AN EXPERIMENTAL EVALUATION OF ERROR SEEDING AS A PROGRAM VALIDATION TECHNIQUE

By

Paul E. Ammann and John C. Knight

Computer Science Report TR-85-08

June 6, 1985

AN EXPERIMENTAL EVALUATION OF ERROR SEEDING AS A PROGRAM VALIDATION TECHNIQUE

Paul E. Ammann and John C. Knight

Affiliation Of Authors

Department of Computer Science

University of Virginia

Charlottesville

Virginia, 22903

Financial Acknowledgement

This work was supported by the National Aeronautics and Space Administration under grant number NAG-1-242.

Index Terms

Software validation, software testing, error seeding, software reliability.

Address For Correspondence

John C. Knight

Department of Computer Science

University of Virginia

Charlottesville

Virginia, 22901

Abstract

An experiment is described in which simple syntactic alterations were introduced into program text. The experiment was carried out in order to evaluate the testing strategy known as *error seeding*. The experiment's goal was to determine if randomly placed syntactic manipulations can produce failure characteristics similar to those of the indigenous errors found within unseeded programs. As a result of a separate experiment, several programs were available, each of which was written to the same specifications. Thus the programs were intended to be functionally equivalent. The use of functionally equivalent programs allowed the influence of individual programmer styles to be removed as a variable from the error seeding experiment. Each of six different syntactic manipulations were introduced into each program and the mean times to failure for the seeded errors were observed in repeated trials over a fixed range of inputs. The seeded errors were found to have a broad spectrum of mean times to failure independent of the syntactic alteration used.

The experiment demonstrates that the use of simple syntactic techniques yields seeded errors that are *arbitrarily* difficulty to locate. In addition, several unexpected results indicate that some of the issues involved in error seeding have not been addressed previously. Specifically, certain of the seeded errors were *benign*; they had no effect on the program's functionality. Other seeded errors actually *corrected* indigenous errors. In addition, events inconsistent with assumptions underlying the test method as originally described were observed. Such results have clear implications for error seeding as a testing strategy.

I INTRODUCTION

The availability of reliable techniques for generating seeded errors would be a major step towards making *error seeding* [10] an attractive testing strategy. The goal of the work described here is to examine the feasibility of generating realistic seeded errors *automatically* using relatively *unsophisticated* algorithms. This paper proposes several algorithms to seed errors in programs written in Pascal using only simple methods based on the syntax of the program being seeded, and reports the results of an experimental evaluation of these seeding techniques.

Testing is the traditional way of attempting to show that a computer program does what it is required to do, i.e. validating the program. In most cases, testing cannot provide a guarantee of correctness because that would require execution of the program for all possible inputs [7], and, for realistic programs, the input space is usually too large. In practice, only an *extremely small* subset of the input space can be selected as test cases.

Satisfactory execution of these test cases may give the tester "confidence" in the program but this is usually *not* a scientific assessment of the quality of the program because of the informal way in which test cases are selected. For critical applications such as defense systems, robotics, and avionics, formal methods of validation such as proofs of correctness would be preferred. However, at present such methods are infeasible for most programs of interest and so software engineers continue to turn to testing for validation.

Theories of program testing and various formal test methods have been proposed [8]. If they were applied to a particular program, these formal methods would allow useful scientific statements to be made about the quality of the subject program. In practice these methods either make various restrictions on the program text with which they can deal, or they require an infeasible number of test cases. For example, domain testing [14] is presently only defined for programs that do not

involve arrays and have linear predicates as their conditions.

The testing that is usually performed is a combination of functional testing in which required functions are tested, and random testing in which random samples from the input space are used. For each of these methods, the number of test cases that should be used for any particular program is difficult to determine. Clearly at any point in testing, more testing can be expected to reveal more faults in the program. However, as testing proceeds, fewer and fewer faults are revealed per unit time and it is always tempting to assume that the last fault revealed is indeed the *last* fault. The most common criteria for terminating testing is either that the available resources have been used up or that some deadline has been reached. If the exact number of faults introduced unintentionally by the programmer was known, the termination criterion for testing would be obvious. The faults introduced unintentionally are referred to as *indigenous*.

The error seeding testing method [10] was proposed as a technique for determining when a program has been adequately tested using functional or random testing. The method resulted from a desire to apply statistical methods to the problem of predicting the number of errors in a program. The hope was that the number of errors discovered during testing could be used to estimate the number of remaining undetected errors. Analysis of this method is presented in [10, 11, 12].

The method involves deliberately introducing or *seeding* errors into a program and then testing the program. Briefly, the approach is as follows. A program to be tested is assumed to contain an unknown number of indigenous errors, N, and n additional errors are seeded into the program. Suppose that during testing a total of r errors are detected, and of these k were seeded. Assuming all errors are equally difficult to detect, the maximum likelihood estimate of N is given by [12]:

$$N \approx \frac{n\,(r-k)}{k}$$

Error seeding has the desirable properties that it is apparently simple to employ and it provides a stopping condition for testing. Unfortunately it has the major drawback that, in order to work effectively, the seeded errors must have the same characteristics as indigenous errors. Intuitively this seems very hard to do. Any attempt to deliberately introduce errors by *conscious* effort is unlikely to produce errors that are similar to those introduced *unconsciously* during the program development process. If the processes involved in the generation of indigenous errors were known, this information would be used to prevent errors rather than to guide error seeding.

An alternative approach is to use an automated method to seed errors. If an algorithm could be designed that could seed realistic errors reliably, it would be a useful adjunct to the error seeding method. Again, this seems very hard to do because it implies that the seeding algorithm would somehow have detailed semantic information of the program it is to seed, and would be able to mimic human-generated indigenous errors using that semantic information. It is not clear how this could be done.

It seems unlikely that simple methods could generate seeded errors that are realistic. Making minor changes to a program such as changing operators in expressions or rearranging statements are unlike typical indigenous errors and one would suspect that such errors would survive very few test cases, if any. Budd [2] goes so far as to state that "it would seem that generating artificial errors with the same features and likelihood of detection as natural ones would be an intractable problem."

There is no evidence to justify this view, however. The work described here examines the possibility of generating realistic seeded errors automatically using relatively unsophisticated algorithms. The seeding algorithms proposed in this paper are based solely on the syntax of the program being seeded.

Crucial to the validity of the error seeding approach are three assumptions:

(1) indigenous errors are all approximately equally difficult to locate,

(2) seeded errors are approximately as difficult to locate as indigenous errors,

(3) errors, whether indigenous or seeded, do not interfere with one another.

A priori, there is no reason to believe that any of these assumptions hold. In view of the assumed difficulty with the second assumption listed above, a modification of the error seeding scheme known as capture/recapture [12] has been proposed. The capture/recapture strategy uses no artificial errors and works in a two step process. Tests are run on a program for an arbitrary amount of time. As errors are located, they are removed from the program and their existence is recorded on a list. Then these n errors are all reinstalled and the program is *independently* subjected to a second testing process. This time r errors are located. The two lists of detected errors are compared. Based upon k, the number of common errors in the two lists, a maximum likelihood estimator of N, the total number of errors, is given by [12]:

$$N \approx \frac{n\,r}{k}$$

Capture/recapture replaces the assumption that seeded errors are as difficult to locate as indigenous ones by the assumption that there are sufficient errors in the program to make the technique statistically meaningful; requiring n to exceed 20 is a commonly used condition [12]. Unfortunately, programs thought to contain at least 20 indigenous errors are not suitable candidates for high reliability applications.

The first and third assumptions listed for errors seeding are also required for capture/recapture. Indeed if errors are not equally difficult to locate then both lists of errors will contain large numbers of the simple, easy to find errors and few or none of the subtle, hard to find errors. Under such circumstances capture/recapture will make excessively optimistic predictions of the number of remaining errors.

There is also no allowance in either method for the possibility that a certain error may partially or completely hide the presence of another.

Another approach which avoids all assumptions invoked above is adopted in the testing strategy known as *mutation testing* [2, 3]. Mutation testing is based on the idea that programs are often almost correct, in the sense that a small modification can turn an incorrect program into a correct one. Therefore minor modifications to a program's text are introduced in a thorough and systematic manner, and tests are run to discriminate the modified programs or mutants from the original program. A set of tests is comprehensive if it can distinguish all of the generated mutants from the original program.

The techniques used to create mutants are very similar to the seeding techniques evaluated in this experiment. Despite the similarity between the types of faults introduced, the goals of mutation testing are completely distinct from those of error seeding. Mutation testing is concerned solely with the construction of a set of test cases which thoroughly exercises every statement in the tested program. Error seeding attempts to make statements about the number of errors remaining in a program based upon the discovery of seeded errors.

In the experiment described in this paper the effort is directed at producing errors durable enough to be used for seeding. In evaluating a seeding algorithm, it is not sufficient merely to apply the algorithm to a single program and then test that program until the seeded error is found. If an error were difficult to locate, this might be a coincidence or might result from the style of the programmer responsible for the subject program rather than any inherent quality of the seeding algorithm. It is important to remove the biases of programmer style and to have enough samples of use of the algorithm to eliminate coincidence.

As part of a separate experiment [9], twenty-seven Pascal programs have been written independently by *different* programmers from *identical* specifications. Thus all

twenty-seven are intended to perform the same function, the processing of radar data in a simple antimissile system. The programs have all been subjected to one million random tests as a part of the other experiment. The one million tests used four different computer architectures and five distinct Pascal compilers. A great deal is known about the indigenous errors present in the programs as well as how different environments affect test results [1].

For the evaluation of the error seeding algorithms proposed in this paper, seventeen of the twenty-seven programs were selected at random, errors were seeded into all seventeen, and the resulting programs were tested. All of the programs were not used because of limited resources; the experiment required large amounts of computer time. The results of the evaluation are the cumulative results of testing seventeen programs that are supposedly functionally identical, and so the influences of a particular programmer's style do not dominate the results. In view of the problems associated with multiple computing environments, all of the testing in this experiment was done on PRIME computers using the HullV Pascal Compiler [5].

The remainder of this paper is organized as follows. In section II we present the seeding algorithms used in the experiment, and in section III we describe the organization of the experiment itself. Section IV contains an overview of the results of the tests performed on programs containing seeded errors. In section V we present some observations on the results, and in section VI we compare the behavior of the seeded errors with the behavior of the known indigenous errors introduced by the programmers. Our conclusions appear in section VII.

II THE GENERATION OF SEEDED ERRORS

The approach taken here to seeding errors is to make changes that are purely syntactic. Six different algorithms of three basic types are used; two introduce range errors into **for** statements, three introduce errors into the conditions of **if** statements,

Grammar Rule	Syntactic Alteration
for_stmt ::=	1. *expr1* -> *expr1* + 1
for *var* := *expr1* **to** *expr2* **do** *stmt*	2. *expr2* -> *expr2* - 1
if_stmt ::=	3. *condition* -> true
if *condition* **then** *stmt* **else** *stmt*	4. *condition* -> false
	5. *condition* -> not *condition*
assignment_stmt ::=	6. *statement deleted*
var := *expr*	

Table 1. Seeding Algorithms

and one modifies **assignment** statements. These particular modifications were chosen because, in the authors' opinion, they do have some similarity to indigenous errors. Many similar algorithms could be easily designed; no claim is made for completeness. Cleverness on the part of any particular seeded error is not required for this experiment to succeed. Table 1 illustrates the six chosen syntactic alterations.

The two algorithms that seed errors into **for** statements attempt to simulate erroneous treatment of end cases. The range of the control variable in both cases is reduced. Errors in which the range is extended were not considered in order to avoid the mass of execution-time errors that would have resulted. Obviously these modifications introduce syntax errors if the **for** statement in question ranges over an enumerated type instead of the integers. A general solution is to use the **succ** and **pred** functions to alter the **for** statement's range.

Three algorithms introduce errors into **if** statements. The first modification simulates the case where a special condition is suddenly forced to always (or never) occur; the second modification simulates the reverse. The third modification

simulates the case where the programmer reverses the actions he wants performed. The analogous modifications to the **if** statement with no **else** part were not considered because of resource limitations.

The single error that is introduced into the **assignment** statement is to delete the statement from the program text. This modification attempts to simulate failures to properly initialize variables, increment counters, etc.

The program used to seed these errors is based on an LALR(1) parser for Pascal built with the Mystro system [4]. The parser uses a simple Pascal grammar and does not maintain a symbol table. As the lexical analyzer processes the input source text, it places the input into a character array. When a token is recognized by the lexical analyzer, a record is returned to the parser describing where the token resides in the character array. As the parser uses any production, for example A ::= XYZ, the semantic action that is invoked locates the first character of X and the last of Z and sets these two values as the location of A. This information is kept on the semantic stack. This scheme allows easy manipulation of the text associated with any symbol in a given production.

The seeding program seeds a particular error by parsing the subject program to determine how many of the productions in the seeding algorithm of interest are generated by the parse. One of these productions is then selected at random (without replacement) and when that particular occurrence of the production is applied during a *second* parse, the text in the character array is modified by the semantic action associated with the production. Finally the erroneous program text is written to an output file.

Only one error is seeded into a given program at a time. Such a restriction is necessary to determine which error is responsible for a particular failure during the testing phase. The strategy also minimizes interference between errors.

III EXPERIMENTAL EVALUATION

The 17 program versions used in this evaluation each implement a simple (but realistic) antimissile system whose specifications came originally from an aerospace company. The program is required to read some data that represents radar reflections and, using a collection of conditions, has to decide whether the reflections come from an object that is a threat. If the program decides that the object is a threat, it must generate a signal to launch an interceptor.

Version	Source Lines	Indigenous Errors	Program Reliability	for statements	if statements	assignment statements
1	855	1	0.999998	29	16	186
2	327	0	1.000000	9	21	108
3	764	4	0.997703	8	35	295
4	815	0	1.000000	20	22	126
5	950	0	1.000000	10	9	155
6	643	3	0.998851	7	23	216
7	654	1	0.999929	10	64	257
8	598	2	0.999677	4	39	213
9	1004	2	0.999947	7	17	161
10	837	0	1.000000	21	21	126
11	604	2	0.999573	5	22	218
12	502	1	0.999996	8	16	157
13	604	2	0.998632	10	43	159
14	520	0	1.000000	7	16	147
15	967	2	0.999938	25	22	199
16	488	2	0.999731	25	17	120
17	395	1	0.999885	4	28	141

Table 2. Program Characteristics

Table 2 illustrates some of the characteristics of the 17 chosen programs. The programs varied in length from 395 to 1004 source lines, with a mean of 678. The variations in length are due to differing programming styles. Some programmers commented code heavily; others used no comments at all. Some programs are broken into more functions and procedures than others.

Over the course of the 1,000,000 test case experiment, various errors manifested themselves; the number of indigenous errors observed in each program is shown in Table 2 along with the associated reliability. Reliability is defined as 1 minus the ratio of observed failures to 1,000,000. The reliabilities reflect the combined effect of *all* of the indigenous errors present in a given program. Errors which would have resulted in much lower program reliabilities were screened out as part of an acceptance test to which all programs were subjected. The 1,000,000 test case experiment did not accept a program unless it had passed a filter of 200 randomly generated tests without failure. This requirement tended to produce programs with a mean time to failure much larger than 200 cases. The indigenous errors are discussed in more detail in section VI.

The last three columns in table 2 indicate the number of productions of interest for the 6 seeding algorithms described in section II. Note that a number of the programs contain only 4 instances of the **for** statement; this necessarily limited the number of times that certain seeding algorithms could be applied to these programs.

On a given execution each program produces a 15 by 15 boolean array, a 15 element boolean vector, and a single boolean launch decision, for a total of 241 outputs. The 241 outputs are not independent. The program calculates these outputs based upon simulated radar tracking data and various global parameters. All of the output values are computed using 15 two-dimensional geometric conditions applied to the tracking data. The required conditions are heavily parameterized.

For example, one condition asks whether a set of reflections can be contained within a circle of given radius; the radius is a parameter. Rules for generating tracking data and obtaining the global parameters can be found in [14] together with the complete requirements specification for the program.

In order to evaluate the error seeding algorithms, each of the six was used to seed one error into each of the 17 original programs. This produced 102 separate programs, each of which contained a set of original indigenous errors and one seeded error. This process was repeated four times so that each error seeding algorithm was actually used four different times on each original program. Thus a total of 408 modified programs were produced.

An eighteenth instance of the program was available. This instance was known as the "gold" program since its output was always assumed to be correct and a discrepancy with the gold program was taken to represent failure of a seeded program under test. This gold program was used in the original experiment and has been subjected to many millions of tests in other experiments.

The 408 seeded programs were then tested. In each case, 25,000 randomly generated tests were attempted, and, for each test case, all 241 outputs were checked. Randomly generated test cases representative of actual conditions were used so that realistic measures could be made of a program's mean time to failure. Recent research indicates that random testing can be very effective. [7]

The 25,000 tests were a subset of the million tests carried out on the unseeded programs in the experiment for which they were written originally. Each of the 408 seeded programs was tested with the same test cases. A test driver was responsible for generating successive input points and overseeing the seeded programs' behaviors. The test driver was supplied with a list of the test cases which caused a program to fail in unseeded form, allowing it to distinguish between failures caused by the seeded error and failures caused by the indigenous error(s). For efficiency

and convenience the driver executed all the seeded programs and the gold program for each test case generated. However, a seeded program was scheduled to execute for only as long as it had sustained less than nine failures due solely to the seeded error. Descheduling a seeded program after nine failures yielded a reasonable estimate for its mean time to failure.

IV OVERVIEW OF RESULTS

The subject programs for this experiment were extremely reliable before they were seeded with errors. During the one million test cases to which they had been subjected, none exhibited a reliability less than 0.997. In fact, of the 17 programs selected for the seeding experiment, no errors were found in 5 of them during the original one million tests.

---

(1) The original, unmodified program and the seeded program both succeed. This is a typical result.

(2) The original program succeeds and the seeded program fails. The failure is blamed solely on the seeded error.

(3) The original program and the seeded program both fail. The failure is blamed solely on the indigenous error(s).

(4) The original program fails and the seeded program succeeds. Intuitively one would not expect seeded errors to mask indigenous ones. Nonetheless, such behavior *was* observed.

(5) The seeded program causes an execution-time error. Pascal has no facility for dealing with this irritating event. The mean time to failure due to the seeded error is calculated with the data obtained up to the execution-time fault, and simulation with the seeded program stops.

(6) The seeded program fails to terminate.

Figure 1. Possible Outcomes of a Test Case Execution

When a program containing a seeded error, and perhaps containing indigenous errors, is tested, there are several possible outcomes of the execution of a given input case. The possibilities are outlined in figure 1.

The metric that used in the evaluation of seeded errors is the Mean Time to Failure (MTF), which is defined to be the total number of test cases run divided by the number of failures due to a particular seeded error. Where 25,000 tests were possible, this number is obtained from the complete set of test cases; otherwise it is derived from the number of tests possible before the seeded error generated nine failures. We consider an error to be resilient if it is infrequently found in random testing: i. e. resiliency implies a large MTF.

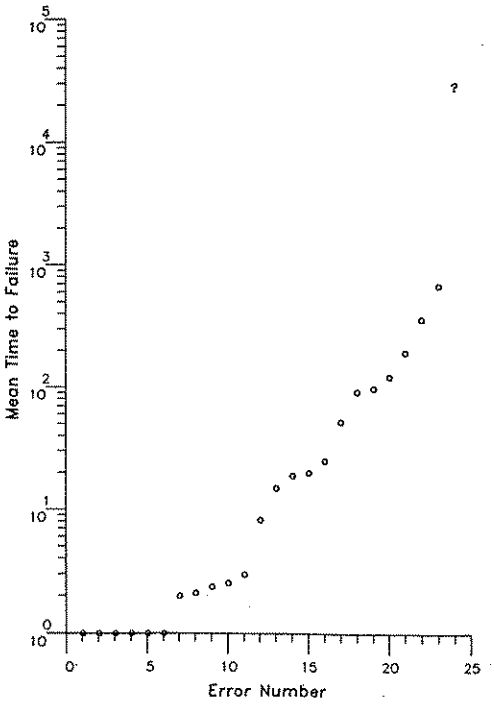

Figure 2: Seeded Errors in Typical Program

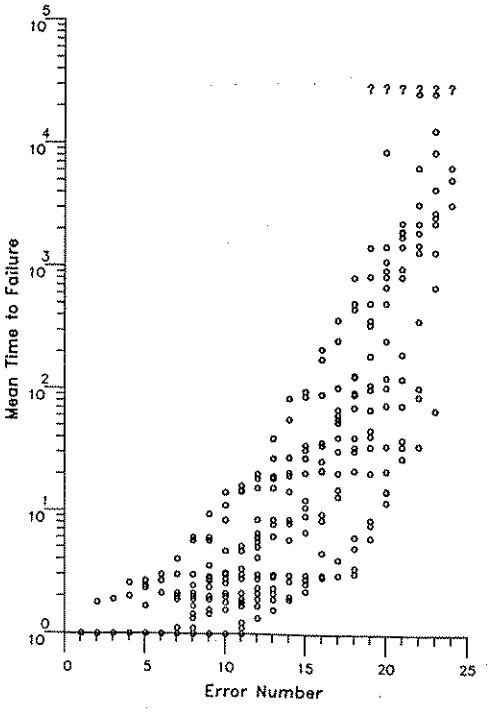

Figure 3: Presentation of All Seeded Errors

A graph for a typical program is shown if figure 2. The graph shows the results obtained from the 24 seeded programs derived from one original program. Recall that the 24 programs were obtained by using each of the six error seeding algorithms four different times on the original program. The vertical axis is the MTF measured in cases. Note that a logarithmic scale is used for the vertical axis. The horizontal axis is just error number and, to make the graphs useful, the errors are sorted into increasing order of MTF. The '?' symbols indicate errors that have an MTF greater than 25,000; these seeded errors did not cause *any* failures during the 25,000 tests that were executed. Altogether, 29 of the 408 seeded errors, or 7.1%, were not discovered in the 25,000 test case run. A discussion of these results is presented in section V.

MTF data for the 24 seeded errors was obtained for all 17 programs. The shapes of the curves were similar for all the programs. Also, for 14 of 17 programs, at least one seeded error had an MTF greater than 25,000. However, no

Seeding Algorithm	MTF of Seeded Error	Executions of Seeded Code	Failures in 1000 cases
6	241	33	5
2	656	1000	2
6	1056	1000	1
5	1272	28103	1
4	3125	266	0
3	6250	313	0
1	8333	1000	0
2	25000	1000	0
4	?	43023	0

Table 3. Frequency of Execution of Seeded Code

single seeding algorithm was responsible for these different resilient errors.

Because of the surprising resiliency of many of the seeded errors, a small test was run as an adjunct to the main experiment. It was suggested by an anonymous questioner in a colliquium that none of the seeded errors were resilient. Rather the code containing an apparently resilient error was merely infrequently exercised. To test this hypothesis nine resilient seeded errors were selected and the corresponding seeded programs were instrumented to count the number on times the seeded coded was executed. The instrumented programs were rerun on 1,000 of the 25,000 test case set. The results of these tests are given in table 3. On average the seeded code is executed several orders of magnitude more often than the program fails. This behavior is independent of the seeding algorithm used. We conclude that the resiliency of the errors is *not* due to infrequent execution.

The MTFs of all 408 seeded errors are shown in figure 3; the figure was obtained by superimposing all 17 of the graphs of the type shown if figure 2 onto one graph. This figure shows an envelope or band into which the MTF of all the seeded errors fall.

Figure 4 shows a summary of the results obtained for the 17 programs. In this figure, the MTFs of all 408 seeded errors have been sorted and placed into buckets of size 17. The average of each bucket was then plotted. The shape of the average curve in figure 4 is similar to all of the graphs of the type shown if figure 2. This curve might be typical for errors seeded using these seeding algorithms although there is no theoretical reason to justify this. If this curve is indeed typical, then it might be used as the basis for a modified analytic approach to error seeding and reliability estimation.

The performance of a representative three of the six seeding algorithms is shown in figures 5, 6, and 7. Each graph shows the MTFs obtained for all 17 programs using just one of the seeding algorithms. Since each algorithm was applied

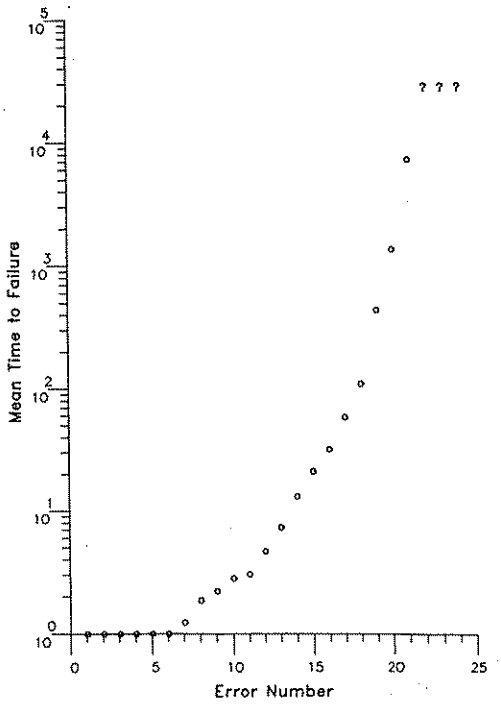

Figure 4: Average of All Seeded Errors

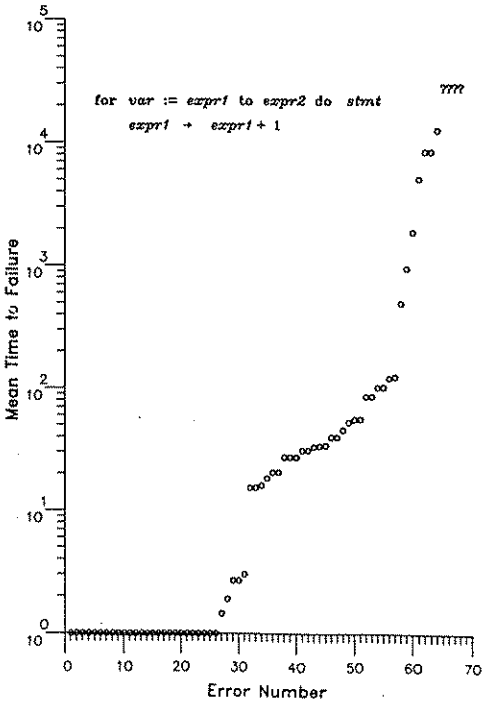

Figure 5: For Statement Modifications

four times to each program, a total of 68 (i.e. 4 * 17) data points are shown in each figure. These particular seeding algorithms yielded MTFs over the entire range. All six seeding algorithms yielded MTF curves with similar shapes when applied to all 17 programs in this way.

The syntax-based seeding algorithms used in this experiment might, of course, introduce infinite loops into programs. This is outcome 6 in Figure 1. Such errors actually were seeded. Consider the application of algorithm 6 to the code shown below. If the third line is deleted, then the program in question will never terminate.

```
i := 1;
while (i < LIMIT) do begin
   i := i + 1;
   {some computation in i}
end;
```

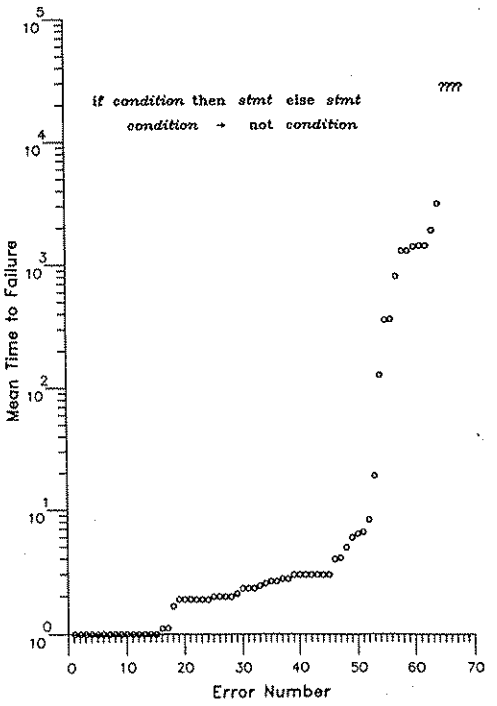

Figure 6: If Statement Modifications

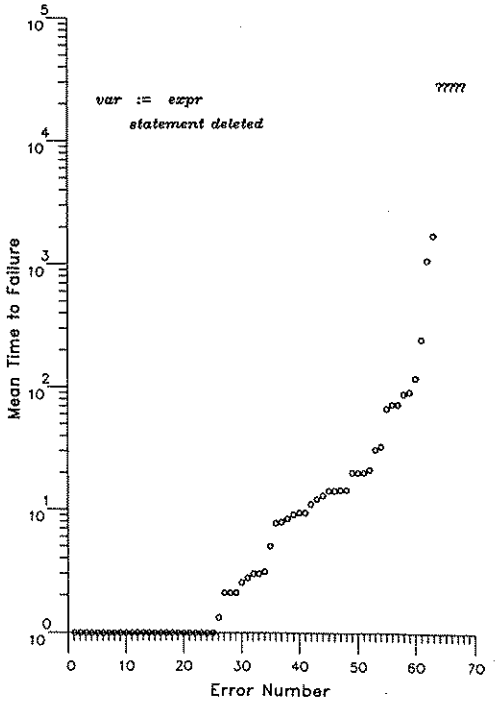

Figure 7: Assignment Statement Deletions

Perhaps the most remarkable result of this evaluation experiment is that for two seeded programs, several instances of the fourth possible test outcome listed in figure 1 actually occurred; a seeded error partially *corrected* an indigenous error. For two of the 408 programs generated, the effect of deliberately introducing an error was that an indigenous error no longer had the same effect. On some of the test cases where the unseeded program failed, the seeded program produced the *correct* output. We refer to this as the *masking effect* since the seeded error is in effect masking the indigenous error. The masking effect is discussed in more detail in section V.

V EFFECTS OF THE SEEDED ERRORS

A typical effect of altering at random a nearly correct program is to cause correctly mapped input points to be mapped incorrectly. However, a textual program alteration need not change the functionality of the program since infinitely many programs can implement the same function. An alteration which normally introduces a bug into code occasionally does *not* alter the functional behavior of the program. Consider cases where defensive programming has been employed:

if *impossible condition* **then** *invoke recovery action*

If the conditional expression truly cannot occur, then replacing it with *false* will not affect the behavior of the code. We term such an alteration *benign*. The experiment makes no attempt to decide which of the 29 undiscovered seeded errors are benign; such a determination is necessarily error prone.

Another example of a benign seeded error is given by the **for** statement modification. Intuitively, one might expect that restricting the range of a computation would guarantee that the code is erroneous at least to the degree that some intended computation inside the **for** loop is left unperformed. But consider the following example extracted from one of the programs:

```
for i := 1 to SIZE do
   for j := 1 to SIZE do begin
      if i <> j then
         MATRIX [i, j] := {some computation in i, j};
      MATRIX [j, i] := MATRIX [i, j];
   end;
```

Limiting the range of either **for** loop (but not both) leaves the resulting computation functionally unchanged. Indeed, the seeded error improves efficiency. Realizing that MATRIX is symmetric and that the diagonal elements are irrelevant, the programmer used the assignment:

```
MATRIX [j, i] := MATRIX [i, j];
```

The programmer failed to carry this observation to its full implication, however, and both indices run over the complete range of MATRIX. Had the programmer "correctly" coded the section it would read:

```
for i := 1 to SIZE do
    for j := i+1 to SIZE do begin
       MATRIX [i, j] := {some computation in i, j};
       MATRIX [j, i] := MATRIX [i, j];
    end;
```

The code section now has fewer instructions and runs about twice as fast, but it can no longer withstand the seeder.

The general case of such a data flow anomaly is the code sequence:

```
definition 1    variable := expression 1;
definition 2    variable := expression 2;
   use              {use of variable}
```

The difficulty is that *variable* is defined twice before it is used; thus the action of the seeder on *definition 1* is irrelevant. Some compilers make an effort to detect such anomalies in the code. No compiler, though, can deal effectively with the MATRIX references in the previous example. Finally, the programmer's inclusion of inefficient code in the above example cannot be used as an indictment of poor program quality. The program from which the example is excerpted is 1 of only 6 of the original 27 programs which passed the 1,000,000 test case N-version experiment without failure. [1, 9]

Another possibility is that certain points mapped *incorrectly* by the original program are mapped *correctly* by the the seeded program. We term such an event the *masking effect*.

Two masking errors were observed in this experiment. They were both varieties of the **if** statement modifications. In one case a programmer attempted to determine which quadrant contained an (x, y) point. The code for such a determination involved a set of nested **if** statements. Because of the way in which

quadrant data were used, an incorrect determination only occasionally caused some of the 241 boolean outputs to be incorrect, thus muting the effect of the error. The programmer failed to examine the appropriate conditions for quadrant determination when a point lay upon the x-axis as illustrated in figure 8. In effect, points were assigned to quadrant 2 even if the x coordinate were positive. This determination was, on average, incorrect in half of such cases. The seeded error of *condition* -> not *condition* had the effect of causing the program to assign points to quadrant 1 even if the x coordinate were negative. Due to the limited number of times that a point appears on the x-axis and the often muted effect of an incorrect determination, few failures were detected in the 241 boolean outputs. The MTF for the seeded error, as well as for the indigenous error, was 1 in 3103 yielding a reliability of 0.999678. Note that the effect of the seeded error was to cause some points in the input set that had been mapped incorrectly to be mapped correctly and others which had been mapped correctly to be mapped incorrectly. The seeded error *partially* corrected the indigenous error; the effect was to *shift* the subset of input points

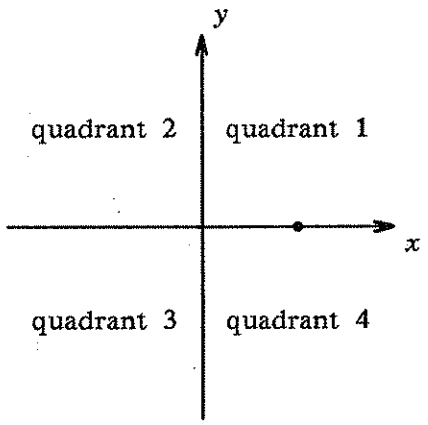

Figure 8. Incorrect Quadrant Determination

mapped incorrectly by the program.

The other observed example of the masking effect occurred when a programmer incorrectly treated a certain condition as a special case:

```
if irrelevant_special_case then
   { erroneous computation }
else
   { correct computation for all  }
   { cases including special_case }
```

The seeder injected the alteration *condition* -> false into the text and the special case code was no longer executed. The effect was to *entirely* correct the indigenous error. The seeded program never mapped an input point incorrectly unless the original program had done so (the original program had several indigenous errors), but the seeded program did map correctly some input points that the original program did not.

VI INDIGENOUS vs. SEEDED ERRORS

Recall from table 2 that the original 17 programs used in the experiment each contained from 0 to 4 indigenous errors; together the programs contained a total of 23 indigenous errors. The failure rates due to each indigenous *error* were determined from the results of the 1,000,000 test case run. A plot of the MTFs for all of the indigenous errors is given in figure 9. The '*'s in figure 9 represent errors whose MTF did not fit on the scale provided. The values represented on figure 9 by '*'s were not given on a rescaled graph so that the y-axis on *all* graphs presented in this paper would maintain an *identical* scale.

The determination of which error is responsible for a given failure was not straightforward in all cases. Sometimes *all* of several errors had to be repaired for a program to map a particular input case correctly. In other cases *any* of several errors could have been repaired. For the purposes of comparison to the seeded

errors, an indigenous error was deemed to be the cause of a failure if, with any subset of other indigenous errors still in the program, it was possible to cause an incorrectly mapped test case to be mapped correctly by fixing only the error in question. This technique yielded the most pessimistic value for reliability that was reasonable.

Figure 10 shows a similar graph which plots the MTF of the most resilient *seeded* errors. Recall that an initial condition for acceptance of a program was that it pass a set of 200 randomly generated test cases without error. The graph in figure 10 was obtained by taking all of the seeded errors with MTFs of more that 200 and forming a graph suitable for comparison with figure 9. Since there were approximately three times as many seeded errors with a known MTF greater than

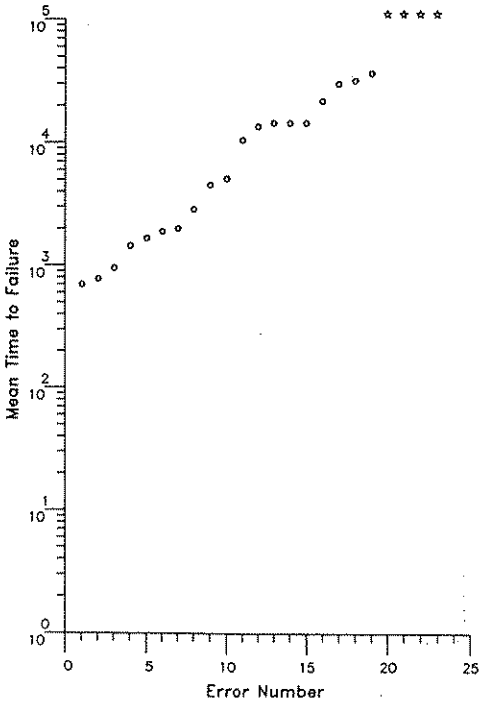

Figure 9: Indigenous Errors

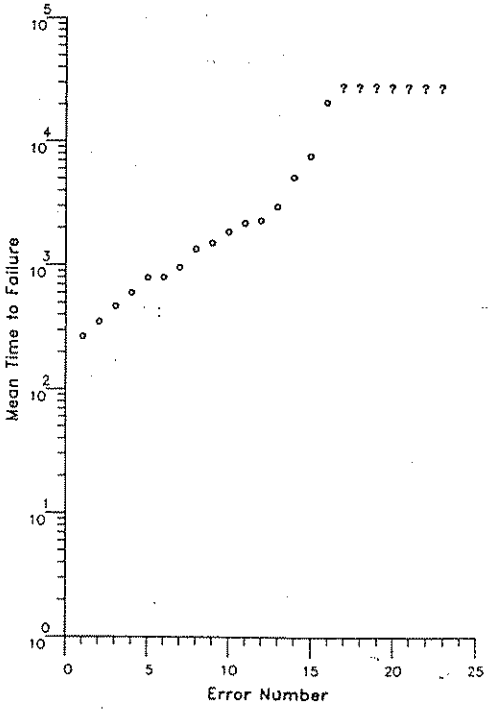

Figure 10: Resilient Seeded Errors

200 as there were indigenous errors with MTFs in the appropriate range of 200 to 25,000 (there were 50 such seeded errors vs. 16 comparable indigenous ones), the seeded errors were sorted by MTF and placed into buckets of size 3. The average of each bucket is presented on figure 10.

The surprising thing about figures 9 and 10 is their remarkable similarity. There is no qualitative difference in the shape of MTF curves between the indigenous errors left by the original programmers and the more resilient errors injected by the seeder. Both curves are approximately log linear in shape. We suspect that the as-yet-undiscovered seeded errors have MTFs which will continue this log linear trend and thus match the indigenous errors in the right hand portion of figure 9.

Neither the seeded errors nor the indigenous errors have MTFs which form a horizontal line, as would be expected if all errors were equally difficult to locate. In fact, the indigenous errors have MTFs ranging from 696 to 500,000. Clearly the assumption made by both the error seeding and capture/recapture techniques that errors are all approximately equally hard to find is violated by this data.

Table 4 shows the impact on capture/recapture of making this invalid assumption. Suppose that all 17 programs are assumed to form one new program which succeeds only if all 17 subprograms succeed. Clearly, the number of indigenous errors in this new program is at least 23 since 23 indigenous errors have been discovered in the 17 original programs. Now suppose that the capture/recapture method is applied to this new program using the 1,000,000 test case results from the N-version experiment.

In table 4, the entry marked "Total Test Cases" is the sum of the test cases used in the capture and the test cases used in the recapture. Half the test cases were used for each phase. Recall that capture/recapture predicts that N, the total number of indigenous errors in a program is related to n, r, and k by:

Total Test Cases (in thousands)	Errors Captured *(n)*	Errors Recaptured *(r)*	Common Errors *(k)*	Errors Predicted *(N)*	Total Discovered *(n+r-k)*
1	3	3	1	9	5
2	5	8	3	13	10
5	9	8	6	12	11
10	8	13	8	13	13
20	10	15	9	17	14
50	16	17	15	18	18
100	20	19	19	20	20
200	20	20	20	20	20
500	21	21	20	22	22
1000	23	23	23	23	23

Table 4. Applications of Capture/Recapture

$$N \approx \frac{n\,r}{k}$$

Note that the predicted value of N is consistently low. Rather than approximating the total number of indigenous errors, which is known to be at least 23, N appears to follow the number of errors discovered up to the current point in testing. Indeed, for the last five entries in the table the method predicts that all of the errors have been found. When a total 18 errors had been found, the method predicted that testing was complete. Similar predictions were made when 20, 22, and 23 indigenous errors had been discovered. Such results do not foster confidence in the capture/recapture method.

VII CONCLUSIONS

Using only simple syntactic algorithms, it is possible to seed into the programs available for this experiment errors that are extremely difficult to locate. In addition, since 17 different programs written to the same specifications were used in this experiment, and the results of seeding were similar for all of them, the efficacy of these seeding algorithms seems to be independent of the programmer who wrote the program being seeded. These results may or may not hold for other applications; other experiments are needed on other programs to obtain more data.

Of the seeding algorithms that tried, none was able to generate resilient errors *predictably*. All the seeding algorithms generated some errors that were trivial to locate and others that were remarkably resilient. It was not possible to predict *a priori* which were which. Unfortunately, this suggests that it might be hard to comply with the requirement used by Mills in his original analysis of error seeding [9]. In that analysis, it was assumed that a specified number of errors could be seeded that were equally likely to be found and as hard to find as indigenous errors.

The effects of benign errors and the masking effect have not been addressed in any consideration of error seeding as a testing strategy, but clearly they must be taken into account. For error seeding to be effective, it must be automated. A system that implements error seeding must therefore expect that some seeded errors will have no effect on a program's functionality and therefore will never be found. Such a system must also allow for the demonstrated possibility that seeded errors will interfere with other errors, indigenous and seeded. In addition, provisions must be made which allow for recovery from errors which generate execution-time faults. Finally, some of the seeded programs will never terminate. Some reliable ad hoc technique for detecting failure to terminate must be built into any reasonable system.

Perhaps the most useful result of this experiment is the light it sheds on the assumptions used by the error seeding technique. Listed below are these three assumptions and an evaluation of each based on the outcome of this experiment:

(1) *Indigenous errors are all approximately equally difficult to locate.* The indigenous errors found within the original programs had MTFs spanning *three orders of magnitude*. If either the error seeding or capture/recapture technique is to be effective, this fact must be taken into account. Otherwise, invalid results such as those presented in table 4 will occur.

(2) *Seeded errors are all approximately as difficult to locate as indigenous errors.* Paradoxically, this least-believed assumption seems to be quite reasonable. It *is* possible to seed resilient errors with simple seeding algorithms. Unfortunately, to obtain a few resilient seeded errors with these techniques it is necessary to seed many errors which have very low MTFs. We do not know whether the distribution of MTFs observed in this experiment is typical in general. Even if the distribution were typical, it is not clear whether this information could be made to fit into the original model proposed by Mills.

(3) *Errors, indigenous or seeded, do not interfere with one another.* This assertion is simply not true. Errors *do* interact with one another in subtle ways. Again it is difficult to see how either error seeding or capture/recapture could be adapted to recognize this fact.

Given that it is possible to seed resilient errors with simple syntactic algorithms, we suggest the following validation process:

(1) Debug the program in question and complete all pre-validation procedures.

(2) For validation, prepare a set of test cases and execute the program using these test cases. No faults should be revealed since this testing is intended to perform validation, i.e. showing that the program does what it is required to do.

(3) Seed the program, and rerun the input cases from the testing phase. Clearly if any seeded errors are not discovered, and these errors prove not to be benign, then the testing phase has been inadequate. Under such conditions the test set would have to be expanded.

ACKNOWLEDGEMENTS

Earl Migneault of NASA Langley Research Center provided much insight into the effects observed in this experiment. We thank Sue Brilliant for the data on the indigenous errors within the programs. It is a pleasure to acknowledge the students who wrote the programs that were seeded and tested in this experiment: Carol Finch, Nancy Fitzgerald, Marina Heiss, Edith Irwin, Linda Lauterbach, Shivaji Samanta, Jonathan Watts, and Pat Wilson from the University of Virginia, and Richard Bowles, Paul Higgins, Alastair Milne, Scott Musgrave, Thieu Nguyen, J. Peck, Phillip Ritter, and Robert Sargent from the University of California at Irvine. The Academic Computer Center at the University of Virginia, the AIRLAB facility and the Central Computer Complex at NASA Langley Research Center provided generous amounts of computer time to allow the programs to be tested. This work was supported in part by NASA grant number NAG-1-242.

REFERENCES

[1] S. S. Brilliant, "Analysis of Faults in a Multi-Version Software Experiment", MS Thesis, University of Virginia (May, 1985)

[2] T. A. Budd, "Mutation Analysis: Ideas, Examples, Problems, and Prospects", from *Computer Program Testing*, ed. B. Chandrasekaran, S. Radicchi. Urbino, Italy: North Holland Publishing Company, 1981. pp 129-148.

[3] T. A. Budd, A. DeMillo, R. J. Lipton, and F. G. Sayward, "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs", *Proceedings ACM Symposium on Principles of Programming Languages*, 1980.

[4] W. R. Collins, J. C. Knight, and R. E. Noonan, "A Translator Writing System For Microcomputer High-Level Languages and Assemblers", "*NASA/AIAA*Conference NASA Conference Publication 2158, NASA Goddard Space Flight Center, Greenbelt, MD (November 1980).

[5] B. Cornelius, and I. Thomas, *The Hull V-Mode Pascal Compiler, User Manual Version 3.3*, University of Hull, Hull, England. 1982.

[6] J. W. Duran, and S. Ntafos, "A Report on Random Testing", *Proceedings 1981 Fifth International Conference on Software Engineering*.

[7] J. B. Goodenough, and S. L. Gerhart, "Toward a Theory of Test Data Selection", "*IEEE*Transactions Vol. SE-1, No. 4 (June 1975).

[8] J. S. Gourlay, "A Mathematical Framework for the Investigation of Testing", "*IEEE*Transactions Vol. SE-9, No. 6 (November 1983).

[9] J. C. Knight, and N. G. Leveson, "A Large Scale Experiment in N-Version Programming" Digest of Papers: Fifteenth annual Symposium on Fault-Tolerant Computing, Ann Arbor, Michigan (June 1985).

[10] H. D. Mills, "On The Statistical Validation of Computer Programs", in "*Software*Productivity", Little Brown, Toronto.

[11] G. J. Myers, *Software Reliability, Principles and Practices*. New York, NY: John Wiley & Sons, 1976. pp 336-338.

[12] G. J. Schick, and R. W. Wolverton, "An Analysis of Competing Software Reliability Models", "*IEEE*Transactions Vol. SE-4, No. 2 (March 1978).

[13] L. D. St. Jean, "Testing Version Independence In Multi-Version Software", MS Thesis, University of Virginia (January, 1985).

[14] L. J. White, and E. I. Cohen, "A Domain Strategy for Computer Program Testing", "*IEEE*Transactions Vol. SE-6, No. 3 (May 1980).