Editorial

PAPERS
Statistical Relational Tables for Statistical Database Management  .......... S. P. Ghosh
Gandalf: Software Development Environments  ............... A. N. Habermann and D. Notkin  1117
Axiomatizing Software Test Data Adequacy  ..................... E. J. Weyuker  1128

CONCISE PAPERS
Guaranteed Response Times in a Distributed Hard-Real-Time Environment  .... D. W. Leinbaugh and M.-R. Yamini  139

CORRESPONDENCE
Comments on "Number of Faults per Line of Code"  .......... F. Stetter

1986 INDEX  Follows page  145

Calls for Papers  1153
Axiomatizing Software Test Data Adequacy

ELAINE J. WEYUKER

Abstract—A test data adequacy criterion is a set of rules used to determine whether or not sufficient testing has been performed. A general axiomatic theory of test data adequacy is developed, and five previously proposed adequacy criteria are examined to see which of the axioms are satisfied. It is shown that the axioms are consistent, but that only two of the criteria satisfy all of the axioms.

Index Terms—Software testing, test data adequacy.

I. INTRODUCTION

ONE of the most important problems in software engineering is how to determine whether or not a program has been tested enough that it can be released to users with reasonable confidence that it will function "acceptably." Of course, what is meant by "acceptable" will vary with the particular application, based on such factors as criticality of function, anticipated consequences of malfunction, and expected frequency of use.

In view of its importance, it is surprising that this area has seen relatively little research activity. Most of the research effort in software testing has involved the development of test data selection strategies rather than adequacy criteria. Furthermore, industry standards for determining test adequacy have been close to nonexistent.

Myers [22] 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."

We shall call a criterion used to determine whether testing may terminate, an adequacy criterion. Such a criterion represents minimal standards for testing a program, and as such measures how well the testing process has been performed. The criterion should relate a test set to the program, the specification, or both. In addition, it could also relate the test set to the program’s intended environment or operational profile. More will be said about this below.

We consider here only dynamic testing. This means that the program is run on one or more input vectors, and the outputs produced are assessed. There are other ways of validating a program which do not involve running the program on test cases, including static analysis and formal verification, but our concern here is only with ways of assessing the adequacy of dynamic testing.

An example of an adequacy criterion is branch adequacy. If a program P is represented by a flowchart, then a branch is an edge of the flowchart. Test set T is branch adequate for P, provided for every branch b of P, there is some t in T which causes b to be traversed. This is an example of an adequacy criterion which is entirely program-based in the sense that it is independent of the specification (except, of course, for comparing the results produced by the program for a given input with the intended results as defined in the specification). Other adequacy criteria are discussed in Section IV.

We are primarily interested here in adequacy criteria which are largely program dependent, and will thus generally omit reference to the specification. In such a case we may speak of "a program being adequately tested by a test set." We have chosen to consider program-based adequacy criteria since most proposed adequacy criteria are of this type. Such strategies are more easily mechanizable than specification-based ones, as they permit the program to be treated as a purely syntactic object. This means that well-understood graph theoretic concepts can be applied.

Rather than defining a particular criterion for test data adequacy, we develop a general axiomatic theory of adequacy in this paper. Thus we shall attempt to identify and abstract essential properties which should hold for any such criterion.

This work has two primary motivations. Although, as mentioned above, there has been relatively little research done to find good, usable, adequacy criteria, some criteria have been defined [1], [2], [4], [6], [7]. The work in this paper should help in understanding the strengths and weaknesses of these previously proposed criteria. In addition, the axiomatization should guide the definition of new adequacy criteria.

A question central to the formulation and formalization of our intuition on adequacy is: What should be the relationship between an adequately tested program and a correct program? An initial reaction might be that they should be intimately connected, perhaps even that an adequately tested program should be correct. But the purpose of testing is to uncover errors, not to certify correctness, and the
to be one plus the number of arithmetic operations in \( q \).

Note that with this definition, there are only finitely many different programs \( P \) such that \( |P| < n \), for each positive integer \( n \), since we have only finitely many identifiers and constants.

There have been many other proposals for measures of the size of a program [3], [7], [8], [11], [13], [14], [16], [19], [20], [21], [27]. Our definition in [4] was chosen to facilitate upper and lower bound calculations. Other more familiar choices would have yielded similar, but less arithmetically neat, results. Since our only use of size in the current paper is in the definition of the adequacy criteria presented in [4], we shall use that definition.

In Section III we will need notions of what it means for two programs to be close to one another. Such a notion could refer to either syntactic or semantic closeness, or some combination of the two. We shall say that two programs \( P \) and \( Q \) are almost the same if \( P = Q \) and \( P \) can be transformed into \( Q \) by applying exactly one instance of the following changes to \( P \):

1. Replace relational operator \( r_1 \) in a predicate in \( P \) with relational operator \( r_2 \).
2. Replace constant \( c_1 \) in a predicate in \( P \) with constant \( c_2 \).
3. Replace constant \( c_1 \) in an assignment statement in \( P \) with constant \( c_2 \).
4. Replace arithmetic operator \( a_1 \) in an assignment statement in \( P \) with arithmetic operator \( a_2 \).

Two different programs which are almost the same are the same size, have the same form, and compute the same function in essentially the same way, using the same variables.

An obvious question is: Which of these permitted changes are likely to preserve equivalence? One would not expect replacing a “<” by an “=” to preserve equivalence. Replacing a “<” by a “\( \leq \)” on the other hand, might preserve equivalence if, for example, the boundary value is treated the same on both outcomes of the predicate. Similarly, replacing a constant “0” by “159” would be unlikely to preserve equivalence, but replacing “0” by “1” might if the constant were used to control a loop which was computing a running sum.

A notion of closeness which is somewhat less restrictive than “almost the same” permits several changes to the program of the type permitted by rules 1–4, provided these changes do not alter the semantics of the program. We shall say that \( P \) and \( Q \) are very close provided \( P = Q \) and \( P \) can be transformed into \( Q \) by applying the above change rules 1–4 any number of times. Like programs which are almost the same, two very close programs are the same size, have the same form, and compute the same function using the same variables in the same roles, but now there may be somewhat more substantial differences in the way the computation is performed. In both cases syntactic data flow characteristics are maintained.

An example of “very close” programs might involve a program \( P \) which includes a loop executed \( k \) times. This is governed by two constants \( c_1 \) and \( c_2 \) such that \( c_2 - c_1 = k \). Program \( Q \) is identical to \( P \) except that \( c_1 \) is replaced by constant \( d_1 \) and \( d_2 \) by \( d_2 \). If \( d_2 - d_1 = k \), \( P \) and \( Q \) could well be equivalent and hence “very close.”

We shall say that \( P \) and \( Q \) are the same shape if \( P \) can be transformed into \( Q \) by applying the above change rules 1–4 any number of times. Whereas the notions “almost the same” and “very close” require that the two programs be both semantically and syntactically related, the present notion requires only syntactic closeness. Note that all three notions are reflexive and symmetric and that “very close” and “the same shape” are transitive relations.

A test set is a set of input vectors. We require that a program halt on every member of a test set. Of course it is not decidable whether or not a given program actually does halt on a given input [5], [26] and there are certainly programs that are not intended to halt on some or all inputs. Thus, we can pick some large fixed bound and stipulate that if an input causes a program to run longer than this amount of time (or execute more than this number of statements), it is not a suitable test case. Such an excessive running time may also signal a problem and indicate that the code should be carefully scrutinized.

In Section III we develop an axiomatic theory of program-based adequacy notions. Our purpose is to identify characteristics which should hold for any program-based adequacy criterion. We also introduce properties which we consider desirable, but not essential, for adequacy criteria. In Section IV we consider five previously defined adequacy criteria, and in each case consider which axioms and properties are satisfied.

III. AN AXIOMATIC THEORY OF TEST DATA ADEQUACY

We shall use the following notation in the sequel. \( P, Q \)

\( \) denote programs, \( S \) denotes a specification, and \( T, T', T_i \), \( i = 1, 2, \ldots \) denote test sets.

Axiomatic theories have traditionally been used in two complementary ways. On the one hand they serve to make underlying assumptions explicit. For example, Euclid introduced the axiomatic method to make explicit the assumptions underlying our geometric intuition, and Peano’s postulates make explicit the assumptions needed to derive the properties of the natural numbers. On the other hand, axiomatic theories enable one to derive the properties common to a collection of different structures. Thus the axioms for groups can be used to derive properties common to all groups. The axiomatic theory we develop is mainly in the spirit of the first use, although our axioms also serve to demarcate the sets of possible adequacy criteria satisfying subsets of our axioms.

An adequacy criterion tells us whether or not it is reasonable to terminate testing. If the adequacy criterion is applied to a program, specification, and test set, and we determine that the criterion has not been fulfilled, then it implies that we have not sufficiently tested the program. If there were programs for which no adequate test set existed using a given criterion, then when we determined that the criterion had not been fulfilled by a test set, we
would not know whether the problem was that testing was not comprehensive enough, or the program was simply not adequately testable.

In Section IV we shall see that this is precisely the situation for branch testing. If a program has unexecutable branches (an undecidable property [26]) then no amount of testing can cause every branch to be exercised. Therefore, when we find out that a given test set exercised 80 percent of the branches, we do not know whether we should continue trying to find test cases to exercise the remaining 20 percent of the branches, or stop testing because 100 percent of the executable branches have been exercised. Thus, the first and most important property of an adequacy criterion is applicability.

**Axiom 1:** (Applicability). For every program, there exists an adequate test set.

Due to our requirement that there be only finitely many representable points even when the domain is infinite, Axiom 1 can be rephrased as follows:

**Axiom 1**: For every program, there exists a finite adequate test set.

Of course, requiring that an adequate test set be finite certainly does not guarantee that one can generate all the required test cases. An interesting possible refinement of this axiom would require that for every program there exist some “reasonably sized” adequate test set. This might be defined in terms of the program’s size or complexity. We shall not pursue this idea further here, and leave it as an interesting open problem.

We shall say that a program has been **exhaustively tested** if it has been tested on all representable points of the specification’s domain. Such a test set, called an **exhaustive** test set, should be adequate even when the domain is infinite. But, of course, an important point of testing is to be able to select a subset of the domain which in some sense stands in for the entire domain. Programs intended to fulfill specifications with very small (finite) domains, however, might well require exhaustive testing using any reasonable criterion. In fact one only needs to be able to do nonexhaustive testing when the domain is large. The extreme case is a domain of size one. In this case the only alternative to exhaustive testing is no testing at all, surely an unacceptable solution. Thus, although a criterion may well require exhaustive testing in some cases, one which **always requires** exhaustive testing is unacceptable. Formalizing this we have the following:

**Axiom 2:** (Nonexhaustive Applicability). There is a program \( P \) and test set \( T \) such that \( P \) is adequately tested by \( T \), and \( T \) is not an exhaustive test set.

It is easy to argue that our next axiom is a reasonable one on intuitive grounds. Since an adequacy criterion represents a minimum degree of testing sufficiency, surely if a program has been adequately tested, running it on some additional (“unecessary”) tests should not make it inadequately tested.

**Axiom 3:** (Monotonicity). If \( T \) is adequate for \( P \), and \( T \subseteq T' \) then \( T' \) is adequate for \( P \).

We shall call the requirement that \( P(t) = S(t) \) for every \( t \) in \( T \), the **correctness condition**. A certification that \( T \) adequately tests \( P \) implies that it has been tested enough that it is reasonable to terminate testing. Ideally, as long as errors are being uncovered, testing should not end, and thus it might be argued that an adequacy criterion should only be invoked after \( P \) agrees with the specification on all of \( T \). In that case, a certification of adequacy implies that the correctness condition holds. In practice, however, noncritical errors are sometimes ignored, and programs with known errors are sometimes released. Therefore the adequacy criterion might reasonably be invoked even though \( P(t) \neq S(t) \) for some \( t \) in \( T \). Furthermore, if one of the requirements of an adequacy criterion is the correctness condition, then unless the criterion guarantees the correctness of the program, this requirement implies that the criterion does not satisfy the monotonicity axiom. That is:

**Theorem:** If a test data adequacy criterion is monotonic and implies the correctness condition, then the existence of any adequate test set for \( P \) implies that \( P \) is correct.

**Proof:** Let \( T \) be adequate for \( P \). Assume the criterion is monotonic and implies the correctness condition, and there is a \( t \) such that \( P(t) \neq S(t) \). Since the criterion is monotonic, \( T \cup \{t\} \) is adequate for \( P \). But this contradicts the requirement that \( P \) be correct on every element in the test set.

We also have the following immediate corollary:

**Corollary:** If a test data adequacy criterion is monotonic and implies the correctness condition, then no incorrect program can be adequately tested, and hence the applicability axiom does not hold.

It is this theorem, and our conviction that monotonicity is of central importance, that led us to conclude that the correctness condition should not be included in a notion of adequacy. However, an adequacy criterion would not normally be applied until the tester believed that testing was complete; this implies that the program is correct on the test set or the errors are below some level of criticality. Hence, one can expect that the correctness condition will generally hold when an adequacy criterion is invoked.

Since the correctness condition is not required for adequacy, if \( P \) has been adequately tested by \( T \) using a monotonic adequacy criterion, and \( P(T) = S(T) \), then even if new test data \( T' \) is added including some points on which the program is not correct, the new set \( T \cup T' \) is adequate for \( P \). In practice, when the presence of an error is detected by the test data in \( T' \), the program would be returned for debugging, and retested before being certified as adequately tested and released.

Another fundamental property of dynamic test data adequacy criteria is that a program cannot be deemed adequately tested if it has not been tested at all. As we stressed above, adequacy is a measure of how well the dynamic testing process has been performed, not the program’s correctness. Note that we have assumed that every program has at least one input variable. Otherwise, it might be reasonably argued that the empty set is an adequate test.
set for inputless programs. Therefore we propose the following:

**Axiom 4**: (Inadequate Empty Set). The empty set is not adequate for any program.

This does not imply that other ways of validating programs do not exist. For example, a program could be formally proved correct using verification techniques. It should then be known to be correct, however, pragmatic experience warns us that this is frequently not true [10], [12], and thus dynamic testing might be performed in addition. But even when some other validation technique has been perfectly applied, if a program has not been dynamically tested, it should not, in our view, be deemed adequately tested.

We next consider the question of how the semantic closeness of two programs should affect the way they are tested. Since we are studying program-based adequacy criteria, and by definition such criteria depend primarily on the implementation, the fact that two programs compute the same function should not necessarily imply that they require the same test data for adequate testing. Of course, we do not want to say that no two equivalent programs can be adequately tested by the same test set. That would be inconsistent with monotonicity. Thus we propose the following axiom:

**Axiom 5**: (Antiequationality). There are programs $P$ and $Q$ such that $P \equiv Q$, $T$ is adequate for $P$, but $T$ is not adequate for $Q$.

Having decided that semantic closeness (equivalence) is not enough to ensure that two programs require the same test data, we now consider syntactically close programs. Two programs have the same shape provided one can be transformed into the other using a set of four simple change rules. Viewed as graphs, such programs have the same structure and the same syntactic data flow characteristics, but there is no necessary relationship between the functions computed by the two programs or the paths traversed by a test set. Clearly, we should not expect such programs to necessarily require the same test data for adequacy, even if we restrict attention to program-based adequacy criteria. That is, the syntactic closeness of two programs is also not sufficient to demand that they require the same test data.

Again, in view of the monotonicity axiom, one does not want to say that whenever two programs are the same shape but not equivalent they should require different test data for adequacy. But there should certainly be some pair of programs which, although they are the same shape, require different adequate test sets.

**Axiom 6**: (General Multiple Change). There are programs $P$ and $Q$ which are the same shape, and a test set $T$ such that $T$ is adequate for $P$, but $T$ is not adequate for $Q$.

The next property we introduce is superficially analogous to the monotonicity axiom. Monotonicity required that if a test set $T$ is adequate for a given $P$, then a superset of $T$ should certainly be adequate. Similar intuition might lead one to feel that if $Q$ is a “subprogram” of $P$, then $T$ should be adequate for $Q$. Of course, we do not really mean that $T$ should be adequate for $Q$, but rather that the values that the elements of $T$ are transformed into on “entrance” to $Q$ should be adequate for $Q$.

We have to be careful about specifying just what we mean by a “subprogram” or else there may be multiple entry points to $Q$. Also, although a statement may look like an entry point syntactically, it may in fact never be executable as the first statement of the subprogram.

To deal with these problems, we introduce the notion of a component. A component of $P$ is any contiguous sequence of statements of $P$. By definition, a component is single-entrant, and represents a (contiguous) subcomputation within $P$.

Given a way to divide a program into its component pieces, a natural question arises: Should it follow that if a program has been adequately tested, each of the pieces that make up the program have been adequately tested? That is, if $T$ is adequate for $P$, and $Q$ is a component of $P$, and $T'$ is the set of all vectors of values that variables can assume on entrance to $Q$ for some input of $T$, then should $T'$ be adequate for $Q$? Of course, it is not decidable whether or not $Q$ can actually ever be executed within $P$, but since by assumption $P$ halts on every element of $T$, $T'$ can be effectively obtained. Although an initial reaction might be that such a property should hold, deeper consideration causes us to reject such a decomposition property as an axiom. The core of the problem is that although $P$ may appear to be more "complicated" than $Q$, in the sense that it physically contains $Q$, it may actually be simpler in some other (semantic) sense. This is particularly important if $Q$ is unexecutable in $P$.

In particular, let $T'$ be adequate for $P$ and $T'$ be the set of all vectors of values that variables can assume on entrance to $Q$ for some input of $T$. Since $Q$ is unexecutable in $P$, it follows that $T'$ is the empty set. Therefore, if $P'$'s adequate testing implied $Q$'s adequate testing, then the empty set would be deemed adequate for testing $Q$. Hence the adequacy criterion would not satisfy Axiom 4 (Inadequate Empty Set Axiom). In addition, since we have argued that every program should be adequately testable (Applicability Axiom) and there is no algorithm to decide of an arbitrary program whether or not it contains unexecutable code [26], this “decomposition property” should not hold.

But unexecutable components are not the only reason why we do not want this type of decomposition property to hold in general, despite its intuitive appeal. Let the domain of a component $Q$ be the set of all vectors of values that the variables may take on at entrance to $Q$ for any element in $P$'s domain. The problem is that $Q$'s domain may be very small (and is in general indeterminable), so even exhaustive testing of $P$ may mean that $Q$ is only lightly tested for its function. The extreme case of this is the problem cited above in which $Q$ is unexecutable in $P$, and hence its domain is the empty set.

To make this somewhat more concrete, consider the example of a program which has as a component a sorting
routine. Whenever the routine is entered, however, the
data are already sorted, and hence the component can only
be tested within the context of the larger program on al-
ready sorted data. Now surely we would not consider al-
ready sorted data alone, an adequate test set for a sort
routine, even though its domain within the larger program
contains only such data. With this discussion in mind, we
propose the following axiom:

**Axiom 7:** (Antidecomposition). There exists a program
P and component Q such that T is adequate for P, T’ is
the set of vectors of values that variables can assume on
entrance to Q for some t of T, and T’ is not adequate for
Q.

Now what if each of the pieces of a program have been
adequately tested? Should the entire program be consid-
ered adequately tested? That is, if T is adequate for P and
P(T) is adequate for Q, should T be deemed adequate for
P; Q?

A first reaction might be that this seems intuitively rea-
sonable, but of little practical value. After all, how often
can one expect the outputs produced on an adequate test
set for one program to be adequate for another indepen-
dent program. One might therefore suggest the following
intuitively stronger composition property: If T₁ is ade-
quate for P and T₂ is a test set such that P(T₂) = T where
T is adequate for Q, then T₁ ∪ T₂ is adequate for P; Q.

However, it can be shown that for monotonic adequacy
criteria, these two composition properties are the same.
The problem with these properties is that they do not take
into account the added complexity and interactions which
may result when two programs are composed. A tra-
ditional way to test programs has been in a “bottom-up”
fashion. The lowest level modules are tested first, and
successively combined to form larger and larger pieces
until the entire program is tested. Acceptance of the com-
position properties as axioms would be analogous to say-
ing that it is sufficient to stop testing once the lowest level
modules have been tested, with no testing required for
their integration. This is clearly not reasonable. Thus we
propose our final axiom:

**Axiom 8:** (Anticomposition). There exist programs P
and Q such that T is adequate for P and P(T) is adequate
for Q, but T is not adequate for P; Q.

We close this section by considering proposals for no-
tions of two programs being both syntactically and se-
mantically close. If P and Q perform the same computa-
tion in “substantially the same way,” one could argue
that they should require the same test data. We first use
the notion of “almost the same” as a way of defining
closeness. One might therefore consider the following
property.

**Equivalent Single Change Property:** If T is adequate
for P, and Q is almost the same as P, then T is adequate
for Q.

A somewhat less restricted notion of closeness is based
on our definition of “very close.”

**Equivalent Multiple Change Property:** If T is adequate
for P, and Q is very close to P, then T is adequate for Q.

Since it is not clear that either of these proposals is the
most appropriate way to capture the idea of syntactic and
semantic closeness, we feel they should not determine
whether or not an adequacy criterion is acceptable. Also
note that antiextensionality, the general multiple change
axiom, and these two equivalent change properties are not
independent. Axiom 5 states that semantically close pro-
grams should not necessarily be tested the same way. Ax-
iom 6 states the same for syntactic closeness. The two
equivalent change properties, however, state that pro-
garms which are both semantically and syntactically close
should require the same test data. Thus we have that the
failure of either Axioms 5 or 6 implies that the equivalent
multiple change property, and hence the equivalent single
change property, holds. Of course it also follows then that
one way to guarantee that Axioms 5 and 6 hold is to de-
vise a criterion for which the equivalent change properties
fail.

We introduce one final notion of syntactic and semantic
closeness. If Q is a component of P and \( P = Q \), then Q
might be considered a simpler implementation of the com-
putation performed by P in a much more real sense. It
thus might be reasonable to argue that an adequate test set
for P should necessarily be adequate for Q.

**Equivalent Component Property:** If Q is a component
of P and \( P = Q \), then if T is adequate for P, T is adequate
for Q.

IV. A Survey of Program-Based Adequacy
Criteria

Having proposed a set of axioms, we now investigate
what extent various program-based adequacy criteria
satisfy our theory. The first adequacy criterion we con-
ider is path adequacy: If P is a program represented by
a flowchart, a path in P is a finite sequence of nodes \( n₁, n₂, \ldots, nₖ \) \( k \geq 2 \) such that there is an edge from \( n_i \) to
\( n_{i+1} \) for \( i = 1, 2, \ldots, k - 1 \) in the flowchart rep-
resenting P. Our definition of path is a purely syntactic one.
We assume there is a path from the entry statement to
every statement of the program. We say that T is path ade-
quate for P if for every path p of P, there is some t in T
which causes p to be traversed. It is easy to show that
path adequacy implies branch adequacy, which implies
statement adequacy (for every statement q of P, there is
some t in T which causes q to be executed).

The applicability axiom fails for path adequacy, as there
may be no adequate test set for any program which con-
tains an unexecutable path. In contrast, the nonexhaustive
applicability axiom, monotonicity, and the inadequate
empty set axiom clearly hold for path adequacy.

The equivalent single change property, and hence the
equivalent multiple change property, does not hold for
path adequacy. Consider the P1 and P2:

\[
P1: \quad \text{input } x \\
\quad \text{if } x \leq 1 \quad \text{then } x \leftarrow -x \\
\quad \text{else } x \leftarrow 1 - x \\
\text{end}
\]
if $x = -1$ then $x \leftarrow 1$
      else $x \leftarrow x + 1$
end
def P2:
      input $x$
      if $x \leq 0$ then $x \leftarrow -x$
      else $x \leftarrow 1 - x$
end
      if $x = -1$ then $x \leftarrow 1$
      else $x \leftarrow x + 1$
end
output $x$

$P1$ is adequately path tested by the set $\{0, 1, 2, 3\}$. In contrast, $P2$, which is almost the same as $P1$, is not adequately path tested by $\{0, 1, 2, 3\}$ as the path formed by taking the true exit from both decisions is not traversed. In fact, the situation is even worse than that. $P2$ cannot be adequately path tested by any test set, since there can be no input which will cause the true exit to be taken from both decisions.

This example brings out an interesting point. It might seem that the change properties require too much. Programs which perform the same computation in essentially the same way, as made precise in Section II, should certainly require test sets which are very close, but maybe not the same. Perhaps if $Q$ is very close to $P$ and can be formed by applying the change rules $k$ times, we should only require that there be points $t_1, \ldots, t_k$ and $t_{k+1}, \ldots, t_{2k}$ such that if $T$ is adequate for $P$, then $(T - \{t_1, \ldots, t_k\}) \cup \{t_{k+1}, \ldots, t_{2k}\}$ is adequate for $Q$. That is, for each change made to $P$ to form $Q$, one point in the adequate test set may be changed. We have just seen, however, that there are cases for which no amount of modification of an adequate test set for $P$ will make it path adequate for $Q$ which is almost the same as $P$. $P2$ is also an example which shows that the applicability axiom is not satisfied for path adequacy.

The failure to satisfy these change properties shows that the antiextensionality axiom and the general multiple change axiom hold. Furthermore, the antidecomposition axiom does not hold. Notice that if a program $P$ contains unexecutable paths, there is no path adequate test set for $P$. Even exhaustive testing is not adequate in such a case.

Anticomposition does hold. Consider the following programs:

$P3$:
      input $x$
      if $x < 11$ then $x \leftarrow 0$
      else $x \leftarrow 1$
end
output $x$

$P4$:
      input $x$
      if $x = 0$ then $x \leftarrow 0$
      else $x \leftarrow 1$
end
output $x$

Let $T = \{10, 11\}$. $T$ is path adequate for $P3$. $P3(T) = \{0, 1\}$, and is path adequate for $P4$. However, $T$ is not path adequate for $P3;P4$ since, for example, no input of $T$ takes the then exit of the predicate $x < 11$ followed by the else exit of the predicate $x = 0$. In fact, that is an unexecutable path in the sense that there can be no set of test data which causes it to be traversed.

Finally, the equivalent component property does not hold.

$P5$:
      input $x$
      $x \leftarrow -x$
      if $x \geq 0$ then if $x > 3$ then continue
      else continue
      else continue
      end
      end
output $x$

$P6$:
      input $x$
      if $x \geq 0$ then if $x > 3$ then continue
      else continue
      else continue
      end
      end
output $x$

$P6$ is a component of $P5$ and $P5 = P6$. The test set $T = \{-4, 0, 1\}$ causes every path of $P5$ to be executed, but the first continue statement of $P6$ (i.e., the one corresponding to the predicate "$x > 3$" being true) is never executed as a result of running $P6$ on $T$.

Similar arguments and examples demonstrate that, except for the anticomposition axiom, exactly the same set of axioms and properties hold for branch and statement adequacy, as hold for path adequacy. A straightforward argument shows that the anticomposition axiom does not hold for statement and branch adequacy. These results are summarized in Section V.

Since these control flow criteria are widely used [17], it is important to understand the implications of these results. The most serious problem is the failure of the applicability axiom to hold, implying that there are programs which are not adequately testable. But if one of the criteria is applied and a determination is made that some elements of a program have never been executed by a given set of test data, this could either mean that additional (or more carefully selected) test data are required, or that the unexecuted portion of the program is not executable. Since there can be no algorithm to decide for an arbitrary program whether or not it contains unexecutable code, or whether a particular statement, branch, or path is executable [26], one cannot in general hope to determine which situation prevails.

Mutation analysis [1], [2], [6] is an adequacy criterion which is substantially different from the control flow criteria considered above. Given a program $P$, specification
S, and a test set \( T \) such that \( P \) is correct on every member of \( T \), a set of alternative programs known as mutations or mutants of \( P \) is produced. Each mutant \( P_i \) is formed by modifying a single statement of \( P \) in some predefined way, similar to the transformations permitted by our definition of ‘almost the same.’ Each mutant is then run on every element of \( T \), and \( T \) is said to be mutation adequate for \( P \) provided that for every inequivalent mutant \( P_i \) of \( P \), there is a \( t \) in \( T \) such that \( P_i(t) \neq P(t) \). A similar idea was proposed and implemented by Hamlet, and is described in [15].

Unlike the other adequacy criteria mentioned so far, mutation adequacy is not monotonic because it requires that the correctness condition hold. If this condition were removed from the requirements, however, mutation adequacy would be monotonic. Again, due to the correctness condition, the applicability axiom does not hold for mutation adequacy. Certainly it is true that a mutation system can produce only finitely many mutants, and therefore a finite set always suffices to distinguish a program from its inequivalent mutants. But, if \( P \) is incorrect at point \( t \), and \( P' \) is a mutant which differs from \( P \) only at \( t \), then no test set is mutation adequate for \( P \). Since the correctness condition does not play any fundamental role in mutation adequacy, however, and if it were removed both of these axioms would be satisfied, we will consider that mutation adequacy satisfies these two axioms for comparison purposes and list them as such in the table of Section V.

Mutation adequacy, like all the other program-based adequacy criteria discussed, clearly satisfies the inadequate empty set axiom. It is also clearly antitelescopic. Two programs which perform the same computation in substantially different ways will surely have different sets of mutations and require different sets of test data to distinguish the program from the mutants in general. It is less obvious, however, that two programs which are equivalent and perform the computation in essentially the same way may also require different test data. Consider, however, the following equivalent programs \( P_7 \) and \( P_8 \) which return the index of the first occurrence of a maximal element in an integer vector \( A \) of length at least 2. They appear in [6] and have been rewritten to conform more closely to the syntax of our programming language.

We have selected this example since all the necessary details of mutation testing \( P_7 \) are described in [6].

### \( P_7 \): input \( I, N, R, A(N) \)

\[
R \leftarrow 1
\]

\[
I \leftarrow 2
\]

While \( I \leq N \) do if \( A(I) > A(R) \) then \( R \leftarrow I \) else continue

\[
I \leftarrow I + 1
\]

end

Output \( R \)

### \( P_8 \): input \( I, N, R, A(N) \)

\[
R \leftarrow 1
\]

\[
I \leftarrow 1
\]

While \( I \leq N \) do if \( A(I) > A(R) \) then \( R \leftarrow I \) else continue

\[
I \leftarrow I + 1
\]

end

Output \( R \)

In [6], the authors outline the argument that the test set \( T = \{(1 \ 2 \ 3), \ (1 \ 3 \ 2), \ (3 \ 1 \ 2), \ (1 \ 2 \ 2)\} \) constitutes a mutation adequate test set for \( P_7 \). (Technically, the test data should also include values for variables other than the elements of \( A \). These are omitted to simplify notation and focus attention on the characteristics of the test data which are important to us.) However, \( T \) is not mutation adequate for \( P_8 \). Consider \( P_9 \), a mutant of \( P_8 \):

### \( P_9 \): input \( I, N, R, A(N) \)

\[
R \leftarrow 2
\]

\[
I \leftarrow 1
\]

While \( I \leq N \) do if \( A(I) > A(R) \) then \( R \leftarrow I \) else continue

\[
I \leftarrow I + 1
\]

end

Output \( R \)

Even though this program is not equivalent to \( P_8 \) (for example \( (2 \ 1 \ 1) \) distinguishes the two programs), they produce the same output on every input in \( T \). This example thus serves to show that mutation adequacy does not satisfy the equivalent change properties. As mentioned above, \( T \) is mutation adequate for \( P_7 \), and \( P_7 \) is almost the same as \( P_8 \), yet \( T \) is not mutation adequate for \( P_8 \). The failure of mutation adequacy to satisfy these properties guarantees that the general multiple change and antitelescopicity axioms hold. The example also shows that the nonexhaustive applicability axiom holds since \( T \) is a nonexhaustive mutation adequate test set for \( P_7 \).

Since it is possible to have a mutation adequate test set for a program containing unexecutable code, and the empty set is not mutation adequate for such a program, the antidecomposition axiom holds. The anticomposition axiom also holds for mutation adequacy. To see this, let \( P \) and \( Q \) be programs and \( T \) a test set such that \( T \) is mutation adequate for \( P \) and \( P(T) \) is mutation adequate for \( Q \). Let \( P' \) be an inequivalent mutant of \( P \). Since \( T \) is mutation adequate for \( P \), there must be a \( t \) in \( T \) such that \( P(t) \neq P'(t) \). Let \( Q \) be such that \( Q(P(T)) = Q(P'(T)) \) but such that \( P \) is equivalent to \( P' \); \( Q \). In that case, even though \( T \) is mutation adequate for \( P \), and thus distinguishes \( P \) from every inequivalent mutant, and \( P(T) \) is mutation adequate for \( Q \), and thus distinguishes \( Q \) from every inequivalent mutant, \( T \) does not distinguish \( P \) from the inequivalent mutant \( P' \).

In contrast, the equivalent component property does not hold for mutation adequacy. If \( Q \) is a component of \( P \) and \( P \equiv Q \) but \( Q \) is not executable in \( P \), then any mutant of \( P \) which involves a change within \( Q \), yields a program which is equivalent to \( P \). But this change to \( Q \) (when \( Q \)
is considered an independent program) could obviously yield a program which is inequivalent to \( Q \) but not distinguised from \( Q \) by \( T \).

The class of changes to a program that are made in order to create the set of mutants is closely related to the class of changes permitted by our definition of "almost the same," and in fact provided the basis for our decision about which changes should be permitted. All of our changes are examples of mutations. We rejected the replacement of one identifier by another, which is permitted in forming a mutant, since such a change would alter the data flow characteristics of the program, and there is evidence [24], [25], [9], [23], [18] that such characteristics are important for testing. Since a mutant of \( P \) is formed by making a single modification of \( P \), we differentiated between single changes and multiple changes.

In [4] we introduced the notion of size adequacy. We say that a test set \( T \) is size adequate for a program \( P \) if for every program \( P' \) which is not equivalent to \( P \), but for which \( P'(T) = P(T) \), we have \( |P'| > |P| \). Since we cannot hope for a test set to distinguish a program from all inequivalent programs, the above might be considered a reasonable approximation to the ideal. However, we showed that:

**Theorem:** If \( P \) is a nonminimal program, then no test set \( T \) can be size adequate for \( P \) unless \( T \) is exhaustive.

Thus, using size adequacy as the criterion, the only programs which can be adequately tested with less than an exhaustive test set are programs which are in a sense "optimal." This is a serious weakness of size adequacy since an important part of the intelligent selection of test data involves choosing a subset of the domain which in some sense stands in for the entire domain during testing. The difficulty arises from the possibility of constructing programs in which an equivalent of \( P \) is "embedded." This led us to introduce [4] the last adequacy criterion which we consider. In Section III we defined a component of a program. We now introduce a second less restrictive notion of what it means for one program to be a part of another. The definition of a component required that the statements be physically adjacent to one another. The notion of "reduction" removes this requirement. We first introduce seven simplification rules. Each one represents a way that statements may be deleted (or omitted) from the program.

1. Replace some assignment statement by continue.
2. Replace: 
   - if PRED then \( P \) 
   - else \( Q \) 
   - end 
   by \( P \).
3. Replace: 
   - if PRED then \( P \) 
   - else \( Q \) 
   - end 
   by \( Q \).
4. Replace: 
   - if PRED then \( P \) end 
   by \( P \).
5. Replace: 
   if PRED then \( P \) end 
   by continue.
6. Replace: 
   while PRED do \( P \) end 
   by \( P \).
7. Replace: 
   while PRED do \( P \) end 
   by continue.

Program \( M \) reduces to program \( N \) if \( N \) can be obtained from \( M \) by applying these reduction rules, 0 or more times. We say that \( P \) is embedded in \( Q \) if \( Q \) reduces to some program which is equivalent to \( P \). Clearly, if \( M = N \) then \( M \) is embedded in \( N \) and \( N \) is embedded in \( M \). The converse of this statement is not true. Also, if \( M = N \) then \( M \) is embedded in \( P \) if and only if \( P \) is embedded in \( N \). Finally, a program \( P \) is self-embedded if there is a program \( Q \) such that \( P \) reduces to \( Q \), \( Q = P \), and \( Q \) is not identical to \( P \). It is easy to see that every component of a program \( P \) is embedded in \( P \). Of course the semantic relationship between a program and one of its components is generally much clearer than that of an arbitrary program which is embedded in another.

We say a finite test set \( T \) is modified size adequate for a program \( P \) if for each program \( P' \) such that \( P \) is not embedded in \( P' \), but for which \( P'(T) = P(T) \), we have \( |P'| > |P| \). Thus, instead of requiring that \( T \) be sufficient to distinguish \( P \) from all programs which are no longer than \( P \), we require that \( T \) distinguish \( P \) from a very large nonpathological subset.

We have shown [4] that modified size adequacy essentially subsumes branch adequacy and mutation adequacy. In particular we have shown the following:

**Theorem:** If \( P \) is not self-embedded, and \( T \) is a modified size adequate test set for \( P \), then \( T \) is branch adequate for \( P \).

**Theorem:** If \( P \) is not embedded in any of its inequivalent mutants and \( T \) is modified size adequate for \( P \), then \( T \) is mutation adequate for \( P \).

It is obvious that modified size adequacy is monotonic and satisfies the inadequate empty set axiom. The anti-decomposition axiom holds since there can be modified size adequate test sets for programs containing unexecutable code for which the empty set would not be modified size adequate. The applicability axiom holds since exhaustive testing will distinguish any pair of programs which differ on some point of the specification's domain.

To see that the equivalent multiple changes property, and hence the equivalent single change property, hold for modified size adequacy, one need first recognize that the permitted changes are all size preserving. That is, if \( P \) and \( Q \) are the same shape, then \( |P| = |Q| \). If \( P \) is very close to \( Q \), then \( P \approx Q \), and hence embedded in exactly the same programs. Thus, if \( P' \) is a program which is no longer than \( P \), and such that \( P \) is not embedded in \( P' \), \( P' \) is no longer than \( Q \), and \( Q \) is not embedded in \( P' \). Furthermore, \( P(t) = P'(t) \) if and only if \( Q(t) = P'(t) \), since \( P \approx Q \). Thus, \( T \) is modified size adequate for \( P \) if and only if it is modified size adequate for \( Q \).
We next check the general multiple change axiom. Consider P10 and P11:

**P10:** input \( x \)

\[
\begin{align*}
\text{if } x = 100 & \text{ then } x \leftarrow 1 \\
\text{else } x & \leftarrow 4 \\
\text{end}
\end{align*}
\]

output \( x \)

**P11:** input \( x \)

\[
\begin{align*}
\text{if } x < 2 & \text{ then } x \leftarrow 1 \\
\text{else } x & \leftarrow 4 \\
\text{end}
\end{align*}
\]

output \( x \)

P10 and P11 are the same shape and clearly inequivalent. The test set \( T = \{0, 1, 2, 4\} \) is modified size adequate for P11 but not for P10, which requires that any modified size adequate test set include values around 100. (See [4] for a definition of critical points and a proof of this statement.) Since \( T \) is a nonexhaustive test set which is modified size adequate for P11, it follows that the non-exhaustive applicability axiom is satisfied. The equivalent component property holds since if \( P = Q \), \( P \) is not embedded in \( R \) if and only if \( Q \) is not embedded in \( R \), and \( R(T) = P(T) \), if and only if \( R(T) = Q(T) \). Also, if \( Q \) is a component of \( P \), \( |Q| < |P| \), so \( |R| > |P| \) implies \( |R| > |Q| \). In fact, the same arguments work to show that the following stronger statement is true:

If \( P = Q \), \( |Q| < |P| \), and \( T \) is adequate for \( P \),

then \( T \) is adequate for \( Q \).

To see that the anticomposition axiom holds for modified size adequacy, consider P12 and P11 of the previous example:

**P12:** input \( x \)

\[
\begin{align*}
\text{if } x < 1 & \text{ then } x \leftarrow 0 \\
\text{else continue} & \\
\text{end}
\end{align*}
\]

output \( x \)

Since the continue statement is an abbreviation for a statement of the form \( \text{VAR} \leftarrow \text{VAR} \), each such statement adds one to the count. Therefore, \( |P11| = 2 = |P12| \). The set \( T = \{-1, 0, 1, 2, 4\} \) is modified size adequate for P12, and \( P12(T) = \{0, 1, 2, 4\} \) is modified size adequate for P11. However, \( T \) is not modified size adequate for P12; P11. To see this, consider P13:

**P13:** input \( x \)

\[
\begin{align*}
\text{if } x < 2 & \text{ then } x \leftarrow 1 \\
\text{else if } x = 2 & \text{ then } x \leftarrow 2 * x \\
\text{else continue} & \\
\text{end}
\end{align*}
\]

output \( x \)

\( |P13| = 4 = |P12| \; P11 \) and \( P12; P11(t) = P13(t) \) for every \( t \) in \( T \). However, \( P12 \; P11 \) is not equivalent to \( P13 \) since, for example, \( P12; P11(3) = 4 \) while \( P13(3) = 3 \). Note that neither \( P12 \; P11 \) nor \( P13 \) is embedded in the other. Therefore, \( T \) is not modified size adequate for \( P12; P11 \).

The above example also serves to show that modified size adequacy is anticonstructive. \( P12; P11 = P11 \), the set \( \{0, 1, 2, 4\} \) is modified size adequate for \( P11 \), but as mentioned above, it is not modified size adequate for \( P12; P11 \).

**V. Summary**

We have developed an axiomatization of software test data adequacy for program-based adequacy criteria. The first four axioms stated that every program must be testable, that an adequacy criterion must be satisfiable in a nontrivial way, and that a program which has not been tested at all should not be deemed adequately tested. Also, once a program has been adequately tested no amount of additional test data can transform it into an inadequately tested program.

Four axioms were in a sense “negative” axioms. They said that programs which are closely related either syntactically or semantically but not both, may well require different test data. Also, the fact that each of the pieces of a program has been adequately tested, does not necessarily imply that the entire program has been adequately tested. Finally, and less intuitively obvious, even though a program has been adequately tested, it does not necessarily follow that each of its components has been tested. This is due in part to the fact that programs may contain unexecutable code.

Having developed this system of axioms, we considered five previously defined adequacy criteria to see which of the axioms each satisfied. We found that the two most widely used (statement and branch adequacy) satisfied only five of the axioms. Only two of the criteria satisfied all eight of the axioms, and only one of these criteria fulfilled the three desirable properties as well. Table I summarizes these results.
At this point, we have a consistent set of axioms, as evidenced by the fact that modified size adequacy and mutation adequacy satisfy all of the axioms. We propose to continue developing this theory, and attempt to classify adequacy criteria according to the axioms they satisfy. Hopefully this work will also encourage others to identify essential characteristics of adequacy criteria and propose additional axioms.

ACKNOWLEDGMENT

I am grateful to M. Davis, P. Frankl, D. Hamlet, J. Gourlay, S. Weiss, and an anonymous referee for many helpful suggestions. I am particularly indebted to M. Evangelist for the time and thought he put into reading and criticizing this work. Not only is the organization of this paper improved, but several interesting ideas were added as a result of his probing questions.

REFERENCES


---


Elaine J. Weyuker received the Ph.D. degree in computer science from Rutgers University, New Brunswick, NJ, in 1977. She has been on the faculty of the Courant Institute of Mathematical Sciences of New York University since 1977, and is currently an Associate Professor of Computer Science. Before coming to NYU, she was a Systems Engineer for IBM and was on the faculty of the City University of New York from 1969 to 1975. Her research interests are in software engineering, particularly software complexity measures, software testing and reliability, and in the theory of computation. She is the author of a book (with Martin Davis), Computability, Complexity, and Languages.

Dr. Weyuker has been an ACM National Lecturer and is currently a member of the Executive Committee of the IEEE Computer Society Technical Committee on Software Engineering.