Computer Science Department

TECHNICAL REPORT

ON TESTING NONTESTABLE PROGRAMS

BY

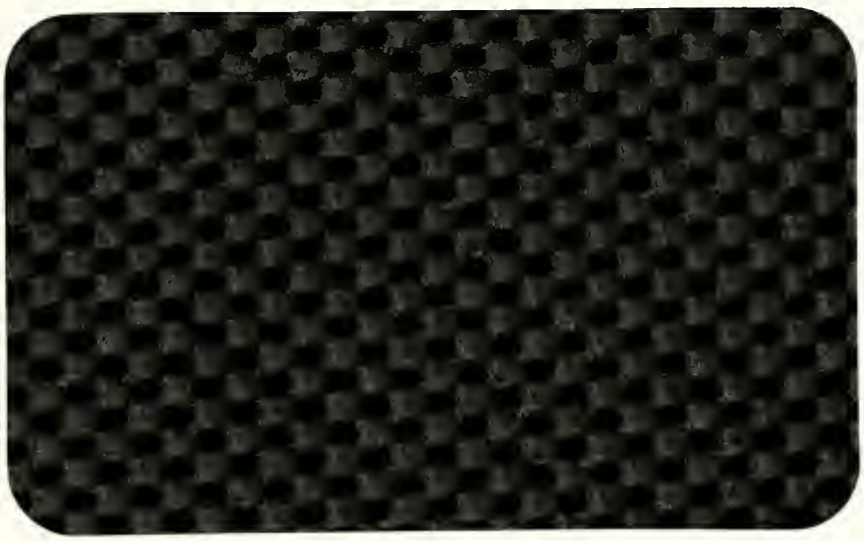
ELAINE J. WEYUKER

OCTOBER 1980

REPORT NO. 025

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012



ON TESTING NONTESTABLE PROGRAMS

BY

ELAINE J. WEYUKER

OCTOBER 1980

REPORT NO. 025

ABSTRACT

A frequently invoked assumption in program testing is that there is an oracle (i.e., the tester or an external mechanism can accurately decide whether or not the output produced by a program is correct.) A program is nontestable if either an oracle does not exist or the tester must expend some extraordinary amount of time to determine whether or not the output is correct.

The reasonableness of the oracle assumption is examined and the conclusion is reached that in many cases this is not a realistic assumption. The consequences of assuming the availability of an oracle are examined and alternatives investigated.

C.2

1. INTRODUCTION

It is widely accepted that the fundamental limitation of using program testing techniques to determine the correctness of a program is the inability to extrapolate from the correctness of results for a proper subset of the input domain to the program's correctness for all elements of the domain. In particular, for any proper subset of the domain there are infinitely many programs which produce the correct output on those elements, but produce an incorrect output for some other domain element. Nonetheless we routinely test programs to increase our confidence in their correctness, and a great deal of research is currently being devoted to improving the effectiveness of program testing. These efforts fall into three primary categories:

1) The development of a sound theoretical basis for testing,

2) Devising and improving testing methodologies, particularly mechanizable ones,

3) The definition of accurate measures of and criteria for test data thoroughness.

Almost all of the research on software testing therefore focuses on the development and analysis of input data. In particular there is an underlying assumption that once this phase is complete, the remaining tasks are straightforward. This consists of running the program on the selected data, producing output which is then examined to determine the program's correctness on the test data. That this last phase of output examination is indeed routine is frequently expressed as the oracle assumption. This states that the tester is able to

determine whether or not the output produced on the test data is correct. The mechanism which checks this correctness is known as an oracle. (See for example [13] or [5].)

Intuitively, it does not seem unreasonable to require that the tester be able to determine the correct answer in some "reasonable" amount of time while expending some "reasonable" amount of effort. Therefore if either of the following two conditions occur, a program should be considered nontestable.

1) There does not exist an oracle.

2) It is theoretically possible, but practically too difficult to determine the correct output.

The term nontestable is used since if one cannot decide whether or not the output is correct or must expend some extraordinary amount of time to do so, from the point of view of correctness testing, there is nothing to be gained by performing the test.

In Section 2 we examine the reasonableness of the oracle and related assumptions and discuss characteristics of programs for which such assumptions are not valid. Section 3 considers how to test such programs and Section 4 discusses the consequences of accepting the oracle assumption. Section 5 concludes with suggestions for software users and procurers.

2. THE ORACLE ASSUMPTION AND NONTESTABLE PROGRAMS

Although much of the testing literature describes methodologies which are predicated on both the theoretical and practical availability of an oracle, it is our belief that it is

unusual for such an oracle to be pragmatically attainable or even to exist. That is not to say that the tester has no idea what the correct answer is. Frequently the tester is able to state with assurance that a result is incorrect without actually knowing the correct answer. This corresponds to the concept of a decision problem being partially solvable but not recursively solvable, and will be known as a partial oracle. Recall that many well-known problems of recursive function theory, including the halting problem, fall into the partially solvable category. (See [7] or [9] for definitions of these terms and a discussion of these concepts.) Furthermore, it has been shown ([10], [11]) that many properties of programs which are of particular interest in program testing have partially solvable, but not recursively solvable decision problems. Included are whether a statement of a program is semantically reachable, whether a given branch is traversable, and whether a given path can be traversed.

As a simple example of the ability to detect incorrect results without knowing the correct result, consider the calculation of the total assets of a large corporation. $100 is not a plausible result, nor is $1000. $1,000,000 might be considered by the tester a potentially correct result, but a specialist, such as the corporation's comptroller, might have enough information to determine that it is incorrect while $1,100,000 is plausible. It is unlikely that the comptroller can readily determine that $1,134,906.43 is correct, and $1,135,627.85 is incorrect. Thus even an expert may accept incorrect but plausible answers as correct results. The

expertise generally permits the restricting of the range of plausible results so that fewer incorrect results fall into this range.

Even if the tester does not know the correct answer, it is sometimes possible to assign some measure of likelihood to different values. For example, if a program which is to compute the sine function is to be tested, and one of the test data selected is 42°, one could begin by saying that if the output is less than -1 or greater than 1, then there is an error. Thus an answer of 3 is known to be incorrect without the actual value of $\sin 42^\circ$ being known. What one frequently tries to do is repeatedly refine the range of plausible outputs until very few possible answers are acceptable. To continue with the sine example, we know that $\sin 30^\circ = .5000$ and $\sin 45^\circ = .7071$ and that the sine function is strictly increasing on that interval. Thus the plausible range for $\sin 42^\circ$ has been further restricted. Furthermore, since the curve between 30° and 45° is convex upwards, a straight line approximation provides a lower bound of .6657 for $\sin 42^\circ$. This type of analysis has allowed us to say that $.6657 < \sin 42^\circ < .7071$. At this point the tester may not have available any additional information to allow the further restriction of the range of allowable values. Nonetheless, the tester may know that since the sine curve is relatively flat between 30° and 45°, the straight line approximation should be quite good. Therefore it follows that the actual value of $\sin 42^\circ$ is much more likely to fall near the low end of the acceptable range than near the high end. Note that we have not

assumed or established a probability distribution, but nonetheless have a notion of likelihood.

The foregoing examples were intended to contrast the disparity between the assumptions made in the testing research literature, and the situations faced by testing practitioners. A secondary purpose of the example is to demonstrate one technique for restricting the range of possible outputs when the precise answer cannot be readily determined. Other examples and partial solutions are given in later sections.

It is interesting to attempt to identify classes of programs which are nontestable. These include:

1) Programs which were written to determine the answer. If the correct answer were known, there would have been no need to write the program.

2) Programs which produce so much output that it is impractical to verify all of it.

3) Programs for which the tester has a misconception. In such a case, there are two distinct sets of specifications. The tester is comparing the output against a set of specifications which differs from the original problem specifications.

We note that the existence of tester misconceptions argues convincingly for testing groups which are independent from programming groups, and when possible the involvement of several different testers in order to minimize the likelihood of coinciding misconceptions. It also argues for precise specifications and documentation, before implementation begins.

For a program to fall into the second category described

above, it need not produce reams of output. A single output page containing columns of 30 digit numbers may simply be too tedious to check completely. Typically, programs in this class are accepted either by the tester "eyeballing" the output to see that it "looks okay" or by examining in detail portions of the output (particularly portions known to be error-prone) and inferring the correctness of all the output from the correctness of these portions.

A common solution to the problem for programs in both categories 1 and 2 is to restrict attention to "simple" data. This will be discussed in the next section.

Testing programs in the third category presents radically different problems. In the other cases, the tester is aware that there is no oracle available (either because of lack of existence or inaccessibility) and must find approximations to an oracle. In this third case, however, the tester believes there is an oracle, i.e., he believes he knows or can ascertain the correct answers. This implies that if an input is selected which is not processed in accordance with the tester's misconception, but is rather processed correctly, the tester/oracle believes the program contains an error and therefore attempts to debug it. The consequences of this situation are discussed in Section 4.

3. TESTING WITHOUT AN ORACLE

Having argued that many, if not most programs are by our definition nontestable, we are faced with two obvious questions. The first is why do researchers assume the availability of an

oracle? There seem to be two primary reasons. Many of the programs which appear in the testing literature are simple enough to make this a realistic assumption. Furthermore, it simply allows one to proceed. The second and more fundamental question is how does one proceed when it is known that no oracle is available? We are certainly not arguing for the abandonment of testing as a primary means of determining the presence or absence of software errors, and feel strongly that the systematization and improvement of testing techniques is one of the foremost problems in software engineering today.

Perhaps the ideal way to test a program when we do not have an oracle is to write another program based on an independent algorithm and compare the results. This comparison might be done manually, although frequently a comparison program will also be necessary. In particular, if the reason that the program was deemed nontestable was due to the volume or tediousness of the output, it would be just as impractical to compare the results manually as to verify them initially.

In a sense, this new program might be considered an oracle, for if the results of the two programs agree, the tester will consider the original results correct. If the results do not match, the validity of the original results is at least called into question.

The notion of writing multiple independent programs or subroutines to accomplish a single goal has been proposed in other contexts, particularly in the area of fault tolerant computing [1], [4], [8]. The motivation there, however, is

fundamentally different. In the case of fault tolerant systems, alternate programs are frequently run only after it has been determined that the original routine contains an error. In that case a partial oracle must also be assumed. There have also been suggestions of "voting" systems [1] for which the programmer writes multiple versions of routines and a consensus is used to determine the correct output. This is generally only proposed for highly critical software - not for routine cases as discussed above. We, in contrast, are discussing the use of an alternate program or programs to determine whether or not the original program functions correctly on some inputs.

There are two reasons why the use of multiple programs for testing is generally not practical. Obviously such a treatment requires a great deal of overhead. At least two programs must be written, and if the output comparison is to be done automatically three programs are required to produce what one hopes will be a single result. Furthermore, each of these programs must be debugged and tested. That such overhead might be worthwhile in some cases is obvious, but for most applications it is simply unreasonable.

The other primary practical objection is the requirement that the algorithms be independent. One is frequently lucky to have one algorithm to solve a problem. Finding a second algorithm, and showing that the two algorithms are independent, may well be formidable tasks.

A different, and frequently employed course of action is to run the program on "simplified" data for which the correctness of

the results can be accurately and readily determined. The tester then extrapolates from the correctness of the test results on these simple cases to correctness for more complicated cases. In a sense, that is what is always done when testing a program. Short of exhaustive testing, we are always left to deduce the correctness of the program for untested portions of the domain. But in this case we are deliberately omitting test cases even though these cases may have been identified as important. They are not being omitted because it is not expected that they will yield substantial additional information, but rather because they are too difficult, expensive or impossible to check.

For those programs deemed nontestable due to a lack of knowledge in general of the correct answer, there are nonetheless frequently simple cases for which the exact correct result is known. A program to generate base 2 logarithms might be tested only on numbers of the form 2^n. A program to find the largest prime less than some integer n might be tested on small values of n.

In the case of programs which produce excessive amounts of output, testing on simplified data might involve minor modifications of the program. For example, a program intended to generate all permutations of ten elements should output 3,628,800 permutations, surely too many to check or even count manually. The addition of a counter to the program could be used to verify that the correct quantity of output is produced, but not that the output is correct. One might modify the program slightly and make the number of elements to be permuted a program parameter.

If the tester then tests the program on a small input such as 4, the correctness of these results could be readily checked, as there are only 24 such permutations.

Note that this example is of interest for another reason: it is an inputless program. In other words it is a program that is intended to do a single computation. If the proper result is not known in such a case, the only way to test it is to modify it in some way, usually by making some fixed value a parameter, and then testing the more general program on input for which the results are known.

The problem with relying upon results obtained by testing only on simple cases is obvious. Experience tells us that it is frequently the "complicated" cases that are most error-prone. It is common for central cases to work perfectly while boundary cases cause errors. And of course by looking only at simple cases, errors due to overflow conditions, roundoff problems and truncation errors are likely to be missed.

We have now argued that programs are frequently nontestable, in the sense of lacking ready access to an oracle, and suggested two ways of testing such programs. The first of these suggestions, writing multiple independent routines, is frequently discarded as being impractical. The second technique of looking at simplified data is commonly used by testers and is satisfactory for locating certain types of errors but is unsatisfactory for errors which are particularly associated with large or boundary values.

The third alternative is to simply accept plausible results,

but with an awareness that they have not been certified as correct. As in the case of the sine program described in Section 2, a useful technique is to attempt to successively narrow the range of plausible results and even assign a probabilistic measure to potential plausible answers or at least some relative measure of likelihood. The sine example allowed us to demonstrate this technique by employing information known about the sine function's behavior.

One other class of nontestable programs deserves mention. These are programs for which not only an oracle is lacking, but it is not even possible to determine the plausibility of the output. One cannot be expected to have any intuition about the correct value of the one thousandth digit of π. Furthermore there is no acceptable tolerance of error. The result is either right or wrong. Since plausibility may be thought of as an unspecified, yet intuitively understood, level of acceptable error, the tester is faced with a serious dilemma. One way to deal with this problem is to devise an independent algorithm and compare the results. The other method is to apply the technique of testing with simplified data. The limitations associated with these approaches must be borne in mind. For this example, the program could be slightly modified to generate the first n digits of π rather than just the desired one thousandth digit, and then tested with n = 10. Since these values are well-known, and can be easily checked, one might deduce, subject to the limitations discussed earlier, that provided these digits are correct, the desired one is also correct. This too may be considered an

example of acceptance by plausibility.

4. THE CONSEQUENCES OF TESTING WITHOUT AN ORACLE

We now consider the consequences of accepting the oracle assumption. There are two distinct situations which deserve mention and consideration. The first is when an output result is actually correct, but the tester/oracle determines that it is incorrect. This is the less common of the two cases and frequently represents a tester misconception.

There are several possible consequences of such an incorrect "oracle" output. In any case, time is wasted while someone tries to locate the nonexistent error. It may also cost time if it causes a delay in the release of the program while useless debugging is going on. Of course an even more serious problem occurs when the tester or debugger modifies the correct program in order to "fix" it and thereby makes it incorrect.

The other, and more common situation, is when the actual result is incorrect, but the tester/oracle believes it is correct. It is well known that many (if not most) programs which have been tested and validated and released to the field, still contain errors. However, we are discussing a fundamentally different situation. In general whenever nonexhaustive testing has been done, there remains a potential for error. But it is expected that the aspects of the program which have been tested and accepted as correct, actually are correct. At the very least the specific data points on which the program has been run are understood to yield correct results. When this is not the case,

even exhaustive testing does not guarantee freedom from error.

5. CONCLUSIONS

Although much of the testing literature routinely assumes the availability of an oracle, it appears, based on discussions with testing practitioners (i.e., people who work in independent testing groups) that testers are frequently aware that they do not have an oracle available. They recognize that they have at best a good idea of the correct result (i.e., plausibility on a restricted range) and sometimes very little idea what the correct result should be.

It is apparent that the software user community has by and large willingly accepted a caveat emptor attitude. We suggest that the following five items be considered an absolute minimal standard part of documentation:

1) The test data the program was run on.

2) The criteria used to select the test data. For example, were they selected to cause the traversal of each program branch, were they cases that proved troublesome in previous versions of the software, were data selected to test each program function, or were the test cases simply chosen at random?

3) The degree to which the criteria were fulfilled. Were 100% of the branches traversed or 30%?

4) The output produced for each test datum.

5) The reasons why the test results were deemed correct or acceptable.

Although such information does not solve the problem of

nontestable programs, it does at least give the user more information to decide whether or not the program should be accepted as adequately tested, rather than simply accepting the programmer's or tester's assurances that the software is ready for use or distribution.

As the fields of software engineering in general, and program testing in particular develop, it appears likely that increased emphasis will be placed upon the development of criteria for determining the adequacy of test data. Not only will we have to write programs to fulfill specified tasks, we will also have to be able to certify that they work as claimed. This is routinely required of hardware producers.

To develop such criteria, we must be able to state precisely what we have been able to show about the program. One of the currently used criteria for adequacy requires the traversal of each branch of the program [6]. Many people including [2], [3], and [12] have discussed at length why such a criterion is a poor indicator of program test adequacy. It might be argued, however, that its virtue is clear. We are able to state precisely what has been demonstrated; i.e., we are able to make statements such as "all but three of the branches of the program have been traversed", or "96% of the branches have been traversed." But even these are not quite accurate statements of what is known. Implicit in such statements is the assumption that the branches have been traversed and yielded the correct results. But as we have argued, this cannot in general be determined. Hence this and any other such criterion of adequacy suffers from the

fundamental flaws which we have discussed. Therefore, as testing research progresses and testing methodologies continually improve, we see that there are two fundamental barriers which must be faced. The first is the unsolvability results mentioned earlier in this paper and discussed elsewhere in the literature. But these are largely of a theoretical nature. The second barrier, however, is a real, pragmatic problem which must in some sense be faced each time a program is tested. We must ask, and be able to determine, whether or not the results obtained are correct. This, we believe, is the fundamental limitation that testers must face.

ACKNOWLEDGMENTS

I am grateful to Paul Abrahams, Tom Anderson, Tom Ostrand, and Sandi Rapps for their comments and helpful suggestions. Thanks also to Marilyn Grossman for her careful and cheerful typing of the paper.

REFERENCES

1. Avizienis, A. and L. Chen. On the Implementation of N-Version Programming for Software Fault-Tolerance During Program Execution, Proceedings of COMPSAC Conference, 1977, 149-155.

2. DeMillo, R.A., R.J. Lipton, and F.G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer, Computer, Vol. 11, No. 4, April 1978, 34-41.

3. Goodenough, J.B. and S.L. Gerhart. Toward a Theory of Testing: Data Selection Criteria, in Current Trends in Programming Methodology Vol. 2, ed. R.T. Yeh, Prentice-Hall, 1977, 44-79.

4. Horning, J.J., H.C. Lauer, P.M. Melliar-Smith, and B. Randell. A Program Structure for Error Detection and Recovery, in Lecture Notes in Computer Science, Vol. 16, Springer, 1974, 177-193.

5. Howden, W.E. and P. Eichhorst. Proving Properties of Programs from Program Traces, in Tutorial: Software Testing Validation Techniques, eds. E. Miller and W.E. Howden, IEEE Computer Society, 1978, 46-56.

6. Huang, J.C. An Approach to Program Testing, Computing Surveys, Vol. 7, 1975, 113-128.

7. Manna, Z. Mathematical Theory of Computation, McGraw-Hill, 1974.

8. Randell, B. System Structure for Software Fault Tolerance, IEEE Trans. Software Eng., Vol. SE-1, June 1975, 220-232.

9. Rogers, H. Jr. Theory of Recursive Functions and Effective Computability, McGraw-Hill, 1967.

10. Weyuker, E.J. Program Schemas with Semantic Restrictions, Ph.D. Thesis, Dept. Comp. Sci. Tech. Report DCS-TR-60, Rutgers University, New Brunswick, N.J., June 1977.

11. Weyuker, E.J. The Applicability of Program Schema Results to Programs, Int. J. Comput. Inf. Sci., Vol. 8, No.5, Oct 1979, 387-403.

12. Weyuker, E.J. and T.J. Ostrand. Theories of Program Testing and the Application of Revealing Subdomains, IEEE Trans. Software Eng., Vol. SE-6, May 1980.

13. White, L.J. and E.I. Cohen. A Domain Strategy for Computer Program Testing, IEEE Trans. Software Eng., Vol. SE-6, May 1980, pp.247-257.

c.2

NYU
Comp. Sci. Dept.
TR-025
Weyuker
On testing nontestable programs.

c.2

NYU
Comp. Sci. Dept.
TR-025 Weyuker

AUTHOR

On testing nontestable

TITLE

programs.

DATE DUE	BORROWER'S NAME

This book may be kept

FOURTEEN DAYS

A fine will be charged for each day the book is kept overtime.

GAYLORD 142 PRINTED IN U.S.A.