Increasing Confidence in Software Through Program Perturbations[†]

David R. Hanson, Richard J. Lipton,
and Frederick G. Sayward

Research Report #100

Department of Computer Science
Yale University
New Haven, Connecticut 06520

[†] This work was supported by the Office of Naval Research under Grant #N00014-75-C-0752.

ABSTRACT

A new method for increasing confidence in software based on the premise that competent programmers write correct or "nearly" correct software is presented. The envisioned system takes as input a program and a set of test data. It produces and executes a set of perturbation programs, and generates a list indicating which perturbation programs are indistinguishable from the original program (with the given data). A non-empty list indicates that the data is not adequate, that there exist equivalent programs in the list, or that the original program is incorrect. An empty list indicates that the original program is either correct or "far" from correct. While the set of perturbation programs should be large enough to include many commonly made errors, it appears that there is a coupling effect suggesting that errors not present in the set of perturbation programs are still checked by this method. Two examples of the use of the method are given.

1. INTRODUCTION

It is well-known that the design and construction of reliable software is a difficult task. The purpose of this paper is to present a new method that may aid in constructing reliable software, and to illustrate the application of this method to Fortran subroutines.

Current approaches to reliable software fall into three categories: constraining, proving, and testing. By constraining we mean those methods that place restrictions or constraints on programmers in an effort to force them to design reliable software. The whole of structured programming [1] with its restrictions on the use of data and control structures falls into this category. Also included in this category are the attempts to create a variety of methodologies, such as those of Parnas [2] and Wirth [3], that enable programmers to avoid certain common errors. Although these methods have met with encouraging success they by no means seem to solve the entire problem.

Another approach is to rely on proving that a program satisfies certain formal properties. These methods, which are usually referred to as verification methods [4,5], hold the promise of correct software. However, for a variety of reasons -- principally the difficulty of specifying software in formal terms and the difficulty of the proofs [6] -- these methods are not yet practical for "real" programs. Indeed, one significant problem is that while verification techniques may work on "clean" languages such as Alphard [7] and Pascal [8], there has been little evidence that they will be successful on "dirty" languages such as Fortran. Consequently, verification schemes do not appear applicable to the large bulk of existing software, and their payoff may be far in the future.

Program testing has long been in the paradoxical position of being the traditional method of checking real software and yet, until recently [9-14], receiving little attention in the literature. Our belief is that program testing, while not on the sound mathematical foundation of the other methods, can be used to develop quite powerful and useful methods for constructing reliable software.

Program testing consists of determining if some program P works correctly on some given data I. The basic question is then "if P works on I, is P correct?" The answer is, of course, not necessarily: P may work on I and yet fail on all other input data. The central problem in program testing is to find a way of determining whether I is an adequate test of P, not in any formal sense but rather in an empirical sense. More precisely, testing is an <u>inductive</u> process whereas other approaches, such as verification, are <u>deductive</u> approaches.

The majority of work in program testing has been concerned with the use of path analysis and symbolic execution to generate adequate test data [11]. These systems generate test data by solving a system of inequalities constructed by symbolically executing all control paths of a program. Although there is some evidence that the use of conditional statements in languages such as Fortran often results in a linear system of inequalities, the general problem of producing test data for all execution paths is an unsolvable problem [14].

All of the methods described above ignore the fact that programs produced by competent programmers are usually "almost" correct. Our approach relies on this observation and attempts to provide a comprehensive evaluation of both the program and its associated test data. We assume that if a program is not correct, then it is a small "perturbation" of the correct one. The basic idea is to take a program and its associated test data and generate all the possible simple perturbations of that program, and run each one with the given data. If all the perturbation programs give incorrect results, it is very likely that the original program is correct. If some of the perturbation programs yield correct results, the data is inadequate, there is an error, or the perturbations are equivalent programs.

Superficially, this method would appear capable of detecting only simple errors such as typographical mistakes leading to undefined variables. However, there exists a coupling effect; test data that distinguishes all simple perturbations is so sensitive that it also implicitly distinguishes complex perturbations. This effect is due to the observation that competent programmers design programs that are very sensitive to even mild alterations.

The motivation for this approach comes from the fault detection problem in hardware theory. For example, if C is a circuit that forms the complement of a 32-bit number, then to test an arbitrary circuit we need to check 2^{32} inputs. But circuits, like programs, are highly structured objects, and if there is an error in C, it is very likely to be a single fault error. By comparing the original circuit with all possible perturbations of C involving single fault errors, it is possible to reduce the number of inputs from 2^{32} to approximately 100. The basis for presuming that C is correct is that the probability of C containing a double fault is extremely small.

Section 2 describes the perturbation method in more detail and indicates the type of simple perturbations we are considering. Two examples are given in section 3. The first example illustrates the application of the method to a correct program, and the second illustrates its application to a program containing a complex error. This second example demonstrates the coupling effect. Section 4 describes how the method can be incorporated into a system and be used by both programmers and managers in large software projects.

2. DESCRIPTION OF THE METHOD

Let P be a program and let I be a set of data. We wish to determine if I is an adequate set of test data. (We do not mean that P is guaranteed to be correct, but rather that there is empirical evidence that P is correct -- remembering that P is not a "random" program but has been constructed by a competent programmer.) Our method relies the construction of a set of <u>perturbation programs</u> $P_1,\ldots,P_k$. Initially, each P_i can be thought of as corresponding to one of all possible errors that could be made in constructing P. We will see, however, that a coupling effect suggests that this is an unnecessarily conservative view. The set of data I is <u>adequate</u> if

(i) P works correctly on I, and

(ii) none of the P_i works correctly on I.

Clearly, if I is adequate then P is <u>correct</u> or the set of perturbation programs was improperly constructed.

There are two reasons to believe that this method of considering only simple perturbations should work. First, there is empirical evidence that most programming errors are relatively simple. For example, Youngs [15] found that 15% of all non-syntax errors were merely instances of the use of the wrong variable. Most of the errors studied by Gannon [16], which were used to direct language design, were also relatively simple. Indeed, there are numerous stories about large software systems failing because of incredibly simple errors.

Second, we have evidence that checking only simple perturbation programs will force the test data to be so sensitive that even more complex errors will be checked. The significance of this <u>coupling effect</u> is that the potential exists for catching a wide range of errors while only testing for simple ones. An example of this effect is given below in connection with Hoare's FIND program [17].

Errors of the type found by Youngs and Gannon might be called terminal errors. This type of error provides a starting point for the construction of the programs P_i by making perturbations to P. Let G be a context-free grammar for the language L. For program P in L define term(Q) to be all programs Q in L such that the parse tree of Q differs from the parse tree of P only at the leaves. For example, if S is a Fortran IF statement, then term(S) would contain only IF statements. If S is the statement

```
IF (A .EQ. C) B = 2
```

then the statements

```
IF (A .NE. C) B = 1
IF (A .EQ. B) C = 2
```

are members of term(s).

If some P_i differs from P by at most k terminal symbols, then P_i is called a k-terminal perturbation of P. The set of k-terminal perturbations of P is denoted by $\text{term}_k(P)$.

There are simple errors that are not k-terminal. For example, an error in the parentheses structure of a Fortran arithmetic expression is not k-terminal. Errors of this type, however, are caught during compilation since P is no longer in L. Permuting the order of program statements and failure to initialize a variable are other examples of errors that are not k-terminal. A more sophisticated system than the one described here could examine the structure of a program and produce perturbations that reflect the permutation of statements, variations of loop boundaries, and changes in the flow of control.

3. TWO EXAMPLES

In this section, two experiments are described to illustrate the perturbation method.

3.1 MAX

The first example involves the MAX program analyzed by Naur [18]. The problem is to set R to the index of the <u>first</u> occurrence of the maximum element in the array A(1),...,A(N). The following Fortran subroutine performs this operation.

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

For this subroutine, the following three classes of 1-terminal perturbations were considered.

<u>Relational operator</u>: Replace the relational operator .GT. in statement 3 by all the alternatives selected from the set of relational operators {.EQ., .NE., .LE., .LT., .GE., .GT.}.

<u>Constants</u>: Replace the three occurrences of constants by members of the set {0, 1, 2}.

<u>Variables</u>: Replace the seven occurrences of variables by members of the set {R, I, N, A(I), A(R)}.

Applying these perturbations to MAX yields 39 perturbation programs, P_1 through P_{39}, whose characteristics are summarized in table 1. Fourteen of these programs can be eliminated from further consideration by inspection. Four programs do not compile, seven lead to subscript errors due mainly to the use of uninitialized variables, and three have ill-formed loops. The initial

Table 1
The MAX Experiment

program perturbation	line	name	data 1 (1,2,3)	data 2 (1,3,2)	data 2 (3,1,2)	1,2&3	data 4 (2,2,1)	1,2,3&4
.GT. -> .EQ.	3	P_{1}			M			
.GT. -> .NE.	3	P_{2}	M					
.GT. -> .LT.	3	P_{3}						
.GT. -> .LE.	3	P_{4}						
.GT. -> .GE.	3	P_{5}	M	M	M	M		
1 -> 0	1	P_{6}	M	M				
1 -> 2	1	P_{7}	M	M				
2 -> 0	* 2	P_{8}						
2 -> 1	2	P_{9}	M	M	M	M	M	M
1 -> 0	* 2	P_{10}						
1 -> 2	2	P_{11}		M	M		M	
R -> I	* 1	P_{12}						
R -> N	* 1	P_{13}						
R -> A(I)	* 1	P_{14}						
R -> A(R)	* 1	P_{15}						
I -> R	* 2	P_{16}						
I -> N	* 2	P_{17}						
I -> A(I)	* 2	P_{18}						
I -> A(R)	* 2	P_{19}						
N -> I	* 2	P_{20}						
N -> R	2	P_{21}		M	M		M	
N -> A(I)	* 2	P_{22}						
N -> A(R)	2	P_{23}		M	M		M	
A(I) -> I	3	P_{24}	M	M	M	M		
A(I) -> R	3	P_{25}			M		M	
A(I) -> N	3	P_{26}	M	M	M	M		
A(I) -> A(R)	3	P_{27}			M		M	
A(R) -> I	3	P_{28}		M	M		M	
A(R) -> R	3	P_{29}	M	M				
A(R) -> N	3	P_{30}			M		M	
A(R) -> A(I)	3	P_{31}			M		M	
R -> I	* 3	P_{32}						
R -> N	* 3	P_{33}						
R -> A(I)	3	P_{34}			M		M	
R -> A(R)	3	P_{35}			M		M	
I -> R	3	P_{36}			M		M	
I -> N	3	P_{37}	M		M		M	
I -> A(I)	3	P_{38}	M		M		M	
I -> A(R)	3	P_{39}	M		M		M	

* indicates that the perturbation was eliminated before execution.
M indicates that the execution results are the same as for MAX.
x -> y represents substituting y for x in the indicated line.

test data consisted of three cases; a three-element vector in which the maximum is varied over the three positions.

The initial set of test data eliminates all but perturbations P_5, P_9, P_{24}, and P_{26}. That is, these perturbation programs gave the same results as the original version of MAX. The presence of P_5 indicated that the inadequacy of the test data might be due to the absence of repeated array elements. Hence a fourth case was added to the test data to resolve this inadequacy. The results of this test are given in the rightmost column of table 1, which shows that all perturbations except P_9 have been eliminated. P_9 as formed by a change of constants -- the DO loop is started at 1 instead of 2. Although this change results in a slightly longer execution time, close examination reveals that P_9 and MAX are functionally equivalent programs. There is no test data that can be used to distinguish these two programs. Consequently, MAX has passed the 1-terminal analysis.

3.2 FIND

The second example, which is described in less detail, involves Hoare's FIND program [17]. FIND has two input parameters: an integer array A and an array index F. FIND is to transform A such that $A(I) \leq A(F)$ for all $I < F$ and $A(I) \geq A(F)$ for all $I > F$. This problem is of particular interest because a subtle 2-terminal perturbation of FIND, called BUGGYFIND, has been extensively analyzed by SELECT [19], a system that generates test data. The subtle change is as follows. In FIND the elements of A are interchanged depending on a conditional of the form

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

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

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

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

Taking BUGGYFIND as the original program, consider the following 1-terminal perturbations.

BF1: the conditional is X .LE. A(F) .AND. R .LE. Y.

BF2: the conditional is X .LE. R .AND. A(F) .LE. Y.

The results of running these programs on all permutations of (0, 1, 2, 3) are listed in table 2. Observe that in all cases BUGGYFIND, BF1, and BF2 produce identical results. Consequently, since BF1 and BF2 cannot be eliminated, BUGGYFIND must be viewed with some suspicion. The important point of this example is that a 2-terminal error was detected using only 1-terminal perturbations. This illustrates the coupling effect, which indicates that simple perturbations are capable of detecting more complex errors.

4. IMPLEMENTATION CONSIDERATIONS

We envision a system in which there is some offline programmer-system interaction. The programmer submits a "well-analyzed" program and test data to the system. The system returns a list of perturbation programs that are indistinguishable from the original progam using the given data. If the list is long, the programmer may wish to simply enrich the data and try again. If the list is short -- on the order of 10 programs -- the programmer may analyze the perturbations by hand to determine whether they are equivalent programs or

Table 2
The Find Experiment, F = 3

A	FIND	BUGGYFIND	BF1	BF2
0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
0 1 3 2	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
0 2 1 3	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
0 2 3 1	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
0 3 1 2	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
0 3 2 1	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
1 0 2 3	1 0 2 3	1 0 2 3	1 0 2 3	1 0 2 3
1 0 3 2	1 0 2 3	1 0 2 3	1 0 2 3	1 0 2 3
1 2 0 3	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
1 2 3 0	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
1 3 0 2	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
1 3 2 0	1 0 2 3	1 0 2 3	1 0 2 3	1 0 2 3
2 0 1 3	1 0 2 3	1 0 2 3	1 0 2 3	1 0 2 3
2 0 3 1	1 0 2 3	1 0 2 3	1 0 2 3	1 0 2 3
2 1 0 3	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
2 1 3 0	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
2 3 0 1	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
2 3 1 0	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
3 0 1 2	1 0 2 3	1 0 2 3	1 0 2 3	1 0 2 3
3 0 2 1	1 0 2 3	1 0 2 3	1 0 2 3	1 0 2 3
3 1 0 2	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
3 1 2 0	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
3 2 0 1	0 1 2 3	0 2 1 3	0 2 1 3	0 2 1 3
3 2 1 0	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3

there is reason to suspect the original program. The key point is that <u>all</u> perturbation programs thought to be equivalent to the original program must be "signed off" by the programmer before the program is accepted. Thus by having the system generate reports indicating who has signed off various programs, subsequent failure of a program can be readily attributed to the proper source.

An apparent drawback to this method is the potentially explosive number of perturbation programs, even at the 1-terminal level. The brute-force approach leads to a large number of programs to be compiled and executed. There seems to be little that can be done to reduce the execution time necessary to run all interesting perturbations. However, considering the enormous amount of time currently spent on ad hoc testing and debugging techniques, the amount of CPU time required does not seem excessive.

For example, the Fortran version of FIND consists of about 30 statements. This size is typical for a module in a well-structured system. Asssuming that it requires 0.01 seconds to run a test case for a four-element array, running all 24 permutations described above requires 0.24 seconds. There are approximately 1000 1-terminal perturbations in the Fortran version of FIND. Thus a complete analysis of 1-terminal perturbations requires less than 5 minutes of CPU time.

There are several ways to reduce the number of the programs to be executed. A significant number of perturbations will be rejected by the compiler. The techniques of some current program validation systems can be applied to further reduce the number of programs. For example, the DAVE system [20] can be used to eliminate programs having uninitialized variables. Presumably, the competent programmer rarely makes such errors. The set of perturbation programs may be further reduced by using a symbolic execution system to eliminate programs containing non-executable statements or unreachable paths.

A more serious problem arises when a perturbation program does not halt. To handle this, we assume that the original program completes in at least t seconds. The system then stops any perturbation program that runs longer than ct seconds, where c is some constant supplied by the programmer. These non-halting programs have not been eliminated; rather, they are reported to the programmer who must decide to either eliminate them by hand or increase the value of c and try again.

5. CONCLUSIONS

The system described in this paper represents a practical approach to program testing. The method rests on the validity of the coupling effect, that is, that simple perturbations are sufficient to catch more complex errors than actually tested by the simple perturbations. Although the coupling effect is mathematically unprovable, initial experience -- as demonstrated by BUGGYFIND -- suggests its validity. The perturbation approach also offers the possibility of testing existing programs, while most of the other approaches to software validation, such as program verification, the design of new languages, or specialized methodologies, are applicable only to new programs or programs written in a specific language.

In addition, by the appropriate choice of perturbations, the method includes several other testing schemes as subcases. For example, given a program and its test data we can determine if a given statement is executed by simply changing that statement to one that divides by zero. If the program does not halt due to a divide-by-zero error, the statement is not executed using the given data. Similar techniques can be used to determine if certain control paths are traversed or if a given variable is referenced before its definition.

The system described here is also useful as a management tool. The software manager can use the reports generated by such a system to monitor and control the development of the modules in a large project. As mentioned above, a module P might be considered acceptable if its associated test data distinguished it from all its perturbations P_i, or in the event that some small number of P_i were indistinguishable from P, the programmer responsible for P certified that those P_i were equivalent programs. If P subsequently failed, the perturbations of P that were thought to be equivalent could provide an indication of why P failed. The manager might also use this information in evaluating programmer performance.

Recent research in programming language design [16] and programming methodology [9] indicates that better languages and specialized methodologies can significantly improve software reliability. Empirical data obtained by testing programs using the perturbation scheme may also offer some insight into what specific kinds of language features and methodologies actually reduce errors. For example, a high incidence of a particular error that causes certain perturbations to be consistently indistinguishable from the original program would suggest that the language be changed to prevent that error. The undistinguishable perturbations would point to the undesirable language feature. In addition, the method might show the presence of deficiencies in module specifications, which, in turn, would suggest deficiencies in the programming methodology used. Again, the perturbation programs would help determine the nature of the deficiency.

Finally, as with any system that collects empirical data, the possibility exists for self-improvement. Initially, the perturbations of a program are produced without any real knowledge of which ones will be helpful in correcting errors. Continued use of the system, however, would provide data on which types of perturbations are most useful. Such data could then be used to "tune" the system for better performance.

REFERENCES

[1] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Structured Programming, Academic Press, New York, 1972.

[2] D. L. Parnas, A technique for software module specification with examples, Communications of the ACM, vol. 15, no. 5, May 1972, 330-336.

[3] N. Wirth, Program development by stepwise refinement, Communications of the ACM, vol. 14, no. 4, April 1971, 221-227.

[4] B. Elspas, K. N. Levitt, R. J. Waldinger, and A. Waksman, An assessment of techniques for proving program correctness, Computing Surveys, vol. 4 no. 2, June 1972, 97-147.

[5] R. L. London, A view of program verification, SIGPLAN Notices, vol. 10, no. 6, June 1975, 534-545.

[6] R. A. DeMillo, R. J. Lipton, and A. J. Perlis, Social processes and proofs of theorems and programs, Conference Record of the Fifth Symposium on Principles of Programming Languages, to appear.

[7] W. A. Wulf, R. L. London, and M. Shaw, Abstraction and verification in Alphard: Introduction to language and methodology, Research report, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, June 1976.

[8] K. Jensen and N. Wirth, PASCAL User Manual and Report, Lecture Notes in Computer Science, vol. 18, Springer-Verlag, New York, 1974.

[9] S. L. Gerhart and L. Yelowitz, Observations of fallibility in applications of modern programming methodologies, IEEE Trans. on Software Engineering, vol. SE-2, no. 3, September 1976, 195-207.

[10] W. E. Howden, Reliability of the path analysis testing strategy, IEEE Trans. on Software Engineering, vol. SE-2, no. 3, September 1976, 208-214.

[11] L. A. Clarke, A system to generate test data and symbolically execute programs, IEEE Trans. on Software Engineering, vol. SE-2, no. 3, September 1976, 215-222.

[12] W. Miller and D. L. Spooner, Automatic generation of floating-point test data, IEEE Trans. on Software Engineering, vol. SE-2, no. 3, September 1976, 223-226.

[13] H. N. Gabow, S. N. Maheshwari, and L. J. Osterweil, On two problems in the generation of program test paths, IEEE Trans. on Software Engineering, vol. SE-2, no. 3, September 1976, 227-231.

[14] J. C. Huang, An approach to program testing, Computing Surveys, vol. 7, no. 3, September 1975, 113-128.

[15] E. A. Youngs, Human errors in programming, Int. Journal on Man-Machine Studies, vol. 6, 1974, 361-376.

[16] J. D. Gannon and J. J. Horning, Language design for reliable software, IEEE Trans. on Software Engineering, vol. SE-1, no. 2, June, 1975, 208-214.

[17] C. A. R. Hoare, Proof of a program: FIND, Communications of the ACM, vol. 14, no. 1, January 1971, 31-35.

[18] P. Naur, Programming by action clusters, BIT, vol. 9, 1967, 250-258.

[19] R. S. Boyer, B. Elspas, and K. N. Levitt, SELECT - A formal system for testing and debugging programs by symbolic execution, SIGPLAN Notices, vol. 10, no. 6, June 1975, 234-245.

[20] L. J. Osterweil and L. D. Fosdick, DAVE - A validation error detection and documentation system for Fortran programs, Software -- Practice and Experience, vol. 6, no. 4, October-December 1976, 473-486.