# PSEUDO-ORACLES FOR NON-TESTABLE PROGRAMS

Martin D. Davis[1] and Elaine J. Weyuker
Courant Institute of Mathematical Sciences, New York University

## 1. Introduction

The most commonly used method of validating a program is by testing. The programmer typically runs the program on some test cases, and if and when they run correctly, the program is considered to be correct.

We know that many difficult problems are associated with testing. One such problem is that it is a fundamental part of the testing process to require the ability to infer properties of a program by observing the program's behavior on selected inputs. The most common property that one hopes to infer through testing is correctness. But unless the program is run on the entire input domain, there are infinitely many programs which produce the correct output on the selected inputs, but produce incorrect output for some other element of the domain.

The question inevitably arises: What criteria should be used to select input values to test? A related question is how should test data adequacy be defined? How does one decide what percentage of the testing process has been completed, and when has sufficient testing been done?

What these questions, and in fact most of the questions addressed by the testing research community focus on is the development and analysis of input data. It is generally assumed that this is the difficult, significant part of testing, and the remaining tasks are straightforward.

[1] Work supported under NSF Grant MCS 80-02438.

This assumption was examined in [11,12]. In particular, the question of determining whether or not a program produced correct output on the selected input data was discussed. It is usually assumed (see [4] or [13] for example) that the tester or some mechanism (a so-called oracle) can make this determination in some "reasonable" amount of time while expending some "reasonable" amount of effort. We will call a program for which either no oracle exists or for which one exists in theory, but the practical difficulties are simply too great to make it usable, a non-testable program. It was concluded in [12] that in many cases it is not realistic to assume the existence of an oracle. It was argued that although the tester frequently has some notion of what sort of results are plausible, this is hardly the same thing as being able to determine whether or not the output is actually correct. The following classes of programs were identified as being non-testable according to the definition given:

(1) Programs which were written in order to determine the answer in the first place. There would be no need to write such programs, if the correct answer were known.

(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. This may be thought of as a case in which the tester is comparing the outputs against a specification which is different from the original (given) specification.

In this paper we propose a technique for software testing designed explicitly so that an oracle is not required. Since, as was mentioned earlier, it is relatively rare that a (complete) oracle does exist, we conclude that the idea to be introduced should be valuable as a pragmatic testing method.

## 2. Testing Without an Oracle.

We now address the question: How does one test a program for which no oracle is attainable? Our answer depends on the notion of pseudo-oracle. A pseudo-oracle is an independently produced program intended to fulfill the same specification as the original program. The two programs, which are to be produced in parallel by totally independent programming teams, are run on identical sets of input data, and the results compared. Naturally, the input selected must satisfy some predetermined test data adequacy criterion. If the outputs are the same (or acceptably close in the case of numerical programs) the original program is considered to be validated. If, on the other hand, the outputs of the two programs do not agree, the two programs are examined using standard debugging techniques. The process is repeated until all discrepancies are resolved.

Since our proposal involves the comparison of the outputs of two programs, it must be possible to do this accurately and efficiently. Therefore a monitor which runs the two versions of the programs, and compares their outputs must be produced. The production of such a program is a straightforward task.

An alternative proposal is to produce several pseudo-oracles, and use some kind of consensus algorithm to determine whether or not the original program's output is correct. This has the virtue of indicating whether or not the output of the original program can be considered correct and in addition, indicates to which program debugging effort should be applied.

In order that our proposal be practically useful, it will be essential that the required pseudo-oracle (or pseudo-oracles) be produced relatively quickly and easily. For this reason, we envision the use of a very high level language, such as PROLOG [6], or SETL [10], for implementing the testing program. Programs in these languages can usually be written quite quickly. Such languages can even be used as executable specification languages.

It is true that such a very high level language program might not be considered suitable for a production program due to the inefficiency of the code produced by compilers for such languages. Since, however, pseudo-oracle programs will be in use only during the relatively short testing and debugging phases, such inefficiencies are not too important, especially when balanced against the ease and speed of development of programs in such languages.

Note that although our proposed idea is particularly valuable for non-testable programs, it has some distinctly positive features for use in the testing of any program. The use of a monitor allows the output to be automatically verified. This in turn implies that it is feasible to run the program on much larger quantities of data than would ordinarily be feasible.

It is important to compare our proposal which involves the use of multiple implementations of a specification, with what is commonly known as fault tolerant programming [1,3,9]. We are proposing the use of an additional program or programs to determine whether or not the original program functions correctly on the test data. Pseudo-oracles are envisioned as being of use only during the program's development stage. By contrast, fault tolerant systems use alternate versions of a fully developed and debugged program when it has been determined during execution that the primary version contains an error. Such systems are generally only used for highly critical software. There have also been suggestions for "voting" systems [1] in which multiple versions of routines are produced. During the operation phase of the software, all of the versions are run simultaneously and a consensus is used to determine the correct output. Again this is intended only for highly critical software, not for routine cases, and for the operation phase rather than the testing phase of the software.

Before proceeding to analyze the assumptions under which our proposal could be used as a pragmatically reasonable testing methodology, we reconsider the three classes of non-testable programs previously discussed, from the point of view of the usefulness of our proposed pseudo-oracles in dealing with them. The first class included programs for which the correct output was not known because the programs were written in order to determine the answer. Such programs may be thought of as being the prototypical example of a non-testable program, and hence obviously well suited for our proposal.

The second class we mentioned included programs which produced too much output to make it reasonable to verify its correctness. Since all we have done is swap the examination of the output for the comparison of two sets of output data, at first glance it may be thought that we are no better off than before. However, since we envision the use of a monitor to do this comparison, the tedious task of comparing the outputs of two programs can be completely automated. As mentioned above, this would not only make it feasible to check very large quantities of output, but would also relieve the tester of a boring and tedious task even in the context of rather moderate quantities of data.

The final class consists of programs about which the tester had a misconception. That is, the data which the tester was using as an oracle was in fact not an oracle at all with respect to the given specification. By making the production of the original program and the pseudo-oracle completely independent, it becomes extremely unlikely that the same misconception would be embodied in both. More will be said about this in our discussion of our assumptions.

A recent paper [8] describes a similar (independently proposed) idea. The primary difference is the level of the language in which the pseudo-oracle is written. The proposal in [8] involves the production of two versions of a program in a single language. It is our opinion that this will involve prohibitive costs, and we consider the use of a very high level language as central to the practicality of our proposal. A second difference between the proposals is our insistence that the pseudo-oracle be produced completely independently of the program to be tested. This minimizes the likelihood of propagation of errors and misconceptions.

## 3. Assumptions.

In order that our proposal be useful in practical contexts, certain assumptions must be fulfilled, and we now make these explicit:

I. INDEPENDENCE OF THE PSEUDO-ORACLE.

This assumption is really central to our proposed methodology. The two (or several) programs must be developed completely independently by different programming teams. This is essential in order to eliminate the possibility of the some programmer's misconceptions being inserted into both the original program and the pseudo-oracle.

II. AVAILABILITY OF A CONVENIENT VERY HIGH LEVEL LANGUAGE.

Obviously the team charged with the development of the pseudo-oracle must have available a compiler for a language in which code can be written quickly and easily. The compiler should have substantial debugging features in order to facilitate the development.

III. EXTENSIVE USE ENVISIONED FOR ORIGINAL PROGRAM.

The overhead involved in producing a second program (even in a very high level language) can only be justified if the original program is intended to be run often.

IV. COMPLETE AND PRECISE SPECIFICATION

There must be a complete and precise specification available to both programming teams. This is obviously crucial; it is hardly to be expected that two (or more) programs written to meet some vague incomplete specification will turn out to be equivalent.

V. ADEQUACY OF TEST DATA

This thorny and much discussed issue [2,5,7,14] arises irrespective of whether testing is carried out with respect to an oracle or a pseudo-oracle. We limit ourselves to a few remarks. Naturally any input values deemed to be "critical" by those who produced the specification or by members of the programming teams should be included among the test data. This last possibility could take the form of a challenge. One of the teams could say in effect, "We have reason to believe our program behaves correctly on the following data. Does yours?" In addition, input data should be provided by an independent testing group. The number of input values required will presumably depend in part on the number of bits contained in each output value. Clearly it constitutes far weaker confirmation that two independently written programs both produced the output "yes" on a given input than that they both produced the output 3795714802.

## 4. Conclusions.

We have proposed a novel methodology for software testing involving the parallel productions of two or more programs meeting a given set of specifications. It is our belief that this methodology can be quite useful in testing software that would otherwise have to be regarded as being "non-testable". However it is perhaps worth noting that oracles and pseudo-oracles are not so very different in principle as might at first be thought. The information that a tester uses as constituting an oracle is after all just data produced by a computation, carried out either by hand, or by computer. As such it is subject to human fallibility. Of course, by definition, a genuine oracle can never be wrong. But, in actuality one can rarely be absolutely certain that an "oracle" is a genuine oracle. In the case of disagreement between program and "oracle" on a piece of test data, there is always the possibility that the program is right and the "oracle" is wrong.

## 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. 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.

4. 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, 46-50.

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

6. Kowalski, R. A. Logic for Problem Solving, North Holland, 1979.

7. Myers, G.J. The Art of Software Testing, John Wiley & Sons, New York, 1979.

8. Panzl, D. J. Experience with Automatic Program Testing, Proceedings Trends and Applications 1981, National Bureau of Standards, May 1981.

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

10. Schwartz, J.T. Automatic Data Structure Choice in a Language of Very High Level, CACM, Vol. 18 (1975), pp. 722-728.

11. Weyuker, E.J. The Oracle Assumption of Program Testing, Proc. of the 13th Hawaii International Conf. on System Sciences, Hawaii, Jan. 1980.

12. Weyuker, E. J. On Testing Nontestable Programs, Dept. of Computer Science Report No. 025, New York University, New York, October 1980.

13. White, L. J., E. I. Cohen, and B. Chandrasekaran. A Domain Strategy for Computer Program Testing, Comp. and Info. Sci. Res. Center Tech. Report, The Ohio State University, Columbus, Ohio, Aug. 1978.

14. Woodward, M. R., D. Hedley, and M. A. Hennell. Experience with Path Analysis and Testing Programs, IEEE Trans. Software Eng., Vol. SE-6, May 1980, pp. 247-257.