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 conducted 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.

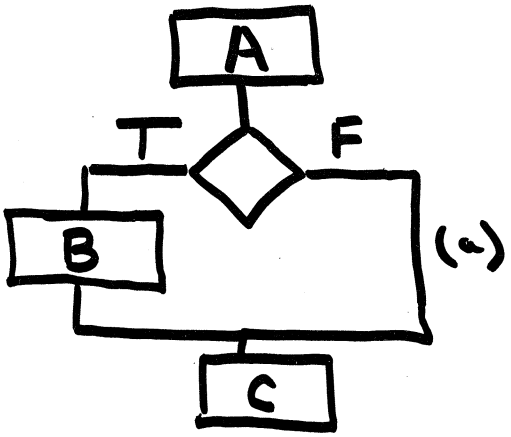


figure 2

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).

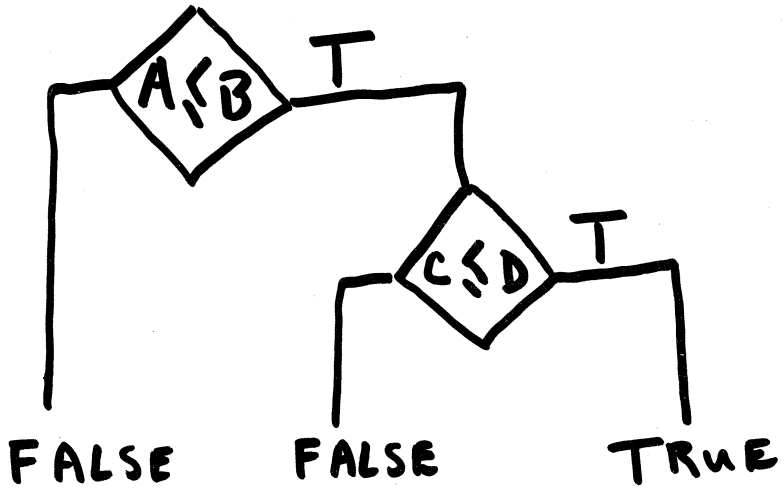


figure 3

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

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

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

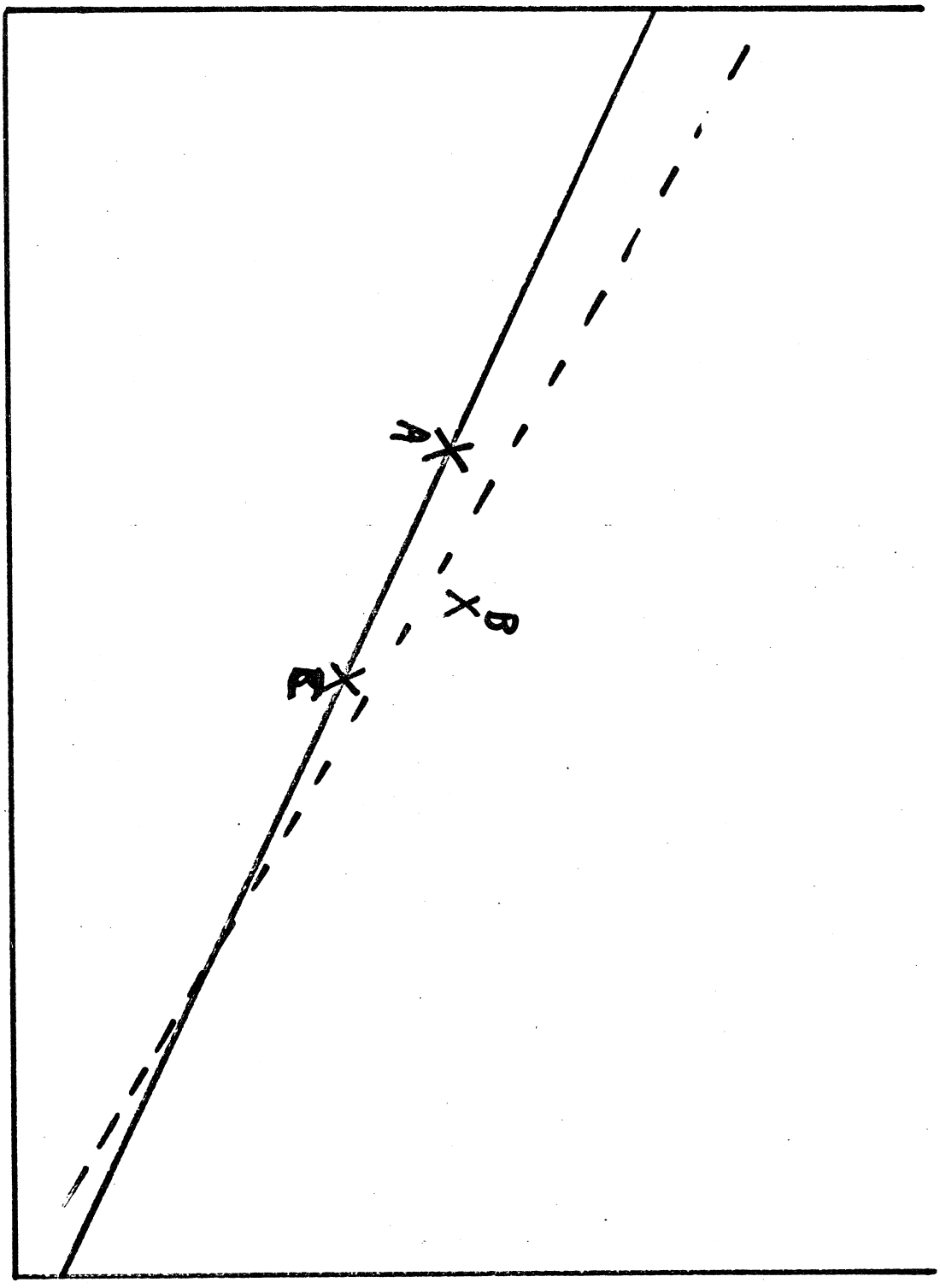


figure 5

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≦-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.

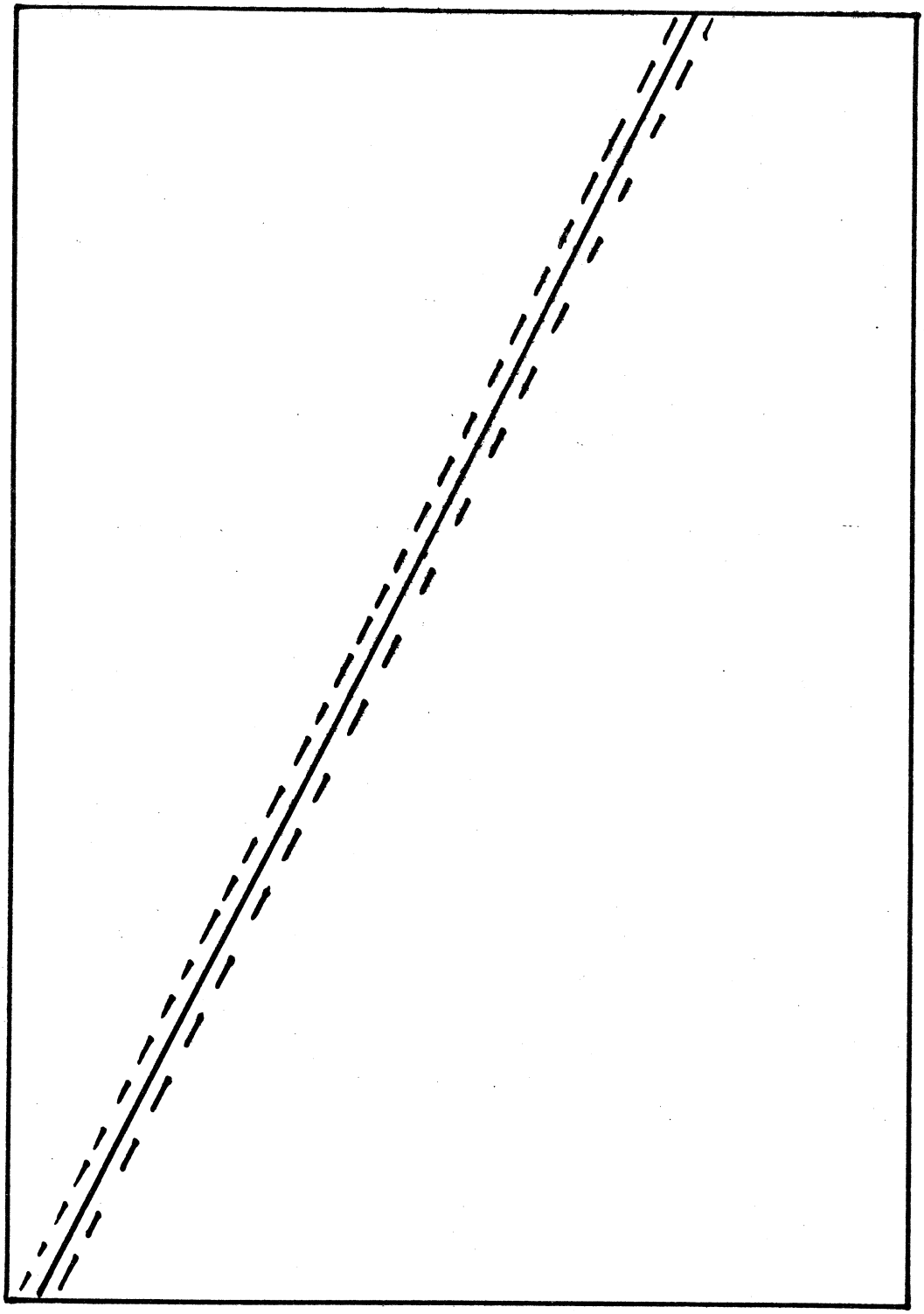


figure 9

```
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≤J+1, K≥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 ≦ J+1. Some of the choices are redundant, for example ++I ≦ J+1 and I ≦ --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 ≧ I+1 should be K ≧ 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 ≧ 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

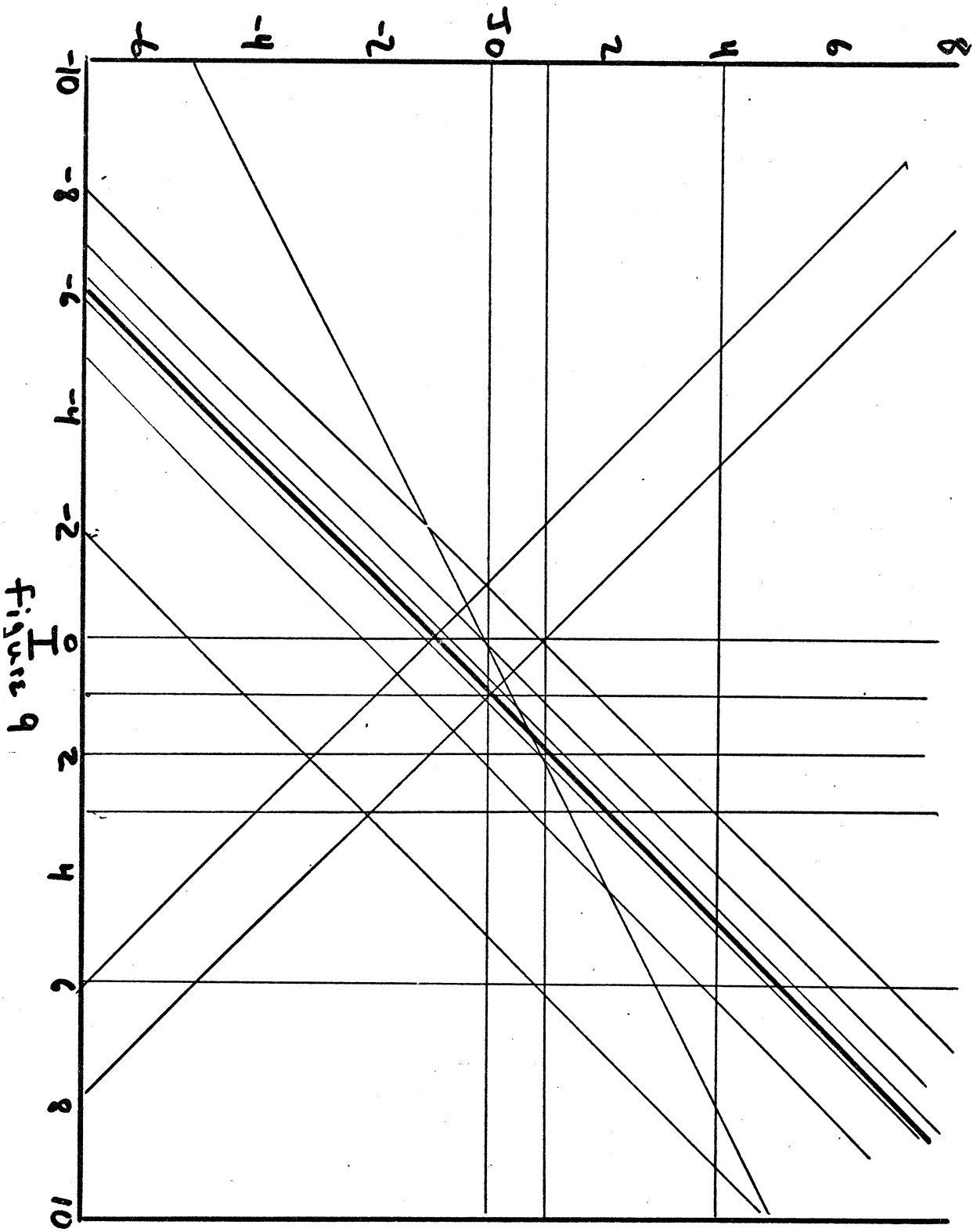
```
A = B + C
```

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


-10
-8
-6
-4
-2
0
2
4
6
8
10
height
h

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.

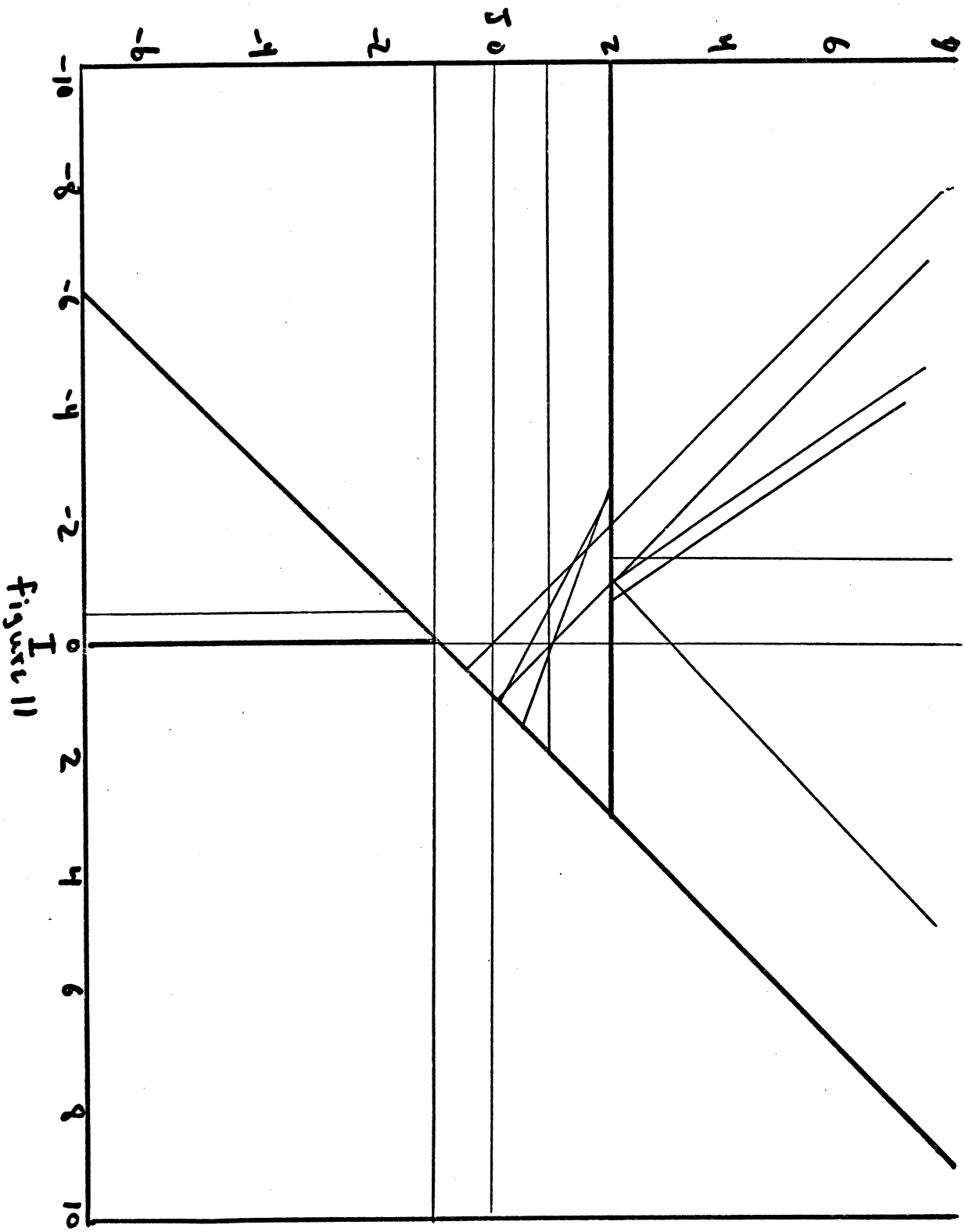


figure 11

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
 incharacter(cw);
 IF 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

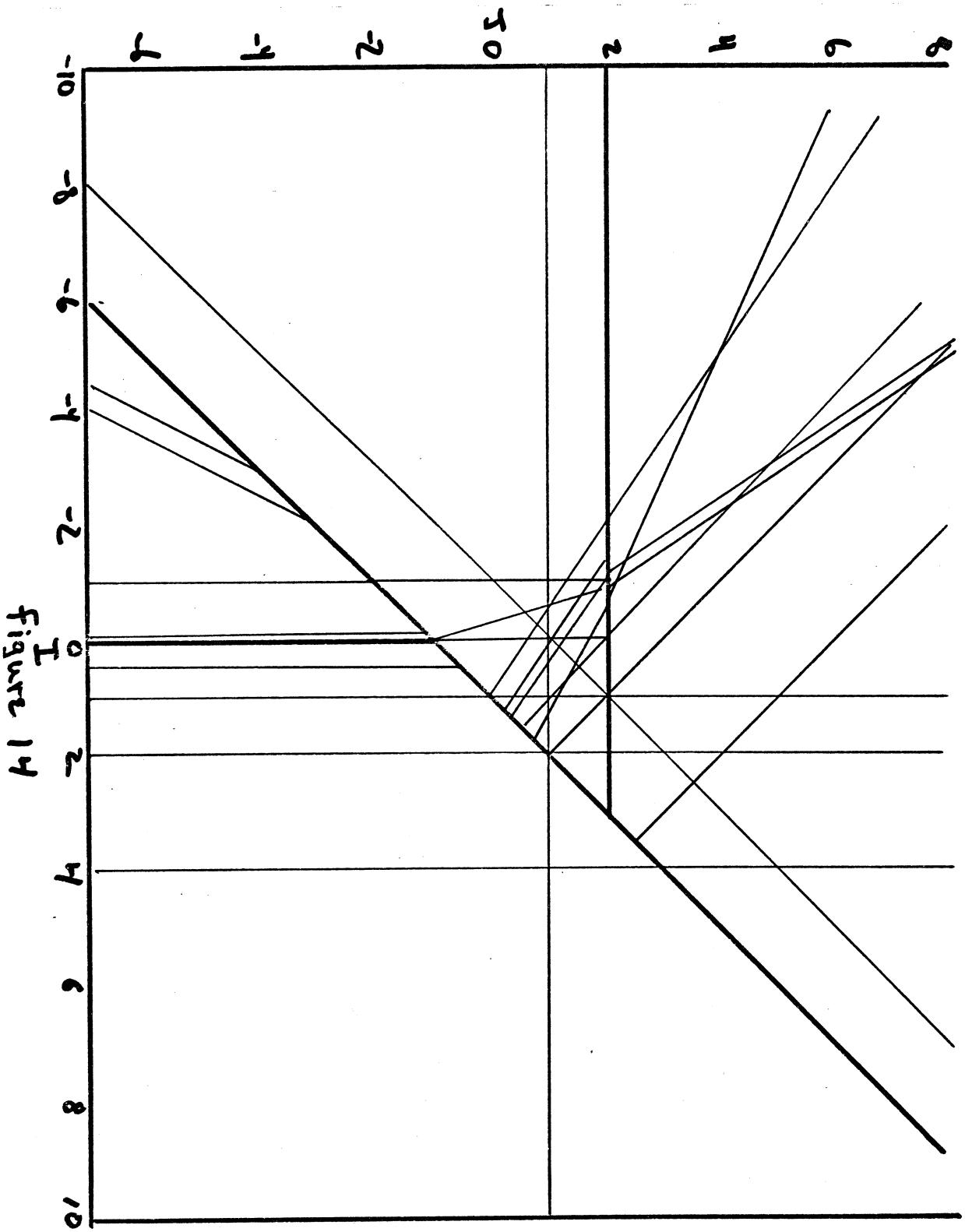


figure 14

```
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.