Assessing Test Data Adequacy through Program Inference

ELAINE J. WEYUKER

Courant Institute of Mathematical Sciences

Despite the almost universal reliance on testing as the means of locating software errors and its long history of use, few criteria have been proposed for deciding when software has been thoroughly tested. As a basis for the development of usable notions of test data adequacy, an abstract definition is proposed and examined, and approximations to this definition are considered.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging. F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; I.2.2 [Artificial Intelligence]: Automatic Programming—program synthesis; I.2.6 [Artificial Intelligence]: Learning—induction

General Terms: Reliability, Verification

Additional Key Words and Phrases: Program testing, software testing, test data adequacy, program inference, inductive inference

1. INTRODUCTION

The testing phase of the software development cycle attempts to expose the presence of as many errors as possible in a program and ultimately provide the developer and user with a belief that the program is likely to be correct. The ideal goal is to guarantee correctness, but in all except very simple cases this is impossible to accomplish through testing on a finite set of data [13, 23]. It is common for commercially produced programs, which have apparently been thoroughly tested, to exhibit incorrect behavior long after they have been released and used.

Testing can be viewed as an inference process in the course of which the tester attempts to deduce properties of a program by observing its behavior on selected inputs. When the property one desires to infer is correctness, the inputs are usually selected to cause the program to exhibit all potential aspects of its behavior or to cover all facets of the specification. If the selected inputs are processed correctly, one then infers that the program will correctly process its entire input domain.

© 1983 ACM 0164-0925/83/1000-0641 \$00.75

This research was supported in part by the National Science Foundation under grant MCS-82-01167. Author's address: Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

ACM Transactions on Programming Languages and Systems, Vol. 5, No. 4, October 1983, Pages 641-655.

Two key problems of program testing are

- (1) given a program and specification, how to select data which test the program most effectively;
- (2) given a program, specification, and test data which are processed correctly, how to determine whether or not the testing has been sufficient to justify a claim that the program has been adequately tested.

A good survey of theoretical work on the first problem can be found in [8]. The purpose of this paper is to consider the second problem. Myers states:

The completion criteria typically used in practice are both meaningless and counterproductive. The two most common criteria are

- 1. Stop when the scheduled time for testing expires.
- 2. Stop when all the test cases execute without detecting errors. [16, p. 122]

In this paper we examine other proposals for criteria for test data adequacy and discuss problems associated with their use as practical guides to whether or not testing is complete. In the spirit of [7], we first propose an ideal criterion for adequacy and discuss its practical limitations. We then consider approximations to this adequacy criterion.

It is important to consider what the relationship should be between adequacy and test data selection criteria. One might argue that every test data selection criterion is automatically an adequacy criterion, for we could simply say that the program has been adequately tested if and only if the given selection criterion has been satisfied. However, most currently proposed adequacy and test data selection criteria represent conditions which are necessary for a program to be tested completely, but certainly are not sufficient. Usually they are not even "nearly sufficient," as it tends to be easy to construct programs containing errors which nonetheless fulfill each of these criteria. In spite of this, it is not uncommon for an adequacy criterion to be defined in terms of a test data selection criterion. For example, Huang [14] advocates the selection of data to traverse each branch of a flow graph; once this, or a reasonable approximation to it, is accomplished, the program is considered adequately tested.

Owing to the crudeness of presently available adequacy notions, we believe that test case selection should not be directed toward the fulfillment of the criterion used as the adequacy notion. Rather, test data should be generated by some appropriate independent method, and only when the tester feels that the program has been tested on sufficient data should an (independent) adequacy criterion be invoked. If the adequacy criterion is fulfilled, then the program is deemed thoroughly tested; otherwise, additional test data must be generated and the process repeated. Note that even at this stage, test data selection should not be driven by adequacy criteria. In short, we should test to locate errors, not to fulfill some (imperfect) criterion.

Another important question to consider is the desired relationship between the correctness of the program being tested and the adequacy of the test data. Should the adequate testing of a program guarantee its correctness? We argue that even though in the ideal case testing stops only after all the errors have been located and removed, in practice such a requirement of correctness is not realistic.

On the other hand, it is clear that the correctness of a program should not automatically guarantee that an arbitrary test set is adequate. One of the weaknesses of the notion of a reliable and valid criterion as defined in [7] is that for a correct program any criterion is reliable and valid, and hence any set of test data is ideal. It is crucial that the properties which determine the adequacy of a set of test data depend on the quality of the test data rather than rely solely on the qualities of the program.

With these concerns in mind, we introduce in Section 3 an abstract notion of adequacy which has the property that if a program has been adequately tested, then it is correct, but the correctness of the program does not imply that it has been adequately tested. In Section 4 we compare this notion to other adequacy criteria, and in Section 5 we consider pragmatic approximations to this ideal notion of adequacy. Section 6 demonstrates the use of these notions on some examples.

2. EXISTING NOTIONS OF TEST DATA ADEQUACY

Goodenough and Gerhart [7] define an *ideal set of tests* to have properties that imply that the tests are capable of exposing all errors in a program. Thus, if a program produces correct results on a set of ideal tests, it must be correct. However, these properties are nonconstructive in the sense that they do not tell us how to produce ideal tests for a given program. In addition, it is generally impossible to determine whether a given set of tests for a program is ideal.

Lacking a guaranteed way to create tests that can conclusively demonstrate correctness, software test developers need a method of determining when sufficient testing has been done. Such an *adequacy criterion* for test data should reflect the test's ability to expose errors in the program. Many of the adequacy criteria which have been proposed and are in use today approach this natural goal only indirectly. It is common, for example, to require that every statement, branch, or path fulfilling some condition be traversed in order that test data be deemed adequate [14, 24], even though it has been pointed out [3, 7, 23] that these notions of adequacy suffer from deficiencies. In particular, it is easy to devise simple programs and test data such that, even though the program contains errors, the requirements of each of these criteria are fulfilled. Since the goal of testing is to detect the presence of errors, and these notions of adequacy measure code traversal, it is not too surprising that they are not really satisfactory indicators of how thoroughly the program has been tested. Furthermore, these criteria are themselves untestable in general, in the sense that there can be no algorithm to decide for an arbitrary program whether there exist test data that will cause a given statement, branch, or path to be traversed, or whether every statement, branch, or path can be traversed [21].

Several other criteria for test data adequacy have been proposed and discussed. Error seeding [16] consists of the deliberate implantation of bugs in the program being tested. The buggy version of the program is then run on the set of test data which has been proposed as adequate to see how many of the implanted bugs are exposed. If k percent of the implanted bugs are located, it is then assumed that k percent of the original bugs have been found. This technique assumes that the

types and distribution of bugs which occur unintentionally are the same as those implanted, a convenient, but usually inaccurate, assumption.

Another proposed adequacy criterion is known as the program mutation method [1, 3]. This system makes a series of minor changes to the program being tested, creating a set of programs known as mutations. Some of these modifications cause program errors, while others simply yield equivalent programs. A proposed set of test data is considered adequate if it causes every inequivalent mutation to give an incorrect answer on some input in the set. What the authors have done is to define implicitly what they consider to be the class of most likely simple errors. By showing that these errors do not occur, they have not guaranteed the absence of all errors, but rather that the program is either correct or radically incorrect. Since the authors assume that the program being tested was written by a "competent programmer," that is, a person who writes programs which are "close" to being correct, the second alternative can be eliminated. A closely related system was implemented by Hamlet and is described in [9].

All the proposed criteria for test data adequacy discussed above are programbased. That is, they rely solely on the written code of the program being tested. It is now being recognized that program testing techniques should be based on both the specification and the program [6, 7, 16, 17, 22, 23]. It is certainly important to develop test data based on all sources of information, but we believe it is even more important to develop criteria for adequacy which judge the test data's quality by considering all information sources. After all, test data derived from one source may coincidentally reflect characteristics of other sources. But an adequacy criterion is being used to judge the test data's quality, and therefore *must* assess that quality against all sources.

3. AN ABSTRACT NOTION OF ADEQUACY

Goodenough and Gerhart [7] use the concept of an ideal test as the basis of their theory of program testing. The theory which they propose describes characteristics, or sufficient conditions, for a set of tests to be ideal, but does not provide a means of determining whether the conditions are fulfilled.

Informally, we expect a test set to be adequate or to test a program thoroughly relative to a given specification, if the tests cover all aspects of the actual computation performed by the program, as well as the computation intended by the specification. Goodenough and Gerhart suggest that a test set is more likely to be ideal if it takes account of each of the following factors:

- (1) every individual branching condition in the program is represented in the tests;
- (2) every potential termination condition in the program is represented in the tests;
- (3) every variable mentioned in a program decision is partitioned correctly into classes that are "treated the same" by the program;
- (4) every condition relevant to the correct operation of the program that is implied by the specification, knowledge of the program's data structures, or knowledge of the general method being implemented by the program is represented in the tests.

To capture precisely the notion of test set adequacy, we use the concept of *program inference*, the derivation of a program from a sample of its input/output behavior. Program testing and program inference can be thought of as being inverse processes. The testing process begins with a program and specification, and looks for input/output pairs that characterize every aspect of both the intended and actual computations. Program inference starts with a set of input/output pairs and specification, and derives a "simplest" program to fit this given behavior.

In order to infer a program from test data, one would almost certainly need several "central" examples to indicate the general pattern. In contrast, we might well consider it sufficient to test a program on only one or two such central test cases. For both testing and inference, however, boundary points have to be explicitly described. For program inference, it is clearly necessary to identify where each type of computation begins and ends. In the case of testing, we know that these boundary points, and points near them, are particularly error prone and thus must be included in the test set.

For program P, we let P(x) denote the result of P executing on input x. We let I_T denote the program inferred from the set of input/output pairs T, using some fixed (but unspecified) inference procedure.

Several inference systems have been implemented [2, 18-20], but the precise algorithm used to infer the program is not central to our discussion. In the examples of Section 6, we use the system developed by Summers to demonstrate our ideas.

For programs P and Q we write P = Q (P is equivalent to Q) to mean that P(x) = Q(x) for every input x (and hence P(x) is defined if and only if Q(x) is defined).

The specification S for a program P need not be executable; in particular, S may well be written in a natural language. Since people run programs on illegal input data (i.e., values not included in the input domain of S), and, in fact, one might deliberately want to include examples of such data in a test set, we have to be able to talk about S(x) for any input x. For x in the domain of the specification, S(x) is the value which a program intended to fulfill S should produce on input x. For x not in the domain of S, we say that S(x) is undefined. It thus makes sense to extend our notion of equivalence to speak of the equivalence of a program and a specification. In certain situations one is willing to accept as both a necessary and sufficient condition for specification S being fulfilled, that the program P produce the correct result on every element of the input domain given by the specification. In that case it does not matter whether the program produces output on some illegal input, and we consider S(x) = P(x) provided that both are defined and equal, both are undefined, or P(x) is defined and S(x) is undefined. Under different circumstances, one might take the position that not only must the program behave correctly on data in the domain, but also it must not produce "normal" output for inputs outside the domain. (It may, of course, produce an error message indicating illegal input.) In that case we would consider S(x) =P(x) provided either both are defined and equal or both are undefined. We might be willing to similarly weaken our notion of equivalence for two programs.

Although T is a set of input/output pairs, we frequently speak of an input t of T. By this we mean that t is an input value such that there exists a t', with (t, t') a member of T.

A set of input/output pairs T is an *inference adequate test set* for program P intended to fulfill specification S if and only if

(1)
$$I_T \equiv P$$
 and

(2) $I_T \equiv S$.

That is, T is adequate if and only if T contains sufficient data to infer the computations defined by both P and S.

Note that we only check the adequacy of a test set after it fails to expose errors. This is consistent with our view that the role of testing is to expose errors. As long as there is a t in T such that $P(t) \neq S(t)$, there is no question that the test data are doing their job. It is only once P(t) = S(t) for every t in T that we have to determine whether or not it is time to stop testing. Ideally the process ends when the program is correct and the test data are sufficient to determine this.

If a set of test data is to be inference adequate as defined above, then the test data must truly test each portion of the program code as well as the specification. Furthermore, the fact that T is adequate means that I_T is equivalent to both P and S, and hence P is correct. We thus have a definition of test adequacy which implies program correctness but which is not implied by the correctness of P. This is consistent with our position stated in Section 1 regarding the desired relationship between correctness and test data adequacy.

Since the determination of inference adequacy depends on the determination of equivalence, and equivalence is, in general, undecidable, we must be willing to consider approximations in order to make this theoretical adequacy criterion usable. This is discussed in Section 5 and illustrated with examples in Section 6.

A related idea has been recently proposed by Hamlet [11]. His notion of a determining test set is intended to capture the idea of a finite amount of test data being sufficient to distinguish a program from all other programs, both equivalent and inequivalent. Instead of considering only the input/output behavior of a program, Hamlet includes a record of how a computation was performed in his notion of equality. Thus, in order to be determining, there must be enough data to allow any program with an identical computation to be identified, given the computation details for each of the inputs in the test set. In Section 4, we compare a similar restricted form of inference adequacy to branch adequacy.

There are three distinct types of difficulties that need to be considered in connection with a proposed definition of test data adequacy. The first type involves unsolvability problems. Our proposal requires the ability to infer a program from data and to determine whether or not it is equivalent to the original program, even though program equivalence is not, in general, a recursively solvable problem. Note, however, that similar unsolvable problems must be faced, although not always as directly, when using each of the other adequacy proposals discussed above. In Section 5 we consider approximations to our adequacy definition, since the determination of equivalence is so central to the criterion.

The second type of problem concerns usability: Is it reasonable to require the fulfillment of the criterion? In the case of the code traversal measures, for

example, there are frequently too many paths to be able to traverse all of them. With the mutation method, an n line program produces on the order of n^2 mutants, each of which must be distinguished from, or shown to be equivalent to, the original. For a moderate-sized program this can require that far too many programs be run. Our proposed criterion for adequacy encounters a different type of intractability. In particular, the state of the art of program inference systems is currently not well developed. In addition, if we could really show that a program inferred from data were equivalent to the specification, we would not have to write the program to begin with. Why not simply rely on such "automatic programming" systems? Although this is a possible methodology for program production in the future, it does not appear realistic today.

The third point to examine is whether the definition is really appropriate. We have discussed this for other proposed criteria and claim that our definition reflects what is meant intuitively by test data adequacy, since the goal is to completely characterize by test data, all portions of both the intended and actual computations.

In the next section we consider relationships among various adequacy criteria. In particular, we are interested in studying the relative strength of these proposals.

4. RELATIONS AMONG ADEQUACY CRITERIA

One would like to be able to formally compare the inference criterion with the other adequacy criteria discussed earlier. It has been shown [21] that branch adequacy implies statement adequacy. We would like to be able to show that inference adequacy implies branch adequacy. Unfortunately, however, this is not true. Let P be a program containing a branch which is *nontraversable* (i.e., for which there is no input value such that the branch is traversed). The program of Figure 1 is an example. Then no set of test data can be branch adequate for P. Nonetheless, it is certainly possible to infer from some set of test data T a program P' which is equivalent to both S and P. Then T is inference adequate but not branch adequate for P. A similar argument can be made for programs P which have inessential branches. An *inessential branch* is one which is traversable but preceded by a decision which is not necessary.

As a simple example consider the flowchart fragment of Figure 2. Precisely because the decision is *not* necessary for the computation, data can be selected which do not traverse both the T and F branches, yet which sufficiently characterize the computation to enable the inference of a different, but equivalent, program. These data would be inference adequate, but not branch adequate.

But it is not only these "anomalous" cases that prevent the implication from being true. Suppose, for example, we were asked to write a program which performs some actions whenever the input string is of even length. For some reason, the program written and being tested checks explicitly for the cases of length 0, 2, 4, 6, and 8 and then has a loop to test for all other cases. One might say that in this implementation, the loop has been unwound five times. Now assume that a test set including inputs of length 0, 2, and 4 is sufficient to infer the program which performs the action with just a single test (i.e., without the loop unwound). Then the test set would be inference adequate for the given program, but not branch adequate. If instead of requiring the inference of a



program equivalent to P, we required the program P itself to be inferred, this problem would be solved. But that would eliminate from consideration programs containing unreachable code or inessential branches, even though we know that people do include (presumably unintentionally) such code in their programs. A more important problem is that since the goal of most implemented inference systems is to infer the simplest program consistent with the data, such a system would only be usable to test the adequacy of "optimal" programs, and we know that most programs do not represent the simplest possible implementation of a specification.

Clearly, without a precise definition of inference, it is not possible to prove the desired theorems formally. Although the intuition behind our inference adequacy proposal is independent of the particular program inference system used, demonstrating our ideas with examples requires the use of some specific inference system. In Section 6, we present four examples of the use of inference adequacy, using the inference system developed by Summers [20]. This system allows us to make a precise comparison between inference adequacy and branch adequacy.

Summers states that "what the system is to do is to produce the simplest program which satisfies the examples." Unfortunately, he does not formally define the term "simplest"; we shall assume that such a program contains neither nontraversable nor inessential branches, and that if some statements of a program

P are deleted to yield a new program P', then P' is "simpler" than P. We then have the following theorem which relates a restricted form of inference adequacy and branch adequacy.

THEOREM. If P can be inferred from T, then T is branch adequate for P.

PROOF. If P is inferred from T, then P is the simplest program which is consistent with the input/output pairs of T. Assume T is not branch adequate for P. Then there exists a branch b which is not traversed by any t in T. Thus the removal of the decision preceding b leads to a program P' such that P(t) = P'(t) for all t in T and P' is simpler than P. \Box

The next result is an immediate consequence of this theorem:

COROLLARY. Let P be a "simplest" program to fulfill specification S and let T be a set of inference-adequate test data for P relative to S. Then T is branch adequate for P.

It is easier to compare inference adequacy to Goodenough and Gerhart's notion of an ideal test. The following two theorems show that inference adequacy is strictly stronger than idealness.

THEOREM. Let P be a program intended to fulfill specification S. If test set T is inference adequate for P relative to S, then T is an ideal test set for P.

PROOF. Since T is inference adequate for P, it follows that P is correct and hence any test set is ideal for P. \Box

THEOREM. Let P be a program intended to fulfill specification S. There exists a test set T which is ideal for P but is not inference adequate for P relative to S.

PROOF. If P is not correct, then no test set is inference adequate for P relative to S. Hence let P be a correct program. Then any test set, including the empty set, is ideal for P. But clearly the empty set, and many nonempty sets, are not inference adequate for the given program. \Box

It is interesting to compare the philosophy underlying inference adequacy with that of the program mutation method which is outlined in Section 2. A primary difference is that using our definition of adequacy, a test set is always considered to be adequate or inadequate for a given program *relative to* a given specification. In contrast, as indicated previously, the mutation method is a program-based strategy. Still, the basic philosophies are similar. Our definition requires that sufficient test data be generated to distinguish both the computation intended by the specification and the computation actually performed by the program from those produced by *all* nonequivalent programs. The mutation method, in contrast, requires that the test data be sufficient to distinguish the program from only *some* nonequivalent programs, namely, the programs which the authors have deemed most likely to have been written as the result of errors in the original program. In that sense, mutation testing may be thought of as an approximation to our definition of adequacy.

In the next section we consider other ways of approximating inference adequacy more directly, while addressing the practical difficulties.

5. APPROXIMATIONS TO THE INFERENCE CRITERION

The preceding discussion of the general difficulties of using inference adequacy as a pragmatic criterion, coupled with the knowledge that it is even stronger than criteria which we have previously argued are not pragmatically usable, makes it clear that our notion can at best be used as a guide. The next task, therefore, is to consider practically attainable approximations to inference adequacy. There are several ways of proceeding.

The first possibility is to place sufficient restrictions on the programs to be considered so that questions that are unsolvable or intractable in general are possible for programs in the restricted class. For example, inference is feasible in many cases for programs whose behavior may be modeled by a finite-state machine. Inference for such machines can be accomplished by performing checking experiments. In addition, equivalence is decidable for finite-state machines, and hence for such programs. Nevertheless, serious practical limitations and difficulties are associated with such experiments, and there is an extensive discussion of these problems in Hennie's book [12]. Hamlet [10] has discussed these limitations vis à vis testing. We concur with Hamlet's assessment that this direction is not likely to be productive.

A second way to proceed would be to look directly for practical approximations to program inference and equivalence, and consider the relaxation of some of the requirements. One might, for example, remove the requirement that the inferred program be equivalent to *both* the specification and the program being tested. Such a relaxation would eliminate the guarantee that an adequately tested program is correct. If $I_T \equiv P$, we say that T is program-adequate, and if $I_T \equiv S$, T is specification-adequate.

The decision as to which of these two requirements to relax might depend heavily on the type of test data selection criterion used. In general, if a programbased selection criterion were used, then we would be more willing to eliminate the requirement that I_T be shown equivalent to P. Similarly, if a specificationbased selection criterion were used, then the $I_T \equiv S$ requirement might reasonably be eased. In either case we are left with determining at least one equivalence.

The assessment of specification-adequacy can be made easier by producing I_T in a very-high-level language such as SETL [4] or Prolog [15]. Although the general equivalence problem is still undecidable, a major virtue of considering programs in such languages is that the programs look very much like the specifications, and hence as a practical matter it is easier to determine equivalence.

In the case of determining program-adequacy, we can approximate checking for equivalence by the following technique, which is essentially an extension of testing to I_T and has the benefit of suggesting additional tests for the original program if T is not adequate. It may also indicate the type of error present if the program is incorrect.

Suppose we have specification S, program P, and test set T such that P(t) = S(t) for every t in T. I_T is the program inferred from T. To judge whether T is an adequate test set, we generate an additional set of tests R by some means, possibly by random selection. (For an interesting discussion of the effectiveness of random testing, see [5].) We require only that the tests in R be independent of

those in T. If random selection is used to create R, one might well want to augment this set by requiring that certain special or boundary cases be included. The next step is to test P on R.

It is worthwhile considering the implications of the possible outcomes of the additional tests R. If $P(r) \neq S(r)$ for some r in R, then P is not correct and testing must continue. This is illustrated in Example 2. Assuming P(r) = S(r) for every r in R, we now test I_T on R. If $I_T(r) = P(r)$ for every r in R, then T is accepted as an adequate test set for P and S. This may be thought of as approximating the equivalence of I_T and P, and is illustrated in Example 4. If $I_T(r) \neq P(r)$ for some r in R, then I_T is incorrect on some elements of R, since P(r) = S(r) for all r in R. This indicates that we have not tested P sufficiently, since $I_T \neq P$. Thus we must continue testing. In particular, test data should be similar to the elements of R where I_T was incorrect, since that part of the problem's domain was not characterized sufficiently well by the original tests. A new program $I_{T'}$ must then be inferred from the augmented test set T' and the process repeated. Example 3 illustrates this case.

Note that we have suggested here that the inadequacy of a test set relative to the inference adequacy criterion be used as a guide for the generation of additional test data. Although we stated strongly in the introduction that such a strategy is ill advised in the case of most adequacy criteria, we feel that, since inference adequacy so directly describes what is intended when we call a test set "adequate," this type of iterative procedure is reasonable in this case. In addition, we require that a new program $I_{T'}$ be inferred and shown to be equvalent to P and S. This requires the generation of a new test set R' whose elements are independent from those of $T' = T \cup R$.

An interesting and important question to consider is what is a reasonable size for R. Obviously if R contains only one piece of test data, we feel far less assured than if R contains many pieces of test data. We suggest the requirement that $|R| \ge |T|$.

6. EXAMPLES

In this section we demonstrate the application of our adequacy notion using an example drawn from Summers [20] with some simple variations. The system developed by Summers was selected because it is a real, implemented inference system which infers programs in a subset of LISP. The programs being tested are written in PL/I.

Since LISP and PL/I have different input and output format conventions, we require that the same input (up to formatting differences) be given to both programs and that they produce the same output (up to formatting differences). In particular, in our examples, the input to the PL/I program will be a string of letters with no embedded blanks, surrounded by parentheses, whereas the input to the LISP program will be a list of single-letter atoms, separated by blanks. Thus, if the input to the PL/I program is (ABCD), the input to the LISP program is (A B C D). Similar formatting differences hold for the outputs of the programs.

Example 1

S: The program accepts as input a list X of length $n \leq 78$, each of whose elements is a single letter. Parentheses surround the list. The program prints the input and the first half

of X surrounded by parentheses if n is even or the first (n + 1)/2 elements of X surrounded by parentheses if n is odd. For all other inputs, the program's output is undefined.

```
P1: halfeven: procedure options (main);
```

```
dcl instring char (80) varying, outstring char (80) varying;
dcl halflength fixed;
put list('input:');
get list(instring);
put list(instring);
halflength = (length(instring) - 1)/2;
if length(instring) - 2 = (2 * halflength)
then do;
outstring = substr(instring, 2, halflength);
outstring = '(' || outstring || ')';
put list(outstring);
end;
end;
end halfeven;
```

 $T: \{((), ()), ((A), (A)), ((AB), (A)), ((ABC), (AB)), ((ABCD), (AB)), ((ABCDE), (ABC)), ((ABCDEF), (ABC))\}$

The program P1 is shown to be incorrect by the test set. In particular, P1 does not produce the required output on inputs (A), (ABC), and (ABCDE). Thus, no program is inferred.

```
\begin{array}{l} Example \ 2\\ S: \quad \text{As in Example 1}\\ P2: \ P1 \ \text{of Example 1}\\ T: \quad \{((\ ),\ (\ )),\ ((\text{AB},\ (\text{A})),\ ((\text{ABCD}),\ (\text{AB})),\ ((\text{ABCDEF}),\ (\text{ABC}))\}\\ I_T: \quad \text{half}[x] \leftarrow h[x;\ x]\\ \quad h[x;\ y] \leftarrow [\text{atom}[y] \rightarrow \text{nil};\\ \quad T \rightarrow \text{cons}[\text{car}[x];\ h[\text{cdr}[x];\ \text{cddr}[y]]]] \end{array}
```

Since S(t) = P2(t) for all t in T, I_T was inferred. In order to approximate the determination of the equivalence of I_T to S and P, we generate a set of random input test data R. The values $\{7, 1, 6, 8\}$, to be used as lengths of input strings, were generated by using the SETL [4] random number generator, requesting four integers between 1 and 15.

If an input to P2 is not of even length, P2 prints the input and terminates, producing no other output. Thus $P2((A)) \neq S((A))$ and $P2((ABCDEFG)) \neq S((ABCDEFG))$. Some of the randomly generated data indicate, therefore, that the program P2 is incorrect. Note that in addition, $I_T((ABCDEFG))$ and $I_T((A))$ are undefined (car is applied to an atom). This indicates that the test data did not sufficiently characterize the intended computation. Since the program does not agree with the specification on odd length lists, it must be corrected and retested.

Example 3

```
S: As in Example 1
```

```
P3: halfall: procedure options (main);
dcl instring char (80) varying, outstring char (80) varying;
dcl halflength fixed;
put list('input:');
get list (instring);
put list (instring);
```

```
ACM Transactions on Programming Languages and Systems, Vol. 5, No. 4, October 1983.
```

halflength = (length(instring) - 1)/2; outstring = substr(instring, 2, halflength); outstring = '(' || outstring || ')'; put list (outstring); end halfall;

T: {((), ()), ((AB), (A)), ((ABCD), (AB)), ((ABCDEF), (ABC))} I_T : As in Example 2

Using the same set of random data R as in the previous example, we see that S(r) = P3(r) for all r in R, but $I_T((ABCDEFG))$ and $I_T((A))$ are undefined. This indicates that T is not sufficient to adequately test P3 relative to S. Unlike the situation in Example 2, the additional test data of R do not indicate an error in P3. (P3 is in fact equivalent to S.) We note that whereas all the input lists in T were of even length, two of the lists of R were odd, and I_T produced the incorrect output for these inputs. This indicates to us that our test set T must be augmented by some lists of odd length. Our new test set T' is

 $T': \{((), ()), ((A), (A)), ((AB), (A)), ((ABC), (AB)), ((ABCD), (AB)), ((ABCDE), (ABC)), ((ABCDEF), (ABC))\}$

A new program $I_{T'}$ must now be inferred. The system would then infer the program $I_{T'}$:

 $\begin{array}{l} \operatorname{half}[x] \leftarrow h[x;x] \\ h[x;y] \leftarrow [\operatorname{atom}[y] \rightarrow \operatorname{nil}; \\ \operatorname{atom}[\operatorname{cdr}[y]] \rightarrow \operatorname{cons}[\operatorname{car}[x];\operatorname{nil}]; \\ T \rightarrow \operatorname{cons}[\operatorname{car}[x];h[\operatorname{cdr}[x];\operatorname{cddr}[y]]] \end{array}$

We must now generate a new set of random test data R'. Since |T'| = 7, we generate seven random numbers between 1 and 15 to use as our test set to approximate equivalence. The set of numbers generated was $\{15, 9, 14, 5, 6, 4, 13\}$. $P3(r) = S(r) = I_{T'}(r)$ for each r in R'. P3 is thus considered adequately tested by T' relative to the specification.

Example 4

S: As in Example 1 P4: P3 of Example 3 T: {((), ()), ((A), (A)), ((AB), (A)), ((ABC), (AB)), ((ABCD), (AB)), ((ABCDE), (ABC)), ((ABCDEF), (ABC))) I_T: $half[x] \leftarrow h[x;x]$ $h[x;y] \leftarrow [atom[y] \rightarrow nil;$ $atom[cdr[y]] \rightarrow cons[car[x];nil];$ $T \rightarrow cons[car[x];h[cdr[x];cddr[y]]]]$

 $P4(t) = S(t) = I_T(t)$ for all t in T. Since |T| = 7, a set R of 7 random inputs was generated, as in Example 3. $P4(r) = S(r) = I_T(r)$ for every r in R. Thus P4 is considered adequately tested by T relative to the specification.

7. CONCLUSIONS

We have introduced a definition of test data adequacy which requires that the test data be sufficient to infer both the intended and actual computations. We have pointed out pragmatic limitations of this definition and considered plausible approximations to the requirements. In particular, we have considered ways of approximating the determination of equivalence of programs. 654 • Elaine J. Weyuker

Although there exist several implemented inference systems, and we in fact have demonstrated our adequacy criterion using one, it is not clear that such systems will ever be practically available. Therefore, just as we have used testing with random data as an approximation to the (unsolvable) problem of determining equivalence, it would be both interesting and useful to attempt to develop practical approximations to program inference.

ACKNOWLEDGMENTS

I am grateful to Tom Ostrand, Martin Davis, and Dick Hamlet for their many helpful comments and suggestions.

REFERENCES

- 1. ACREE, A.T., DEMILLO, R.A., BUDD, T.J., LIPTON, R.J., AND SAYWARD, F.G. Mutation analysis. Tech. Rep. GIT-ICS-79/08, Georgia Inst. of Technology, Sept. 1979.
- 2. BIERMANN, A.W., AND KRISHNASWAMY, R. Constructing programs from example computations. *IEEE Trans. Softw. Eng. SE-2* (Sept. 1976), 141–153.
- 3. DEMILLO, R.A., LIPTON, R.J., AND SAYWARD, F.G. Hints on test data selection: Help for the practicing programmer. Computer 11, 4 (Apr. 1978), 34-41.
- 4. DEWAR, R.B.K., GRAND, A., LIU, S-C., AND SCHWARTZ, J.T. Programming by refinement, as exemplified by the SETL representation sublanguage. ACM Trans. Program. Lang. Syst. 1, 1 (July 1979), 27-49.
- DURAN, J.W., AND NTAFOS, S. A Report on Random Testing. In Proceedings 5th International Conference on Software Engineering (San Diego, Calif., Mar. 9-12, 1981), ACM, New York, 1981, pp. 179–183.
- 6. GANNON, J., MCMULLIN, P., AND HAMLET, R. Data-abstraction implementation, specification, and testing. ACM Trans. Program. Lang. Syst. 3, 3 (July 1981), 211–223.
- 7. GOODENOUGH, J.B., AND GERHART, S.L. Toward a theory of testing: Data selection criteria. In *Current Trends in Programming Methodology*, vol. 2, R.T. Yeh (Ed.). Prentice-Hall, Englewood Cliffs, N.J., 1977, pp. 44-79.
- 8. GOURLAY, J.S. Theory of testing computer programs. Ph.D. dissertation, Dept. of Computer and Communication Sciences, Univ. of Michigan, Ann Arbor, Mich., 1981.
- 9. HAMLET, R.G. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng. SE-3* (July 1977), 279-290.
- HAMLET, R.G. Critique of reliability theory. In Proceedings IEEE Workshop on Software Testing and Test Documentation (Fort Lauderdale, Fla., Dec. 1978), IEEE, New York, 1978, pp. 57-69.
- 11. HAMLET, R.G. Reliability theory of program testing. Acta Inf. 16 (1981), 31-43.
- 12. HENNIE, F.C. Finite-State Models for Logical Machines. Wiley, New York, 1968.
- HOWDEN, W.E. Reliability of the path analysis testing strategy. IEEE Trans. Softw. Eng. SE-2 (Sept. 1976), 208-215.
- 14. HUANG, J.C. An approach to program testing. Comput. Surv. 7 (ACM), 3 (Sept. 1975), 113-128.
- 15. KOWALSKI, R.A. Logic for Problem Solving. Elsevier North-Holland, New York, 1979.
- 16. MYERS, G.J. The Art of Software Testing. Wiley, New York, 1979.
- RICHARDSON, D.J., AND CLARKE, L.A. A partition analysis method to increase program reliability. In Proceedings 5th International Conference on Software Engineering (San Diego, Calif., Mar. 9-12, 1981), ACM, New York, 1981, pp. 244-253.
- SHAW, D.E., SWARTOUT, W.R., AND GREEN, C.C. Inferring LISP programs from examples. In Proceedings 4th International Joint Conference on Artificial Intelligence (Tbilisi, USSR, 1975), pp. 260-267.
- SIKLOSSY, L. AND SYKES, D.A. Automatic program synthesis from example problems. In Proceedings 4th International Joint Conference on Artificial Intelligence (Tbilisi, USSR, 1975), 268-273.
- 20. SUMMERS, P.D. A methodology for LISP program construction from examples. J. ACM 24, 1 (Jan. 1977), 161-75.

- WEYUKER, E.J. The applicability of program schema results to programs. Int. J. Comput. Inf. Sci. 8, 5 (Nov. 1979), 387-403.
- 22. WEYUKER, E.J. An error-based testing strategy. Tech. Rep. 027, Dept. of Computer Science, Courant Institute of Mathematical Sciences, New York Univ., New York, Jan. 1981.
- 23. WEYUKER, E.J., AND OSTRAND, T.J. Theories of program testing and the application of revealing subdomains. *IEEE Trans. Softw. Eng. SE-6* (May 1980), 236-246.
- 24. WOODWARD, M.R., HEDLEY, D., AND HENNELL, M.A. Experience with path analysis and testing of programs. *IEEE Trans. Softw. Eng. SE-6* (May 1980), 278–286.

Received September 1981; revised March 1982, January 1983; accepted February 1983