



THE EVALUATION OF PROGRAM-BASED SOFTWARE TEST DATA ADEQUACY CRITERIA

In earlier work, a preliminary set of axioms for software test data adequacy was introduced in order to formalize properties which should be satisfied by any good program-based adequacy criterion. Here, we extend this work by augmenting the set with additional axioms which substantially strengthen the set. In doing so, we rule out several types of unsuitable notions of adequacy.

ELAINE J. WEYUKER

For much of the brief history of computers, software systems consisted of a few thousand lines of code produced by an individual in a short period of time. Under such circumstances, the use of ad hoc techniques was perhaps acceptable. Today, however, software systems of tens of thousands or even millions of lines are commonplace. Such systems are produced by large teams of developers, frequently situated in several locations, over a period of years. These realities of modern software production demand that software engineers pay more attention to formalizing the production of software, including the definition of accurate models of the phases of software production and an increased use of formal criteria to evaluate these models.

In this article we investigate fundamental properties for program-based test data adequacy criteria. In particular, we need to be able to recognize a good adequacy criterion when we see one, and we need to be able to recognize that a proposed adequacy criterion is a poor choice. Furthermore, we must have an objective, well-defined basis for this assessment.

Several recent papers reflect the growing awareness of the importance of this type of software engineering theory. Hoare et al. [10] presented a set of algebraic

laws, or axioms, for a programming language. In [17], we presented properties for software complexity measures and evaluated several well-known measures based on their satisfaction of the properties. Similar research was described by Iannino et al. in [11] for software reliability models.

We describe how we have extended our earlier work which contained an axiomatization of program-based software test data adequacy criteria [16]. The philosophy behind this work is that software testing is more than just the selection of test data and the execution of the software on that test set. We need to evaluate test data by using adequacy criteria and assess proposed criteria.

Like many axiomatic theories, this work is intended to make explicit our intuition; in this case about the testing process. Thus, each of the properties presented grows out of our experience with practical testing and our observations about the strengths and weaknesses of proposed test data adequacy criteria, and therefore should not be surprising to the practicing tester.

DEFINITIONS

Here we present necessary definitions and assumptions. We assume a structured programming language in which programs are single-entry/single-exit, that all input statements appear at the start of the program, and all output statements appear at the end. With these assumptions, we can easily define a notion of composi-

This research was supported in part by the National Science Foundation under Grant CCR-85-01614 and by the Office of Naval Research under Contract N00014-85-K-0414.

tion of programs. For programs P and Q using the same set of identifiers, we write $P;Q$ to mean the program formed by replacing P 's unique exit and output statements by Q with Q 's input statements deleted. A *component* of a program P is any contiguous sequence of statements of P .

Ideally, a specification is a total function which describes an intended behavior of the program for every possible input. In practice, a *specification* S is a partial function whose *domain* is the set of all values for which S is defined. The specification defines what a program is *intended* to compute for all elements of its domain. Values not included in the specification's domain are considered "don't care" conditions, and any output or no output is acceptable for such inputs.

The *domain of a program* is the set of all values for which the program halts. In theory, of course, one cannot determine the set of values for which the program halts, or the function being computed by the program. For these reasons and since the specification defines what *should* be computed, test cases are selected from the specification's domain. There is not much point in selecting as a test case for a program, an input for which any output or no output is acceptable. This is the case for points outside the specification's domain. Ideally, the specification describes behavior for every possible input and hence any such input could be used as a test case.

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 that 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 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, if $P = Q$, then for each x , $P(x)$ is defined if and only if $Q(x)$ is defined, and hence P and Q have the same domain.

A *test data adequacy criterion* is a set of rules used to determine whether or not testing can be terminated. Such a criterion may involve the program's structure. In this case, the adequacy criterion is said to be *program-based*. Such criteria are generally used during unit testing when relatively small program pieces are tested. Other criteria may ignore the program itself and rely solely on the specification. These criteria are said to be *specification-based*. Still other criteria may ignore both the program and the specification in the selection or evaluation of test data. An example of such a criterion is random selection.

Although in industrial settings, program-based adequacy criteria are rarely used, it is generally acknowledged that when such a criterion is employed, it is likely to be either statement or branch adequacy. If a program is represented by a flowchart, then a branch is an edge of the flowchart. Test set T is *statement (branch) adequate* for P , provided for every statement (branch) of P , there is some test case in T which causes the statement (branch) to be exercised. They are both clearly examples of program-based adequacy criteria.

PROPERTIES TO ASSESS PROGRAM-BASED ADEQUACY CRITERIA

The properties, or axioms, proposed in [16] were intended to be used as the basis for assessing program-based adequacy criteria. Our ultimate goal is to define a set of properties that would be satisfied by all and only "good" program-based test data adequacy criteria and in that sense would truly characterize such criteria. In practice, however, a tester might well decide to employ an adequacy criterion in spite of the fact it fails to satisfy certain of the desirable properties. It may be that even though the criterion has some known deficiencies, its strengths nonetheless make it especially appealing for the particular circumstances.

For example, it was shown that statement and branch adequacy fail to satisfy three of the original eight properties. Nonetheless, a tester might choose to use these criteria in spite of these limitations because there is a statement or branch coverage tool available in-house which is relatively easy and inexpensive to use. If the alternative to using a less than ideal criterion is that no adequacy criterion will be used, then that might well be a compelling argument for a criterion's use, even though it has known deficiencies.

Since we presented a complete discussion of the importance of each property in earlier work, we omit arguments here for the rationale behind these properties. In each instance, when we refer to an "adequate test set" or a program being "adequately tested by" a test set, that adequacy is being assessed using some fixed adequacy criterion.

We now present the eight properties. The first four properties apply equally to any adequacy criterion; the later properties are specialized to program-based criteria. As mentioned earlier, the intent of these properties is to formalize the intuition we have gained by testing "real" systems.

The first property we present is in a sense the central property for *any* adequacy criterion. It requires that, as assessed by a given adequacy criterion, every program must be adequately testable.

1. APPLICABILITY PROPERTY. For every program, there exists an adequate test set.

There are many ways that this property could be refined. We could require that the language contain only finitely-many representable points. In that case this property can be rephrased as:

Applicability Property. For every program, there exists a *finite* adequate test set.

Of course, even when test sets are guaranteed to be finite, there can be a very large number of required test cases. An additional refinement might be to require that test sets be "reasonably sized," as assessed by the program's size or complexity.

Another possible refinement of the Applicability Property might be to consider adequacy criteria which are only applicable for restricted domains. For example,

one might be able to define a meaningful adequacy criterion which applied specifically to graphics programs, but was not useful for other types of programs. Thus, if one could precisely define a class of programs for which it was algorithmically possible to determine whether or not a given program was a member of that class, a less demanding property might be proposed as follows:

Relativized Applicability Property. Let R be a class of programs and A be an adequacy criterion. For every program in R there exists an A -adequate test set.

Exhaustive testing of a substantial size domain would generally be prohibitively expensive. Testing is designed explicitly to address this issue.

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 surely be adequate no matter what criterion is used since there *can* be no additional testing possible. Exhaustive testing of a substantial size domain, however, would generally be prohibitively expensive. Testing is designed explicitly to address this issue. It is inherently a process in which a relatively small set (the test set) is selected in such a way that it is a reasonable approximation to a much larger set (the domain). Thus, although an adequacy criterion may well require exhaustive testing in some cases, particularly when the domain is small, a criterion which always requires exhaustive testing is unacceptable. Formalizing this we have:

2. NON-EXHAUSTIVE APPLICABILITY PROPERTY. 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.

The next property states that once a program has been adequately tested, running some additional tests cannot cause the program to be deemed inadequately tested.

3. MONOTONICITY PROPERTY. If T is adequate for P , and $T \subseteq T'$ then T' is adequate for P .

It follows immediately from Properties 1 and 3 that an exhaustive test set is always adequate, as desired.

The next property reflects the fact that a test data adequacy criterion is intended to measure how well the *testing process* has been performed. Certainly, if a program has not been tested at all, the process cannot have been thorough, and the program cannot be considered adequately tested.

4. INADEQUATE EMPTY SET PROPERTY. The empty set is not an adequate test set for any program.

5. ANTIEXTENSIONALITY PROPERTY. There are programs P and Q such that $P \equiv Q$, T is adequate for P , but T is not adequate for Q .

The antiextensionality property reflects the fact that we are assessing *program-based* adequacy criteria, not adequacy criteria in general. It says the semantic equality of two programs is not sufficient to imply that they should necessarily be tested the same way. Program-based testing should depend upon the implementation, and not simply the functions computed by the program.

To introduce a notion of syntactic closeness, we shall say that two programs are the *same shape* if one can be transformed into the other by applying the following rules any number of times:

- (a) Replace relational operator r_1 in a predicate with relational operator r_2 .
- (b) Replace constant c_1 in a predicate or assignment statement with constant c_2 .
- (c) Replace arithmetic operator a_1 in an assignment statement with arithmetic operator a_2 .

6. GENERAL MULTIPLE CHANGE PROPERTY.

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 .

Just as the antiextensionality property said that semantic "closeness" (equality) is not sufficient to imply that two programs should necessarily be tested the same way, this change property states that the syntactic closeness of two programs is not sufficient to imply that they should necessarily be tested the same way either.

7. ANTIDECOMPOSITION PROPERTY. 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 in T , and T' is not adequate for Q .

This antidecomposition property may at first appear to be somewhat counterintuitive. It states that although a program has been adequately tested, it does not necessarily imply that each of its component pieces has been properly tested. That is, a routine which has been adequately tested in some environment or context has not necessarily been adequately tested for other environments. Furthermore, even though P appears to be more complicated than Q , in the sense that P syntactically contains Q , semantically Q may actually be more complex than P . For example, if Q lies on an unexecutable path of P , then even if T is a test set which is adequate for P , T' , which in this case would be the

empty set, would presumably not be adequate for Q. Similarly, even if a component Q were executable in P, it may only be executable with data in a very restricted form which the criterion would not assess as adequate for Q.

8. ANTICOMPOSITION PROPERTY. There exist programs P and Q, and test set T, such that T is adequate for P, and the set of vectors of values that variables can assume on entrance to Q for inputs in T is adequate for Q, but T is not adequate for P; Q.

This final proposed property states that testing each piece of a program in isolation is not necessarily sufficient to deem the entire program adequately tested. Such a scheme fails to take into account the added interactions and interfaces which must be tested when programs are composed.

THE INSUFFICIENCY OF THE PROPERTIES

In [16] we demonstrated that the preceding eight properties are useful in exposing weaknesses in several well-known program-based adequacy criteria. For example, it was shown that the statement and branch adequacy criteria fail to satisfy the Applicability Property. A criterion which fails to satisfy this property is fundamentally unsatisfactory as an adequacy criterion since after evaluating the testing process and determining that the criterion has not been satisfied (for example, only 50 percent of the statements have been executed) one cannot algorithmically determine whether more testing must be performed, or the criterion is simply not satisfiable for this program (for example, it contains a substantial amount of unexecutable code.) It was also shown that statement and branch adequacy fail to satisfy the Antidecomposition and Anticomposition properties. In essence, the Antidecomposition Property rules out criteria that do not recognize that the context of a piece of code may well determine what testing is appropriate. The Anticomposition Property eliminates criteria that do not have provision for testing the interaction of program pieces.

Rapps and Weyuker [13] have defined a family of adequacy criteria, known as the data flow testing criteria, most of which are more demanding than branch testing. For this family of adequacy criteria, test data must be selected so as to cause the execution of paths connecting various program locations at which a variable is given a value (called a definition) and places where the selected variable definition may subsequently be used. Each of these criteria require that some or all path segments of a certain type be executed. Like branch testing, these criteria fail to satisfy the Applicability Property. In [7], Frankl and Weyuker have explored the properties of a family of criteria which are based on the data flow criteria, but which satisfy the Applicability Property.

In spite of the properties' usefulness in exposing

flaws in proposed adequacy criteria, we shall now demonstrate that, taken together, they are still weak. To do this, we define a criterion which satisfies each of the properties, but does not conform to one's intuition about what such an adequacy criterion should be like. We then introduce new properties aimed at eliminating such inappropriate criteria. Eventually we hope to have a complete set of axioms in the sense that they can be satisfied by all and only adequacy criteria that conform to one's intuitive ideas of program-based test data adequacy.

A *Gödel numbering* is a way of assigning a unique numerical value to each program in such a way that the program can be algorithmically retrieved from this value [2]. Let P be a program and let p be its Gödel number. Then we shall say that T is *Gödel adequate* for P provided $p \in T$. That is, any test set is deemed adequate by this criterion provided it contains among its inputs, the program's number. Such a criterion does not require that test data be chosen in a way that has anything fundamental to do with either the program's syntax, semantics, or intended semantics (specifications). It should also be noted that regardless of how complex a program is, there will always be an adequate test set of size one.

Our intuition tells us that an "inessential" change in a program, such as changing the variables' names, should not change the test data required to adequately test the program.

It is easy to verify that each of the previously proposed properties is satisfied by this criterion. Since Gödel adequacy is clearly not a "good" or even an appropriate adequacy criterion, it helps elucidate some fundamental gaps in the set of properties. One prominent weakness is that although the properties state the (semantic) equivalence of two programs should not be sufficient to require that they be tested the same way (Antiextensionality Property) and that the syntactic closeness of two programs should not be sufficient either (General Multiple Change Property), we did not consider the case of two programs which are both semantically and syntactically the same. We shall call a program P a *renaming* of Q if P is identical to Q except that all instances of an identifier x_i of Q have been replaced in P by an identifier x_j , where x_j does not appear in Q, or if there exists a sequence $Q = P_1, P_2, \dots, P_n = P$ where P_{i+1} is a renaming of P_i for $i = 1, \dots, n - 1$.

Our intuition tells us that an "inessential" change in a program, such as changing the variables' names, should not change the test data required to adequately test the program. Two programs which are renamings

of each other are as close as two programs can be without being textually identical. Since they are essentially the same program, they should require the same test cases. We are therefore led to propose the next property:

RENAMING PROPERTY. Let P be a renaming of Q . Then T is adequate for P if and only if T is adequate for Q .

Clearly, Gödel adequacy does not satisfy this property since if P and Q are distinct programs with Gödel numbers respectively p and q , then $p \neq q$ even if P is a renaming of Q . By definition, $\{p\}$ is Gödel adequate for P , but is not Gödel adequate for Q . However, we shall now present a minor modification of this criterion, which, although just as intuitively inappropriate as Gödel adequacy, *does* satisfy the Renaming Property as well as the original eight properties. Thus, we see that this enlarged set of properties is not sufficient to eliminate all inappropriate criteria.

We first assume, without loss of generality, that there is some standard ordering of variables in the language, and an associated canonical representation of any program. For example, assume for a program involving k distinct variable names, that the canonical representation of the program uses the variable names $x_1, x_2, \dots, x_k$ and that x_1 is the first variable encountered in a sequential scan of the program, x_2 is the second distinct variable, etc. Then every program can be algorithmically converted to its canonical form. Clearly, if P is a renaming of Q , then Q is a renaming of P and P and Q have the same canonical form. Now if instead of assigning a unique number to each syntactically distinct program, we assigned the number of a program's canonical form to it and every other member of that class, we could call this the *Gödel-class number*. Therefore, if P and Q are renamings of each other, they have the same Gödel-class number. We shall then say that a test set T is *Gödel-class adequate* for a program P provided that $p \in T$, where p is P 's Gödel-class number. This criterion is essentially the same as Gödel adequacy, with the same lack of intuitive acceptability, but was defined in such a way as to assure that it satisfies the Renaming Property as well as the original eight properties.

We now consider other possible strengthenings of the set of properties. As previously noted, Gödel adequacy is essentially independent of both the program's semantics and syntax. Gödel-class adequacy addresses this problem to a small extent by grouping together programs whose syntax and semantics are essentially identical. Both criteria share a related weakness, however. Intuitively, as programs become more complex, they should require more testing. Both Gödel adequacy and Gödel-class adequacy ignore this fundamental insight. For both criteria, every program has a size one adequate test set, and every size one test set is Gödel and Gödel-class adequate for some program. We are there-

fore led to the following property which neither of these adequacy criteria satisfy:

COMPLEXITY PROPERTY. For every n , there is a program P , such that P is adequately tested by a size n test set, but not by any size $n - 1$ test set.

That is, for every program there are other programs that require more testing. As mentioned, neither Gödel or Gödel-class adequacy satisfies this property since in both cases every program is testable with a size one test set. However, by modifying these criteria slightly once again, we are able to define an equally inappropriate criterion which satisfies all of the proposed properties. Let p be program P 's Gödel-class number. Then we shall say that test set T is *Gödel-class-interval adequate*, provided $\{q \mid 1 \leq q \leq p\} \subseteq T$. Thus, a program with Gödel-class number n is adequately tested using this criterion, by a test set of size n , but by no test set of size $n - 1$. The criterion therefore satisfies the Complexity Property, and, it can be shown, the original eight properties as well as the Renaming Property. Once again, however, the test cases really have nothing to do with either the program's syntax or semantics.

The obvious question is how can there be a criterion which satisfies all of the proposed properties, even though it is obviously inappropriate? The answer is that we have not included among the properties, anything which *characterizes* what a program-based adequacy criterion is supposed to be. In a sense, it was tacitly assumed that we would only use these properties to assess something that was, in fact, a plausible adequacy criterion. In [16] we used the first eight properties to evaluate several program-based adequacy criteria. It allowed us to determine, in a concrete way, the strengths and weaknesses of each of these adequacy criteria.

Since our goal is to ultimately present a set of properties which are satisfiable by all and only good program-based adequacy criteria, we now consider the fundamental role of a program-based adequacy criterion. Such a criterion should assess the quality of testing by determining whether or not a set of test data has completely exercised a given program or the extent to which this has been done. What a particular program-based adequacy criterion defines, then, is how the word "exercises" is to be interpreted.

We will now consider various program-based adequacy criteria in light of this insight, and ultimately propose a property which captures this intuition.

Two well-known program-based adequacy criteria are statement and branch coverage. For these criteria, the interpretation of the term "exercised" is clear. What is required is that certain parts of the program code must be *executed*. Another class of program-based adequacy criteria are the data flow testing criteria [7, 13] mentioned earlier. Each of these criteria require that some or all path segments of a certain type be exercised. Once again, the interpretation of "exercised" is that certain parts of the code be executed.

A substantially different type of program-based adequacy criterion is known as mutation analysis [1, 5]. Using this criterion, 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 *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 "the same shape." 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 by Hamlet in [8]. For mutation adequacy, the word "exercised" includes executed. Suppose some statement of the program has never been executed by any test case. Then that statement could be replaced by a different statement without affecting the outcome of any test case, and hence the test data would not have distinguished P from this mutated program.

We will consider now one final family of program-based notions of test data adequacy towards our goal of determining a central property for program-based adequacy criteria. In [3] we defined an adequacy criterion, known as modified size adequacy, and in [16] we demonstrated that it satisfied the original eight properties. This criterion, which can be viewed as a theoretical generalization of mutation adequacy, requires that sufficient test data be included so that the program being tested is distinguished from certain inequivalent programs which are no longer than the tested program, using a simple syntactic measure of size. The underlying intuition behind this criterion is that ideally a test set should be able to distinguish a given program from *all* inequivalent programs. Since there are infinitely many such programs, such an ideal is not practically feasible. A finite approximation to this ideal is therefore necessary, and modified size adequacy represents one such finite approximation.

A further generalization of this adequacy criterion was proposed in [4]. Since the decision to restrict attention to programs which are no longer than the one being tested is somewhat ad hoc, we defined a family of program-based adequacy criteria which requires that the given program be distinguished by test data from inequivalent programs of distance no greater than some value for some suitable notion of distance.

In what sense do these criteria which assess adequacy in terms of whether or not a program has been distinguished from other inequivalent programs by test data, require that the program be completely exercised? Once again, for virtually any plausible notion of distance, it is required that every statement of the given program be executed. If that were not the case, then an unexecuted statement of the program could be replaced by a different one which renders the resulting program inequivalent to the original one. The resulting program, which could presumably be chosen to be within a small distance of the original one, would not be distinguished from the original by the test data, even though they are not equivalent.

Having looked at several fundamentally different types of program-based adequacy criteria, we see that one common pattern emerges. What is common to each of these adequacy criteria is the requirement that every statement must be executed. The philosophy underlying such a requirement is straightforward: if some portion of the program has never been executed, then that portion could be performing any arbitrarily wrong computation and testing would be unable to detect these faults.

We therefore propose:

STATEMENT COVERAGE PROPERTY. If T is adequate for P , then T causes every *executable* statement of P to be executed.

It should be noted that this property does not quite require that a criterion imply statement adequacy. If it did, such a criterion would also fail to satisfy the Applicability Property. Suppose P is a program containing some unexecutable code and that A is a program-based adequacy criterion. If the Statement Coverage Property required that *every* statement of P be traversed in order that a test set be deemed adequately tested using criterion A , then no test set would be adequate for P since no test set can cause the unexecutable statements to be executed. Thus A would fail to satisfy the Applicability Property. The importance of the Statement Coverage Property is that it requires test data to be included that relate to the program. This is in contrast to criteria such as those based on a program's Gödel number.

Practical experience has convinced us that human beings are quite good at determining whether or not code is executable.

In one of the earliest books containing a substantial discussion of testing, Myers [12] presents a set of fundamental principles of testing, or what he calls "testing axioms." His second principle is:

One of the most difficult problems in testing is knowing when to stop.

He subsequently elaborates on this point and states: "One basic criterion for a set of test cases is ensuring that they cause every instruction in the module to be executed at least once. The criterion is certainly necessary, but it is not the place to stop."

Myers's perspective throughout this book is to present a practical, non-theoretical, discussion of software reliability in general, and software testing in particular. Since his viewpoint, which is based primarily on pragmatic experience, is substantially different from ours, it is encouraging to note that the same conclusion is drawn.

We agree that statement adequacy "is not the place to stop," and argue that it is a poor criterion since it does not satisfy the Applicability, Antidecomposition, and Anticomposition properties. This points out why it is not a sufficient criterion. Still, we agree with Myers that it is a necessary criterion, and believe that we have argued this point persuasively. If a test set does not satisfy statement adequacy for a program, some portion of the program has never been executed, and therefore that portion could contain arbitrary faults that would go undetected.

One final comment of a theoretical nature is in order about the Statement Coverage Property. It requires that the tester be able to determine which statements of a program are executable. But it is a well-known theoretical result that there can be no algorithm to determine whether or not a particular statement of a program is executable [2]. Nonetheless, practical experience has convinced us that human beings are, in fact, quite good at determining whether or not code is executable. In empirical experiments using our data flow testing tool ASSET [18], for example, this has become apparent.

A NON-PROGRAM-BASED ADEQUACY CRITERION

In this section we investigate one additional adequacy criterion. Our intent is to show that a criterion which is *not* program-based does not satisfy the properties which describe program-based criteria, but does satisfy the more general properties. We shall say that a program has been *k-adequately tested* provided it has been tested on at least $k > 0$ points. Then any test set of size at least k is *k-adequate* for any program. *k-adequacy* is an ad hoc form of testing which can be thought of as an extremely crude model of random testing. Although it might be argued that it is a plausible notion of test data adequacy, it is certainly *not* a program-based criterion. Therefore, we examine which of the proposed properties this criterion satisfies.

It is easy to verify that *k-adequacy* satisfies the Applicability, Non-Exhaustive Applicability, Monotonicity, and Inadequate Empty Set properties. This is not surprising since all of these properties represent characteristics of adequacy criteria in general, not necessarily program-based ones.

Antiextensionality emphasizes that for a program-based adequacy criterion, it is the *implementation* which serves as the basis of evaluation, not the function being computed. Thus, as the implementation changes, even if the same function is computed, different test data may be required to satisfy a given program-based adequacy criterion. Therefore, *k-adequacy* fails to satisfy the Anti-Extensionality Property.

If T is *k-adequate* for any program, then it is *k-adequate* for every other program. It thus follows that *k-adequacy* also fails to satisfy the General Multiple Change Property, and the Anticomposition Property. *k-adequacy* does satisfy the Antidecomposition Property because if T is a test set of size k , and V is the set of values variables can assume on entrance to Q , it is possible that V contains fewer than k -elements.

The Renaming Property is also satisfied by *k-adequacy* since any two programs are adequately tested by any size k test set, not only semantically and syntactically close programs. On the other hand, it does not satisfy the Complexity Property since all programs require the same amount of testing using the *k-adequacy* criterion.

Finally, it clearly does not satisfy the Statement Coverage Property. This is as it should be since, as we have argued, Statement Coverage is in a sense the fundamental property of program-based test data adequacy criteria. Thus, it should not only preclude notions which are inappropriate to serve as test data adequacy criteria, as it does for the Gödel number criteria, but it should also eliminate plausible adequacy criteria which, like *k-adequacy*, are *not* program-based.

SUMMARY AND FUTURE DIRECTIONS

In [16] a preliminary set of axioms or properties for software test data adequacy was introduced. The purpose of that investigation was to abstract and formalize properties which should be satisfied by any "good" program-based test data adequacy criterion. In this way we are able to assess proposed models of program-based adequacy in a concrete way. We extended this work in this article. We showed that even though the properties were useful in assessing the strengths and weaknesses of proposed program-based adequacy criteria, they were still not complete in the sense that they could all be simultaneously satisfied by entirely unsuitable adequacy criteria.

We then added three new properties which substantially strengthen the set and, in particular, rule out these unsuitable notions. We intend to continue this formal investigation of software test data adequacy criteria, adding and modifying properties when necessary until it is possible to prove a type of completeness theorem that says, in effect, that the only notions of program-based test data adequacy that can satisfy all of the properties are appropriate ones, and conversely, that any "good" notion satisfies the entire set of properties. At that point, we will have a clear understanding of this important part of the software process, and have a formal means of evaluating newly proposed models. The properties should also serve as a foundation for the definition of "good" program-based adequacy criteria.

REFERENCES

Note: References [6], [9], [14], and [15] are not cited in text.

1. Budd, T.A. "Mutation Analysis: Ideas, Examples, Problems and Prospects." In *Computer Program Testing*, Chandrasekaran and Radicchi, 129-148. North-Holland, New York, 1981.
2. Davis, M.D. and Weyuker, E.J. *Computability, Complexity, and Languages*. Academic Press, New York, 1983.
3. Davis, M.D. and Weyuker, E.J. A formal notion of program-based test data adequacy. *Inf. and Control*, 56, 1-2 (Jan.-Feb. 1983), 52-71.
4. Davis, M.D. and Weyuker, E.J. Metric space based test data adequacy criteria. *The Computer Journal*, 30, 4 (1987), 17-24.
5. DeMillo, R.A., Lipton, R.J. and Sayward, F.G. Hints on test data selection: Help for the practicing programmer. *Computer*, 11, 4 (Apr. 1978), 34-41.
6. Frankl, P.G. "The Use of Data Flow Information for the Selection and Evaluation of Software Test Data." Ph.D. diss., New York University, 1987.

7. Frankl, P.G. and Weyuker, E.J. Data flow testing in the presence of unexecutable paths. In *Proc. Workshop on Software Testing* (Banff, Alberta, Canada, July 15-17, 1986), 4-13.
8. Hamlet, R.G. Testing programs with the aid of a computer. *IEEE Trans. Software Eng.*, SE-3, 4 (July 1977), 279-290.
9. Hamlet, R.G. Reliability theory of program testing. *Acta Informatica*, 16, (1981), 31-43.
10. Hoare, C.A.R., Hayes, I.J., Jifeng, He, Morgan, C.C., Roscoe, A.W., Sanders, J.W., Sorensen, I.H., Spivey, J.M. and Sufrin, B.A. Laws of programming. *Comm. ACM*, 30, 8 (Aug. 1987), 672-686.
11. Iannino, A., Musa, J.D., Okumoto, K. and Littlewood, B. Criteria for software reliability model comparisons. *IEEE Trans. Software Eng.*, SE-10, 6 (Nov. 1984), 687-691.
12. Myers, G.J. *Software Reliability, Principles and Practices*. John Wiley & Sons, New York, 1976.
13. Rapps, S. and Weyuker, E.J. Selecting software test data using data flow information. *IEEE Trans. Software Eng.*, SE-11, 4 (April 1985), 367-375.
14. Weyuker, E.J. The applicability of program schema results to programs. *Int. J. Computer and Information Sci.*, 8, 5 (Nov. 1979), 387-403.
15. Weyuker, E.J. and Ostrand, T.J. Theories of program testing and the application of revealing subdomains. *IEEE Trans. Software Eng.*, SE-6, 3 (May 1980), 236-246.
16. Weyuker, E.J. Axiomatizing software test data adequacy. *IEEE Trans. Software Eng.*, SE-12, 12 (Dec. 1986), 1128-1138.
17. Weyuker, E.J. Evaluating software complexity measures. *IEEE Trans. Software Eng.*, to be published.
18. Weyuker, E.J. An empirical study of the complexity of data flow testing. In *Proc. Second Workshop Testing, Verification and Analysis* (Banff, Alberta, Canada, July 19-21, 1988), to appear.

CR Categories and Subject Descriptors: D.2.5 [Testing and Debugging]: F.3.1 [Specifying and Verifying and Reasoning about Programs]: Specification techniques

Additional Key Words and Phrases: Software testing, software validation, test data, adequacy, axiomatic evaluation

Author's Present Address: Elaine J. Weyuker, Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer St., New York, N.Y. 10012.

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

ACM SPECIAL INTEREST GROUPS

ARE YOUR TECHNICAL INTERESTS HERE?

The ACM Special Interest Groups further the advancement of computer science and practice in many specialized areas. Members of each SIG receive as one of their benefits a periodical exclusively devoted to the special interest. The following are the publications that are available—through membership or special subscription.

SIGACT NEWS (Automata and Computability Theory)

SIGAda Letters (Ada)

SIGAPL Quote Quad (APL)

SIGARCH Computer Architecture News (Architecture of Computer Systems)

SIGART Newsletter (Artificial Intelligence)

SIGBDP DATABASE (Business Data Processing)

SIGBIO Newsletter (Biomedical Computing)

SIGCAPH Newsletter (Computers and the Physically Handicapped) Print Edition

SIGCAPH Newsletter, Cassette Edition

SIGCAPH Newsletter, Print and Cassette Editions

SIGCAS Newsletter (Computers and Society)

SIGCHI Bulletin (Computer and Human Interaction)

SIGCOMM Computer Communication Review (Data Communication)

SIGCPR Newsletter (Computer Personnel Research)

SIGCSE Bulletin (Computer Science Education)

SIGCUE Bulletin (Computer Uses in Education)

SIGDA Newsletter (Design Automation)

SIGDOC Asterisk (Systems Documentation)

SIGGRAPH Computer Graphics (Computer Graphics)

SIGIR Forum (Information Retrieval)

SIGMETRICS Performance Evaluation Review (Measurement and Evaluation)

SIGMICRO Newsletter (Microprogramming)

SIGMOD Record (Management of Data)

SIGNUM Newsletter (Numerical Mathematics)

SIGOIS Newsletter (Office Information Systems)

SIGOPS Operating Systems Review (Operating Systems)

SIGPLAN Notices (Programming Languages)

SIGPLAN FORTRAN FORUM (FORTRAN)

SIGPLAN LISP Pointers

SIGSAC Newsletter (Security, Audit, and Control)

SIGSAM Bulletin (Symbolic and Algebraic Manipulation)

SIGSIM Simuletter (Simulation and Modeling)

SIGSMALL/PC Newsletter (Small and Personal Computing Systems and Applications)

SIGSOFT Software Engineering Notes (Software Engineering)

SIGUCCS Newsletter (University and College Computing Services)