AXIOMATIZING SOFTWARE TEST DATA ADEQUACY

BY

Elaine J. Weyuker

January, 1984

Report #99

This research was supported in part by the National Science Foundation under grant MCS-82-01167.

C.1

AXIOMATIZING SOFTWARE TEST DATA ADEQUACY

Elaine J. Weyuker

1. 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 factors such as criticality of function, anticipated consequences of malfunction, and expected frequency of use.

In view of its importance, it is surprising that there has been relatively little research activity in this area. 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 whether or not enough testing has been performed appear to be close to nonexistent. Myers [MY79] 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 the criterion used to determine whether or not testing may terminate, an adequacy criterion. Rather than developing a particular criterion for test data adequacy, we develop a general axiomatic theory of test data adequacy in this paper. Thus we shall attempt to identify and abstract

This research was supported in part by the National Science Foundation under grant MCS-82-01167.

essential properties which must hold for any such criterion. We will also discuss properties which, although desirable, do not appear to be really essential.

There are two primary motivations for this work. Although, as mentioned above, there has been relatively little research done to find good, usable, adequacy criteria, there have been some criteria defined. The work in this paper should help in understanding the strengths and weaknesses of these previously proposed criteria. In addition, the axiomatization should facilitate the definition of good adequacy criteria by indicating the most important characteristics of such a criterion.

We say that a set of test data T is adequate to test program P relative to specification S provided that it fulfills some set of prespecified characteristics. We call this set of characteristics, which may relate T to the program, the specification, or both, an adequacy criterion. One well-known adequacy criterion, for example, is branch adequacy. If a program is represented by a flowchart, then a branch is an edge of the flowchart. 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 3.

We are primarily interested in this paper 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 almost

all adequacy criteria which have been proposed are program-based. Such strategies are much more amenable to mechanization than specification-based ones, as they permit the program to be treated as a purely syntactic object, and hence the large, well-understood theory of graphs can be applied. In addition, the focus on program-based criteria also reflects an inherent problem associated with specification-based criteria. In particular, given any specification and test set, there are infinitely-many programs which are correct on the test set (i.e. match the specification) but are wrong elsewhere.

We now define our programming language. Although most of the ideas of the paper are not really dependent on the particular details of this language, it is nonetheless necessary to have an explicit syntax.

Our language will contain a finite number of identifiers whose range is the integers (positive, negative, or zero). The language also contains a finite number of constants representing particular integers; we will assume that all numbers encountered as input or output values can be represented by corresponding constants of the language. Thus, although a function may be defined over an infinite set, we will only be able to represent a finite number of these using constants of the language, and thus all test cases should be chosen from this finite set. Arithmetic expressions are to be constructed using constants, identifiers, and the arithmetic operators +, -, *, /, in the usual manner. An assignment statement has the form:

VAR ← EXP

where VAR is an identifier and EXP is an arithmetic expression. A continue statement has the form:

continue

This is a dummy statement, much like the continue statement of Fortran, and can

be thought of as simply an abbreviation for an assignment statement of the form:

VAR ← VAR

where the same identifier occurs on the left as on the right.

A predicate is a condition having one of the forms:

B1=B2, B1≠B2, B1<B2, B1≤B2,

where B1 and B2 are each either a constant or an identifier. A program body is defined recursively:

(1) An assignment statement is a program body.

(2) if PRED then P
else Q
end

is a program body if PRED is a predicate and P and Q are program bodies.

(3) if PRED then P end

is a program body if PRED is a predicate and P is a program body.

(4) while PRED do P
end

is a program body if PRED is a predicate and P is a program body.

(5) P
Q

is a program body if P and Q are program bodies.

A declaration statement has the form:

declare VAR1,VAR2,...,VARn

where VAR1,VAR2,...,VARn are distinct identifiers. They are known as the declared identifiers.

An input statement has the form:

input

This statement causes n constants of the language to be used as input values to be assigned to the n declared identifiers in the order specified in the declaration statement.

An output statement has the form:

output

This statement causes the current values of the n declared identifiers to be output in the order specified in the declaration statement.

Finally, a program consists of a declaration statement, followed by an input statement, followed by a program body, followed by an output statement. Since our language consists of entirely familiar locutions, there is no need for us to specify further its formal semantics.

By definition, programs are single-entry/single-exit. It thus makes sense to speak of composing programs, or more properly composing program bodies. It is true that the formats of the last three statement types are rather artificial and unlike most real programming languages; they are, however, a technical convenience that will be used in Section 2, when composition is discussed. If P and Q are programs, we write P;Q to mean that the instruction following Q's unique input statement is to be executed immediately following the instructions preceding P's unique output statement, and the declaration statement has been modified when appropriate.

The domain of a specification S is the set of all values for which S is defined. This definition may be either an explicit or implicit prescription, or a proscription. Values not included in the domain are considered "don't care" conditions.

The domain of a program is the set of all values for which the program is defined. A program can be undefined for an input either because it abnormally terminates (yielding an error message for example) or because it enters a loop

and fails to halt. The former type of situation can be detected algorithmically, the latter cannot. By definition, therefore, a program halts on every element of its domain, although one cannot, in general, determine the set of values for which the program halts, or the function being computed by the program. Due to these problems, and the fact that the specification defines what should be computed, we take the position that test cases should be selected from the specification's domain. There is not much point in testing a program on an input if any output (or no output) is acceptable, as is the case for points outside the specification's domain.

For program P, we let P(x) denote the result of P executing on input vector x. If x is in the specification's domain, then we let S(x) denote the value which a program intended to fulfill S should produce on input x. For x not in the domain of S, we shall say that S(x) is undefined. If T is a set of input vectors and P a program, we let P(T) denote the set of output vectors produced by P on each of the members of T. If P and Q are programs, we write $P \equiv Q$ (P is equivalent to Q) if and only if P(x)=Q(x) for every element x. In particular, this implies that for each x, P(x) is defined if and only if Q(x) is defined, and hence that P and Q have the same domain. It is tempting to extend the notion of equivalence to permit us to speak of "the equivalence of program P and specification S". Informally, this would mean that the program fulfills the given specification, or, more formally, that S(x) = P(x) for all x in the domain of S. The problem with such an extension is that this notion of equivalence would not be an equivalence relation. In particular, for x outside the domain of S, we are willing to accept any program behavior. Thus, it is perfectly possible that both P and Q fulfill specification S, but P and Q are not equivalent. Thus, transitivity would not hold, and hence equivalence would

not be an equivalence relation. We thus retain our original restriction that equivalence is defined only for pairs of programs.

We do want to have a way of indicating that a program fulfills a specification. Thus, we introduce a notion of correctness. We shall say a program P is correct for a specification S if P(x) = S(x) for every element in the domain of S. Note that it is perfectly reasonable for two programs to be correct for S without being equivalent.

In Section 3, we will need a notion of size of a program. The question of how to measure the size or complexity of a program is a difficult one which many people have considered [C, E, EM, G, HAL75, HAL77, HAN, MC, MI, MY77, WHH]. As in [DAW], we shall find it most useful to define the size of P (denoted |P|) to be the maximum of two quantities associated with the program P:

(1) The number of arithmetic operations in P plus the number of ←'s. (Note that since the continue statement is an abbreviation for a statement of the form VAR ← VAR, each such statement adds one to this count.)

(2) The number of occurrences of predicates in P.

We also compute |R|, where R is a program body, in the same way, and say that |q|, where q is an assignment statement, is 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.

In Section 2, we will need notions of what we mean when we say that two programs are close to one another. Of course, 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 provided P≡Q

and P can be transformed into Q by applying exactly one instance of the following textual changes to P:

(1) Replace a relational operator r1 in a predicate in P with relational operator r2.

(2) Replace a constant c1 in a predicate in P with constant c2.

(3) Replace a constant c1 in an assignment statement in P with constant c2.

(4) Replace an arithmetic operator a1 in an assignment statement in P with arithmetic operator a2.

Two programs which are almost the same are as close as two programs can be without being identical. They are the same size, have the same form, and compute the same function in essentially the same way, using the same variables.

A notion of closeness which is somewhat less restrictive than "almost the same" permits multiple textual 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 similar provided $P \equiv Q$ 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 similar 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. Again syntactic data flow characteristics are maintained. In Section 3, we shall mention the reason for distinguishing the special case of a single change.

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 "similar" require that the two programs be both

semantically and syntactically related, this notion requires only syntactic closeness. Note that all three are reflexive and symmetric and that similarity and the same shape are transitive relations.

We shall 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 and there are certainly programs which are not intended to halt on some or all inputs. Still, we can pick some large, fixed, upper bound and say that if a program runs longer than this amount of time (or executes more than this number of statements), that it is not a suitable test case. Of course, such an excessive running time may well signal a problem and indicate that the code should be carefully scrutinized, but that is tangential to the current discussion.

Now, it is possible that an adequacy criterion requires that $P(t) = S(t)$ for every t in T. We shall call such a requirement the correctness condition. It is our view that a certification that a program has been adequately tested by a test set implies that it is reasonable to terminate the testing phase of the development cycle. Surely, as long as errors are being uncovered, testing is not complete. Thus, we believe that an adequacy criterion should only be invoked after the program agrees with the specification on the entire test set. In that case, a certification of adequacy implies that the correctness condition holds.

In Section 2 we develop an axiomatic theory of program-based adequacy notions. Our purpose is to identify characteristics which should hold for any adequacy criterion which is program-based. We also clarify the reasons for our decision to exclude an explicit requirement of correctness on the test set from the notion of test data adequacy. In addition, we discuss properties which have an intuitive appeal, and thus might seem to be potential axioms. We

explain the reasons why, on deeper consideration, each such proposal was discarded. Finally, we introduce properties which we consider desirable, but not essential, for adequacy criteria. In Section 3 we consider six previously defined adequacy criteria, and in each case consider which of our axioms and properties are satisfied.

2. AN AXIOMATIC THEORY OF TEST DATA ADEQUACY

We shall use the following notation in the sequel. P, Q, P_i i=1,2,... denote programs or program bodies, S denotes a specification, and T, T', T_i i=1,2,... denote sets of test data.

The first and most important property of an adequacy criterion is applicability. Every program must be testable, regardless of the quality of the program. There are certainly properties of programs which are desirable and should be encouraged, and other properties which are to be highly discouraged. Still, the solution to poor programming practice is not to make poor programs untestable. Thus we have:

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:

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

We shall say that a program has been exhaustively tested if it has been tested on all representable points of the domain. Such a test set, called an

exhaustive test set, should always be adequate. 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. If a criterion satisfies the applicability axiom, it follows that if a program has an infinite domain, we can always select a proper subset which adequately tests the program. Programs 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 non-exhaustive testing when the domain is large. The extreme case is, of course, the case of a domain of size one. In this case the only alternative to exhaustive testing is not testing the program on any member of the domain 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.

With this in mind, we introduce our next axiom. Intuitively it says that a criterion must be fulfillable by some non-exhaustive test set.

AXIOM 2 (Non-exhaustive 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.

In fact, we really want to require more than Axiom 2. Not only do we want to mandate that there be some program which is adequately testable by a non-exhaustive test set, we want to make sure that this does not occur only for our "best" programs. We shall say a program is optimal provided there is no shorter equivalent program. Obviously, people do not in general write optimal programs, and we must therefore be able to adequately test non-optimal ones.

AXIOM 3 (Non-optimal Applicability): There is a non-optimal program P and test

set T such that P is adequately tested by T, and T is not an exhaustive test set.

Clearly, the satisfaction of Axiom 3 implies satisfaction of Axiom 2, but not the converse. In Section 3 we shall see an example of a criterion which satisfies Axiom 2 but not Axiom 3. Furthermore, these axioms are not simply refinements of Axiom 1. In fact, they are independent of Axiom 1 in the sense that the truth value of Axiom 1 for a given test data adequacy criterion is independent of the truth value of Axioms 2 and 3. In Section 3, we shall see examples of adequacy criteria such that Axiom 1 is true while Axiom 3 is false, Axiom 1 is false while Axiom 3 is true, and Axioms 1 and 3 are both true.

A final comment on exhaustive testing. There is one truly fundamental and ideal property of test data selection and adequacy criteria which has been frequently discussed: a program should be correct on every element of the test set, if and only if the program is correct. But we know that this is an impossible goal even though for every program such a test set exists. Whenever a program has not been tested on every possible input, there are infinitely-many programs which are correct on a given test set but incorrect elsewhere. Thus, any criterion which permits anything less than exhaustive testing of any program is not going to satisfy this property. But, in fact, the point of testing is <u>not</u> to guarantee correctness. The goal is to uncover errors.

It is easy to argue that our next axiom is a reasonable one on intuitive grounds. Surely if a program has been adequately tested, running the program on some additional ("unnecessary") tests should not make the program inadequately tested.

AXIOM 4 (Monotonicity): If T is adequate for P, and $T \subseteq T'$, then T' is adequate for P.

Deeper reflection on this axiom, however, reveals an unfortunate implication. If one of the requirements of the adequacy criterion is that the program produce the correct results on every element of the test set, then unless the criterion guarantees correctness, this requirement implies that the criterion does not satisfy the monotonicity axiom. That is:

THEOREM 1: 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 an adequate test set for P. Assume the adequacy criterion is monotonic and implies the correctness condition. Assume 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, in conjunction with our conviction that monotonicity is an important axiom, that led us to conclude that the correctness condition should not be included in a notion of adequacy. As mentioned in Section 1,

however, we believe that in general an adequacy criterion should not be applied until the tester believes that the testing process is complete. This then implies that the program is correct on the test set. Hence, one can expect that the correctness condition will in practice hold when the 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) for all t in T, then even if new test data T′ is added including some points on which the program is not correct, the new set T 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. However, a certification that a program has been adequately tested does not guarantee that the program is correct. It means, rather, that the program is ready for release and hopefully, if the adequacy criterion is a good one, the program contains few errors which occur rarely.

Suppose a criterion C satisfies the applicability axiom. Then for program P there is an adequate test set T, where T is a finite subset of the domain D. If C is also monotonic, then D is adequate for P as it should be. Thus we have:

THEOREM 2: If C is a monotonic adequacy criterion which satisfies the applicability axiom, then exhaustive testing is adequate for every program.

We next consider a property which categorizes specification-based (black-box) adequacy criteria. It states that a semantic closeness is sufficient to imply that two programs require the same test data.

Extensionality Property: If T is adequate for P and P ≡ Q then T is adequate for Q.

It is clear that this is a property which is inappropriate for program-based adequacy criteria. Essentially it says that the adequacy of test data is independent of the implementation. Since, by definition, a program-based criterion depends primarily on the implementation, this is clearly an unacceptable characteristic. (See the introduction for a discussion of the reasons why we consider program-based criteria.) Thus we have the following axiom:

AXIOM 5 (Antiextensionality): There are programs P and Q such that P ≡ Q, T is adequate for P, but T is not adequate for Q.

We do not want to say that no two equivalent programs can be adequately tested by the same test set. Certainly that would be inconsistent with monotonicity. Thus, all antiextensionality requires is that the adequacy criterion consider the algorithm used in implementation, at least in some cases.

Having decided that semantic closeness (equivalence) is not enough to insure that two programs require the same test data, we now consider syntactically close programs. We said 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. Clearly, we cannot 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, however, that whenever two programs are the same shape but not similar (and hence inequivalent), they should require different test data for adequacy. But there should certainly be some pair of programs which, although they are the same shape, have different adequate test sets.

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

The next theorem underscores a consequence of using an adequacy criterion which fails to satisfy axiom 6, but requires the correctness condition to hold. The intuition is that test data that satisfies an adequacy criterion which does not satisfy the general multiple change axiom and which requires the correctness condition, will not detect errors. Of course, what one would hope is just the opposite, namely that T would be sensitive enough to expose errors in inequivalent programs even though they are the same shape.

THEOREM 3: Let C be an adequacy criterion which does not satisfy the general multiple change axiom and implies the correctness condition. Then the existence of any adequate test set T for program P implies that any program which has the same shape as P is correct on every element of T.

PROOF: Let T be adequate for P and let P_i, i=1,...,n be the set of all programs which have the same shape as P. Since C does not satisfy the general multiple

change axiom, T is adequate for every P_i. Since adequacy implies the correctness condition, each P_i is correct on every element of T. □

The next property we introduce is superficially analogous to the monotonicity axiom. Monotonicity required that if a set of test data 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 don't 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 a program P is any program body of P. By definition, a component is single-entrant, and represents a (contiguous) subcomputation within P. We will sometimes speak of a program Q being a component of program P, by which we mean that the program body formed by removing the declaration, input, and output statements from Q is a component of P.

Note that our definition of a component is a purely syntactic one. In particular, one can obviously recursively decide whether or not Q is a component of P. Furthermore, this can be determined relatively quickly. When we consider components as programs, of course, we assume that appropriate declaration, input, and output statements have been added.

Component Decomposition Property: Let T be adequate for P, let Q be a component

of P, and let T' be the set of all vectors of values that variables can assume on entrance to Q for some input of T. Then T' is adequate for Q.

Thus, if a program has been adequately tested, each of the pieces that make up the program have been adequately tested. 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, we can effectively obtain T'. However, it is possible that although P appears to be more "complicated" than Q, in the sense that it physically contains Q, it is actually simpler by some other (semantic) measure of complexity. This is particularly significant when Q is unexecutable in P. The following theorem underscores this point.

THEOREM 4: Let P be a program containing an unexecutable component Q, and let T be a test set for P fulfilling adequacy criterion C. Then the component decomposition property does not hold for C unless the empty set is an adequate test set for Q.

Intuitively, this theorem states that if a criterion C satisfies the applicability axiom, and if there are programs which are not adequately testable by the empty set, then this decompostion property does not hold for C. Thus, one should not expect the component decomposition property to hold for any criterion which can be satisfied for programs containing unexecutable code. Since people do write programs containing unexecutable code, and there is no algorithm to decide of an arbitrary program whether or not it contains unexecutable code [WEY], one does not want to require fulfillment of the component decomposition property as stated above.

But unexecutable components are not the only reason why we do not want

this 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 is already sorted, and hence the component can only be tested within the context of the larger program on already sorted data. Now surely we would not consider already 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 in mind, we propose the following axiom:

AXIOM 7 (Antidecompostion): There exists a program P and component Q such that T is an adequate test set for P, T′ is the set of vectors of values that variables can assume on entrance to Q for some input 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 considered adequately tested? The next property asserts that it should.

Weak Compositon Property: If T is adequate for P and P(T) is adequate for Q, then T is adequate for P;Q.

At first glance, this property might seem intuitively reasonable, 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 independent program. One might therefore suggest the following intuitively stronger property:

Strong Composition Property: Let T_1 be adequate for P and let T_2 be a test set such that $P(T_2)=T$ where T is adequate for Q. Then $T_1 \cup T_2$ is adequate for P;Q.

Closer reflection indicates that the two properties are equivalent provided monotonicity holds. That is:

THEOREM 5: The strong composition property holds for an adequacy criterion if and only if both the weak composition property and the monotonicity axiom hold for the criterion.

The real problem with the weak composition property is unfortunately not solved by this strengthening. A traditional 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 complete. Acceptance of the composition properties as axioms would be analogous to saying that it is sufficient to stop testing once the lowest level modules have been tested. The composition properties do not take into account the added complexity and interactions which may result when two program bodies are composed. Examples of this will be shown in Section 3.

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 appropriate notions of two programs being both syntactically and semantically close. If two programs perform the same computation in "substantially the same way", one could argue that they should require the same test data. We first use the notion of two programs being "almost the same" as a way of defining closeness. As pointed out earlier, two programs which are almost the same are as close as two programs can be without being identical. 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 "similarity".

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

Of course, antiextensionality, the general multiple change axiom, and these two equivalent change properties are not independent. Axiom 5 states that the semantic closeness of programs is not sufficient reason for them to require the same test data, while axiom 6 states that syntactic closeness is not sufficient either. The two equivalent change properties, however, state that programs 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 devise a criterion for which the equivalent change properties fail. In any case, it is not clear that this is the most appropriate or reasonable way to capture this idea, and thus we feel they should not determine whether or not an adequacy criterion is acceptable.

The transformation rules used to define "almost the same" and "similar" are purposely highly restrictive. Instead of rules 3 and 4, for example, we could have a single rule such as:

(3′) Replace any assignment statement q1 in P with an assignment statement q2 provided that |q1|=|q2|.

Such a rule subsumes our rules 3 and 4, and would permit many operations to be changed at once within an assignment statement, and also permit such things as the replacement of one variable by another. Recent work to develop criteria for the selection of test data recognized the need to include data flow information in such decisions [RW, N, LK]. Similar arguments hold for test data adequacy criteria. Thus the use of rule 3′ as the basis of a desirable property of adequacy is unsatisfactory, for it would say that programs which may have different data flow characteristics would nonetheless necessarily have the same adequate test sets provided they were equivalent and syntactically close.

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 computation 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. Note, too, that the adequate empty set problem of Theorem 4 does not arise here since we are not looking at the set of values which the elements of T can assume at entrance to Q, but rather to T itself. Also, whereas we rejected the extensionality property as a potential axiom on the grounds that it stated that adequacy should be independent of implementation, here P and Q are intimately related both syntactically and semantically. Thus it is interes ting to consider:

Equivalent Component Property: If Q is a component of P and Q ≡ P, then if T is adequate for P, T is adequate for Q.

We feel these last three properties are desirable, but not necessary characteristics of adequacy criteria. We therefore do not include them among our set of axioms.

3. A SURVEY OF PROGRAM-BASED ADEQUACY CRITERIA

Having proposed a set of axioms, we now investigate to what extent various program-based adequacy criteria satisfy our theory.

The first adequacy criterion we consider is statement adequacy: T is statement adequate for P provided for every statement q of P, there is some t in T which causes q to be executed. For this criterion, the applicability axiom fails. For any program which contains unexecutable statements, there can be no adequate test set. In contrast, the non-exhaustive applicability axiom, non-optimal applicability axiom, and monotonicity clearly hold for statement adequacy.

The equivalent single change property, and hence the equivalent multiple change property, does not hold for statement adequacy. For example, consider the program P1:

```
P1: declare x
    input
    if x≤0 then x←0
           else continue
           end
    output
```

The set {0,1} is statement adequate for this program. However, {0,1} is not statement adequate for program P2 which is almost the same as P1 and hence equivalent to P1.

```
P2: declare x
    input
    if x<0 then x←0
           else continue
           end
    output
```

In fact the situation can be substantially worse than this example illustrates. As the next example shows, it is possible to have two programs which are almost the same, such that one can be adequately tested using statement adequacy, while the other cannot be.

```
P3:  declare x
     input
     if x<2 then x←x-1
            else x←0
            end
     if x=1 then x←0
            else continue
            end
     output
```

For program P3, the test set {2, 3} is statement adequate.

```
P4:  declare x
     input
     if x<0 then x←x-1
            else x←0
            end
     if x=1 then x←0
            else continue
            end
     output
```

P4 is almost the same as P3, but there can be no statement adequate test set for P4 since the predicate x=1 can never evaluate to true and hence the statement "x←0" in the second if statement can never be executed using any test set. This example brings out an important and interesting point. It might seem that the change axioms require too much. Programs which perform the same

computation in essentially the same way, as made precise in Sections 1 and 2, should certainly require test sets which are very close, but maybe not the same. Perhaps if Q is similar 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 that there are cases for statement adequacy, for which no amount of modification of an adequate test set for P will make it statement adequate for Q which is almost the same as P. P4 is also an example which shows that the applicability axiom is not satisfied for statement adequacy.

Of course, the failure to satisfy these change axioms shows that the antiextensionality axiom and the general multiple change axiom hold. Furthermore, the component decomposition property holds and hence the antidecomposition axiom does not hold. Notice that if a program P contains unexecutable code, there is no statement adequate test set for P. Even exhaustive testing is not adequate in such a case. It is also easy to see that the composition properties hold for statement adequacy, and hence the anticomposition axiom does not hold.

Finally, the equivalent component property does not hold. Consider the following programs:

```
P5:  declare x
     input
     x ← -x
     if x>0 then if x>3 then continue
                        else continue
                        end
            else continue
            end
     x ← 0
     output
```

```
P6:  declare x
     input
     if x>0 then if x>3 then continue
                        else continue
                        end
            else continue
            end
     x ← 0
     output
```

P6 is a component of P5. The test set T = {-4, 0, 1} causes every statement 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.

Thus, of the eight axioms and three desirable properties, only five hold for statement adequacy.

Branch adequacy was defined in Section 1, and is reported to be one of the most widely used non-trivial adequacy criteria. It is easy to see that branch adequacy implies statement adequacy. It is also easy to see that precisely the same five axioms hold for branch adequacy as held for statement adequacy. These results are summarized in Table 1 at the end of Section 3.

If P is a program represented by a flowchart, a path in P is a finite sequence of nodes $(n_1,\ldots,n_k)$ $k\geq 2$ such that there is an edge from n_i to n_{i+1} for $i=1,2,\ldots,k-1$ in the flowchart representing P. Our definition of path is a purely syntactic one. We assume there is a path from the declaration statement to every statement of the program. We say that T is path adequate 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 see that path adequacy implies branch adequacy. Again, non-exhaustive applicability, non-optimal applicability, monotonicity, antiextensionality, and the general multiple change axioms hold, while applicability, antidecomposition, and the equivalent component, equivalent single and multiple change properties do not hold. However, unlike the cases of statement and branch adequacy, anticomposition does hold. Consider the following programs:

```
P7:  declare x
     input
     if x<11 then x←0
             else x←1
             end
     output
```

```
P8:  declare x
     input
     if x=0 then x←0
            else x←1
            end
     output
```

Let T={10,11}. T is path adequate for P7. P7(T)={0,1}, and is path adequate for P8. However, T is not path adequate for P7;P8 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.

Programs P3 and P4 showed that it was possible to have a pair of programs which are almost the same and such that one is adequately testable using statement adequacy as the criterion, while the other is not. P3 and P4 also demonstrate this for branch adequacy. We now show that the same situation can occur if path adequacy is used as the adequacy criterion:

```
P9:  declare x
     input
     if x<1  then x ← -x
             else x ← 1-x
             end
     if x=-1 then x ← 1
             else x ← x+1
             end
     output
```

```
P10: declare x
     input
     if x<0  then x ← -x
             else x ← 1-x
             end
     if x=-1 then x ← 1
             else x ← x+1
             end
     output
```

P9 is adequately path tested by the set {0, 1, 2, 3}. P10, in contrast, cannot be adequately path tested by any test set as there can be no input which will cause the true exit to be taken from both decisions.

A final overall problem with these three code-coverage measures of adequacy is: a determination that certain statements of a program have never been executed by a given set of test data could either mean that the program requires additional (or more carefully selected) test data, or that the unexecuted portion of the program is not executable. But, 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 [WEY], one cannot in general hope to determine which situation prevails.

Mutation analysis [B, BDLS, DLS] is an adequacy criterion which is substantially different from the criteria considered so far. 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 [HAM].

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 chart at the end of this section.

Mutation adequacy, like all the other program-based adequacy criteria discussed, is clearly antiextensional. 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 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 [DLS] and have been rewritten to conform to the syntax of our programming language. (We consider each element of the vector A as a separate identifier to conform to the syntax of our programming language.)

```
P11: declare I, N, R, A(N)
     input
     R←1
     I←2
     while I≤N do if A(I) > A(R) then R←I
                                 else continue
                                 end
                  I←I+1
             end
     output
```

```
P12: declare I, N, R, A(N)
     input
     R←1
     I←1
     while I≤N do if A(I) > A(R) then R←I
                                 else continue
                                 end
                  I←I+1
             end
     output
```

In [DLS], 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 P11. (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.) If the domain of P11 is some set which properly includes T, then since T is not an exhaustive test set, mutation adequacy satisfies the non-exhaustive applicablility axiom. However, T is not mutation adequate for P12. Consider P13, a mutation of P12:

```
P13: declare I, N, R, A(N)
     input
     R←2
     I←1
     while I≤N do if A(I) > A(R) then R←I
                                 else continue
                                 end
                 I←I+1
             end
     output
```

Even though this program is not equivalent to P12, (for example (2 2 1) distinguishes between 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 single change property, and hence the equivalent multiple change property. As mentioned above, T = {(1 2 3), (1 3 2), (3 1 2), (1 2 2)} is mutation adequate for P11, and P11 is almost the same as P12, yet T

is not mutation adequate for P12. As we pointed out in Section 2, the failure of mutation adequacy to satisfy these properties guarantees that the general multiple change and antiextensionality axioms hold for mutation adequacy.

A variation of this example shows that the non-optimal applicability axiom holds for mutation adequacy. Consider the following program:

```
P14: declare I, N, R, A(N)
     input
     R←1
     I←1
     I←2
     while I<N do if A(I) > A(R) then R←I
                                 else continue
                                 end
                  I←I+1
             end
     output
```

Clearly P11≡P14 and |P14|=7 while |P11|=6, so P14 is non-optimal. T, a non-exhaustive test set, is mutation adequate for P14.

Since it is possible to have a mutation adequate test set for a program containing unexecutable code, and the empty set would not in general be 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)≠P′(t). Let Q be

such that Q(P(t))=Q(P'(t)) but such that P;Q is not 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;Q from the inequivalent mutant P';Q.

In contrast, the equivalent component property does not hold for mutation adequacy. If Q is a component of P and P ≡ 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 distinguished from Q by T.

The class of changes to a program which 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 or should not be permitted. All or our changes are examples of mutations. However, a mutation system would permit the replacement of one identifier by another, a change we rejected for reasons discussed earlier. Also, mutation analysis is the reason we differentiated between single changes and multiple changes. A mutant of a program P is formed by making a single modification of P.

The next adequacy criterion to be considered was introduced in [DAW]. 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) for each t in 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. If each mutant P_i of a program P had the property $|P_i| \leq |P|$, mutation adequacy could in turn be thought of as an approximation to size adequacy. Instead of requiring test data to be sufficient to distinguish P from all inequivalent shorter programs, it need only distinguish P from a predefined subset of these programs. In fact, it is possible in some implemented mutation systems for $|P_i| = |P|+1$. In that case, our definition of size would have to be slightly, but not essentially, modified for this to be true.

It is obvious that size adequacy is monotonic. The antidecomposition axiom holds since there can be size adequate test sets for programs containing unexecutable code for which the empty set would not be size adequate. The applicability axiom holds since exhaustive testing will distinguish any pair of inequivalent programs.

To see that the equivalent multiple changes property, and hence the equivalent single change property, hold for 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 similar to Q, then $P \equiv Q$. Thus, any program which is inequivalent to P and no longer than P, is inequivalent to Q and no longer than Q, so T is size adequate for P if and only if it is size adequate for Q.

We next check the general multiple change axiom. Consider programs P15 and P16:

```
P15: declare x
     input
     if x=100 then x←1
              else x←4
              end
     output
```

```
P16: declare x
     input
     if x<2 then x←1
            else x←4
            end
     output
```

P15 and P16 are the same shape and clearly inequivalent. The test set T={0, 1, 2, 4} is size adequate for P16 but clearly not for P15, which requires that any size adequate test set include values around 100. (See [DAW] for a definition of critical points and a proof of this statement.) Since T is not an exhaustive test set, and is size adequate for P16, size adequacy satisfies the non-exhaustive applicability axiom.

That the equivalent component property holds follows immediately from the observation that if Q is a component of P, then $|Q| \leq |P|$.

To see that the anticomposition axiom holds for size adequacy, consider the following program P17 and program P16 of the previous example:

```
P17:  declare x
      input
      if x<1 then x←0
             else continue
             end
      output
```

|P16|=2=|P17|. The set T = {-1, 0, 1, 2, 4} is size adequate for P17, and P17(T) = {0, 1, 2, 4} is size adequate for P16. However, T is not size adequate for P17;P16. To see this, consider the following program:

```
P18:  declare x
      input
      if x<2 then x←1
             else if x=2 then x←2*x
                         else continue
                         end
             end
      output
```

|P18|=4=|P17;P16| and P17;P16(t)=P18(t) for every t in T. However, P17;P16 is not equivalent to P18 since, for example, P17;P16(3)=4 while P18(3)=3. Therefore, T is not size adequate for P17;P16.

The above example also serves to show that size adequacy is antiextensional. P17;P16≡P16, the set {0, 1, 2, 4} is size adequate for P16, but as shown above, it is not size adequate for P17;P16.

Even though each of the axioms 1 and 2, and 4 through 8 holds for size

adequacy, as well as the equivalent component and two equivalent change properties, the next theorem (the proof of which appears in [DAW]) makes it clear that this is not a completely satisfactory adequacy criterion.

THEOREM 6: Suppose that the program P is non-optimal. Then no test set T can be size adequate for P unless T is exhaustive.

Thus, the non-optimal applicability axiom does not hold for size adequacy, a very important deficiency. The difficulty arises from the possibility of constructing programs in which an equivalent of P is "embedded". This led us to introduce [DAW] the last adequacy criterion which we consider. In Section 2 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:

(1) Replace some assignment statement by continue

(2) Replace the program body:

```
if PRED then P
        else Q
        end
```

by P.

(3) Replace the program body:

```
if PRED then P
        else Q
        end
```

by Q.

(4) Replace the program body:

```
if PRED then P end
```

by P.

(5) Replace the program body:

```
if PRED then P end
```

by continue.

(6) Replace the program body:

```
while PRED do P end
```

by P.

(7) Replace the program body:

```
while PRED do P end
```

by continue.

We say that program M reduces to N if the program 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 $M \equiv N$ then M is embedded in N and N is embedded in M. The converse of this statement is not true. Also, if $M \equiv N$ then M is embedded in P if and only if N is embedded in P. Finally, we shall say that a program P is self-embedded if there is a program Q such that P reduces to Q, $Q \equiv P$, 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 $P'(t)=P(t)$

for each t in T, we have |P'|>|P|. We are requiring here that T be sufficient to distinguish P from a very large class of non-pathological shorter programs.

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

THEOREM 7: If P is a program which is not self-embedded, and T a modified size adequate test set for P, then T is branch adequate for P.

THEOREM 8: If P is a program which is not embedded in any of its inequivalent mutants and T is modified size adequate for P, then T is mutation adequate for P.

Each of the axioms 1 through 8 holds for modified size adequacy as well as the equivalent component and two equivalent change properties. The examples used to show that the anticomposition axiom and antiextensionality hold for size adequacy, also demonstrate that they hold for modified size adequacy. One need only observe that P17;P16 is not embedded in P18. The example used to show that the general multiple change axiom is satisfied for size adequacy, also demonstrates that it holds for modified size adequacy.

Similarly, the same argument which was used to prove that the equivalent single and multiple change properties hold for size adequacy can be applied for modified size adequacy, provided one observe that since P16≡P17, they are embedded in exactly the same programs. Monotonicity obviously holds, and the arguments used to show that the applicability and antidecomposition axioms hold for mutation adequacy and size adequacy, can be used for modified size adequacy. Finally, 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, for both size adequacy and modified size adequacy, essentially the same arguments work to show that the following stronger statement is true:

If P ≡ Q and |Q|≤|P| then if T is adequate for P, T is adequate for Q.

From the point of view of our theory, the critical difference between these two adequacy criteria is that the non-optimal applicability axiom does hold for modified size adequacy. As mentioned above, this adequacy criterion was originally introduced in [DAW] because of the result described in Theorem 6.

To see that the non-optimal applicability axiom, and hence the non-exhaustive applicability axiom, hold for modified size adequacy, consider programs P19 and P20:

```
P19:  declare x, y
      input
      y←x-x
      output

P20:  declare x, y
      input
      y←0
      output
```

Clearly, P19 ≡ P20, and since |P19|=2 while |P20|=1, P19 is non-optimal.

It has been shown [DAW] that for any test set T, if T is modified size adequate for P19, then $|T| \geq 3$. In fact any size 3 test set is modifed size adequate for P19. For such a test set to be inadequate for P19, there would have to be some program Q such that P19 is not embedded in Q, $|Q| \leq 2$, and Q agrees with P19 at three or more points. This has been shown in [DAW] to be impossible.

Table 1 summarizes the results for the eight axioms and three properties, and the six adequacy criteria surveyed above.

TABLE 1

	statement	branch	path	mutation	size	modified size
applicability	no	no	no	yes	yes	yes
non-exhaus applicability	yes	yes	yes	yes	yes	yes
non-opt applicability	yes	yes	yes	yes	no	yes
monotonicity	yes	yes	yes	yes	yes	yes
antiextensionality	yes	yes	yes	yes	yes	yes
general multiple change	yes	yes	yes	yes	yes	yes
antidecomposition	no	no	no	yes	yes	yes
anticomposition	no	no	yes	yes	yes	yes
equiv single change	no	no	no	no	yes	yes
equiv multiple change	no	no	no	no	yes	yes
equiv component	no	no	no	no	yes	yes

4. 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, and that an adequacy criterion must be satisfiable in a non-trivial way for arbitrary programs. 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 have 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 have been tested. This is due in part to the fact that programs may contain unexecutable code.

Having developed this system of axioms, we considered six previously defined adequacy criteria to see which of the axioms each satisfied. We found that the two best known of these criteria 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.

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. On the other hand, there is no reason to believe that they are at present a complete set of axioms in the sense of being satisfied by every intuitively appropriate adequacy notion, and only by such. 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. Eventually it should be possible to prove an appropriate completeness theorem for the system of axioms which is ultimately developed.

REFERENCES

[B] T.A. Budd, "Mutation Analysis: Ideas, Examples, Problems and Prospects," in Computer Program Testing, Chandrasekaran and Radicchi, eds., North-Holland, New York, pp.129-148.

[BDLS] T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs," Proc. 7th Annual Symp. on Principles of Programming Languages, Las Vegas, 19 80, pp.220-233.

[C] N. Chapin,"A Measure of Software Complexity", Proc. of the 1979 National Computer Conference, New York, pp.995-1002.

[DAW] M.D. Davis and E.J. Weyuker, "A Formal Notion of Program-Based Test Data Adequacy," Inf. and Control, to appear.

[DLS] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", Computer, 11(4), April 1978, pp.34-41.

[E] J.L. Elshoff, "An Investigation into the Effects of the Counting Method Used on Software Science Measurements," SIGPLAN Notices, Vol.13, No.2, Feb 1978, pp.30-45.

[EM] J.L. Elshoff and M. Marcotty, "On the Use of the Cyclomatic Number to Measure Program Complexity," SIGPLAN Notices, Vol.13, No.12, Dec 1978, pp.29-40.

[G] T. Gilb, Software Metrics, Winthrop, Cambridge, Ma., 1977.

[HAL75] M.H. Halstead, "Toward a Theoretical Basis for Estimating Programming Effort," Proc. of ACM 1975, New York, pp.222-224.

[HAL77] M.H. Halstead, Elements of Software Science, Elsevier North-Holland, New York, 1977.

[HAM] R.G. Hamlet, "Testing Programs with the Aid of a Compiler," IEEE Trans. Software Eng., Vol.SE-3, July 1977, pp.279-290.

[HAN] W.J. Hansen, "Measurement of Program Complexity by the Pair (Cyclomatic Number,Operator Count), SIGPLAN Notices, Vol.13, No.3, March 1978, pp.29-33.

[LK] J.W. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy," IEEE Trans. Software Eng., Vol.SE-9, No.3, May 1983, pp.347-354.

[MC] T.J. McCabe, "A Complexity Measure," IEEE Trans. Software Eng., Vol.SE-2, No.4, Dec 1976, pp.308-320.

[MI] H.D. Mills, "The Complexity of Programs," in Program Test Methods, W.C. Hetzel, ed, 1973.

[MY77] G.J. Myers, "An Extension to the Cyclomatic Measure of Program Complexity," SIGPLAN Notices, Vol.12, No.10, Oct 1977, pp.61-64.

[MY79] G.J. Myers, The Art of Software Testing, John Wiley and Sons, New York, 1979.

[N] S. Ntafos, "On Testing With Required Elements," U. Texas at Dallas Technical Report 90, July 1981.

[RW] S. Rapps and E.J. Weyuker, "Data Flow Analysis Techniques For Test Data Selection," Proc. 6th International Conference on Software Engineering, Tokyo, Japan, September 1982, pp.272-278.

[WEY] E.J. Weyuker, "The Applicability of Program Schema Results to Programs," Int. J. Comput. Inf. Sci., Volume 8, Number 5, Oct. 1979, pp.387-403.

[WHH] M.R. Woodward, M.A. Hennell, and D. Hedley, "A Measure of Control Flow Complexity in Program Text," IEEE Trans. Software Eng., Vol.SE-5, No.1, Jan 1979, pp.45-50.