On Testing Non-testable Programs

Elaine J. Weyuker

Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, New York 10012, USA

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 non-testable 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.

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. None the less 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 adequacy.

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. These consist of running the program on the selected data, producing output which is then examined to determine the program's correctness on the test data. The mechanism which checks this correctness is known as an *oracle*, and the belief that the tester is routinely able to determine whether or not the test output is correct is the *oracle assumption*.[1, 2]

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 *non-testable*: (1) there does not exist an oracle; (2) it is theoretically possible, but practically too difficult to determine the correct output.

The term non-testable is used since, from the point of view of correctness testing, if one cannot decide whether or not the output is correct or must expend some extraordinary amount of time to do so, 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, Section 4 looks at techniques which are particularly applicable to numerical and scientific computations, and Section 5 discusses the consequences of accepting the oracle assumption. Section 6 concludes with suggestions for software users and procurers.

2. THE ORACLE ASSUMPTION AND NON-TESTABLE PROGRAMS

Although much of the testing literature describes methodologies which are predicated on both the theoretical and practical availability of an oracle, in many cases such an oracle is pragmatically unattainable. 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. In such a case we shall speak of a *partial oracle*.

Some years ago, while working as a programmer for a major oil company, the author received a telephone call from the company comptroller's office. As standard practice, the company ran a program at the beginning of each month to determine the company's total assets and liabilities. The program had been running without apparent error for years. The person in the comptroller's office said that the program had 'suddenly gone crazy'—it had stated that the company's assets totalled $300. This is a simple example of the ability to detect incorrect results without knowing the correct result. $300 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 although $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 expert can usually restrict the range of plausible results to exclude more incorrect results than a non-expert.

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

CCC-0010-4620/82/0025-0465 $03.00

one of the test values 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° being known. What one frequently tries to do is repeatedly refine the range of plausible outputs until very few answers are acceptable. To continue with the sine example, we know that $\sin 30° = 0.5000$ and $\sin 45° = 0.7071$ and that the sine function is strictly increasing on that interval. Thus the plausible range for sin 42° has been further restricted. Furthermore, since the curve between 30° and 45° is convex upwards, a straight line approximation provides a lower bound of 0.6657. This type of analysis has allowed us to say that $0.6657 < \sin 42° < 0.7071$. At this point the tester may not have available any additional information to allow the further restriction of the range of allowable values. None the less, 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° 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 none the less have a notion of likelihood.

It is interesting to attempt to identify classes of programs which are non-testable. 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 specifications. The tester is comparing the output against a specification which differs from the original problem specification.

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 two sections. Note that a program need not produce reams of output to fall into the second category described above. 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.

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 that there is an oracle, i.e. he believes he knows or can ascertain the correct answers. This implies that if selected test data are processed correctly by the program rather than in accordance with the tester's misconception, the tester/oracle will believe that the program contains an error and will therefore attempt to debug it. The consequences of this situation are discussed in Section 5.

The existence of tester misconceptions argues convincingly for testing by people who are independent of the programmers, and when possible the involvement of several different testers in order to minimize the likelihood of coinciding misconceptions. It also argues for precise specification and documentation, *before* implementation begins.

3. TESTING WITHOUT AN ORACLE

Since we believe that many programs are by our definition non-testable, 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 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 produce a *pseudo-oracle*, an independently written program intended to fulfill the same specification as the original program. This technique is frequently called *dual coding*, and has been used historically only for highly critical software. The two programs are run on identical sets of input data and the results compared. The comparison might be done manually, although frequently a comparison program will also be necessary. In particular, if the program was deemed non-testable because of 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 pseudo-oracle 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.[3–5] The motivation there, however, is fundamentally different. In the case of fault tolerant systems, an alternative program is 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[3] for which the programmer writes multiple versions of routines and a consensus is used to determine the correct output. Again, this is generally only proposed for highly critical software. We, in contrast, are discussing the use of an alternative program to determine whether or not the original program functions correctly on some inputs.

Of course the use of a pseudo-oracle for testing may not be 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. In order for the use of a pseudo-oracle to be reasonable, Davis and Weyuker[6] note that it is essential that such a program be produced

relatively quickly and easily. For this reason, a very high level language such as SETL[7] might be used for implementing the pseudo-oracle. The advantages of writing in very high level languages are that programs can be produced quickly, they are less likely to be incorrect than programs in traditional languages, and they can be quickly and easily modified. Developers and experienced users of SETL indicate that the ratio of the number of lines of SETL code to the number of lines of PL/I or a similar high level language range from1:4 to 1:10 depending on the nature of the program and primitive data structures used. Similar ratios are predicted for the total time to produce a running version of the program. Of course, even more advantageous ratios can be expected for programs written in an assembly language.

It is true that a very high level language program may not be suitable as a production program owing to the inefficiency of the compiled object code. However, since pseudo-oracle programs will be in use only during the testing and debugging phases, such inefficiencies do not present a serious problem, especially when balanced against the ease and speed of development of the pseudo-oracle.

Gilb[8] argues that writing dual code does not necessarily add substantially to the overall cost of software since writing source code contributes only a minor part of the total cost. In addition, he states that this practice will cut down substantially the amount of work which a human tester/oracle must perform, and therefore offset the increased coding costs. More experimental research should be done to determine the real costs of using multiple versions of programs as a testing technique.

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 infer 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 impossible, too difficult, or too expensive to check.

For those programs deemed non-testable due to a lack of knowledge of the correct answer in general, there are, none the less, 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 police officer stopped by The Courant Institute a few years ago to ask whether we could produce a listing of all possible orders in which the teams of the American Baseball League could finish. Since there were ten teams in the league, such a program would have to generate all 3,628,800 permutations of ten elements, 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 intended to do a single computation. If the proper result is not known in such a case, the program should be parameterized and the more general program tested 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 whereas boundary cases cause errors. And of course by looking only at simple cases, errors due to overflow conditions, round-off problems, and truncation errors may well be missed.

We have now argued that programs are frequently non-testable, 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.

One other class of non-testable 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. Both the use of a pseudo-oracle and the use of simplified data might be useful. 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 serious limitations discussed earlier, that provided these digits are correct, the desired one is also correct.

4. NUMERICAL AND SCIENTIFIC COMPUTATIONS

Although we believe that non-testable programs occur in

all areas of data processing, the problem is undoubtedly most acute in the area of numerical computations, particularly when floating point arithmetic is used. For this reason, numerical analysts and scientific programmers have developed some additional techniques to deal with these problems. Note that we will not consider the question of techniques to minimize the error in a computation. Although this is an important and related area, we are concerned only with the question of determining whether or not the computed results are correct.

One should bear in mind when performing such computations that there are three distinct sources of error: the mathematical model used to describe the problem, the program written to implement the computation, and features of the environment such as round-off error. In general, techniques developed to assess one of these aspects assume that the other two are error-free.

Another point to keep in mind is that there is rarely a single correct answer in these types of computations. Rather, the goal is generally an approximation which is within a designated tolerance of the exact solution. Sometimes knowing that the result falls within the specified tolerance in all but a small percentage of the cases is sufficient.

The first rule-of-thumb used is obvious: test the program on data for which the correct solution is known. Frequently one has analytic or empirical solutions for at least a few points in the domain. Even if these are the simplest cases, they do permit one to determine whether the program 'works at all.'

Although the value of dual code or redundancy of at least certain computations is well recognized, it seems generally to be discarded as being too costly. However, a commonly used technique is related to this and the rule-of-thumb cited above: run the program on a set of 'standard problems' which have been used to test and compare programs intended to perform the same task. In essence, the results are being compared to those of independently produced software without the overhead of having to produce the alternative versions.

An important and useful technique involves the use of properties of the computation which are known from the theory. The theory might tell us that some conservation laws hold, or that certain properties should remain invariant, or give an indication of an error bound. These should be checked repeatedly throughout the computation. If such a property fails to hold at some point, we know immediately that the results will not be correct. We know when calculating values of the sine and cosine, for example, that $\sin^2(x) + \cos^2(x) = 1$. Care should be taken to see that this check is really being performed at each stage. If the cosine is calculated by finding the square root of $(1 - \sin^2(x))$, checking for this invariant cannot be expected to be revealing.

A more sophisticated example involves the use of Newton's method to find the zeros of a function f. This iterative method uses local linearization to find each successive iterate: $x_{k+1} = x_k - (f(x_k)/f'(x_k))$ provided $f'(x_k) \neq 0$.

To test a program which uses Newton's method, we first note that we need routines to compute $f(x)$ and $f'(x)$. Considering these routines as black boxes, they should at this stage be checked for consistency. That is, is f' really the derivative of f? Using the Taylor series expansion we know that

$$f(x + e) = f(x) + ef'(x) + O(e^2)$$

By computing a table which includes the value of $f(x + e) - (f(x) + ef'(x))$ for such values of e as 1, 0.1, 0.01, 0.001, etc. one can see at a glance whether f' could be the derivative of f. Once it has been determined that these routines are consistent, it remains to check the code intended to implement Newton's method.

The rate of convergence can frequently be used to detect errors in iterative methods. In discussing the use of multi-level adaptive techniques to solve partial differential equations, for example, Brandt[9] states: 'Because of the iterative character of the method, programming (or conceptual) errors often do not manifest themselves in catastrophic results, but rather in considerably slower convergence rates.'

The theory tells us that Newton's method is a quadratically convergent method; i.e. each successive iteration approximately squares the error. Thus we can expect that the number of correct digits in the computation should be roughly doubled at each iteration. That this is actually happening should be checked.

One must always be careful to check, however, that the theory is really applicable. Rice and Rice[10] give the example of the use of an algorithm to estimate the value of π. The theory says that even estimates are smaller than π and odd estimates are greater than π. Using the algorithm it is determined that the 10000th estimate is 3.14134410 and the 10001st estimate is 3.14154407, so $3.14134410 < \pi < 3.14154407$. Use of the 49999th and 50000th estimates gives $3.14074582 < \pi < 3.14078581$. This is obviously a contradiction. The point is that the theory assumes perfect arithmetic, whereas computer arithmetic involves round-off errors which may render the theory inapplicable.

The determination of the effects of finite precision of computer arithmetic on the accuracy of results is a fundamental problem in numerical calculations. This problem of significance arises, for example, when automatic normalizing floating point arithmetic is used. In recognition of this important problem, Goldstein and Hoffberg[11] modified the CDC 6600 Fortran compiler to associate with each floating point number an extra word which indicated the number of significant bits. This was determined by keeping track of the number of shifts necessary to normalize the number following an operation. A similar tool, SigPac, was developed by Bright and Coles.[12] A program run under SigPac will produce its regular numerical output as well as a bound on the number of valid digits for selected quantities. This permits the tester to determine whether or not there has potentially been a degradation of significance in the course of the calculation.

Essentially, these techniques address the question: 'What is the worst error that can occur?' Like all worst case analysis, such techniques tend to be highly pessimistic and are most useful for telling the tester that there is no problem, not for indicating that a problem really does exist.

In contrast to this type of analysis is the notion of *backwards error analysis*. Essentially the question asked in this type of analysis is not 'How close is the computed solution to the actual solution?' but rather the reverse: 'For what problem of the same type is this the exact

solution?' This is a particularly interesting idea since it does not require that the correct solution be known. In addition, numerical values in the statement of the problem are frequently not exact—they may be the result of empirical observations or previous inexact computations. Thus, as long as we have computed the exact solution of a 'relatively close' problem, we may be perfectly satisfied. A large backward error, i.e. a large difference between the correct input data and the input data for which the computed solution is the exact solution, may indicate that the algorithm used was not acceptable. Rice[13] gives the following example:

Solving $f(x) = 2.23x^2\cos(x/4) + 2.41x^4 - 5.06e^{-6.4x} + 0.832 = 0$, assume the result $x = 0.256$ was obtained. Substituting this into f we get $f(0.256) = 0.005086$. Thus 0.256 is the exact solution to

$$2.23x^2\cos(x/4) + 2.41x^4 - 5.06e^{-6.4x} + 0.82691 = 0$$

Since the data have been changed by more than the implied accuracy of three digits in the problem's original coefficients, we would probably not accept this solution.

Another way of proceeding to check a computation for significance when tools such as SigPac are not available is to redo the computation in some different but theoretically equivalent way, and compare the results. For example, although $(1-x^2)$ and $(1-x)(1+x)$ and $((0.5-x)+0.5)(1+x)$ are all mathematically equivalent, they may not yield the same results. If it turns out that they agree in some, but not all of the digits, an estimate of the number of correct digits has been provided. Frequently it is sufficient to simply perturb the order in which the operations are performed in order to check whether significance has been lost. For example, rearranging the order in which three variables x, y, and z are added may be sufficient to produce different results, particularly when the values of some of the variables are much greater than the final result. A very simple way to perform this perturbation is to run a given high-level language program using different compilers. Since each compiler has its own way of translating the high-level language code into machine language, it is likely that this is sufficient to expose problems and give an estimate of the number of correct digits.

Rice and Rice[10] suggest running a given (high-level language) program on machines with different amounts of precision. They present an example in which a computation was performed using 4, 8, and 16 digit precision. For a particular value of the argument, the 4 digit and 8 digit results were completely different, whereas the 8 and 16 digit results agreed. Thus they conclude 'we feel justified in assuming these several digits are correct.'

Occasionally, knowledge of which errors are most commonly made can be helpful. Hartree[14] mentions, for example, that the interchange of adjacent digits is an easy error to make. Since the result of such an error is always a multiple of 9 and the difference between the interchanged digits, this fact coupled with the knowledge that this is a likely error may be useful in locating and identifying a mistake of this type.

A final technique which is sometimes useful in detecting a computational error in a table of results is difference checks. The idea is that the effect of a difference of ξ in one element of the table will be magnified by computing successive differences, and thereby permit the tester to detect an error. Dahlquist and Bjorck[15] include the following exercise:

Locate and correct the misprint in the following table of the values of a 'well-behaved' function:

0852 2251 3651 5045 6458 7864 9272

Computing the differences we get the following table:

0852			
	1399		
2251		+1	
	1400		−7
3651		−6	
	1394		+25
5045		+19	
	1413		−26
6458		−7	
	1406		+9
7864		+2	
	1408		
9272			

Since a difference of ξ will be reflected in the third differences by $+\xi$, -3ξ, $+3\xi$, $-\xi$, it follows that ξ is approximately $-26/3 \approx -9$, and hence the fourth entry in the table should have been 5054 rather than 5045.

5. THE CONSEQUENCES OF TESTING WITHOUT AN ORACLE

We now consider the consequences of accepting the oracle assumption. Two distinct situations 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 non-existent 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 non-exhaustive 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.

6. 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 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?
(2) The degree to which the criteria were fulfilled. Were 100% of the branches traversed or 30%?
(3) The test data the program was run on.
(4) The output produced for each test datum.
(5) How the test results were determined to be correct or acceptable.

Although such information does not solve the problem of non-testable programs, it does at least give the user more information to use in deciding 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. Such certification is routinely required of hardware producers.

To develop adequacy 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.[16] Many people including Goodenough and Gerhart[17] and Weyuker and Ostrand[18] have discussed at length why this 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 involves unsolvability results,[19,20] 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.

Acknowledgements

I am grateful to Paul Abrahams, Tom Anderson, Tom Ostrand and Sandi Rapps for their comments and helpful suggestions. I am also grateful to my colleagues at the Courant Institute for their interesting and helpful discussions of how to deal with these problems in numerical and scientific computations. In particular I thank Octavio Betancourt, Martin Davis, Max Goldstein, Mal Kalos, Peter Lax, Michael Overton, Charlie Peskin and Michael Weinstein.

REFERENCES

1. W. E. Howden and P. Eichhorst, Proving properties of programs from program traces, in *Tutorial: Software Testing and Validation Techniques*, ed. by E. Miller and W. E. Howden, pp. 46–56. IEEE Computer Society, New York (1978).
2. L. J. White and E. I. Cohen, A domain strategy for computer program testing. *IEEE Transactions on Software Engineering* **SE-6**, 247–257 (1980).
3. A. Avizienis and L. Chen, On the implementation of N-version programming for software fault-tolerance during program execution. *Proceedings of COMPSAC Conference*, 149–155 (1977).
4. J. J. Horning, H. C. Lauer, P. M. Melliar-Smith and B. Randell, A program structure for error detection and recovery, in *Lecture Notes in Computer Science* **16**, pp. 177–193. Springer, Berlin (1974).
5. B. Randell, System structure for software fault tolerance. *IEEE Transactions on Software Engineering* **SE-1**, 220–232 (1975).
6. M. Davis and E. Weyuker, Pseudo-oracles for non-testable programs. *Proceedings of ACM 81 Conference* (1981).
7. R. B. K. Dewar, A. Grand, S.-C. Liu and J. T. Schwartz, Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM TOPLAS* **1** (1), 27–49 (1979).
8. T. Gilb, *Software Metrics*, Winthrop, Englewood Cliffs, New Jersey (1977).
9. A. Brandt, Multi-level adaptive techniques for partial differential equations: ideas and software, in *Mathematical Software III*, ed. by J. R. Rice, pp. 277–318. Academic Press, London (1977).
10. J. K. Rice and J. R. Rice, *Introduction to Computer Science*, Holt Rinehart Winston, New York (1969).
11. M. Goldstein and S. Hoffberg, The estimation of significance, in *Mathematical Software*, ed. by J. R. Rice, pp. 93–104. Academic Press, New York (1971).
12. H. S. Bright and I. J. Coles, A method of testing programs for data sensitivity, in *Program Test Methods*, ed. by W. C. Hetzel, pp. 143–162. Prentice-Hall, Englewood Cliffs, New Jersey (1973).
13. J. R. Rice, *Matrix Computations and Mathematical Software*, McGraw-Hill, New York (1981).
14. D. R. Hartree, *Numerical Analysis*, 2nd Edn, Oxford (1958).
15. G. Dahlquist and A. Bjorck, *Numerical Methods*, Prentice-Hall, Englewood Cliffs, New Jersey (1974).
16. J. C. Huang, An approach to program testing. *Computing Surveys* **7**, 113–128 (1975).
17. J. B. Goodenough and S. L. Gerhart, Toward a theory of testing: data selection criteria, in *Current Trends in Programming Methodology* Vol. 2, ed. by R. T. Yeh, pp. 44–79. Prentice-Hall, Englewood Cliffs, New Jersey (1977).
18. E. J. Weyuker and T. J. Ostrand, Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering* **SE-6**, 236–246 (1980).
19. W. E. Howden, Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering* **SE-2**, 208–215 (1976).
20. E. J. Weyuker, The applicability of program schema results to programs. *International Journal of Computer & Information Sciences* **8** (5), 387–403 (1980).

Received November 1981