn are code names of mutant types (see Appendix C). Mutants of the listed types will be considered for this and subsequent PIMS runs. Part (5) is then re-executed by PIMS.

SENSE T1 T2 ... Tn ----- PIMS displays which of the listed mutant types are currently being considered and which are not. Part (5) is then re-executed by PIMS

SENSE ----- PIMS displays which of the possible mutant types are currently being considered and which are not. Part (5) is then re-executed by PIMS.

SELECT T1 T2 ... Tn ----- The user specifies to PIMS which of the mutants he wishes to know about.

(6) Display and Output of Past Results

In order to inform the user, in non-initial runs, that part (6) has begun, PIMS displays the following message:

REVIEW PREVIOUS RUN RESULTS

At this point the user has three options: (1) he may request that certain information concerning the mutant status before this run be displayed, (2) he may request that similar information be included in his output file, or (3) he may request that the mutation phase of PIMS be started. Option three is requested by typing MUTATE. The other two options cause this display to be re-executed, accomplished by a repeated displaying of the "REVIEW..." message. The information which can be displayed or included in the output file is the following:

(a) Displaying Information

All requests to display information on the screen begin with the word DISPLAY. Next there is a space followed by a keyword which describes the information to be put on the screen. The keywords are the following:

HEADER --- The program subroutine statement and the classification of the program's formal parameters are displayed.

CORRECTNESS --- The method of determining mutant correctness and, possibly, the subroutine statement of the predicate subroutine are displayed.

TITLE --- The PIMS run title is displayed.

STATUS --- The mutants' status before this run is displayed

(b) Outputting Information

All requests to have information included in the user's output file begin with the word OUTPUT. Next there is a space followed by a keyword describing the type of information to be included in the output file. The keyword is the following:

TESTCASES --- The previous test cases are included.

TESTCASE n --- The specified case, "n" is displayed.

The Mutation Phase

There is no user interaction during the mutation phase. In the event of a fatal processing error the host operating system will issue appropriate diagnostic messages.

The Post-run Phase

The Post-Run Phase consists of one part which is similar to part (6) (Display and Output of Past results) of the Pre-Run Phase. It can be called "Display and Output of New Results." In order to inform the user that the Post-Run Phase has begun, PIMS displays the following message:

```
POST-RUN PHASE
```

At this point the user has three options: he may request that certain information concerning the mutant results for this run and the mutant status after this run be displayed, he may request that similar information as well as mutant program listings be included in his Output File, or he may request that the PIMS run terminate. The first two options will be described below. The third option is requested by typing STOP and in each of the former two options the Post-Run Phase rè-cycles by PIMS displaying the following message:

```
POST RUN RESULTS
```

The information which can be displayed or included in the output file is the following:

(1) Display of Information

All requests to display information on the screen begin with the word DISPLAY. Next there is a space followed by a keyword describing the information to be put on the screen. The keywords are the following:

(i) HEADER - Same as in the Pre-Run Phase.

(ii) CORRECTNESS - Same as in the Pre-Run Phase.

(iii) TITLE - The FIMS run title is displayed.

(iv) RESULTS - The mutant results for this run are displayed.

(v) STATUS - The mutants' status after this run is displayed.

(2) Output of Information

All requests to have information included in the user's output file begin with the word OUTPUT. Next there is a space followed by a keyword describing the type of information to be included in the output file. The keywords are the following:

(i) TESTCASES - The new test cases are included.

(ii) MUTANTS - This keyword must be followed by additional keywords as follows: (The absence of keywords implies the ALL keyword.)

(a) ALL - A listing of all the live mutants is included.

(b) RANDOM - A listing of one randomly selected live mutant of each possible mutant type having live mutants is included.

(c) ALL T1 T2 ... Tn - A listing of all the live mutants for each of the given types is included.

(d) RANDOM T1 T2 ... Tn - A listing of one randomly selected live mutant for each of the given mutant types is included.

(e) HELP - FIMS displays the code names of the mutant types as described in Appendix D.

The default for all mutant types is no listing. Once a user decides to list a mutant type, via either an ALL or a RANDOM, he cannot later switch to no listing for the type. However, he may switch from ALL to RANDOM or from RANDOM to ALL for any mutant type.

CHAPTER IV

IMPLEMENTATION AND PORTABILITY DISCUSSIONS

Implementation

The PIMS program is written in FORTRAN as a feasibility study of automatic program mutations. In other words, we addressed the question: "Can the concept of program mutation be implemented in an automated system with reasonable runtime and computational simplicity?" The top levels are depicted in the following diagram.

```
Driver-----PRERUN-----CLRTTY,CIFILE,CCFILE,CTFILE,CMFILE,
         |            DISPLY,FILEOF,GETNAM,GETTYF,LTINFO,
         |            MNRECS,NEWTST,WPAST,WCSTAT
         |
         --MPHASE-----CLRTTY,DISPLY,MERGED,MERGEM,XNEWMU,
         |            YOLDMU
         |
         --POSTRN-----CLRTTY,CLENUP,DISPLY,NEWRES,WNEW
```

Beyond these levels, the control paths are relatively difficult to analyze from a maintenance programmer's point of view. The system data structures are almost entirely parametric and allocation is contained in multiple COMMON blocks. The large number of these blocks and their extensive use permits many side-effects to take place as the result of procedures invoked at every level. These side effects also greatly complicate the issues concerning the scope-of-control of procedures over their variables.

PIMS executes in a paged environment as a single image with about 67K bytes of address space required for both the data and the executable code. Since the program is logically divided into three distinct phases, the address space could be reduced with little impact on execution or operation by implementating the task as overlays or separately executed programs. This approach is recommended for implementing the PIMS program on most minicomputers.

Portability

During the implementation of PIMS on the PRIME-400, much collaboration took place between the research groups at Yale University and at Georgia Tech. My efforts used a sixteen-bit machine, the PRIME-400, while the Yale effort used a 36-bit machine, the DECsystem-10. The only medium available for transporting programs and data between these two systems was nine-track magnetic tape.

Although both vendors claimed to support ANSI compatible magnetic tapes, files could not be written by one system and read by the other without some form of intermediate processing. A list of this processing includes:

1) Records which were written with 80 characters per record and one record per block used different methods

of indicating the end-of-record. Specifically, DEC wrote an 82-character record with two trailing nulls (binary zeroes) while PRIME expected an 80-character record which included the two nulls.

2) Disk files with embedded carriage-return/line-feed sequences caused general havoc on both systems. The line-feeds usually had to be removed before any progress was made in processing the files.

A second and larger set of problems was encountered when FORTRAN source files were moved between the two systems. Obvious problems developed as a result of the differing word lengths and associated integer magnitude. The impact of many of these problems was lessened by the PRIME FORTRAN declaration for long-integers, INTEGER*4, which specifies a 32-bit integer. In order to perform the same functions on divers computers, it would be necessary to constrain all implementations. A discussion of those constraints is presented below.

First, all integer quantities should be kept within the range -32,767 to +32,767. This would allow the system to function on sixteen-bit machines that do not provide any long integer forms.

Second, the packing of multiple fields of data per integer variable should be avoided. This packing is also inefficient for the large word machines, but there are severe unpacking problems for the sixteen-bit machines. In our case, a 36-bit word was used on the DECsystem-10 to contain two nine-bit and one eighteen-bit fields. We were able to implement this using a long integer and obtain two eight-bit and one sixteen-bit fields. This loss of magnitude has not caused problems to date. Expanded range can be obtained by segmenting the use of the system to process smaller programs.

Third, the character processing that is done in the compiler and command processor should be done either with integer tokens subject to the first constraint. If integer tokens are not desirable, then at least characters should be processed in FORTRAN A1 format. The A1 processing will decrease the efficiency for the large word machines again, but the character routines will be portable. A good machine independent "string subroutine" package would probably be a better choice here.

Fourth, vendor supplied features and all I/O should be imbedded in user written procedures. In some cases, this will merely add a layer of run-time linkage with th parameters to the user routine being passed directly to the vendor feature or I/O routine. However, this layer allows other system routines to be substituted and code added to provided for the behavior of another technique. Modifying a single imbedded routine is much easier than searching for all of the uses of a specific statement throughout an entire system implementation.

We believe that the above techniques should be used in future implementation efforts. They were not used in our system but the benefits of these techniques became apparent as we tried to pass more and more FORTRAN source data between machines.

CHAPTER V

SUMMARY

A program mutation system was built and is now operational on a PRIME-400 minicomputer. The user may specify an input data file that contains a subroutine which is valid in a certain subset of the FORTRAN programming language. This subroutine is parsed, interpreted with user specified test data and the user is given the opportunity of determining the correctness of this test data either manually or through the use of a predicate subroutine that will determine the correctness of this base routine.

Once the user thinks he has an adequate test data set, this base program is modified in several ways and executed again after each modification. These modifications are called <u>mutants</u> and each mutant will either survive or die during its execution. All mutants that produce incorrect results or will not be valid subset programs will die. Those mutants that produce correct results will indicate to the user that further analysis is needed.

When further analysis is necessary the user must determine that either a live mutant is equivalent with other mutants and discard the equivalent mutants manually, or a live mutant might be eliminated by augmenting the test data set. The test data set is then modified as required and the mutants that remain live are executed again. Each time, the program will report various statistics about the live mutants remaining. When the user is satisfied with completeness of data achieved, the process stops. We now say that an acceptable level of test data adequacy has been reached. In more quantitative terms, some percentage of the total mutants will remain live at this point. For tests of the same subroutine, we interpret this percentage to mean, **The test data that shows the lowest number of live mutants at the time of comparison, has the most adequate test data.** We use this measure of test data adequacy to infer that the subroutine with the more adequate test data has been tested more thoroughly. In addition, the subroutine that has been tested more thoroughly is more probably the most correct.

Conclusion

Program Mutation is a valuable asset in program testing. The methodology greatly reduces the time and effort required to find errors in those programs studied thus far. Although there is a wealth of FORTRAN software in the world today, it is difficult to obtain and modify real-world software for anaylsis. This problem is the topic of current research.

The PIMS system discussed above was not designed with the property of such topics as portability and maintainability. I would like to suggest that these topics are suitable for investigation in their own right. The idea of program mutation is currently being extended to full ANSI-1969 FORTRAN, COBOL, and PASCAL with the hope that by examining the effects of the methodology in several programming languages, some insight may be obtained into the methodologies of program testing in general.

APPENDIX A

ERROR MESSAGES

This document describes the errors detected by PIMS during the interactive phases of a PIMS run, the messages displayed by PIMS on detection of an error, and the actions taken by PIMS after finding an error. The errors are divided into two classes: fatal and non-fatal, with fatal errors resulting in an abort of the PIMS run. Fatal errors only occur during the Pre-Run Phase of PIMS. The occurence of a fatal error or the user entering KILL during the Pre-Run Phase of PIMS leaves all transient files as they were before the PIMS run began. Once the user issues a GO command, thus signalling the end of the Pre-Run Phase, he will not be able to issue a KILL.

We also group the errors into those which occur during parsing the program and those which occur strictly as bad responses to prompts made by PIMS. Parser errors always are fatal and the PIMS parser is designed to abort on detection of a first error. That is, if the program has multiple syntactic errors, the PIMS parser will print an error message for only the first of them.

NOTE:
The user should never end a PIMS run by a CTRL-P. The CTRL-P termination will cause further PIMS runs to perform unpredictably.

Parser Error Messages

The messages which the user may encounter during the parsing of either the routine which is being tested or of the predicate subroutine are very similar to those generated by any FORTRAN compiler. Since a knowledge of FORTRAN is prerequisite to a meaningful use of the FIMS system, an understanding of typical compiler diagnostics is assumed.

One aspect of the FIMS diagnostics that is different from that of a typical compiler is the fatal nature of compiler diagnostics. In the event that any compile-time error is encountered within a module, the error is reported at that point and the compile is aborted. There is no attempt at compile-time error trace-back or recovery. If any errors are reported, the user is advised to scan the remainder of the routine manually for other syntax errors prior to a resubmission to FIMS. This manual scan should save man and machine time during the Pre-run Phase.

Interactive Error Messages

Pre-Run Phase

(a) The Program Name

(1) Message: ILLEGAL FILE NAME
Action: Repeat part (a).

(2) Message: NON-EXISTENT RAW PROGRAM FILE
Action: Repeat part (a).

(3) Message: FILE NAME CONFLICT - OUTPUT FILE ALREADY EXISTS
Action: None. Serves as a warning.

(b) The Run Type

(1) Message: ILLEGAL REPLY
Action: Repeat part (b).

(2) Message: PROGRAM NOT IN THE FIMS FORTRAN SUBSET
Action: Abort

(3) Message: THE FOLLOWING TRANSIENT FILES ARE MISSING:
Action: Abort

(4) Message: THE FOLLOWING TRANSIENT FILES ALREADY EXIST:
Action: Abort

(c) Program and Test Cases

(1) Message: ILLEGAL CLASSIFICATION
Action: (A) Display the legal classification codes.
(B) Repeat part (c-1) or the same parameter.

(2) Message: ILLEGAL REPLY
Action: Repeat Program and Test Cases

(3) Message: PREDICATE SUBROUTINE FILE DOES NOT EXIST
Action: Abort

(4) Message: BAD PREDICATE SUBROUTINE CALLING SEQUENCE
Action: (A) Display the program's formal parameters and their classifications.
(B) Display the predicate subroutine statement.
(C) Abort

(5) Message: ILLEGAL VALUE
Action: Repeat the request for data on the same input formal parameter(s). User's input ignored.

(6) Message: NOT ENOUGH DATA SUPPLIED
Action: Repeat the request for data on the same input formal parameter(s). User's input ignored.

(d) Additional Test Cases

(1) Message: ILLEGAL VALUE
Action: Repeat the request for data on the same input formal parameter(s). User's input ignored.

(2) Message: NOT ENOUGH DATA SUPPLIED
Action: Repeat the request for data on the same input formal parameter(s). User's input ignored.

(e) Addition of and Status of Mutant Types Considered

(1) Message: ILLEGAL REPLY
Action: (A) Display all legal replies.
(B) Repeat part (e).

(2) Message: ILLEGAL MUTANT TYPE
Action: (A) Display the coded names of the mutant types.
(B) Repeat part (e).

(3) Message: THESE MUTANT TYPES WERE ALREADY ON:
Action: None. Serves as a warning. The other

specified mutant types which were off are now on.

(f) Displaying and Outputting of Past Results

(1) Message: ILLEGAL REQUEST
Action: (A) Display the legal requests for part (f).
(B) Repeat part (f).

(g) General Errors

(1) Message: PROGRAM FAILS
Action: (A) Display the test case on which it fails.
(B) Display the way in which it failed.
(C) Put (A) and (B) in the output file.
(D) Abort

(2) Message: PREDICATE SUBROUTINE FAILS
Action: (A) Display the test case on which it fails.
(B) Display the way in which it fails.
(C) Put (A) and (B) in the output file.
(D) Abort

(3) Message: OUTPUT FILE EXISTS - TYPE KILL OR CONTINUE.
Action: Abort on KILL, delete output file on CONTINUE.

(4) Message: TOO MUCH DATA OR FAULTY SYNTAX
Action: Repeat the previous prompt for numeric input.

(5) Message: ILLEGAL VALUE
Action: Repeat the previous prompt for numeric input.

__The Post-Run Phase__

```
(a) Message:  ILLEGAL REQUEST
        Action:   (A)  Display  the  legal  requests  for the
        Post-Run Phase.
                  (B) Repeat the Post-Run Phase.

(b) Message:  ILLEGAL MUTANT TYPE.
        Action:  (A) Display all legal mutant types.
                 (B) Repeat the Post-Run Phase.
```

APPENDIX B

FORTRAN LANGUAGE SUBSET

This appendix describes the FORTRAN subset language whose programs can be tested using the Pilot Mutation System. Only the syntax of this subset, specified in an extended BNF (see below), is given. The syntax presented is in a "pure" form with the mundane aspects of FORTRAN syntax assumed. These include the following: 1) statements start on a new line and appear in "card" columns 7-72, 2) column 6 is the statement continuation column, 3) statement labels appear in columns 1-5, 4) names have lengths of 6 or less characters, and 5) comment statements have a C in column 1.

The PIMS FORTRAN subset has the following two semantical restrictions: (1) all variables must be declared, and (2) keywords, such as DO and END, cannot be used as variable names.

Language Subset Overview

The subset of the FORTRAN language chosen for this implementation of PIMS is such that INTEGER processing of numeric data is possible. A program must be a SUBROUTINE subprogram with an optional parameter list. Parameters and other variables **must** be declared using INTEGER or DIMENSION declarations. Arrays may be either one or two dimensional and may be specified in the INTEGER statement.

The acceptable control structures include the logical-IF, GOTO, nested-DO, CONTINUE, and RETURN. Arithmetic expressions may include any of the operators: +, -, *, / or **. Logical expressions are restricted to the IF statement and must be one of: .AND., .OR., or .NOT. as used in many FORTRAN systems. Numeric values should be kept within the range -32,767 to +32,767 due to the nature of the PRIME-400 sixteen bit architecture.

BNF Description of the Language Subset

Standard BNF is augmented with the following four abbreviations:

(1) list appendix - <y> ::= <x-list> is equivalent to

<y> ::= <x> | <x> <y>

(2) commalist appendix - <y> ::= <x-commalist> is equivalent to

<y> ::= <x> | <x> , <y>

(3) option - <y> ::= <x> [<z>] is equivalent to

<y> ::= <x> | <x> <z>

(4) choice - <y> ::= <x> {<w> | <z>} is equivalent to

<y> ::= <x> <w> | <x> <z>

Programs

```
<program> ::= SUBROUTINE <program-name> (
<formal-argument-commalist> )
  <declaration-statement-list>
  <executable-statement-list>
  END
```

Formal Arguments

```
<formal-argument> ::= <variable-name>
```

Declaration Statements

```
<declaration-statement> ::= INTEGER <declaration-commalist>

<declaration> ::= <scalar-decl> | <array-decl>

<scalar-decl> ::= <variable-name>

<array-decl> ::= <one-dim-array-decl> | <two-dim-array-decl>

<one-dim-array-decl> ::= <variable-name> ( <limit> )

<limit> ::= <positive-integer> | <variable-name>

<two-dim-array-decl> ::= <variable-name> ( <limit-pair> )

<limit-pair> ::= <limit> , <limit>
```

Executable Statements

```
<executable-statement> ::= [<label>] <statement>
```

```
<label> ::= <positive-integer>

<statement> ::= <simple-statement> | <conditional-statement>
  | <do-loop-statement>
```

Simple Statements

```
<simple-statement> ::= <goto-statement> |
<assignment-statement>
  <continue-statement> | <return-statement>

<goto-statement> ::= {GO TO | GOTO} <label>

<assignment-statement> ::= <reference> =
<arithmetic-expression>

<continue-statement> ::= CONTINUE

<return-statement> ::= RETURN
```

Conditional Statements

```
<conditional-statement> ::= IF ( <logical-expression> )
<simple-statement>
```

DO-loop Statements

```
<do-loop-statement> ::= <index-part>
  <outer-loop-body>
  <loop-end>

<index-part> ::= DO <label> <index> = <initial> , <terminal>
```

```
[, <increment>]

<outer-loop-body> ::= <outer-loop-statement-list>

<outer-loop-statement> ::= [<label>]
  {<simple-statement> |
  <conditional-statement> |
  <inner-do-loop> }

<inner-do-loop> ::= <index-part>
  <loop-body>
  [<loop-end>]

<loop-body> ::= <loop-statement-list>

<loop-statement> ::= [<label>]
  {<simple-statement> | <conditional-statement>}

<loop-end> ::= <label> <loop-end-statement>

<loop-end-statement> ::= <continue-statement> |
  <assignment-statement> |
  <conditional-statement>

<index> ::= <scalar-reference>

<initial> ::= <scalar-reference> | <positive-integer>

<terminal> ::= <scalar-reference> | <positive-integer>

<increment> ::= <scalar-reference> | <positive-integer>
```

Arithmetic Expressions

```
<arithmetic-expression> ::= [<arithmetic-expression> {+ |
-}] <ae3>

<ae3> ::= [<ae3> {* | /}] <ae2>

<ae2> ::= [<ae2> **] <ae1>

<ae1> ::= <primitive-ae> | - <ae1> | (
<arithmetic-expression> )

<primitive-ae> ::= <reference> | <integer>
```

Logical Expressions

```
<logical-expression> ::= [<logical-expression> .OR.]  <le2>

<le2> ::= [<le2> .AND.]  <le1>

<le1> ::= <primitive-le> | .NOT.  <le1> | (
<logical-expression> )

<primitive-le> ::= <arithmetic-expression> <relational-op>
  <arithmetic-expression>

<relational-op> ::= .LT.  | .LE.  | .EQ.  | .NE.  | .GT.  |
.GE.
```

Data References

```
<reference> ::= <scalar-reference> | <array-one-reference> |
  <array-two-reference>
```

```
<scalar-reference> ::= <variable-name>

<array-one-reference> ::= <variable-name> ( <simple-ae> )

<array-two-reference> ::= <variable-name> ( <simple-ae> ,
<simple-ae> )

<simple-ae> ::= [-] [<positive-integer> *]
<scalar-reference> {+ | -}
  <positive-integer> |
  [-] <scalar-reference> |
  <positive-integer>
```

Identifier Names

```
<program-name> ::= <name>

<variable-name> ::= <name>

<name> ::= <letter> [<alphameric-list>]

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L |
  M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<alphameric> ::= <letter> | <digit>

<digit> ::= <zero> | <positive-digit>

<zero> ::= 0

<positive-digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Constants

```
<constant> ::= <integer>

<integer> ::= <positive-integer> | <zero-list> | -
<positive-integer>

<positive-integer> ::= <positive-digit> [<digit-list>]
```

APPENDIX C

COMMANDS AND ABBREVIATIONS

This appendix describes the commands and their abbreviations that are used to communicate with PIMS during the interactive phases. The commands for specifying mutant types follow.

The user specifies mutant types to PIMS by using the following three character abbreviations. An abbreviation marked * means the mutant type is not currently implemented. There are no "full word" commands for specifying mutant types.

(a) Data Declaration Mutations

(i) ALD - Array Limit Default Insertion

(ii) ALP - Two Dimensional Array Limit Permutation

(b) Data Reference Mutations

(i) CRP k - Constant Replacement. The value k>=1 gives the neighborhood (i.e., c +/- k) of the replacing constants. The user may choose not to specify k, in which case a default value of k=1 is assumed (see Appendix D).

(ii) SVR - Scalar Variable Replacement

(iii) SRC - Scalar Variable for Constant Replacements

(iv) CFS - Constant for Scalar Variable Replacement

(v) CAR - Comparable Array Name Replacement

(vi) CFA - Constant for Array Reference Replacement

(vii) SFA - Scalar Variable for Array Reference Replacement

(viii) AFC - Array Reference for Constant Replacement

(ix) AFS - Array Reference for Scalar Variable Replacement

(x) AIP - Two Dimensional Array Index Permutation

*(xi) SVI k - Scalar Variable Initialization Insertion. The value >0 gives the i+k set of initializing values. The user may choose not to specify k, in which case a default value of k=0 is assumed.

(c) Operator Evaluation Mutations

(i) AOR - Arithmetic Operator Replacement

(ii) ROR - Relational Operator Replacement

(iii) LCR - Logical Connector Replacement

*(iv) APP - Arithmetic Precedence Permutation

*(v) LPP - Logical Precedence Permutation

(a) Control Mutations

(i) GLR - Goto Label Replacement

(ii) PAN - Path Analysis

(iii) CSI - Continue Statement Insertion

(iv) CSD - Continue Statement Deletion

*(v) ILD - Inner Do Loop Decoupling

*(vi) DIA - Do Loop Index Alteration

(vii) RSR - Return Statement Replacement

APPENDIX D

DESCRIPTION OF THE MUTATIONS PERFORMED

This appendix describes the types of first order mutations which the Pilot Mutation System considers and some other mutations, marked with a "*", which may be considered in future extensions of PIMS. The wording used is tied to the syntactic categories defined in the "FORTRAN Language Subset" (see Appendix B). Only those programs which are in the subset language are considered to be valid mutations.

Data Declaration Mutations

Array Limit Default Insertion is accomplished by replacing each scalar reference in an array declaration by 1.

Two Dimensional Array Limit Permutation is accomplished by exchanging each two dimensional array declaration limit pair.

Data Reference Mutations

The following sets are referenced in defining the mutation operations in this section:

A --- set of all array references appearing in the program.

C --- set of all constants appearing in executable statements of the program.

K --- the set $\{-k,-k+1,\ldots,-1,0,1,\ldots,k-1,k\}$ where $k > 0$ is supplied by the user

S --- set of all scalar variable names appearing in executable statements of the program

V1 --- set of all one dimensional array names appearing in executable statements of the program.

V2 --- set of all two dimensional array names appearing in executable statements of the program.

1. Constant Replacement

Each constant c appearing in any executable statement is replaced by members of the set $\{c-k,c-k+1,\ldots,c-1,c+1,\ldots,c+k\}$. If $k=0$ is supplied, then no constant replacements are produced by PIMS.

2. Scalar Variable Replacement

Each scalar variable s appearing in any executable statement is replaced by members of $S-\{s\}$.

3. Scalar Variable for Constant Replacement

Each constant c appearing in any executable statement is replaced by members of S

4. Constant for Scalar Variable Replacement

Each scalar variable s appearing in any executable statement is replaced by members of C.

5. Comparible Array Name Replacemment

Each instance of v1 in V1 appearing in any executable statement is replaced by members of V1-{v1}. Each instance of v2 in V2 appearing in any executable statement is replaced by members of V2-{v2}.

6. Constant for Array Reference Replacement

Each instance of ar in A appearing in an executable statement is replaced by members of C.

7. Scalar Variable for Array Reference Replacement

Each instance of ar in A appearing in an exectable statement is replaced by members of S.

8. Array Reference for Constant Replacemment

Each instance of c in C appearing in any executable statement is replaced by members of A

9. Array Reference for Scalar Variable Replacement

Each instance of s in S appearing in any executable statement is replaced by members of A

10. Two Dimensional Array Index Permutation

Each instance of references to two dimensional arrays has its indecies permuted

*11. Scalar Variable Initialization Insertion

For each s in S the initial value of s is set to members of K.

Operator Evaluation Mutations

1. Arithmetic Operator Replacement

Each instance of a binary operator bo is replaced by members of the set {+,-,*,/,**}-{bo}. Each instance of unary - is eliminated.

2. Relational Operator Replacement

Each instance of a relational operator ro is replaced by members of the set {.LT.,.LE.,.EQ.,.NE.,.GT.,.GE.}-{ro}.

3. Logical Connector Replacement

Each instance of .AND. is replaced by .OR., each instance of .OR. is replaced by .AND., and each instance of .NOT. is eliminated.

*4. Arithmetic Precedence Permutation

Each arithmetic expression containing>1 arithmetic operators is replaced by each of its distinct alternative parses.

*5. Logical Precedence Permutation

Each logical expression containing>1 logical connectors is replaced by each of its distinct alternative parses.

Control Mutations

The following sets of statement labels are referenced in defining the mutation operations in this section:

L --- set of all statement labels in the program.

TRAP --- used to represent a statement which is guaranteed to cause a program interrupt

1. Goto Label Replacement

Each instance of l in L in any goto statement is replaced by members of L-{l}.

2. Path Analysis

Each simple statement (including those which are imbedded in conditionals) and each conditional statement is replaced by TRAP. Each index part of each do loop statement has a TRAP inserted as its subsequent statement. This checks that each control path is traversed at least once and can easily be extended to see if each path is traversed any number of times.

3. Continue Statement Insertion

Each do loop which does not end on a continue statement is made to do so

4. Continue Statement Deletion

Each do loop which ends on a continue statement is made to end on the preceeding statement.

*5. Inner Do Loop Decoupling

Inner do loops which end on the same statement as their containing do loop are made to end on a separate, possibly duplicated, statement.

*6. Do Loop Index Alteration

Although this type of mutation is not currently implemented as a separate type, these mutations can be produced as a result of data mutations (see above).

7. Return Statement Replacement

Each non-return simple statement (including those which are imbedded in conditionals) and each conditional statement is replaced by a return statement. Each index part of each do loop statement has a return statement inserted as its subsequent statement.

APPENDIX E

ENTERING AND MODIFYING FILES

Programs are normally entered into the computer using the PRIMOS[19] Text Editor (ED). This editor is a line oriented text processor whose line pointer is always located at the last line processed (whether the processing is printing locating, moving pointer, etc.). The Editor operates in one of two modes, INPUT mode or EDIT mode.

When creating a new file, the Editor is invoked by typing

```
OK, ED
```

which places the Editor in the INPUT mode. When modifying an existing file, the Editor is invoked by typing

```
OK, ED filename
```

which places the Editor in the EDIT mode. The "filename" specified is the six-character name assigned to the raw program file being created or modified. At any time, the user may type a carriage return (c/r) with no other characters preceding it. This is known as a "null response." This null response will switch the Editor from the EDIT mode to the INPUT mode or from INPUT mode to EDIT mode.

INPUT Mode

The INPUT mode is used when entering text information into a file (e.g., creating a program). The word INPUT is displayed at the user's terminal to indicate that the Editor has entered the INPUT mode. The c/r key will terminate the current line of text and prepare the Editor to receive a new line. Tabulation is accomplished with the backslash (\) character. Each backslash represents the first, second, etc. tab setting; the tab stops are at columns 6, 15, and 30. The use of c/r with no text preceding it puts the Editor in EDIT mode.

EDIT Mode

The EDIT mode is used when the contents of a file are to be modified. More than 50 commands are available, although we will only describe a subset of the available commands that should suffice for most purposes. The commands are described later in this appendix.

In the EDIT mode, the Editor maintains an internal line pointer at the current line (the last line processed). The commands TOP, BOTTOM, FIND, and LOCATE, move this pointer. The WHERE command displays the current line number; POINT moves the pointer to a specified line number. The MODE NUMBER command causes the line number to be displayed whenever a line of text is displayed. All commands for location and modification begin processing with the current line. The use of c/r with no text preceding it puts the Editor in INPUT mode.

Typographical Error Correction

In either mode the user may correct errors in typing before the terminating (c/r) is typed. The last character entered is deleted, moving from right to left, one character for each backspace(b/s) typed. The entire current line may be deleted by typing the delete(del) character. The character (b/s) is obtained by holding the key marked "CTRL" or "CONTROL" and then striking the key "H." Any line followed by the delete character is null, and a (c/r) at that point will switch the editor into the alternate mode.

Saving Files

Orderly termination of an Editor session is done from the EDIT mode. The command:

```
FILE filename
```

writes the current version of the edited file to the disk under the specified filename. The file will be created if it did not previously exist or it will be overwritten if it does exist. If an existing file is being modified, the command:

FILE

writes the new version to the disk with the old filename. After the execution of the FILE command, the Editor is terminated and control returns to PRIMOS signified by: "OK," on the user terminal.

Other Useful Techniques

The following general descriptions will aid the user in adapting to the PRIMOS Editor.

Any number of lines may be moved from one location to another using the DUNLOAD command. DUNLOAD deletes these lines as it writes them into an auxiliary file. A LOAD command loads the new auxiliary file data at the desired point. Any number of lines may be copied from one location to another using the UNLOAD command. UNLOAD works the same as DUNLOAD except that UNLOAD does not delete the lines as they are being written.

Any line the begins with a legal FORTRAN statement number may be located with the FIND command.

The MODIFY command is used when a line must be altered but the relative column alignment must remain the same.

EDITOR Command Summary

The following is an alphabetical list of some of the available Editor commands. For a detailed description of all commands, the user is referred to the Editor Reference Section of THE NEW USER'S GUIDE TO EDITOR AND RUNOFF[19]. In the following descriptions, the parameter "string" is any series of ASCII characters including leading, trailing, or embedded blanks.

APPEND string................	Appends string to the end of the current line.
BOTTOM......................	Moves the pointer beyond the last line of the file.
CHANGE/st1/st2/[n] [G].......	Replaces st1 with st2 for n lines. If G is omitted, only the first occurrence of st1 on each line is changed; if G is present, all occurrences on n lines are changed.
DELETE [n]..................	Deletes n lines, including the current line. The default value of n is one.

DUNLOAD filename [n].........	Deletes n lines from the current file and writes them into filename. The default value of n is one.
FILE [filename]..............	Writes the contents of the current file into filename and QUITS to FRIMCS.
FIND string..................	Moves the pointer to the first line beginning with string.
INSERT string................	Inserts the string after the current line.
LOAD filename................	Loads text from filename into the current file following the current line.
LOCATE string................	Moves the pointer forward to the first line containing string. The string may contain leading and trailing blanks.
MODE NUMBER..................	Displays line numbers in front of displayed lines.
MODE NNUMBER.................	Turns off the line number

	display.
NEXT [{+|-}] [n]...........	Moves the pointer n lines, forward if n is positive and backward if n is negative.
POINT [n]..................	Moves the pointer to line n.
PRINT [n]..................	Displays the current line or n lines beginning with the current line.
QUIT.......................	Terminates the editing session without filing the current file.
RETYPE string..............	The current line is replaced by string.
TOP........................	Moves the pointer one line before the first line of text.
UNLOAD filename [n]........	Copies n lines into filename.
WHERE......................	Displays the current line number.

__Other Capabilities Outside The EDITOR__

From time to time the user will probably wish to view the contents of a file, delete an existing file or change the name of an existing file. These capabilities exist outside of the Editor facilities. In order to view a file at the user's terminal, the user types

```
OK, SLIST filename
```

where filename is the name of the file to be listed. Upon completion of the listing, control is returned to PRIMOS.

Files may be deleted with the PRIMOS command

```
OK, DELETE filename
```

where filename is the name of the file to be deleted. A user may not delete a file that he does not own or that has been appropriately protected.

Files may be renamed with the PRIMOs command

```
OK, CNAME oldname newname
```

where oldname is the current name of the file and newname is the desired new file name. A user may not rename a file that he does not own or that has been appropriately protected.

APPENDIX F

SAMPLE PIMS RUN

The following is a copy of the terminal dialog from an initial PIMS run. Some of the lines were changed to fit them on the page, but the information presented is unchanged.

```
OK, SEG RUN>PIMS
  PRE-RUN PHASE
 ALL INPUT MUST BE IN UPPER CASE
 ENTER THE RAW PROGRAM FILE NAME
JBSTO2
 CATEGORIZE FORMAL PARAMETER N
PROG
     1          SUBROUTINE SORTO2(N,A)
     2  C    *BUBBLE SORT - ALLOW EARLY TERMINATION
     3          INTEGER N,A(N)
     4          INTEGER I
     5          INTEGER T
     6          INTEGER SORTED
     7  C
     8          IF (N.LE.1) GOTO 300
     9      100 CONTINUE
    10          SORTED = 1
    11          DO 200 I = 2,N
    12            IF (A(I-1).LE.A(I)) GOTO 200
    13              T      = A(I-1)
    14              A(I-1)   = A(I)
    15              A(I)= T
    16              SORTED = 0
    17      200 CONTINUE
    18          IF (SORTED.NE.1) GOTO 100
    19      300 CONTINUE
    20          RETURN
    21          END
 TYPE NEXT COMMAND
INPUT
 CATEGORIZE FORMAL PARAMETER A
IO
 IS MUTANT CORRECTNESS DEPENDENT ON A PREDICATE SUBROUTINE?
 TYPE A YES OR NO    ****
NO
```

```
 HOW MANY TEST CASES ARE TO BE SPECIFIED?
2
 SPECIFY TEST CASE       1
 ENTER VALUES FOR
 N      ,
5
 ENTER      5 VALUES FOR ARRAY A
1 2 3 4 5
  TEST CASE NUMBER     1
 PARAMETERS ON INPUT
 N         =          5
 PARAMETERS ON OUTPUT
 A         (   1)=           1
 A         (   2)=           2
 A         (   3)=           3
 A         (   4)=           4
 A         (   5)=           5
   THE RAW PROGRAM TOOK     19 STEPS TO EXECUTE THIS TEST CASE
 HIT RETURN TO CONTINUE

 PLEASE VERIFY THAT DATA IS CORRECT
 TYPE A YES OR NO    ****
YES
 SPECIFY TEST CASE       2
 ENTER VALUES FOR
 N      ,
5
 ENTER      5 VALUES FOR ARRAY A
99 -99 -55 0 50
  TEST CASE NUMBER     2
 PARAMETERS ON INPUT
 N         =          5
 A         (   1)=          99
 A         (   2)=         -99
 A         (   3)=         -55
 A         (   4)=           0
 A         (   5)=          50
 PARAMETERS ON OUTPUT
 A         (   1)=         -99
 A         (   2)=         -55
 A         (   3)=           0
 A         (   4)=          50
 A         (   5)=          99
   THE RAW PROGRAM TOOK     51 STEPS TO EXECUTE THIS TEST CASE
 HIT RETURN TO CONTINUE

 PLEASE VERIFY THAT DATA IS CORRECT
 TYPE A YES OR NO    ****
YES
  WHAT NEW TYPES OF MUTANTS ARE TO BE CONSIDERED ?
ALL
```

```
  MUTATION PHASE
  POST RUN PHASE
 NUMBER OF TEST CASES =    2
 NUMBER OF LIVE MUTANTS =    31
 NUMBER OF MUTANTS =       240
 PERCENTAGE OF ELIMINATED MUTANTS =    87.0%

 MUTANT TYPES AND LIVE MUTANTS PROFILES
 TYPE MUTANTS LIVE* TYPE MUTANTS LIVE* TYPE MUTANTS LIVE*
 ALD     1  0*    CRP    16   4*    SVR    42   3*    SFC    32   6*
 CFS    30  2*    CFA    12   0*    SFA    24   0*    AFC     8   2*
 AFS    12  0*    AOP    12   0*    ROR    15   5*    GLR     4   3*
 PAN    16  1*    CSD     1   0*    RSR    15   5*

 MUTANT ELIMINATION METHOD PROFILE
 METHOD       COUNT* METHOD       COUNT* METHOD       COUNT*
 TIMED-OUT      34* REF UNDVAR    47* SUBSCR RNG    38*
 ARTH FAULT      0* RDONLY VAR     0* TRAP STMT     15*
 EQUIV           0* ZERO DIV       0* WRONG ANS     75*
 POST RUN RESULTS
HELP
   COMMANDS CAN USUALLY BE ABBRIVIATED TO
         TWO LETTERS, COMMANDS ARE AS FOLLOWS :
     HELP        - DISPLAY THIS HELP PAGE (CANNOT ABBRIV.)
     KILL        - ABORT THE CURRENT RUN (CANNOT ABBRIV.)
     PROGRAM     - TYPE THE PROGRAM BEING MUTATED
     TESTCASE N- TYPE THE TESTCASE N
     MUTANTS     - TYPE ALL THE LIVE MUTANTS
     MUTANTS (KEYWORD) (KEYWORD) ... (KEYWORD)
                 - TYPE ONLY MUTANTS OF THE SPECIFIED TYPE
     MUTANTS SELECT - SELECT THE MUTANTS MENU STYLE
     MUTANTS KEYWORDS - SEE THE KEYWORDS FOR MUTANTS TYPES
     HEADER   - DISPLAY THE PIMS RUN HEADER
     CORRECTNESS - DISPLAY THE METHOD OF DETERMINING
         CORRECTNESS
     RESULTS - DISPLAY THE RESULTS FOR MUTANTS CREATED
         IN THIS RUN
     STATUS   - DISPLAY THE STATUS OF ALL MUTANTS (INCLUDING
         PREVIOUS RUNS)
     HALT       - STOP THE CURRENT PIMS RUN
     LOOP    -  ITERATE THE CURRENT RUN
     OUTPUT TESTCASES - JUST THAT,
     OUTPUT MUTANTS - OUTPUT ALL LIVE MUTANTS
     OUTPUT MUTANTS (KEYWORD) (KEYWORD) ... (KEYWORD)
         OUTPUT ONLY MUTANTS OF THE INDICATED TYPE
     OUTPUT MUTANTS RANDOM - OUTPUT ONE RANDOM MUTANT OF
         EACH TYPE
     OUTPUT MUTANTS RANDOM (KEYWORD) (KEYWORD) (KEYWORD)
 POST RUN RESULTS
HALT
```

BIBLIOGRAPHY

[1] C.V.Ramamoorthy, S.F.Ho, and W.T.Chen, "On the Automated Generation of Program Test Data", IEEE Transactions on Software Engineering SE-2,4 (Dec 1976), pp 293-300.

[2] W.E.Howden, "Methodology for the Generation of Program Test Data", IEEE Transactions on Computers C-24,5 (May 1975), pp 554-560.

[3] W.E.Howden, "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering SE-2,3 (Sept 1976), pp 208-214.

[4] J.B.Goodenough and S.L.Gerhart, "Towards a Theory of Test Data Selection", IEEE Transactions on Software Engineering SE-1,2 (June 1975), pp 156-173.

[5] J.C.Huang, "An Approach to Program Testing", Computing Surveys 7,3 (Sept 1975), pp 113-128.

[6] E.F.Miller and R.A.Melton, "Automated Generation of Testcase Datasets", in Procedings of the First International Conference on Reliable Software, SIGPLAN Notices 10,6 (June 1975), pp 51-58.

[7] L.Clarke, "A System to Generate Test Data and Symbolically Execute Programs", IEEE Transactions on Software Engineering SE-2,3 (Sept 1976), pp 215-222.

[8] J.King, "Symbolic Execution and Program Testing", Communications of the ACM 19,7 (July 1976), pp 385-394.

[9] R.London, "The Current State of Proving Programs Correct", in Procedings of the ACM National Conference, 1972, ACM, New York, pp 39-46.

[10] S.Hantler and J.King, "An Introduction to Proving the Correctness of Programs", Computing Surveys 8,3 (Sept 1976), pp 331-353.

[11] F.A.Younos, "Human Errors in Programming", International Journal of Man Machine Studies 6 (1974), pp 361-376.

[12] B.Boehm, "Software Design and Structuring", in Practical Strategies for Developing Large Software Systems, Horowitz (Editor), Addison-Wesley, 1975, pp 103-128.

[13] R.DeMillo, R.Lipton,and F.Sayward, "Hints on Test Data

Selection", IEEE Computer, vol. 11, no. 4, April 1978, pp 34-41.

[14] Special Issue: Programming, ACM Computing Surveys 6,4 (Dec. 1974), pp 209-319

[15] R.DeMillo, R.Lipton, F.Sayward, "PROGRAM MUTATION: A Method of Determining Test Data Adequacy", State of the Art: Program Testing, SRA/INFOTECH, 1978.

[16] R. DeMillo, F. Sayward, "Program Mutation as a Tool for Managing Large Software Development," Trans. 36th Meeting for Quality Control.

[17] SOFTWARE TOOLS SUBSYSTEM USER'S GUIDE, (GIT-ICS-78/02), Georgia Institute of Technology, September, 1978.

[18] FORTRAN PROGRAMMER'S GUIDE, PDR3057, Prime Computer, Incorporated, Framingham, Massachusetts, November, 1977.

[19] THE NEW USER'S GUIDE TO EDITOR AND RUNOFF, PDR3104, Prime Computer, Incorporated, Framingham, Massachusetts, November, 1977.