Acta Informatica 18, 31–45 (1982)

Acta Informatica

© Springer-Verlag 1982

Two Notions of Correctness and Their Relation to Testing

Timothy A. Budd[1], and Dana Angluin[2]

Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, USA

Summary. We consider two interpretations for what it means for test data to demonstrate correctness. For each interpretation, we examine under what conditions data sufficient to demonstrate correctness exists, and whether it can be automatically detected and/or generated. We establish the relation between these questions and the problem of deciding equivalence of two programs.

1. Introduction

It has been remarked that the theory of program testing "can hardly be said to be even in its infancy" [17]. Formal treatment of such notions as "correctness" and "adequate test data" have rarely been presented. In this paper we attempt to clarify certain questions related to program testing: In particular, what kinds of things can we expect test data to show about a program? What is a reasonable definition of "adequate" test data? What problems will we face in attempting automatic generation or validation of test data?

We will point out that there are in the literature at least two interpretations given for what it means for test data to demonstrate correctness. For either of these we can ask when there exist data that satisfy the criteria, and hence can be used to demonstrate correctness, whether one can recognize such data when presented with it, and whether such data can be automatically generated. Finally, we will relate these questions to the problem of deciding the equivalence of two programs.

This paper presents a theoretical study of the properties of these definitions and does not attempt to make practical recommendations concerning the utility of one definition over the other. The proofs presented are not hard or deep, but are intended to give insight into the concepts being considered.

1 The work of this author was supported in part by the National Science Foundation under grant MCS-8109547

2 Yale University, Department of Computer Science, P.O. Box 2158 Yale Station, New Haven, Ct. 06520

0001-5903/82/0018/0031/$03.00

2. Definitions

We consider the following general setting for our investigatons. $\mathscr{P}$ will denote the set of programs we are interested in. $\mathscr{F}$ will denote the set of functions computed by programs in $\mathscr{P}$. For example, if $\mathscr{P}$ is the set of all FORTRAN programs, then $\mathscr{F}$ is the set of all computable functions. Or, $\mathscr{P}$ might be the set of all straight-line programs with integer constants and the operations of addition and multiplication, in which case $\mathscr{F}$ is the set of all polynomial functions with integer coefficients.

$\mathscr{D}$ will represent the set of possible inputs to programs. In order to simplify the proofs, we will use the formal technique of assuming that inputs are encoded as strings of characters, and that there exists a procedure (perhaps non halting) that lists the elements of $\mathscr{D}$ in some order; that is, the set $\mathscr{D}$ is recursively enumerable. The programs in $\mathscr{P}$ are assumed to be represented by strings of characters from some character set; thus we can talk about programs that have other programs as inputs or outputs. For example, the RATFOR preprocessor [28] has FORTRAN programs as both input and output.

We will assume that all programs eventually halt and produce a result. For example, the programs might output the message "error" when given strings for inputs that are not valid for the program. Many of our results would be quite different without this assumption; notice that the example in which $\mathscr{P}$ is all FORTRAN programs does not satisfy this assumption. In practice this is not a serious restriction. For example, a common solution is to assume an arbitrary bound $b(x)$ such that if P fails to terminate within time $b(x)$ on input x, execution is halted and a special failure value produced (e.g., "computing budget exceeded").

A *test set* T is any finite subset of $\mathscr{D}$. A test set is described by listing the elements of the set. The last point may seem trivial, but it is important. One could imagine, for example, a test set being described by a program which generates elements of set. One would then be forced to ask whether this program halted in all cases. This is a troublesome, and largely unnecessary, complication.

Two programs that compute the same function are said to be *equivalent*. If there exists an algorithm that can take any two programs in $\mathscr{P}$ and decide whether they are equivalent, we say that $\mathscr{P}$ has a *decidable* equivalence problem. If no such algorithm exists, we say that the equivalence problem for $\mathscr{P}$ is *undecidable*. For example, it is well known that the equivalence problem for the set of all FORTRAN programs is undecidable, while that for the set of straight line programs described above is decidable.

Some of the results we present establish a connection between the decidability of the equivalence problem and the problem of automatically generating test data. The equivalence problem has been extensively studied for various classes of programs [23]. In most cases it is known to be undecidable. The undecidability of a problem of course does not mean that one should necessarily abandon all work on it; for example Sayward and Baldwin [2] have presented methods for detecting equivalence of FORTRAN programs in a large fraction of cases of practical interest. However, it is important to understand the theoretical limitations one faces.

3. Correctness

What is meant by the assertion that a program P is "correct"? A conventional reply is that P is correct if it computes the desired function. We will call this our first definition of correctness.

Definition. A program P is *correct* with respect to a function F if P computes F.

We note in passing that in practice one is frequently interested in a rather less ambitious task, that of increasing confidence in the program being tested in some rational way. Unfortunately, there is no commonly accepted formal meaning we can attach to this concept, which prevents the sort of analysis we present here from being applied to this less rigorous process.

Howden [24] has discussed what it means for test data to demonstrate this kind of correctness. A test set is called *reliable* for a program P and a function F provided that P and F agree on T if and only if P computes F. If P doesn't compute F, then the test set is required to contain a "witness" to the fact, i.e., a point x such that $F(x) \neq P(x)$. It is easy to see that a reliable test set exists for any program P and function F; it P computes F then any finite set (such as the empty set) will do, and if P doesn't compute F, then $\{x\}$, where x is any point such that $P(x) \neq F(x)$, is a reliable test set for P and F. This observation is formalized in the following:

Theorem 1 [24]. *In any set of programs, a reliable test set exists for each program and function.*

As Howden observed, the reliable test set guaranteed in this theorem must in general depend on both the program and the function. For example, consider the set of programs to compute polynomials. Let P be a program and F the polynomial it computes. Let T be any finite set of inputs. There exists a program Q to compute a polynomial of higher degree that agrees with F on the test points but disagrees elsewhere. Then although T is reliable for P and F, it is not reliable for P and this new polynomial or for Q and F, although they all agree on T. This example provides us with the following theorem:

Theorem 2. *There exist programming systems in which for any program P and finite test set T there exists a function that agrees with P on the test inputs but disagrees elsewhere.*

Thus in order to test using this definition of correctness one must have explicit knowledge of the correct function F. But what form shall this knowledge take? If it is presented in a formal manner and completely specifies the function then one might as well interpret the specifications as a program itself and hence the question of correctness becomes simply one of equivalence. More likely, in practice information concerning the correct function is almost never sufficiently detailed nor formalized.

Note that most automatic test generation procedures [3, 11, 29] produce a test set which depends only on the given program, and do not consider the function which one intends to compute.

If the set of programs being considered is sufficiently small, then a different approach to correctness presents itself. For example, Howden [26] examines the problem of showing that a program that computes a multinomial is correct, with the implicit assumption that *some* multinomial of the same degree or less is in fact the correct function. Thus we trade knowledge of the specific function to be computed for a broader assumption that at least one program in a set of programs is correct. (Note that we have had to make use of the conventional definition of correctness in the assumption; so in one sense nothing has been gained. Nevertheless in practice it is frequently easier to define a set in which we know there exists at least one correct program than it is to formally define the function to be computed.)

The problem of multinomial testing has also been considered by DeMillo and Lipton [13], who present a probabilistic algorithm. Howden's result is extended to linear difference equations in [32]. In [25] Howden considers the problem of testing simple loop programs which implement Lindenmayer Grammars, where the alternatives are all programs which compute such grammars. This type of testing has also been examined by Hamlet [20].

This form of testing arises quite naturally in mutuation analysis [8, 9, 12] where the "competent programmer hypothesis" states that the given program is likely either correct or very nearly so, hence the goal of testing is to establish correctness relative to only a small set of programs which are similar, but not equivalent, to the original. A similar situation is found in the testing system described by Hamlet [18, 19].

When referring to this approach to correctness, we will denote the set of alternative programs by Φ, (with the implicit assumption that Φ is a subset of $\mathscr{P}$). We will allow Φ to depend on the program P, and will thus speak of Φ as being a *neighborhood* of P. (More specifically, $\Phi(P)$ is a mapping from programs to sets of programs). We may assume without loss of generality that P is itself an element of $\Phi(P)$ for all programs P. (Hamlet [20] calls Φ the *alternative set*).

The advantage of this approach is that one can demontrate a program is correct by a process of elimination, by showing that no inequivalent alternative program in Φ is correct. This can be done by generating a test set for which *all* nonequivalent members of Φ fail on at least one point.

Definition. A test set T is *adequate for P (relative to Φ)* if for each program Q in Φ, either Q is equivalent to P or Q fails on at least one test point.

The notion of restricting the set of programs can also be viewed as limiting the types of errors to be considered. Many proposed testing methods are designed specifically to test for a limited class of errors [3, 7, 9, 14, 19, 21, 26, 30, 33]. The advantage is, or course, that viewing testing in this light gives one a tractable problem which avoids the difficulty of formally defining the correct function F.

If we let $\Phi(P)$ be the set of all programs, then assuming the problem to be solved is computable the two notions of correctness are equivalent. More specifically, the following theorem shows the relationship between an adequate test set and correctness.

Theorem 3. *Given a program P and neighborhood $\Phi(P)$, if some program in $\Phi(P)$ computes the correct function F and T is adequate for P (relative to Φ) and P agrees with F on T, then P is correct.*

Proof. Let Q be any program in $\Phi(P)$ which computes F. Since Q computes the correct answers for each test point in T, Q therefore agrees with F on T. Therefore P must agree with Q on T. But by definition P is therefore equivalent to Q. Thus P must compute F. □

This theorem shows that if one is given a program P and a test set T, in order to demonstrate that P is correct one need only show that (a) T is adequate for P (relative to Φ), (b) Some program Q in $\Phi(P)$ is correct, and (c) P computes the correct values on T. Hence except for the implicit reference to the correct function in (b), the only knowledge needed of the correct function are the values on the finite number of test inputs.

Unfortunately, we can show that the existence of an adequate test set may depend upon a careful choice of the neighborhood $\Phi(P)$.

Theorem 4. *There are program sets and neighborhoods for which no finite test set is adequate.*

Proof. The set of polynomials described earlier is one such example, when $\Phi(P) = \{\text{all polynomials}\}$. Notice, however, that if we let $\Phi(P)$ be the set of programs which compute polynomials of the same degree as P or less, then an adequate set does exist.

As another example, let $\Phi(P)$ be the set containing all finite automata. For any automaton P and test set T we can find an automaton Q which agrees with P on T but nowhere else. On the other hand if we let $\Phi(P)$ be finite automata with the same number of states as P, then there is a constructive way to generate an adequate set [10]. □

So if we take our definition of correctness to be the first alternative, a test set sufficient to establish correctness always exists, although we may not be able to find it. The second definition of correctness is in practice easier to use, but an adequate set may or may not exist, depending on the choice of Φ. What follows will investigate further implications of these two different viewpoints.

4. Adequate Test Data, First Definition

Using the first definition of correctness the goal is to demonstrate that a given program P computes a given function F. This definition is commonly used in verification, where the correct function is given as a set of specifications in some formal language [22]. The use of test data as a tool in verification has been considered by Geller [15]. Similarly Goodenough and Gerhart [16] define the correct function in terms of an output predicate $\mathrm{OUT}(x, P(x))$ which is TRUE if $P(x) = F(x)$.

If T is a test set, P a program and F a function, then in this section we say T is *adequate* for P and F provided P agrees with F on T if and only if P computes the function F. Notice that for a given P and F there may be many different adequate test sets. (As noted previously, Howden [24] calls such data *reliable*. We will use our terminology in order to contrast the two concepts of adequate formed using the two different notions of correctness).

We will make the following definitions:

Definition. A procedure $A(P, F, T)$ is an *acceptor* if it returns "true" if T is adequate for P and F, "false" otherwise.

Definition. A procedure $G(P, F)$ is a *generator* if it produces a test set T which is adequate for P and F.

Notice that there is only one acceptor function, whereas there may be many generator functions. That is, although there may be many acceptor procedures they must all produce the same answer given the same input. On the other hand, two generator procedures can produce different test sets even if presented with the same programs as input.

Howden uses the term *reliable test strategy* [24] for what is here called a generator. Our term is intended to focus more closely on the computability of such objects.

In order to discuss the computability of generator or acceptor procedures we must specify how the arguments are to be presented. Clearly P may be presented as the string of characters that represent it, and likewise T. The function F we shall imagine is presented in the form of a "black box", or oracle. That is, the program attempting to compute G (or A) may, as one of its instructions, submit the pair (x, y) to the black box and receive the reply either "true" (if $F(x)=y$) or "false" (if $F(x)\neq y$). These queries are its sole source of information about F (apart from the general fact that F is a member of $\mathscr{F}$.) (This intuitive description is made precise in the notion of "relative" or "oracle" computations in computability theory [31].)

Theorem 5. *If an acceptor procedure is computable, then a generator procedure is computable.*

Proof. Let T_i be the test set consisting of the first i elements of $\mathscr{D}$. By Theorem 1 some test set is adequate, and by definition any superset of this set will also be adequate. Hence T_i must be adequate for some value i. Suppose the acceptor function is computable. Then given P and a black box for F, the generator can successively simulate the acceptor on the inputs P, F and T_i for $i = 1, 2, \ldots$ until it returns "true", and then output T_i for the last value i. □

Theorem 6. *If a generator is computable, then the acceptor is computable.*

Proof. Assume there exists a generator program. Given a program P and a black box for F the acceptor program first simulates the generator to obtain another set T', independent of T, which is by definition adequate. If F and P agree on T' then they are equivalent and any set is adequate, so in particular T is adequate. On the other hand if F and P disagree on T', then the set T is adequate if and only if F and P also disagree on some member of T. □

Theorem 6 illustrates a serious problem with the first definition of adequate, in that it fails to capture the notion of the test set T *showing* the program is correct. Inasmuch as it avoids this problem the second definition of adequate is somewhat more satisfactory.

While Theorem 1 states that an adequate set must always exist, this does not imply that one can either recognize or generate such a set.

Theorem 7. *There are sets $\mathscr{P}$ for which there exists neither a computable acceptor nor a computable generator.*

Proof. Let $\mathscr{P}$ consist of the following programs: P_0 will be a program which computes the constant function 0. Let P_i for $i>0$ be a program for the function that is 0 except on input i, where it is 1. The equivalence problem for $\mathscr{P}$ is decidable, since P_i is equivalent to P_j if and only if $i=j$.

However, suppose there existed a computable generator for this set. We could simulate the generator program on inputs P_0 and the function F which is the constant function zero. The generator may query the black box only a finite number of times before it halts and outputs some test set T. Let i be some integer greater than all the values in the test set and also greater than all values which were queried. Then the generator when given the input P_0 and a black box which computes P_i will again output T, since P_i and the zero function agree on all the values queried. But T is clearly not adequate for P_0 and the function F computed by P_i, contrary to the assumption that a generator procedure is computable. Hence no generator, nor by Theorem 5 the acceptor procedure, is computable for this set. □

By noting that the equivalence problem for this example is decidable, we have the following theorem.

Theorem 8. *$\mathscr{P}$ may have a decidable equivalence problem and still have no computable generator or acceptor.*

As corollary to a later theorem we will show that if $\mathscr{P}$ is finite then a decidable equivalence problem does imply the existence of both an acceptor and a generator. Although for infinite $\mathscr{P}$ a decidable equivalence problem does not imply the existence of a generator, the following theorem shows that the reverse does hold.

Theorem 9. *If a generator procedure is computable then the equivalence problem for $\mathscr{P}$ is decidable.*

Proof. Suppose there exists a generator procedure. Then given programs P and Q in $\mathscr{P}$ the equivalence checker can simulate the generator with inputs P and F, where F is the function computed by Q. The output will be a test set which is adequate for P and F. Thus to test whether P is equivalent to Q, one need only test whether $P(x)=Q(x)$ for all x in this set. □

4.1. First Definition of Adequate, Nonuniform Case

Whereas in Sect. 3 the definition of correctness permitted some knowledge of the correct function F, in the last section the only information actually used

was the value of F on a finite number of inputs. In particular the acceptor and generator procedures were required to be uniform over all possible functions. It is possible, however, that we will have further information about the correct function available to us. For example, Howden [27] describes a system of *testing using certain functional characteristics.*

The difficulty is in trying to formally define what it means for a program to have "knowledge" of the correct function. The last section analyzed one extreme, where the knowledge was reduced to just the value of the function on a finite number of inputs. This section will investigate the case where we allow total knowledge of the function, producing a different acceptor or generator for each function (in a sense "building in" full knowledge of the correct function).

Definition. A procedure $A_F(P, T)$ is an *acceptor for F* if it returns "true" if T is adequate for P and F, "false" otherwise.

Definition. A procedure $G_F(P)$ is a *generator for F* if it produces a test set T which is adequate for P and F.

We will note that these specialized functions may exist in cases where the uniform functions do not; in particular acceptors exist for each of the functions constructed in the example cited in Theorem 7.

Theorems similar to 5 and 6 are true, and are easily proved using a built in procedure for F in place of the black box.

In contrast to Theorem 8, in the nonuniform case a decidable equivalence property is sufficient to establish the existence of a computable acceptor and/or generator. Notice, however, that the following theorem only proves the existence of such a program; it does not show how to effectively construct it.

Theorem 10. *If $\mathscr{P}$ has a decidable equivalence problem then, for every F in $\mathscr{F}$, A_F is computable.*

Proof. Suppose $\mathscr{P}$ has a decidable equivalence problem. Given a function F, let P_0 be a program to compute F. (By definition such a program must exist, however one need not be able to effectively find it.) Then A_F can be computed as follows, if the input to A_F is P and the test set T, then run the equivalence decision procedure to decide whether P_0 and P are equivalent. If the two programs are equivalent, then output "true". Otherwise, check whether $P(x) = P_0(x)$ for all x in T. If so, output "false", otherwise output "true." □

Using the revised of Theorem 5 we can then show that a decidable equivalence problem implies the existence of a computable generator procedure.

We will show a slightly weakened form of the converse. Instead of using the equivalence problem for two arbitrary programs we will use the problem of deciding equivalence of an arbitrary program Q and a specific program P.

Theorem 11. *Let F be a member of $\mathscr{F}$ and suppose P computes F. If some generator function for F is computable then the problem of equivalence to P is decidable for $\mathscr{P}$.*

Proof. In order to decide whether Q is equivalent to P, simply run the generator on Q and try the two programs, P and Q, on the resulting test set. If they agree they are equivalent, otherwise they are not. □

5. Adequate Test Data, Second Definition

The second definition of correctness is intended to make the goals of software testing more tractable. For example, it has been observed that competent programmers will usually produce a program which is "close" to being correct, hence we can narrow our attention to the much smaller set of reasonable alternatives to the given program, rather than concerning ourselves with the universe of all possible programs. Similarly if our goal is simply to show that the program is free of errors of a certain type (and we assume that no errors of any other type occur) then this definition of correctness follows. In one form or another this idea is present in many different testing methodologies [3, 7, 9, 14, 19, 21, 26, 30, 33].

For the first of this section we will make no assumptions concerning the mapping Φ. Section 3 gave the definition for what we shall mean by saying that a test set is adequate for P (relative to Φ), and showed how such a test set could be used to demonstrate correctness.

Although Theorem 4 showed that adequate test data need not exist, we might hope that at least for any given set $\mathscr{P}$ and Φ one could decide whether in this particular situation adequate data existed. The following theorem shows that even this need not be the case.

Theorem 12. *There exists neighborhoods Φ for which the problem of deciding whether an adeaquate test set exists is undecidable.*

Proof. We will construct a particular system of programs with the desired property. Let the set $\mathscr{P}$ contain the following programs: P_0 will be a program for the constant function 0. For each $i>0$, P_i will be a program that on input x runs the ith Turing machine on input i for x steps. If the machine halts on exactly the xth step, then the program outputs 1, otherwise the program outputs 0. Thus these programs always halt for all inputs (even if the associated Turing machine fails to halt). The program P_i is constantly zero if and only if the ith Turing machine fails to halt on input i. Thus the problem of deciding the equivalence of an arbitrary program to P_0 is undecidable.

Define the set $\Phi(P)$ to be $\mathscr{P}$ for all programs P in $\mathscr{P}$. If $P(x)=1$ then $\{x\}$ is certainly adequate. If P is constantly zero then it is easy to show that no finite set T is adequate (let m be the largest element of T and consider some Turing machine, say the ith, that halts on input i in strictly greater than m steps). Thus for a given program P an adequate test set exists if and only if P is not constantly zero. Hence if there existed a program to decide for each P_i whether there was an adequate test set, one could use this procedure to decide whether the ith Turing machine will halt on input i, which is known to be undecidable [31]. Thus the problem of deciding whether there exists an adequate test set is itself undecidable. □

A theorem in the next section will demonstrate that the fact that $\Phi(P)$ is infinite for each program P is critical to Theorem 12.

For the remainder of this section we will assume that $\mathscr{P}$, $\mathscr{F}$ and Φ are such that for every program P there exists an adequate test set. We may then define generator and acceptor procedures analogous to the earlier concepts.

Definition. A procedure $A(P, T)$ is an *acceptor* if it returns "true" if T is adequate for P (relative to the set $\Phi(P)$) and "false" otherwise.

Definition. A procedure $G(P)$ is a *generator* if it produces a test set which is adequate for P (relative to $\Phi(P)$).

Using these new notions, we may again consider the computability of the acceptor and generator procedures.

Theorem 13. *Assume that for every program P there exists adequate test data. Then if the acceptor procedure is computable, there exists a computable generator.*

The proof is analogous to the proof of Theorem 5.

The converse is false.

Theorem 14. *The existence of a generator need not imply the existence of a acceptor.*

Proof. We demonstrate the theorem by giving a set $\mathscr{P}$ and neighborhood Φ for which the result holds. Let $\mathscr{P}$ be the set $\{P_{i,j}\}$ defined as follows: $P_{i,0}$ computes the function that is i on input 0 and 0 for all other inputs. For $j>0$ $P_{i,j}$ computes the function that is i on input 0, j on input 1 and 0 for inputs greater than 2. On input 2, $P_{i,j}$ outputs 0 if the ith Turing machine on input i halts in exactly j steps, otherwise it outputs 1.

Define $\Phi(P_{i,j})$ to be the set of programs $\{P_{i,m}: m \geqq 0\}$. Notice that the set $\{0, 2\}$ is sufficient to distinguish any two programs, hence is adequate. Thus in particular the program which produces this set for every input is a generator for this system.

If the ith Turing machine halts in m steps then $P_{i,m}(2)$ will be 0, and the test set $\{2\}$ will fail to distinguish $P_{i,0}$ and $P_{i,m}$ (which are indeed not equivalent since $P_{i,0}(1) \neq P_{i,m}(1)$). Otherwise, if the ith Turing machine fails to halt on input i then $P_{i,m}(2)=1$ for all values m, and the set $\{2\}$ will be adequate for $P_{i,0}$. Thus $A(P_{i,0}, \{2\})$ yields "false" if and only if the ith Turing machine halts on input i, so $A(P, T)$ cannot be computable. □

Notice that the equivalence problem is decidable for this example.

Theorem 15. *The existence of a decidable equivalence problem for a set Φ is not sufficient to guarantee either a computable acceptor nor a generator.*

Proof. Let the set $\mathscr{P}$ consist of the programs $\{P_{i,j}\}$ defined as follows. $P_{i,j}(x)$ returns i if $x=0$, 1 if $0<x \leqq j$ and the ith Turing machine halts in exactly x steps given input i, and 0 otherwise.

Two programs $P_{i,j}$ and $P_{k,m}$ are equivalent if and only if $i=k$ and the ith Turing machine given input i does not halt in n steps, where $\min(j, m) < n < \max(j, m)$. (That is, either the ith Turing machine halts earlier given input i, or if it halts at all, halts later.) Thus equivalence is decidable.

Let $\Phi(P_{i,j}) = \{P_{i,m}: m \geqq 0\}$. For a given value i, if the ith Turing machine fails to halt on input i, then all the elements of $\Phi(P_{i,j})$ compute the same function and $\{0\}$ is an adequate test set. If the ith Turing machine halts on input i in

exactly m steps, then $\{0, m\}$ is an adequate set. Thus an adequate test set exists for every program in $\mathscr{P}$.

A computable acceptor cannot, however, exist since $A(P_{i,j}, \{0\}$ yields "false" if and only if the ith Turing machine halts on input i.

Suppose there existed a computable generator function G. Let $T = G(P_{i,j})$. The ith Turing machine halts on input i if and only if it halts at the mth step, for some m in T. Since the halting problem is undecidable, no such function G can be computable. □

5.1. *Second Definition of Adequate, Finite Case*

In this section we will make the further assumption that the neighborhood $\Phi(P)$ is finite and that there is a (halting) procedure which, given P as input, will list all the elements of $\Phi(P)$. This case arise frequently in practice. For example, the testing methods described in [7, 9, 14, 19, 30, 33] all consider only finite sets of alternatives.

Theorem 16. *Every program with finite neighborhood has an adequate test set.*

Proof. For each P there are finitely many programs Q in $\Phi(P)$. For each such Q, either Q is equivalent to P or there exists a point x such that $P(x) \neq Q(x)$. In the latter case, add x to the test set. The result will be a finite test set which is adequate for P (relative to Φ). □

We will again consider a restricted form of the equivalence problem. Let $S(P, Q)$ be a procedure which outputs "true" if Q is a member of $\Phi(P)$ and Q is equivalent to P, and "false" otherwise.

Theorem 17. *For programs with finite neighborhoods, the following three statements are equivalent:*

(a) *The acceptor procedure $A(P, T)$ is computable,*
(b) *Some generator procedure $G(P)$ is computable,*
(c) *The equivalence checking procedure $S(P, Q)$ is computable.*

Proof. The argument that (a) $\Rightarrow$ (b) is analogous to the proof of Theorem 5.

(b) $\Rightarrow$ (c) Suppose some generator procedure is computable. Given programs P and Q, compute $\Phi(P)$ and decide whether Q is a member of $\Phi(P)$. If not output "false". If so execute the generator on P, and check whether P and Q agree on the resulting test set. If so output "true" and if not, "false." This procedure correctly computes $S(P, Q)$.

(c) $\rightarrow$ (a) Suppose $S(P, Q)$ is computable. Given program P and test set T, compute $\Phi(P)$. For each Q in $\Phi(P)$ for which $S(P, Q)$ returns "false" verify that $P(x) \neq Q(x)$ for some x in T. If any such verification fails output "false." If all such tests succeed output "true." This procedure correctly computes $A(P, T)$. □

Although an adequate set must always exist, this does not imply that any of the procedures studied in Theorem 17 need be computable.

Theorem 18. *There exist sets $\mathscr{P}$ with finite neighborhoods for which neither an acceptor, a generator nor the equivalence checking procedure are computable.*

Proof. Consider the programming system $\mathscr{P}$ constructed in the proof of Theorem 12. Define $\Phi(P_i)=\{P_0, P_i\}$ for all $i \geqq 0$. Then $\{0\}$ is an adequate test set for P_i if and only if the ith Turing machine on input i does not halt. Hence A is not computable.

Given that A is not computable, Theorem 17 demonstrates that neither $G(P)$ nor $S(P, Q)$ is computable. □

Summary

In this paper we have considered two different definitions for the concept of test data sufficient to demonstrate correctness. The first definition requires knowledge of both the program being tested and the function which the correct program should compute. For this definition we considered both the uniform case (where the testing system is presented with the function as part of its input) and the nonuniform case (where the testing system is designed for a specific function).

The second definition does not require direct knowledge of the correct function, but instead uses a set (neighborhood) of alternative programs. This definition appears to be the one used in the description of a number of different test methods. Using this definition we again considered two cases, corresponding to the neighborhood being infinite and finite, respectively.

For each of these four cases we have defined two functions: An *Acceptor* is a procedure which recognizes adequate test data, and a *Generator* is a procedure which produces it. Table 1 shows that whereas adequate test data always exists in 3 of the 4 cases, in no case must there necessarily be either a computable acceptor or generator.

Let A stand for the property that there exists a computable acceptor, likewise G for generator. Let E represent the system having a decidable equivalence problem. R will represent the restricted equivalence problem where one program is given and the second is chosen arbitrarily. Let S represent the even more restricted equivalence problem where the first program P is fixed and the second program must be a member of $\Phi(P)$. Letting arrows represent implication (for example $E \Rightarrow R$ means that the ability to do general equiva-

Table 1. Existence of Acceptor/Generator for different models

	Does adequate Test Data Always exist?	Must a computable Acceptor or Generator exist?
Uniform in F	Yes	No
Nonuniform in F	Yes	No
Unrestricted neighborhoods	No	No
Finite neighborhoods	Yes	No

Table 2. Summary of Theorems

Uniform in F	$A \Leftrightarrow G \Rightarrow E \quad E \not\Rightarrow A$
Nonuniform in F	$A \Leftrightarrow G \Leftrightarrow R \quad E \Rightarrow R$
Unrestricted neighborhoods	$A \Rightarrow G \quad E \not\Rightarrow G \quad E \not\Rightarrow A \quad G \not\Rightarrow A$
Finite neighborhoods	$A \Leftrightarrow G \Leftrightarrow S$

lence implies the ability to do restricted equivalence), Table 2 then lists the several theorems which we have proved in this paper.

7. Concluding Remarks

The dissimilar results we can show using different definitions of correctness, as illustrated by Table 2, are an indication that in discussing testing and correctness concerns one should be careful to state clearly and formally what one means by the terms "correctness" and "adequate test data." Informal descriptions of terms are dangerous, since they may have several different formal counterparts. As we have seen the implications of two different formal definitions may be startlingly dissimilar.

We have shown that under very general assumptions one is not guaranteed the ability to generate good test data. A logical next step would be to investigate what further restrictions are necessary to insure the existence of a generator and/or acceptor procedure, and whether these restrictions are satisfied in practice.

The theoretical connection between testing and equivalence which we have demonstrated should not really be surprising; for example Sayward and Baldwin [2], and Budd et al. [8] have found in practice that their testing systems required good heuristics for the equivalence problem.

Of course the interesting problem with respect to the second definition of correctness is to find sets Φ which are (a) sufficiently tractable that one can actually generate adequate test data relative to Φ, and (b) sufficiently general that one is confident that a correct program P is contained in $\Phi(P)$. It is our position that we cannot hope to consider this more specific problem until we understand that implications of the model in general, and that has been the goal of this paper.

We have considered the computability of functions which recognize and/or generate good test data. Systems to automatically generate test data have been considered [11, 29] and in fact have been given formal treatment [3]. But in all these cases the test data criteria are much weaker than what we have here considered.

We have tried to relate the study of testing to more traditional problems, in this case the problem of deciding the equivalence of two programs. In this fashion we can relate the body of literature on equivalence [23] to the question of testing. This appears to be a promising endeavor. Future candidates for similar treatment include research on complexity theory [5], inductive inference [1, 6] and automatic programming [4].

References

1. Angluin, D.: On the complexity of minimum inference of regular sets. Information and Control **39(3)**: 337–350 (1978)
2. Baldwin, D., Sayward, F.: Heuristics for determining equivalence of program mutations. Technical Report Number 161, Yale University 1979
3. Barzdin, J.M., Bicevskis, J.J., Kalninsh, A.A.: Construction of complete sample system for correctness testing. In: Lecture Notes in Computer Science, Vol. 32, pp. 1–12. Berlin: Springer-Verlag 1975
4. Biermann, A.W., Krishnaswamy, R.: Constructing programs from example computations. IEEE Transactions on Software Engineering SE-**2(3)**: 141–153 (1976)
5. Blum, M.: On effective procedures for speeding up algorithms. Journal of the ACM**18(2)**: 290–305 (1971)
6. Blum, L., Blum, M.: Toward a mathematical theory of inductive inference. Information and Control **28(2)**: 125–155 (1975)
7. Brooks, M.: Automatic generation of test data for recursive programs having simple errors. PhD Thesis, Stanford University 1980
8. Budd, T.A., Lipton, R.J., DeMillo, R.A., Sayward, F.G.: Mutation analysis. Technical Report Number 155, Yale University 1979
9. Budd, T.A.: Mutation analysis of program test data. PhD Thesis, Yale University 1980
10. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering SE-**4(3)**: 178–187 (1978)
11. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering SE-**2(3)**: 215–222 (1976)
12. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. Computer **11(4)**: 34–43 (1978)
13. DeMillo, R.A., Lipton, R.J.: A probabilistic remark on algebraic program testing. Information Processing Letters **7(4)**: 193–195 (1978)
14. Foster, K.A.: Error sensitive test cases. IEEE Transactions on Software Engineering SE-**6(3)**: 258–264 (1980)
15. Geller, M.: Test data as an aid in proving program correctness. Communications of the ACM **21(5)**: 368–375 (1978)
16. Goodenough, J.B., Gerhart, S.L.: Towards a theory of test data selection. IEEE Transactions on Software Engineering SE-**1(2)**: 156–173 (1975)
17. Goodenough, J.B.: A survey of program testing issues. pp. 316–340. In: Research Directions in Software Technology (P. Wegner, ed.), pp. 316–340. Cambridge, MA: MIT Press, 1979
18. Hamlet, R.: Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering SE-**3(4)**: 279–290 (1977)
19. Hamlet, R.: Testing programs with finite sets of data. The Computer Journal **20(3)**: 232–237 (1977)
20. Hamlet, R.: Test reliability and software maintenance. In: Proceeding of the Compsac conference, pp. 315–320. IEEE 1978
21. Hamlet, R.: Critique of reliability theory. In: Digest of the Workshop on Software Testing and Test Documentation, pp. 57–69. Fort Lauderdale 1978
22. Hantler, S.L., King, J.C.: An introduction to proving the correctness of programs. ACM Computer Surveys **8(3)**: 331–353 (1976)
23. Hopcroft, J.E., Ullman, J.D.: Formal languages and their relation to automata. New York: Addison-Wesley 1969
24. Howden, W.E.: Reliability of the path analysis testing strategy. IEEE Transactions on Software Engineering SE-**2(3)**: 208–214 (1976)
25. Howden, W.E.: Lindenmayer grammers and symbolic testing. Information Processing Letters **7(1)**: 36–39 (1978)
26. Howden, W.E.: Algebraic program testing. Acta Informat. **10(1)**: 53–66 (1978)
27. Howden, W.E.: Functional program testing. IEEE Transactions on Software Engineering SE-**6(2)**: 162–169 (1980)
28. Kernighan, B.W., Plauger, P.J.: Software tools. New York: Addison-Wesley 1976

29. Miller, E.F., Melton, R.A.: Automated generation of testcase datasets. In: Proceeding 1975 International Conference on Reliable Software, pp. 51–58. Los Angeles 1975
30. Ostrand, T.J., Weyuker, E.J.: Remarks on the theory of test data selection. In: Digest of the Workshop on Software Testing and Test Documentation, pp. 1–18. Fort Lauderdale 1978
31. Rogers, H. Jr.: Theory of recursive functions and effective computability. New York: McGraw-Hill 1967
32. *Rowland, J.H., Davis, P.J.: On the selection of test data for recursive mathematical sub-*routines. Siam Journal on Computing **10(1)**: 59–72 (1981)
33. White, L.J., Cohen, E.I.: A domain strategy for computer program testing. IEEE Transactions on Software Engineering SE-**6(3)**: 247–257 (1980)

Received March 17, 1981/May 22, 1982