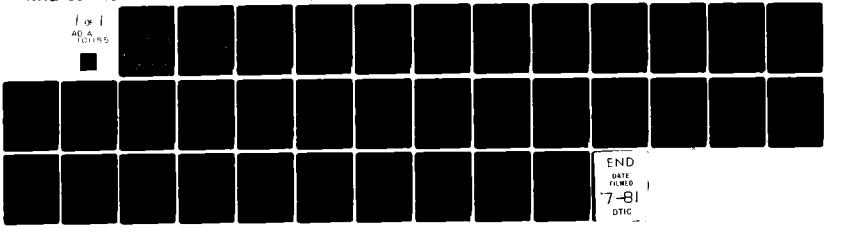
AD-A101 155 GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION A--ETC F/G 9/2
A COMPARISON OF SOME RELIABLE TEST DATA GENERATION PROCEDURES.(U)
APR 81 R A DEMILLO, D E HOCKING, M J MERRITT DAAG29-80-C-0120
UNCLASSIFIED GIT-ICS-81/08 NL



School of

Information and Computer Science

GEORGIA INSTITUTE OF TECHNOLOGY

(12) LEVEL II

GIT-ICS-81/08

A COMPARISON OF SOME RELIABLE TEST DATA GENERATION PROCEDURES †

Richard A. DeMillo*
Daniel E. Hocking**
Michael J. Merritt*

April, 1981

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

**U.S. Army Institute for Research
in Management Information and Computer Science
Rm. 105 O'Keefe Building, GIT
Atlanta, Georgia 30332

†Work supported in part by U.S. Army Research Office, Grant #DAAG29-80-C-0120 and by Office of Naval Research, Grant #N00014-79-C-0231.

A COMPARISON OF SOME RELIABLE TEST DATA GENERATION PROCEDURES [+]

Richard A. DeMillo[*]

Daniel E. Hocking[**]

Michael J. Merritt[*]

April, 1981

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

[**]U.S. Army Institute for Research
in Management Information and Computer Science
Rm. 115 O'Keefe Building, GIT
Atlanta, Georgia 30332

[+]Work supported in part by U.S. Army Research Office,
Grant #DAAG29-80-C-0120 and by Office of Naval Research,
Grant #N00014-79-C-0231.

A COMPARISON OF SOME RELIABLE TEST DATA GENERATION PROCEDURES[+]

Richard A. DeMillo[*], Daniel E. Hocking[**], Michael J. Merritt[*]

Abstract

A set of mutants of a program P, M(P), is a finite subset of the set of all programs written in the language of P, and EM(P) is the set of programs in M(P) which are (functionally) equivalent to P. For a set of test data T, DM(P,T) is the set of programs in M(P) which give results differing from P on at least one point in T. A mutation score for P,T is defined as follows:

$$ms(P,T) = \frac{|DM(P,T)|}{|M(P)|-|EM(P)|}$$

As described elsewhere, it is possible to choose the function M so that ms(P,T) = 1 only if T demonstrates the correctness of P with high probability.

This paper is a case study of four test data generation schemes. For a fixed program P, five sets of test data are generated and mutation scores are calculated using the FMS.2 mutation system. Since each set has a score less than one, the FMS.2 system is used to derive a set T such that ms(P,T)=1.

Keywords: software reliability, program testing, mutation.

---

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

[**]U.S. Army Institute for Research
in Management Information and Computer Science
Rm. 115 O'Keefe Building, GIT
Atlanta, Georgia 30332

[+]Work supported in part by U.S. Army Research Office, Grant #DAAG29-80-C-0120 and by Office of Naval Research, Grant #N00014-79-C-0231.

SECTION 1

INTRODUCTION

There are currently many suggested procedures for choosing input data for software testing [1,7,8,10,12,13]. In this paper we describe an experimental technique for the relative evaluation of different test data generation procedures, and present the results of one such study comparing five testing methodologies. Mutation analysis [1,4,5], a tool for the evaluation of individual test data sets, is used to generate a mutation score, $0 \leq ms(P,T_i) \leq 1$, where P is a program and T_i the test data generated by procedure i. Within certain constraints discussed below, data sets with high mutation scores may be judged superior to those with low scores. Mutation scores for data sets generated by the various methodologies provide an objective evaluation of those methodologies, when applied to the particular program studied. Repeating this procedure with a variety of programs would provide a tool for the overall evaluation of testing methodologies.

SECTION 2

RELIABILITY OF TEST DATA

When a program P behaves correctly on a single test input, t, it is differentiated from an infinite subset of Prog(P) (all programs in the language of P), the subset of programs that behave incorrectly on input t. Since an infinite number of programs in Prog(P) differ from P on only one input, testing alone cannot establish program correctness unless it is exhaustive. This is of course impossible for most practical situations.

Thus, testing cannot be used to establish program correctness-- but it can be used to increase confidence in a program's correctness. Two sets of test data often differ in the levels of confidence they engender--one is said to be more reliable than the other. An example would be two sets of data, T and T', such that $T \subset T'$: T' is more reliable than T.

Measuring Reliability

Let $M(P) \subset Prog(P)$ be a finite subset of the programs in the language of P and let $EM(P) \subseteq M(P)$ be the subset of M(P) of programs equivalent to P (i.e., programs that compute the same function). Finally, let $DM(P,T) \subseteq M(P) - EM(P)$ be the subset of non-equivalent programs in M(P) that behave differently than P on some input from the set T of test data. We define the mutation score for program P and test data set T to be the fraction of

non-equivalent programs differentiated from P by T:

$$ms(P,T) = \frac{|DM(P,T)|}{|M(P)| - |EM(P)|}$$

Notice that ms() is determined by the language and by the set M(P). Thus, the mutation score is a practical method for comparing the reliability of test data sets provided only the set M(P) is chosen to meet two criteria:

I) ms(P,T) is easy to compute, and

II) confidence in the correctness of P increases as ms(P,T) approaches 1.

Furthermore, if P is known to be correct, the mutation score may be used to compare the reliability of test data selection methods; one method is more reliable than another if it produces more reliable test data sets.

Mutation Theory

Mutation analysis is one method of choosing M(P) to satisfy the I and II above. In mutation analysis, each element of M(P) is generated from P by introducing some small change into P--the set M(P) is the set of mutants of P. Each change is meant to simulate a simple programmer error [5]. These changes are introduced according to rules called mutant operators, different types of errors being introduced according to different operators (a complete discussion of mutant operators appears in [1]--explicit examples of program mutants are presented later in this paper).

The errors introduced by mutant operators simulate actual program errors made by competent programmers in practice [1]. Provided P is correct, the elements of M(P) are the programs with single errors that a competent programmer is most likely to produce in place of P. The more frequently such likely errors are detected by a set of test data (the higher the mutation score), the more confidence one can acquire that the program has no such single error. Empirical evidence supports the assumption that test data sufficient to detect single errors suffices to detect erroneous programs with multiple errors as well [1,2,3]. Additional evidence that this choice of M(P) satisfies II is the observed reliability of programs tested by data sets with high mutation scores [1,4,6].

Prototype automated mutation systems, described below, have been used to compute mutation scores for a large number of programs, in three languages [1]. Theoretical studies and run-time observations suggest that mutation scores may be economically computed for even large programs [6,9], so that this choice of M(P) also satisfies I.

A mutation system generates the set M(P) by applying mutant operators to P. Using appropriate optimizing heuristics, it then interprets the program and its mutants, running them on test data provided by the tester. Any mutant which produces output differing from the original program is 'killed,' and removed from further consideration. Some mutant programs will perform identically to the original program on all inputs--these are equivalent mutants, the set EM(P). As the testing process

continues, the tester may view individual mutants or apply automated heuristics in order to determine if they are equivalent. A program passes the mutation analysis once all non-equivalent mutants have been killed by some input. Of course, the original program must have been judged by the tester to have performed correctly on all test data. If P is known to be correct, the set of mutants killed by a set of test data T is exactly the set DM(P,T) (the 'dead mutants' of P) needed to compute ms(P,T).

SECTION 3

CASE STUDY

The remainder of this paper presents a case study in which five test data generation techniques are employed to generate five sets of test data for a simple program, TRITYP, which has been studied elsewhere [1,5,9]. Mutation scores are assigned to each set of data using the interactive FMS.2 mutation system [1]. As all the scores are less than 1, the mutation system is used interactively to derive a set T such that ms(P,T) = 1.

Test Data Generation Methods

The five test data generation techniques we study involve different analyses of the program to be tested. They are: Specifications, Statement, Branch and Domain Analysis (two methods studied are variations of domain analysis).

Specifications analysis is a 'black box' approach to program testing: it involves no analysis of the actual program. Instead, an ad hoc and intuitive analysis of the program's specifications is performed. The tester uses the specifications to try to outguess the programmer and expose errors. Because of the adversary nature of this technique, it has been recommended that programmers not test their own programs, and even that programs be tested by entirely different organizations [10].

The next two methods are 'white box' testing techniques, involving explicit and often complex analysis of the program code. Of these, statement analysis is the simplest, requiring only that a test data set cause every program statement to be executed by at least one test input. Automated systems exist that backtrack from a statement, analyzing branching predicates to produce a single predicate which, when satisfied by input values, causes the appropriate statement to be executed [11].

Branch analysis places a stronger restriction on test data sets, requiring not just that every statement be executed, but that every branch be executed at least once [1,7,8]. Thus, every branching predicate must evaluate to TRUE and to FALSE for some different inputs in the test data set.

Domain analysis, the final test data generation strategy we examine, may be used as either a black box or white box technique [1,12]. In the black box approach, the program specifications are used to partition the input space into contiguous convex regions, called domains, on which the program is to compute different functions. Test data are picked from each domain, each boundary between domains, and points close to such boundaries. The white box approach performs a similar analysis, but examines the domains implicit in the program structure, rather than those which ought to exist, given the program specifications. For large numbers of domains and higher dimensions, the number of test cases required by the black box technique becomes unreasonably large. One heuristic for decreasing the number of test cases is to pick them so as to satisfy several domain

requirements at the same time; a single point may lie within a domain and approach two domain boundaries, thus replacing three separate test cases. For this study, black box domain analysis was carried out twice, once without this heuristic and once with it. We differentiate these slightly different techniques by calling them Domain Analysis and Minimized Domain Analysis, respectively.

Domain analysis was applied to a program with three input variables for this study (published examples usually analyse programs with two inputs). The analysis involves the partitioning of the first orthant of lattice three-space, plus the origin, into one, two and three-dimensional subsets. Figure 1 is a representation of this partitioning. We found this partition fairly difficult to construct--applying this technique to programs with more than three inputs would require partitioning higher-dimensional spaces, while programs with inputs of different types would require the partitioning of heterogeneous input spaces.

The Program TRITYP

The simple FORTRAN program TRITYP in Figure 2 requires three nonnegative integers as input, representing the relative lengths of the sides of a triangle. An element of the set {1,2,3,4} is output, denoting that the input triangle is equilateral, isosceles, scalene or illegal, respectively. Triangles with sides of zero length are legal, but other

degenerate triangles are not (e.g. 3 3 6).

The behavior of TRITYP on negative inputs is not consistent, so that its acceptance of negative inputs at all may be seen as a specifications error. For the purposes of this study, therefore, we will analyze the behavior of the program on nonnegative inputs only. On all such inputs within the integer range of the host machine, the behavior of TRITYP is correct. The program TRITYP has been studied elsewhere [1,5,9], and a very similar program was discussed in [10].

Mutation Scores

Five sets of test data are generated for TRITYP, one according to each of the methods discussed above. Mutation scores are then computed using the FMS.2 system; a summary of the results appears in Figure 3, listed in order of increasing mutation score.

The various mutant operators available on the FMS.2 mutation system are discussed in some detail elsewhere [1]. For this study, all of them are applied, producing 1035 mutants of the program TRITYP. Of these, 69 are equivalent mutants. Thus $|M(TRITYP)| = 1035$, $|EM(TRITYP)| = 69$; that is, there are 966 non-equivalent mutants of TRITYP. The number of these non-equivalent mutants killed by the various test data sets are used to determine the respective mutation scores.

It is apparant from the results in Figure 3 that size alone is not the determining factor in our measure of data set reliability. While the largest set, T_D, is the most reliable, the set T_{MD} rated almost as high with less than half the size (This is also evidence that the minimizing heuristic is reasonable). Similarly, T_B measured significantly better than T_{Sp}, with 30% fewer test cases. This observation contradicts the view that "the more test cases, the better," and demonstrates that a few, well chosen test cases may be more reliable as well as more economical than a larger set of less carefully chosen data.

Surviving Mutants

None of the sets of test data studied killed all the non-equivalent mutants of TRITYP. Since TRITYP is known to be correct, each of these surviving mutants is a possible erroneous program that would not have been detected by the test set it survived. These surviving mutants are thus specific examples of inadequacies in the various testing methodologies--by studying them in some detail, we may hope to discover in more general terms the strengths and weaknesses of these methodologies. The remainder of this section provides a brief discussion of the five methodologies studied above, in light of their surviving mutants.

Statement Analysis

Figure 4 provides examples of three mutants that survive the data set T_{St}, output by the FMS.2 system. The second mutant shown, in which GOTO 60 was replaced by CALL TRAP, was not detected because that line of code was not executed by any input in T_{St}. The two lines of code

```
IF((I+J).LE.K)GOTO 50
GOTO 60
```

were treated as one statement during the generation of the set T_{St}. This statement is executed by the input (3 3 8), but only one branch of the predicate, to GOTO 50, is executed by that input. It is this type of error which branch analysis attempts to detect, by requiring that every branch be executed by some input. The other two mutants shown were executed, but behaved identically to TRITYP on those inputs. Thus, it may not be enough to merely execute a statement or branch on only one input.

Specifications Analysis

Three mutants surviving both T_{St} and T_{Sp} are shown in Figure 5. Once again, the appropriate program branches are not executed by test data, and these errors would be undetected. As an example, the first mutant, replacing IF(I+K.LE.J)GOTO 50 with IF(J+K.LE.J)GOTO 50, will only be detected by input with I equal to K, and I+K$\leq$J. When a tester attempts to exercise paths by altering various input parameters, but without explicit knowledge

of the code or without tracing the logic of the code, such errors may easily remain undetected.

Branch Analysis

Every branch in the program TRITYP is executed by one of the nine inputs in T_B, and this set succeeds in killing the mutants mentioned in the previous sections, despite being a smaller set than T_{Sp}. The simple analysis of TRITYP required to produce T_B has a payoff in high reliability with a small number of test cases. Examples of mutants that do survive appear in Figure 6. In the first one,

```
IF(I+J.LE.K.OR.J+K.LE.I.OR.I+K.LE.J)GOTO 50 ⇒
IF(J+J.LE.K.OR.J+K.LE.I.OR.I+K.LE.J)GOTO 50.
```

This error is not detected because of the complexity of the branching predicate--only a few of the subexpressions are exercised by the test data. This is an example of an error in processing a particular domain, as this predicate defines the region of input space of illegal but distinct integer triples.

Domain Analyses

There were very few mutants that survived the sets T_{MD} and T_D, and in fact those surviving T_D are a subset of those surviving T_{MD}. The survivors are all shown in Figure 7. Many of these involve the ZPUSH operator, which changes its argument only when it is zero. It then evaluates to the largest permitted integer.

This operator is intended to explore the behavior of the program when variables have the value *zero*, a frequently important special case. The last few mutants in Figure 7 are of less debatable significance. These are examples of simple errors in which program constants replace variables, e.g.:

```
40    IF(J+K.LE.I)GOTO 50 ⇒
40    IF(J+3.LE.I)GOTO 50.
```

As Figure 8 shows, each of these mutants computes incorrect values for portions of two domains. Unless test data is chosen from one or more of these regions, the errors go undetected. It is an accident that some of these mutants were killed by each of T_{St}, T_{Sp}, T_B and all of them by T_D. None of the five test data generation schemes studied checks specifically for this type of *code-dependent error*.

Intuitively, domain analysis is a stronger technique than statement or branch analysis, and our study quantifies this qualitative appraisal.

Test Data Generation Using the Mutation System

The mutation operators of the FMS.2 mutation system have been specifically designed to detect *statement, path and domain* errors, among others. During interactive use, an operator may examine mutants not killed by the current test data, and generate new input to kill those particular mutants. Starting with the specifications analysis test data set T_{Sp}, this technique is employed to generate 36 test cases (data set T_{MS}), which kill all

non-equivalent mutants of the program. Thus, $ms(TRITYP, T_{MS}) = 1$. Examination of T_{MS} reveals test cases similar to those generated by domain analysis. In fact, many of these test cases are alternate choices for domain analysis, in that they explore the same domains and domain boundaries. As an example, the input (2 1 0) of mutation analysis explores the same domain boundary (see Figure 1) as (71 40 30), an input from domain analysis; the boundary region described by the equation J + K + 1 = I.

Conclusion

This paper presents a technique for objectively evaluating the reliability of test data generation methods, relative to a particular program. It is possible that for radically different programs, different results could be obtained, although our previous studies have not shown any particular sensitivity to program choice. For the single program studied, three of the generation techniques ranked in order of the complexity of program analysis that each requires (statement, branch and domain analysis). It is an interesting point that the fourth technique, specifications analysis, was less reliable than the relatively simple branch analysis--specifications analysis is so difficult to apply effectively as to be judged an art by its proponents [10]. The slight difference in scores for domain and minimized domain analysis suggest that the small loss in reliability of the latter technique may be effectively sacrificed in return for a smaller set of test data, an important consideration when test runs are expensive. The objective reliability measure presented

here can be combined with economic and efficiency considerations, to permit a data processing manager to make an effective, informed choice between testing methodologies.

SECTION 4

References

[1] A. Acree, T. Budd, R. DeMillo, R. Lipton, and F. Sayward, "Mutation Analysis", Georgia Institute of Technology Technical Report GIT-ICS-79/08, September, 1979.

[2] A. Acree, On Mutation. PhD thesis, Georgia Institute of Technology, 1980.

[3] T. Budd, Mutation Analysis of Program Test Data. Phd thesis, Yale University, in preparation.

[4] T. Budd, R. DeMillo, R. Lipton, and F. Sayward, "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs", Proc. ACM Symp. on Principles of Programming Languages, pp. 220-33. January, 1980.

[5] 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.

[6] J.M. Hanks Testing Cobol Programs by Mutation. MS thesis, Georgia Institute of Technology, 1980.

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

[8] J.C. Huang, "An Approach to Program Testing", ACM Comput-

ing Surveys, September, 1975.

[9] R.J. Lipton and F.G. Sayward, "The Status of Research on Program Mutation", Digest for the Workshop on Software Testing and Test Documentation, pp. 355-73. December, 1978.

[10] G.J. Myers, The Art of Software Testing: John Wiley and Sons (1979) New York, NY.

[11] L.J. Osterweil and L.D. Fosdick, "Experience with DAVE--A Fortran Program Analyzer", Proc. 1976 NCC, AFIPS Conference Record, pp. 909-15.

[12] C.V. Ramamoorthy, S.F. Ho, and W.T. Chen, "On the Automated Generation of Program Test Data", IEEE Transactions on Software Engineering pp. 293-300. December 1976.

[13] L.J. White and E.I. Cohen "A Domain Strategy for Computer Program Testing", Digest for the Workshop on Software Testing and Test Documentation pp. 335-54. December, 1978.

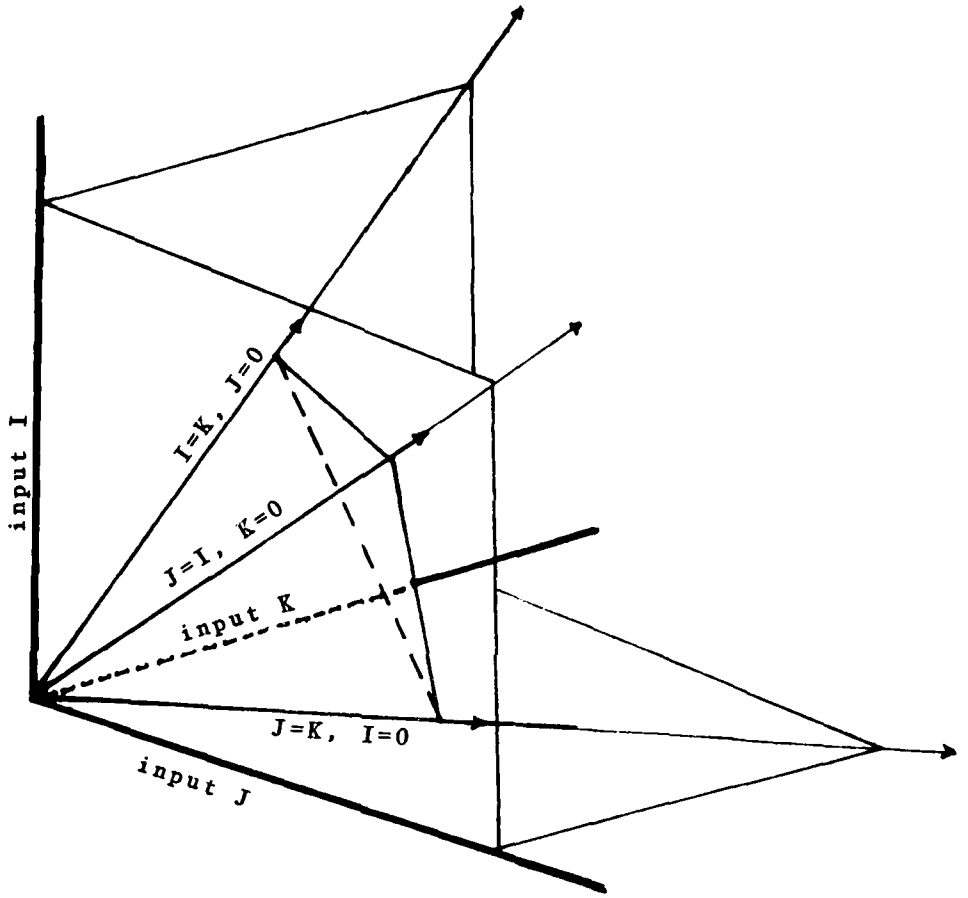


Figure 1.a

Domain analysis: side view of input space. Triangular, pyramidal region contains inputs describing legal triangles.

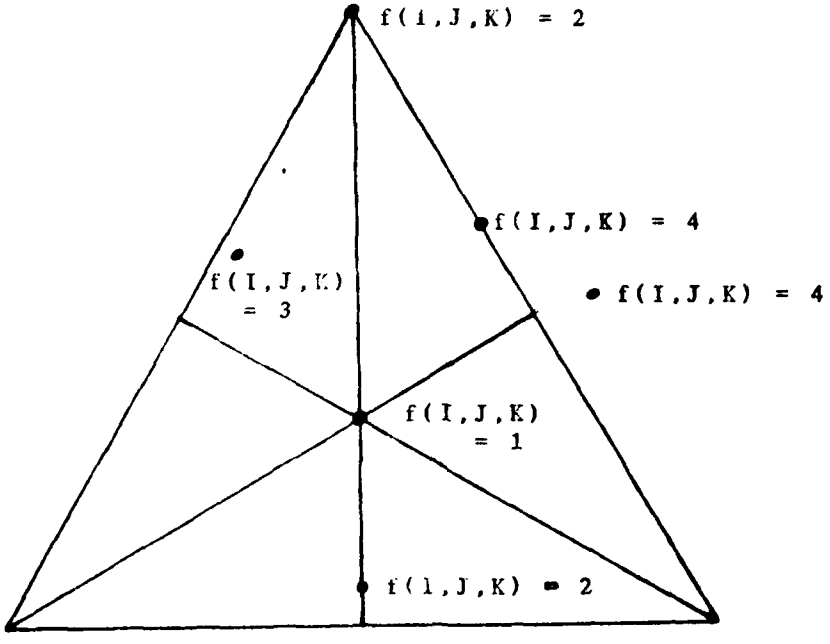


Figure 1.b

Pyramidal region of Figure 1.a, in cross-section perpendicular to the I=J=K ray.

FIGURE 2

```
      SUBROUTINE TRITYP(I,J,K,CODE)
C...I,J, AND K ARE SIDES OF THE PROPOSED TRIANGLE
C...CODE RETURNS THE TYPE OF THE TRIANGLE
C...CODE = 1 FOR EQUILATERAL
C...CODE = 2 FOR ISOSCELES
C...CODE = 3 FOR SCALENE
C...CODE = 4 FOR AN IMPOSSIBLE TRAINGLE
C
      INTEGER I,J,K,CODE
       INPUT I,J,K
       RDONLY I,J,K
       OUTPUT CODE
      INTEGER MATCH
C
C...COUNT MATCHING SIDES
      MATCH = 0
      IF(I.EQ.J)MATCH = MATCH + 100
      IF(I.EQ.K)MATCH = MATCH + 200
      IF(J.EQ.K)MATCH = MATCH + 300
C
C...SELECT POSSIBLE SCALENE TRIANGLES
      IF(MATCH.EQ.0)GOTO 10
C
C...SELECT POSSIBLE ISOSCELES TRIANGLES
      IF(MATCH.EQ.100)GOTO 20
      IF(MATCH.EQ.200)GOTO 30
      IF(MATCH.EQ.300)GOTO 40
C
C...TRIANGLE MUST BE EQUILATERAL
      CODE = 1
      RETURN
C
C...POSSIBLE SCALENE
10    IF((I+J).LE.K.OR.(J+K).LE.I.OR.(I+K).LE.J)GOTO 50
      CODE = 3
      RETURN
C
20    IF((I+J).LE.K)GOTO 50
      GOTO 60
C
30    IF((I+K).LE.J)GOTO 50
      GOTO 60
C
40    IF((J+K).LE.I)GOTO 50
      GOTO 60
C
C...NO TRIANGLE POSSIBLE
50    CODE = 4
      RETURN
C
C...ISOSCELES
60    CODE = 2
      RETURN
      END
```

FIGURE 3

SUMMARY OF RESULTS

Test Data Generation Technique	Size of Test Data Set: |T|	Number of Mutants Killed: |DM(TRITYP,T)|	Mutation Score: ms(TRITYP,T)
Statement Analysis	5	660	.68
Specifications Analysis	13	792	.82
Branch Analysis	9	821	.85
Minimized Domain Analysis	36	943	.976
Domain Analysis	75	951	.984

FIGURE 4

SELECTED MUTANTS SURVIVING DATA SET T_{St}

```
MUTANT NUMBER     40
       IF(I .EQ. J) MATCH = MATCH + 100
         BECOMES
       IF(I .EQ. J) CODE = MATCH + 100

MUTANT NUMBER    930
       GOTO 60
         BECOMES
       CALL TRAP

MUTANT NUMBER    425
 20    IF(I + J .LE. K) GOTO 50
         BECOMES
 20    IF(I + 1 .LE. K) GOTO 50
```

FIGURE 5

SELECTED MUTANTS SURVIVING DATA SETS T_{St} AND T_{Sp}

```
MUTANT NUMBER   150
 30     IF(I + K .LE. J) GOTO 50
          BECOMES
 30     IF(J + K .LE. J) GOTO 50

MUTANT NUMBER   899
 30     IF(I + K .LE. J) GOTO 50
          BECOMES
 30     IF(I + K .LE. -ABS J) GOTO 50

MUTANT NUMBER  1030
 40     IF(J + K .LE. I) GOTO 50
          BECOMES
 40     IF(J + K .LE. I) GOTO 60
```

FIGURE 6

SELECTED MUTANTS THAT SURVIVE T_B

```
MUTANT NUMBER    98
 10     IF(I + J .LE. K .OR. J + K .LE. I .OR. I + K .LE. J) GOTO 50
          BECOMES
 10     IF(J + J .LE. K .OR. J + K .LE. I .OR. I + K .LE. J) GOTO 50

MUTANT NUMBER   375
 10     IF(I + J .LE. K .OR. J + K .LE. I .OR. I + K .LE. J) GOTO 50
          BECOMES
 10     IF(I + J .LE. K .OR. 2 + K .LE. I .OR. I + K .LE. J) GOTO 50

MUTANT NUMBER   798
        IF(I .EQ. J) MATCH = MATCH + 100
          BECOMES
        IF(ZPUSH I .EQ. J) MATCH = MATCH + 100
```

FIGURE 7

MUTANTS SURVIVING BOTH T_{MD} AND T_D

```
MUTANT NUMBER   843
 10     IF(I + J .LE. K .OR. J + K .LE. I .OR. I + K .LE. J) GOTO 50
          BECOMES
 10     IF(ZPUSH I + J .LE. K .OR. J + K .LE. I .OR. I + K .LE. J)
    * GOTO 50

MUTANT NUMBER   846
 10     IF(I + J .LE. K .OR. J + K .LE. I .OR. I + K .LE. J) GOTO 50
          BECOMES
 10     IF(I + ZPUSH J .LE. K .OR. J + K .LE. I .OR. I + K .LE. J)
    * GOTO 50

MUTANT NUMBER   852
 10     IF(I + J .LE. K .OR. J + K .LE. I .OR. I + K .LE. J) GOTO 50
          BECOMES
 10     IF(I + J .LE. ZPUSH K .OR. J + K .LE. I .OR. I + K .LE. J)
    * GOTO 50

MUTANT NUMBER   855
 10     IF(I + J .LE. K .OR. J + K .LE. I .OR. I + K .LE. J) GOTO 50
          BECOMES
 10     IF(I + J .LE. K .OR. ZPUSH J + K .LE. I .OR. I + K .LE. J)
    * GOTO 50
```

FIGURE 7, CONTINUED

```
MUTANT NUMBER   858
 10     IF(I + J .LE. K .OR. J + K .LE. I .OR. I + K .LE. J) GOTO 50
          BECOMES
 10     IF(I + J .LE. K .OR. J + ZPUSH K .LE. I .OR. I + K .LE. J)
    * GOTO 50

MUTANT NUMBER   864
 10     IF(I + J .LE. K .OR. J + K .LE. I .OR. I + K .LE. J) GOTO 50
          BECOMES
 10     IF(I + J .LE. K .OR. J + K .LE. ZPUSH I .OR. I + K .LE. J)
    * GOTO 50

MUTANT NUMBER   867
 10     IF(I + J .LE. K .OR. J + K .LE. I .OR. I + K .LE. J) GOTO 50
          BECOMES
 10     IF(I + J .LE. K .OR. J + K .LE. I .OR. ZPUSH I + K .LE. J)
    * GOTO 50

MUTANT NUMBER   870
 10     IF(I + J .LE. K .OR. J + K .LE. I .OR. I + K .LE. J) GOTO 50
          BECOMES
 10     IF(I + J .LE. K .OR. J + K .LE. I .OR. I + ZPUSH K .LE. J)
    * GOTO 50
```

FIGURE 7, CONTINUED

```
MUTANT NUMBER   876
 10     IF(I + J .LE. K .OR. J + K .LE. I .OR. I + K .LE. J) GOTO 50
          BECOMES
 10     IF(I + J .LE. K .OR. J + K .LE. I .OR. I + K .LE. ZPUSH J)
    * GOTO 50

MUTANT NUMBER   879
 20     IF(I + J .LE. K) GOTO 50
          BECOMES
 20     IF(ZPUSH I + J .LE. K) GOTO 50

MUTANT NUMBER   882
 20     IF(I + J .LE. K) GOTO 50
          BECOMES
 20     IF(I + ZPUSH J .LE. K) GOTO 50

MUTANT NUMBER   885
 20     IF(I + J .LE. K) GOTO 50
          BECOMES
 20     IF(ZPUSH (I + J) .LE. K) GOTO 50

MUTANT NUMBER   903
 40     IF(J + K .LE. I) GOTO 50
          BECOMES
 40     IF(ZPUSH J + K .LE. I) GOTO 50
```

FIGURE 7, CONTINUED

MUTANT NUMBER 906

```
 40     IF(J + K .LE. I) GOTO 50
```

BECOMES

```
 40     IF(J + ZPUSH K .LE. I) GOTO 50
```

MUTANT NUMBER 909

```
 40     IF(J + K .LE. I) GOTO 50
```

BECOMES

```
 40     IF(ZPUSH (J + K) .LE. I) GOTO 50
```

MUTANTS SURVIVING T_{MD}, KILLED BY T_D

MUTANT NUMBER 419

```
 20     IF(I + J .LE. K) GOTO 50
```

BECOMES

```
 20     IF(3 + J .LE. K) GOTO 50
```

MUTANT NUMBER 421

```
 20     IF(I + J .LE. K) GOTO 50
```

BECOMES

```
 20     IF(2 + J .LE. K) GOTO 50
```

MUTANT NUMBER 426

```
 20     IF(I + J .LE. K) GOTO 50
```

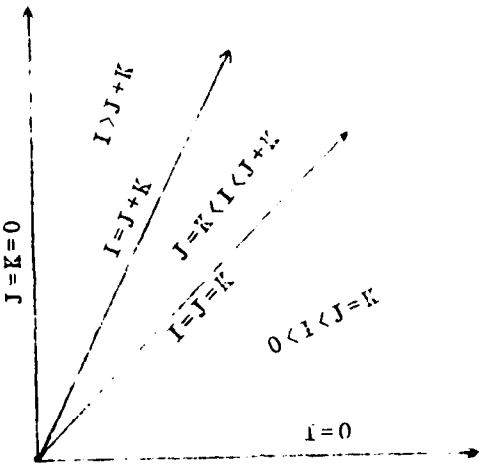
BECOMES

```
 20     IF(I + 3 .LE. K) GOTO 50
```

FIGURE 7, CONTINUED

```
MUTANT NUMBER   428
 20     IF(I + J .LE. K) GOTO 50
          BECOMES
 20     IF(I + 2 .LE. K) GOTO 50

MUTANT NUMBER   465
 40     IF(J + K .LE. I) GOTO 50
          BECOMES
 40     IF(3 + K .LE. I) GOTO 50

MUTANT NUMBER   467
 40     IF(J + K .LE. I) GOTO 50
          BECOMES
 40     IF(2 + K .LE. I) GOTO 50

MUTANT NU' 'R   472
 40     IF(J + K .LE. I) GOTO 50
          BECOMES
 40     IF(J + 3 .LE. I) GOTO 50

MUTANT NUMBER   474
 40     IF(J + K .LE. I) GOTO 50
          BECOMES
 40     IF(J + 2 .LE. I) GOTO 50
```

Figure 8.a

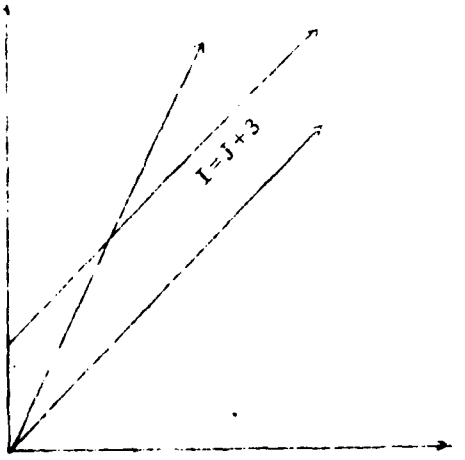


Cross-section of the input space (through the J=K plane), showing portions of four domains. Except for the I=J=K domain, these inputs execute the statement:

```
40    IF((J+K).LE.1)GOTO  50
      GOTO  60
```

Inputs in the shaded region satisfy the condition and follow the first branch.

Figure 8.b



The cross-section of Figure 8.a, with the I=J+3 line added. The points above and on this line satisfy the condition in the mutant:

```
40    IF((J+3).LE.1)GOTO  50
      GOTO  60
```

and follow the first branch. Inputs from the shaded regions are processed incorrectly by this mutant.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: GIT-ICS-81/08	2. GOVT ACCESSION NO.: AD-A101 155
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): A Comparison of Some Reliable Test Data Generation Procedures	5. TYPE OF REPORT & PERIOD COVERED: Interim Technical Report
6. PERFORMING ORG. REPORT NUMBER: GIT-ICS-81/08	
7. AUTHOR(s): Richard A. DeMillo, Daniel E. Hocking, Michael J. Merritt	8. CONTRACT OR GRANT NUMBER(s): ARO-DAAG29-80-C-0120, ONR-N00014-79-C-0231
9. PERFORMING ORGANIZATION NAME AND ADDRESS: School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: Army Research Office, PO Box 12211, Research Triangle Park, NC; Office of Naval Research, 800 N. Quincy Street, Arlington, Virginia	12. REPORT DATE: April 1981
	13. NUMBER OF PAGES: 30 + ii
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):	15. SECURITY CLASS. (of this report): unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

unlimited

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

software reliability, program testing, mutation analysis

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A set of mutants of a program P, M(P), is a finite subset of the set of all programs written in the language of P, and EM(P) is the set of programs in M(P) which are (functionally) equivalent to P. For a set of test data T, DM(P,T) is the set of programs in M(P) which give results differing from P on at least one point in T. A mutation score for P,T is defined as follows:

$$ms(P,T) = \frac{|DM(P,T)|}{|M(P)| - |EM(P)|}$$

(continued over)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

410 017

20. As described elsewhere, it is possible to choose the function M so that ms(P,T) = 1 only if T demonstrates the correctness of P with high probability.

This paper is a case study of four test data generation schemes. For a fixed program P, five sets of test data are generated and mutation scores are calculated using the FMS.2 mutation system. Since each set has a score less than one, the FMS,2 system is used to derive a set T such that ms(P,T) = 1.


F
7