Provable Improvements on Branch Testing

Phyllis G. Frankl * Elaine J. Weyuker[†]

Keywords: fault detection, program testing, software testing data flow testing, mutation testing, condition-coverage testing

Abstract

This paper compares the fault-detecting ability of several software test data adequacy criteria. It has previously been shown that if C_1 *properly covers* C_2, then C_1 is guaranteed to be better at detecting faults than C_2, in the following sense: a test suite selected by independent random selection of one test case from each subdomain induced by C_1 is at least as likely to detect a fault as a test suite similarly selected using C_2. In contrast, if C_1 *subsumes* but does not properly cover C_2, this is not necessarily the case. These results are used to compare a number of criteria, including several that have been proposed as stronger alternatives to branch testing. We compare the relative fault-detecting ability of data flow testing, mutation testing, and the condition-coverage techniques, to branch testing, showing that most of the criteria examined are guaranteed to be better than branch testing according to two probabilistic measures. We also show that there are criteria that can sometimes be poorer at detecting faults than substantially less expensive criteria.

1 Introduction

Although a large number of software testing techniques have been proposed during the last two decades, there has been surprisingly little concrete information about how effectively they detect faults. This is true both in the case of theoretical analyses and

*Author's address: Computer Science Dept., Polytechnic University, 6 Metrotech, Brooklyn, N.Y. 11201. Supported in part by NSF Grant CCR-9206910 and by the New York State Science and Technology Foundation Center for Advanced Technology program.

[†]Author's address: Computer Science Dept., Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012. Supported in part by NSF grant CCR-8920701 and by NASA grant NAG-1-1238.

empirical studies. In fact there are not even generally agreed-upon notions of what it means for one testing strategy to be more effective than another.

Most previous comparisons of software testing criteria have been based on the *subsumes* relation. Criterion C_1 subsumes criterion C_2 if for every program P, every test suite that satisfies C_1 also satisfies C_2. Unfortunately, it is not clear what the fact that C_1 subsumes C_2 tells us about their relative effectiveness. Weiss has argued that subsumption is more useful for comparing the cost of criteria than their effectiveness [22]. Hamlet [13] pointed out that it is possible for C_1 to subsume C_2, yet for some test suite that satisfies C_2 to detect a fault while some test suite that satisfies C_1 does not. Weyuker, Weiss, and Hamlet [24] further investigated the limitations of subsumption and other relations of that nature including Gourlay's *power* relation [12], and a newly proposed relation, the *"better"* relation.

In contrast, our approach is probabilistic. It is based on the fact that for a given program, specification, and criterion, there are typically a large number of test suites that satisfy a given test data adequacy criterion. Often, some of these suites will detect a fault, while others do not. For this reason, we will compare adequacy criteria by comparing the likelihood of detecting a fault and the expected number of faults detected when test suites for each criterion are selected in a particular way. This not only allows us to compare existing criteria in a concrete way, but also corresponds to a very reasonable intuitive notion of what it means for one criterion to be better at detecting faults than another.

In [8, 10] we explored the conditions under which one adequacy criterion is guaranteed to be at least as good as another according to a certain probabilistic measure of fault-detecting ability. That analysis was based on investigating how software testing criteria divide a program's input domain into subsets, or *subdomains*. We showed that it is possible for C_1 to subsume C_2 yet for C_2 to be better at detecting faults according to this measure. A simple example illustrates how this can happen. Assume the domain D of a program P is $\{0, 1, 2\}$, and 0 is the only input on which P fails. Assume that C_1 requires selection of a test case from the subdomain $\{0, 1\}$ and a test case from the subdomain $\{2\}$ and that C_2 requires selection of a test case from the subdomain $\{0, 1\}$ and a test case from the subdomain $\{0, 2\}$. Since every test suite that satisfies C_1 also satisfies C_2, C_1 subsumes C_2. Selecting one test case from each subdomain yields two possible C_1-adequate test suites, $\{0, 2\}$ and $\{1, 2\}$, only one of which will cause P to fail. There are four possible C_2-adequate test suites, $\{0, 0\}$, $\{0, 2\}$, $\{1, 0\}$, and $\{1, 2\}$, three of which will cause P to fail. Thus, a test suite selected to satisfy C_2 is more likely to detect a fault than one selected to satisfy C_1.

Clearly this situation occurred because the only input that caused a failure was a member of both subdomains of C_2, but was a member of only one subdomain of C_1. Furthermore, overlapping subdomains are a common occurrence in software testing; most testing criteria defined in the literature give rise to overlapping subdomains, often even for very simple programs. Thus, as the examples below illustrate, this phenomenon is not a mere theoretical curiosity, but one that occurs with real testing criteria and real programs.

As a response to this type of problem, we introduced a stronger relation between criteria, the *properly covers* relation, and proved that if C_1 properly covers C_2, then when one test case is independently randomly selected from each subdomain using a uniform distribution, the probability that C_1 will detect at least one fault is greater than or equal to the probability that C_2 will detect a fault [8, 10]. This is a powerful result provided that the model of testing used is a reasonable model of reality. We will address this issue in Section 2.1.

If we can show that C_1 properly covers C_2 for all programs in some class $\mathcal{P}$, then we will be *guaranteed* that C_1 is at least as good as C_2 (in the above sense) for testing *any* program P in $\mathcal{P}$, *regardless of the particular faults that occur in P.* On the other hand, if C_1 does not properly cover C_2 for some program P, then even if C_1 subsumes C_2, C_2 may be more likely than C_1 to detect a bug in P. In this paper, we use the above result to explore the relative fault-detecting ability of several well-known testing techniques. We also introduce another measure of fault-detecting ability and show that if C_1 properly covers C_2, then C_1 is also at least as good as C_2 according to this new measure.

Three families of techniques that have been widely investigated are data flow testing, mutation testing, and the condition-coverage techniques. We compare the relative fault-detecting ability of criteria in these families to branch testing, showing that most of the criteria examined are guaranteed to be better than branch testing according to the probabilistic measures mentioned above, but that there are criteria that can sometimes be poorer at detecting faults than substantially less expensive criteria.

2 Background and Terminology

A *multi-set* is a collection of objects in which duplicates may occur, or more formally, a mapping from a set of objects to the non-negative integers, indicating the number of occurrences of each object. We shall delimit multi-sets by curly braces and use set-theoretic operator symbols to denote the corresponding multi-set operators throughout. For a multi-set S_1 to be a *sub-multi-set* of multi-set S_2, there must be at least as many copies of each element of S_1 in S_2 as there are in S_1. In the sequel, when we say that some procedure is applied to each element of a multi-set $S = \{e_1, \ldots, e_n\}$, we will mean that the procedure is applied exactly n times: once to e_1, once to e_2, etc., without regard for whether $e_i = e_j$ for some i, j.

The input domain of a program is the set of possible inputs. We restrict attention to programs with finite input domains, but place no bound on the input domain size. Since real programs run on machines with finite word sizes and with finite amounts of memory, this is not an unrealistic restriction. A *test suite* is a multi-set of test cases, each of which is an element of the input domain. We investigate test *suites* rather than test *sets* because it is easier in practice to allow occasional duplication of test cases than to check for duplicates and eliminate them. A test data adequacy criterion is a relation $C \subseteq \text{Programs} \times \text{Specifications} \times \text{Test Suites}$ that is used to determine whether a given test suite T does a "thorough" job of testing program P for specification S. If $C(P, S, T)$ holds, we will say "T is adequate for testing P with respect to S according to C", or, more simply, "T is C-adequate for P and S". In addition to providing a means for evaluating test suites, adequacy criteria can serve as the basis for test selection strategies, as discussed below.

Many systematic approaches to testing are based on the idea of dividing the input domain of the program into subsets called *subdomains*, then requiring the test suite to include elements from each subdomain. The manner in which the input domain is subdivided may be based on the structure of the program being tested (program-based testing), the structure or semantics of its specification (specification-based testing), or some combination thereof. As a group, these techniques have generally been referred to as *partition testing strategies*, but in fact, most such strategies divide the input domain into overlapping subdomains, and thus do not form true partitions of the input domain. In this paper, we will refer to these strategies as *subdomain-based testing*.

More precisely, a testing criterion C is *subdomain-based* if, for each program P and specification S, there is a non-empty multi-set of subdomains, $\mathcal{SD}_C(P, S)$, such that C requires the selection of one or more test cases from each subdomain in $\mathcal{SD}_C(P, S)$. Note that one could define adequacy criteria that require the selection of at least k test-cases from each subdomain, for some $k > 1$. However, we restrict attention here to criteria that only require at least *one* test case per subdomain. This is not a serious limitation since virtually all criteria discussed in the literature only require the selection of at least *one* test case per subdomain. To model criteria that *do* explicitly require $k > 1$ test cases per subdomain, we could include k copies of each subdomain in $\mathcal{SD}_C(P, S)$ and select one test case from each copy. However, if this is done, the test selection strategy described below may select a test suite with less than k *distinct* test cases from a subdomain.

In general, $\mathcal{SD}_C(P, S)$ is a multi-set, rather than a set, because for some criteria it is possible for two different requirements to correspond to the same subdomain. For example, consider the all-statements criterion, in which each subdomain corresponds to the set of inputs that cause the execution of a particular statement in the program. If two different statements are executed by the same test cases, identical subdomains occur in the multi-set of subdomains. This can happen either because the structure of the flow graph dictates that every path covering one statement also covers another or because the semantics of the program force two seemingly independent statements to be traversed by exactly the same test cases. Of course, given a criterion C for which the multi-set $\mathcal{SD}_C(P, S)$ contains duplicates, one can define a new criterion C' in which the duplicates are eliminated, but, as we shall discuss below, it is frequently easier to keep the duplicates (and hence select "extra" test cases) than to check to see whether duplicates exist.

Note that since $\mathcal{SD}_C(P, S)$ is assumed to be non-empty and at least one test case must be chosen from each subdomain, the empty test suite is not C-adequate for any subdomain-based criterion. A subdomain-based criterion C is *applicable to (P,S)* if and only if there exists a test suite T such that $C(P, S, T)$ holds. C is *universally applicable* if it is applicable to (P, S) for every program, specification pair (P, S). Note that since the empty test suite is not C-adequate, C is applicable to (P, S) if and only if the empty subdomain is not an element of $\mathcal{SD}_C(P, S)$.

Throughout this paper all testing criteria discussed will be universally applicable subdomain-based criteria, unless otherwise noted. In fact, many of the criteria that have been defined and discussed in the software testing literature are *not* universally applicable [7]. For example, the all-statements criterion, which requires every statement in the program to be executed, is not applicable to any program that has a statement that cannot be exercised by any input. However, it is the universally applicable analogs of those criteria that are actually usable in practice. That is, the form of statement testing that is really used requires every *executable* statement to be exercised. Formally, these versions are obtained by removing the empty subdomains from $\mathcal{SD}_C(P, S)$. But determining whether a subdomain is empty is in general undecidable. However, in practice, it is often easy for testers to determine whether a subdomain is empty by inspecting the program code or specification. Another way that testers pragmatically deal with this issue is to make a tacit assumption that there is never more than some percentage x of unexecutable statements (or branches, or whatever the relevant program artifact being covered) and then require that $(100-x)\%$ of the statements be exercised. Of course, it is possible that more that x percent of the code is unexecutable, and then the tester is faced with the same problem. In any case, our analysis will formally assess only the universally applicable versions of any adequacy criterion. Note that the relationship between the universally applicable analogs of criteria may be different than that between the original criteria [7]. This will be discussed in Section 3.

Given a program P and a specification S, a *failure-causing input* t is one such that the output produced by P on input t does not agree with the specified output. We will say that a test suite T *detects a fault* in program P if T contains at least one failure-causing input. Note that we are not concerned here with determining the particular problem in P that caused the failure, only with determining that some problem exists. Intuitively, a good testing strategy is one that is likely to require the selection of one or more failure-causing inputs, if any exist.

2.1 The Model

There are many different ways to select a test suite that satisfies a given adequacy criterion. We would like to be able to compare the likelihoods that test suites that satisfy given adequacy criteria detect faults, without regard to how those test suites were selected. However this problem is too vague to analyze, since the likelihood that a C-adequate test suite detects a fault can only be defined with respect to a given probability distribution on the space of test suites satisfying C. Such distributions may be defined in some precise way, or may arise in practice from testers manually selecting test cases which they consider to be "natural". Since the notion of what is "natural" differs from one person to another, such distributions cannot be formally analyzed and are also extremely difficult to study empirically.

Thus, in order to carry out an analysis of fault-detecting ability, we need to use a test selection model that is well-defined, is not obviously biased in some criterion's favor, and is not too far from reality. With this in mind, we assume that the tester selects a test suite that satisfies a subdomain-based criterion C by first dividing the domain based on $\mathcal{SD}_C(P,S)$, and then for each $D_i \in \mathcal{SD}_C(P,S)$ randomly selecting an element of D_i. In the sequel, we let $d_i = |D_i|$, the size of subdomain D_i, let m_i be the number of failure-causing inputs in D_i and let

$$M(C,P,S) = 1 - \prod_{i=1}^{n}(1 - \frac{m_i}{d_i}).$$

Assuming one test case is independently selected from each subdomain according to a uniform distribution, M gives the probability that a test suite chosen using this test selection strategy will expose at least one fault. This measure has previously been investigated in [4, 10, 14, 23] and was called M_2 in [10]. Note that if $\mathcal{SD}_C(P,S)$ contains a duplicate subdomain $D_i = D_j$, this model requires independent selection of one test case from each copy of the subdomain.

In several earlier works that compared the effectiveness of testing criteria, [4, 14, 23] the test selection procedure was actually specified as part of the criterion, thereby obscuring the distinction between test selection and evaluation of test adequacy. For example, Duran and Ntafos [4] and Hamlet and Taylor [14] compare random testing and partition testing. For partition testing they speak of "dividing a program's input domain into classes whose points are somehow 'the same' " so that "it is sufficient to try one representative from each class"([14], p. 1402). Weyuker and Jeng [23] compared partition testing strategies and random testing using a generalized form of test data selection in which n_i test cases were independently selected from each subdomain D_i. Again the test case selection procedure was integrated into the testing criterion.

Note, that all of these papers based their assessments of effectiveness on the measure M. In addition, in all of these cases, it was assumed that the subdomain division was performed *before* any test cases were selected, and that selection from each subdomain was independent.

In contrast, in this paper we make the role of the selection criterion explicit. It could be argued that this test selection model does not accurately reflect testing practice because testers sometimes allow each test case to "count" toward all of the subdomains to which it belongs. We argue that in fact, our test selection model is not very far from reality in many cases. In particular, specification-based testing is frequently done by first determining the test requirements, and then selecting test cases for each requirement without regard for whether a test case also fulfills additional requirements. This is especially true when system testing is done by an independent testing group, and test cases are derived from the specification even before the implementation is complete. In addition, we expect that as automated test generation tools targeted to program-based testing techniques become more available, it will also become common to select test suites for these criteria in a manner similar to the above strategy. Manually crafting test cases is often far more costly than test case execution, so test suite size is an important issue under these circumstances, but when the generation is done automatically, it may well be easier to generate test cases for each test condition (subdomain) than to generate a test case for a condition, execute it to see which additional conditions have been inadvertently exercised, and remove them from consideration.

One further note: in practice, most proposed adequacy criteria are *monotonic*. This means that if a test suite is adequate for the criterion, then any test suite formed by adding test cases to the suite is also adequate. But by using our selection method, these "extra" test cases cannot be added. In a sense we are comparing test suites containing only elements *required* by the criterion.

2.2 Relations Among Testing Criteria

In [8, 10], we explored several relations R among subdomain-based criteria, asking whether $R(C_1, C_2)$ guarantees $M(C_1, P, S) \geq M(C_2, P, S)$. The most commonly used relation in the literature for comparing criteria is the subsumes relation. Recall that criterion C_1 *subsumes* criterion C_2 if and only if for every program P and specification S, every test suite that satisfies C_1 also satisfies C_2. We showed that the fact that C_1 subsumes C_2 *does not guarantee* that $M(C_1, P, S) \geq M(C_2, P, S)$. A simple example illustrating this appeared in Section 1, above. We therefore introduced a stronger relation among criteria, the *properly covers relation*, which is more relevant for comparing the fault-detecting abilities of criteria.

In order to explain the intuition motivating the properly covers relation and its relationship to subsumption, we first mention two weaker relations, the *narrows* and *covers* relations, also defined in [8, 10].

Definition: Let C_1 and C_2 be criteria. C_1 *narrows* C_2 *for (P,S)* if for every subdomain $D \in \mathcal{SD}_{C2}(P, S)$ there is a subdomain $D' \in \mathcal{SD}_{C1}(P, S)$ such that $D' \subseteq D$. C_1 *universally narrows* C_2 if for every program, specification pair (P, S), C_1 narrows C_2 for (P, S).

In [10], we showed that for criteria that require selection of at least *one* element from each subdomain (as opposed to explicitly requiring selection of k elements for some $k > 1$). C_1 subsumes C_2 if and only if C_1 universally narrows C_2. Thus, for all of the criteria considered in the current paper, the universally narrows relation is equivalent to subsumption.

Definition: Let C_1 and C_2 be criteria. C_1 *covers* C_2 *for (P,S)* if for every subdomain $D \in \mathcal{SD}_{C2}(P,S)$ there is a collection $\{D_1, \ldots, D_n\}$ of subdomains belonging to $\mathcal{SD}_{C1}(P,S)$ such that $D_1 \cup \ldots \cup D_n = D$. C_1 *universally covers* C_2 if for every program, specification pair (P,S), C_1 covers C_2 for (P,S).

In [10] we showed that various well-known criteria are related to one another by the covers relation. We then showed that it is possible for C_1 to cover C_2 for (P,S), and still have $M(C_1,P,S) < M(C_2,P,S)$. The problem arises when one subdomain of C_1 is used in covering two or more subdomains of C_2. We therefore introduced the properly covers relation which overcomes this problem.

Definition: Let $\mathcal{SD}_{C1}(P,S) = \{D_1^1, \ldots, D_m^1\}$, and let $\mathcal{SD}_{C2}(P,S) = \{D_1^2, \ldots, D_n^2\}$. C_1 *properly covers* C_2 *for (P,S)* if there is a multi-set

$$\mathcal{M} = \{D_{1,1}^1, \ldots, D_{1,k_1}^1, \ldots, D_{n,1}^1, \ldots, D_{n,k_n}^1\}$$

such that $\mathcal{M}$ is a sub-multi-set of $\mathcal{SD}_{C1}(P,S)$ and

$$\begin{aligned} D_1^2 &= D_{1,1}^1 \cup \ldots \cup D_{1,k_1}^1 \\ &\vdots \\ D_n^2 &= D_{n,1}^1 \cup \ldots \cup D_{n,k_n}^1 \end{aligned}$$

Note that the number of occurrences of any subdomain $D_{i,j}^1$ in $\mathcal{M}$ is less than or equal to the number of occurrences of that subdomain in the multi-set $\mathcal{SD}_{C1}$. In other words, C_1 properly covers C_2 if each of C_2's subdomains can be "covered" by C_1 subdomains (i.e., can be expressed as a union of some C_1 subdomains), and furthermore, this can be done in such a way that none of C_1's subdomains occurs more often in the covering than it does in $\mathcal{SD}_{C1}$. C_1 *universally properly covers* C_2 if for every program P and specification S, C_1 properly covers C_2 for (P,S).

The following example illustrates the properly covers relation.

Example 1:
Consider a program P whose input domain is $\{x | 1 \leq x \leq 10\}$. Let C_2 be the criterion whose subdomains for program P are $D_1^2 = \{x | 1 \leq x \leq 6\}$ and $D_2^2 = \{x | 4 \leq x \leq 10\}$. Let C_1 be the criterion whose subdomains are $D_1^1 = \{x | 1 \leq x < 4\}$, $D_2^1 = \{x | 3 \leq x \leq 6\}$, $D_3^1 = \{x | 3 \leq x < 8\}$, $D_4^1 = \{x | 4 \leq x \leq 8\}$, $D_5^1 = \{x | 6 \leq x \leq 10\}$. Then C_1 properly covers C_2 for P since $D_1^2 = D_1^1 \cup D_2^1$ and $D_2^2 = D_4^1 \cup D_5^1$. In this case $\mathcal{M} = \{D_{1,1}^1, D_{1,2}^1, D_{2,1}^1, D_{2,2}^1\}$ with $D_{1,1}^1 = D_1^1, D_{1,2}^1 = D_2^1, D_{2,1}^1 = D_4^1, D_{2,2}^1 = D_5^1$.

Also note that it is possible for one criterion to have more subdomains than another, without properly covering it. This is the case for the criterion C_1' obtained from C_1 by removing subdomains D_2^1. C_1' narrows but does not properly cover C_2. □

Observation 1 *For any (P,S), the narrows, covers, and properly covers relations are all transitive. If C_1 properly covers C_2, then C_1 covers C_2. If C_1 covers C_2 then C_1 narrows C_2. If $\mathcal{SD}_{C2}(P,S) \subseteq \mathcal{SD}_{C1}(P,S)$ then C_1 properly covers C_2 for (P,S).*

The following theorem is proven in [10]:

Theorem 1 *If C_1 properly covers C_2 for program P and specification S, then $M(C_1, P, S) \geq M(C_2, P, S)$.*

Thus, if we can show that C_1 universally properly covers C_2, then we are guaranteed that test suites chosen to satisfy C_1 (according to the above test selection strategy) are at least as likely to detect faults as those chosen to satisfy C_2.

Another reasonable measure of the fault-detecting ability of a criterion is the *expected number of failures detected.* Again, let $\mathcal{SD}_C(P, S) = \{D_1, \ldots D_n\}$. Assuming independent random selection of one test case from each subdomain, using a uniform distribution, this is given by

$$E(C, P, S) = \sum_{i=1}^{n} \frac{m_i}{d_i}.$$

We now show that if C_1 properly covers C_2 for a given program, specification pair, then C_1 is also guaranteed to do at least as well as C_2 according to the measure E. Examples 2 and 6, below, show that this is not necessarily the case if C_1 subsumes C_2, but does not properly cover C_2.

Theorem 2 *If C_1 properly covers C_2 for program P and specification S, then $E(C_1, P, S) \geq E(C_2, P, S)$.*

Proof:
We begin by noting that if $D = D_1 \cup \ldots \cup D_n$, then

$$\sum_{i=1}^{n} \frac{m_i}{d_i} \geq \sum_{i=1}^{n} \frac{m_i}{d} \geq \frac{m}{d} \tag{1}$$

where d is the size of D and m is the number of failure-causing inputs in D. Let $\mathcal{SD}_{C1}(P, S) = \{D_1^1, \ldots D_p^1\}$, let $\mathcal{SD}_{C2}(P, S) = \{D_1^2, \ldots D_n^2\}$ and let

$$\mathcal{M} = \{D_{1,1}^1, \ldots, D_{1,k_1}^1, \ldots, D_{n,1}^1, \ldots, D_{n,k_n}^1\}$$

be a sub-multi-set of $\mathcal{SD}_{C1}(P, S)$ such that

$$\begin{aligned} D_1^2 &= D_{1,1}^1 \cup \ldots \cup D_{1,k_1}^1 \\ &\vdots \\ D_n^2 &= D_{n,1}^1 \cup \ldots \cup D_{n,k_n}^1. \end{aligned}$$

For $k = 1, 2$ and for each i, let $d_i^{(k)}$ be the size of D_i^k and $m_i^{(k)}$ be the number of failure-causing inputs in D_i^k. For each i, j, let $d_{i,j}^{(1)}$ be the size of $D_{i,j}^1$ and $m_{i,j}^{(1)}$ be the number of failure-causing inputs in $D^1_{i,j}$. Then

$$\begin{aligned} E(C_2,P,S) &= \sum_{i=1}^{n} \frac{m_i^{(2)}}{d_i^{(2)}} && (2)\\ &\leq \sum_{i=1}^{n}\sum_{j=1}^{k_i} \frac{m_{i,j}^{(1)}}{d_{i,j}^{(1)}} && (3)\\ &\leq \sum_{i=1}^{p} \frac{m_i^{(1)}}{d_i^{(1)}} && (4)\\ &= E(C_1,P,S). && (5)\end{aligned}$$

Note that (3) follows from (1) and (2), since for each i, $D^2_i = D^1_{i,1} \cup \ldots \cup D^1_{i,k_i}$. Since $\mathcal{M}$ is a sub-multi-set of $\mathcal{SD}_{C1}(P,S)$, summation (4) involves all the summands in (3), and perhaps some additional ones. The result follows immediately from the fact that each summand is non-negative. $\square$

Theorems 1 and 2 can easily be generalized to selection strategies in which non-uniform distributions on subdomains are used, provided the distributions on overlapping subdomains satisfy a certain compatibility property. A number of strategies for using an arbitrary distribution on the entire input domain to induce such compatible distributions on the subdomains are discussed in [6].

2.3 Program Structure

Testing criteria that are based only on the structure of the program being tested are called *program-based* (or *structural* or *white-box*) techniques. For such criteria, the multi-set $\mathcal{SD}_{\mathcal{C}}(\mathcal{P},\mathcal{S})$ is independent of the specification. The criteria we consider in the remainder of this paper are all program-based. However, it is important to note that Theorems 1 and 2 hold for any subdomain-based criteria, regardless of the basis for the division of the domain. Thus, *specification-based* (or *functional* or *black-box*) subdomain-based criteria could also be compared using techniques similar to those used in this paper.

We sometimes represent a program by its *flow graph*, a single-entry, single-exit directed graph in which nodes represent sequences of statements or individual statements, and edges represent potential flow of control between nodes. A *path* from node n_1 to node n_k is any sequence $(n_1,\ldots,n_k)$ of nodes such that for each $i<k$, (n_i,n_{i+1}) is an edge. A *suffix* of a path $(n_1,\ldots,n_k)$ is a subpath $(n_j,\ldots,n_k)$ for some $j \geq 1$. A path is *feasible* if there exists some input that causes it to be executed and *infeasible* otherwise. A variable `v` has a *definition* in node n if n contains a statement in which `v` is assigned a value. Variable `v` has a *use* in node n if n contains a statement in which `v`'s value is fetched. A use of a variable occurring in the Boolean expression controlling a conditional or loop statement is sometimes associated with each of the edges leaving that node, and called a *predicate use* or *p-use*. A definition of `v` in node d *reaches* a use of `v` in node or

edge u if there is a *definition-clear path with respect to* `v` *from* d *to* u, i.e., a path from d to u along which `v` is not redefined.

Since the criteria we consider here are based on program structure, it is necessary to select a fixed language for the programs under test. For this reason, we will limit attention to programs written in Pascal. Our results do not depend in any essential way on this choice of language. We will also assume that every program has at least one conditional or repetitive statement, and that at least one variable occurs in every Boolean expression controlling a conditional or repetitive statement in the program. Note that this variable occurrence may be implicit, as in the use of the input file variable in the statement, `while not eof do S`.

If the first requirement is not satisfied, then every input traverses exactly the same path through the program. If the latter requirement is not fulfilled, the Boolean expression will always evaluate to `true` or will always evaluate to `false`, and the other branch will be unexecutable. Note that we *do not* require that programs be "well-structured" or `goto`-less. In contrast, for reasons described in Section 3 below, we did have this requirement in our earlier paper [10].

We also require programs to satisfy the *no feasible anomalies (NFA)* property: every feasible path from the start node to a use of a variable `v` must pass through a node having a definition of `v`. This is a reasonable property to require, since programs that do not satisfy this property have the possibility of referencing an undefined variable.[1] Although there is no algorithm to check whether or not the NFA property holds, it is easy to check the stronger *no anomalies* property, which requires that every path from the start node to a use of `v`, whether feasible or not, pass through a node having a definition of `v`. An algorithm for this is presented in [5]. It is also possible to enforce the no anomalies property by considering the entry node to have definitions of all variables.

Branch testing, also known as decision-coverage, is one of the most widely discussed subdomain-based criteria. A *decision* is a maximal Boolean expression controlling the execution of a conditional statement or loop. For example, in the statement `if (x = 1) and (y = 1) then S`, the Boolean expression `(x=1) and (y=1)` is a decision. In the *decision-coverage* criterion, there are two subdomains for each decision, one consisting of all inputs that cause it to evaluate to `true` at some point during execution and one consisting of all inputs that cause it to evaluate to `false` at some point during execution[2]. Note that these two subdomains are not necessarily disjoint since if the decision is within

[1]Clarke *et al.* [2] have pointed out that a program may legitimately have a feasible definition-clear path with respect to `v` from the start node to a call to procedure `Q` that defines reference parameter `v`. In such cases, the NFA property can be enforced by considering the argument `v` to be defined before it is used in the call to `Q`. Indeed, if this is not the actual data flow, then `Q` or `P` may attempt to reference an undefined variable.

[2]In [10] we investigated a different variant of branch testing, called *all-edges*, in which each edge in the program's flow graph gives rise to a subdomain. The distinction between all-edges and decision coverage is discussed in [9].

a loop, a single test case may cause the decision to evaluate to `true` on one iteration of the loop and to `false` on another iteration.

In the remainder of the paper, we use the universally properly covers relation to compare various criteria to the decision-coverage criterion and to each other. We thereby exhibit criteria that are guaranteed to be at least as good as decision-coverage according to the two measures of fault-detecting ability M and E. In each case, we compare criteria C_1 and C_2 where C_1 strictly subsumes C_2. It therefore follows that for these criteria, C_2 *does not* universally properly cover C_1. We thus focus attention on the question of whether or not C_1 universally properly covers C_2.

3 Data Flow Testing

.

Several of the criteria that have been proposed as more powerful alternatives to branch testing involve the use of data flow information. These criteria are based on data flow analysis, similar to that done by an optimizing compiler, and require that the test data exercise paths from points at which values are assigned to variables, to points at which those values are used. In this section we examine several data-flow based testing criteria and show that each universally properly covers the decision-coverage criterion, and thus can be viewed as being better at detecting faults than branch testing. We also compare various data flow testing criteria to one another.

The *all-uses* criterion [18, 19] requires that test data cover every definition-use association in the program, where a *definition-use association* is a triple $(d, u, \mathtt{v})$ such that d is a node in the program's flow graph in which variable `v` is defined, u is a node or edge in which `v` is used, and there is a definition-clear path with respect to `v` from d to u [3]. We will frequently refer to a definition-use association as an *association*. A test case t *covers* association $(d, u, \mathtt{v})$ if t causes a definition-clear path with respect to `v` from d to u to be executed. Similarly, the *all-p-uses* criterion [18, 19] is a restricted version of all-uses that requires that test data cover every association $(d, u, \mathtt{v})$ in which u is an edge with a p-use of variable `v`. Precise definitions of the criteria for a subset of Pascal similar to the one in question are given in [7].

Lemma 1 *All-uses universally properly covers all-p-uses. All-p-uses universally properly covers decision-coverage.*

Proof:
Since $\mathcal{SD}_{\text{all-p-uses}}(P, S) \subseteq \mathcal{SD}_{\text{all-uses}}(P, S)$, all-uses universally properly covers all-p-uses.

[3]Note that if u is an edge, (n, m), the covering path must be of the form, $(d, \ldots n, m)$, i.e. it must include both the head and the tail of the edge.

The proof that all-p-uses universally properly covers decision-coverage is similar to the proof in [10] that all-p-uses universally covers all-edges. Let P be a program, let d be a decision in P, and let D_d be the subdomain consisting of all inputs that cause decision d to evaluate to `true` (alternatively we could let D_d be the subdomain consisting of all inputs that cause decision d to evaluate to `false`). Let e be the edge that is executed if and only if d evaluates to `true` (or to `false`). Since we are only interested in the *feasible analog* of decision-coverage, we can assume that $D_d \neq \emptyset$, i.e. that e is feasible.

Let `v` be a variable occurring in decision d. Let $\delta_1, \ldots, \delta_n$ be the definitions of `v` for which there is a feasible definition-clear path with respect to `v` from δ_i to e. For each $i, 1 \leq i \leq n$, let D_i be the subdomain $\{t | t \text{ covers association } (\delta_i, e, \mathtt{v})\}$. Recall that we are limiting attention to programs that satisfy the NFA property. Thus, every feasible path from the start node to e passes through at least one of the δ_i, so $D_d \subseteq D_1 \cup \ldots \cup D_n$. Since any test case that exercises one of the $(\delta_i, e, \mathtt{v})$ must exercise edge e, $D_1 \cup \ldots \cup D_n \subseteq D_d$. Since each outcome of each decision in P gives rise to a distinct set of associations, all-p-uses universally properly covers decision-coverage.[4] $\square$

We next consider Ntafos' *required k-tuples criteria* [17]. These criteria require the execution of paths going from a variable definition to a use that is influenced by the definition, via a chain of intervening definitions and uses. A *k-dr interaction* is a sequence $X_1, \ldots, X_{k-1}$ of variables along with a sequence $s_1, \ldots, s_k$ of distinct statements such that variable X_i is defined in s_i, used in s_{i+1}, and there is a definition-clear path with respect to X_i from s_i to s_{i+1}. Note that the value assigned to X_1 in s_1 can influence the value of X_{k-1} which is used at s_k. The *required k-tuples criterion* requires that each k-dr interaction be exercised, i.e., that a path $s_1 p_1 \ldots s_{k-1} p_{k-1} s_k$ be executed where p_i is a definition-clear path with respect to X_i. Certain requirements based on control flow are also included.

Clarke *et al.* [2] pointed out certain technical problems with the original definition and defined the *required k-tuples+* criterion by making the following two modifications:

1. all l-dr interactions must be exercised for $l \leq k$, and

2. the statements s_i in the path need not be distinct.

They showed that required k-tuples+ subsumes required (k$-$1)-tuples+, and that required 2-tuples+ subsumes all-uses. Without modification (1), required k-tuples fails to subsume required (k$-$1)-tuples, and without modification (2), required 2-tuples fails to subsume all-uses.

[4] In [10] we showed that all-p-uses universally covers, but does not universally properly cover all-edges. In order to do so, we restricted attention to programs with no `goto` statements. Both the failure to universally properly cover and the need to restrict the class of programs arose due to edges in P's flow graph that did not have any p-use. Such edges are an artifact of the conventions we used for building flow graphs, not a fundamental aspect of program structure. When we consider the decision-coverage version of branch testing rather than the all-edges version, these extraneous edges cease to exist, and the problems disappear.

Lemma 2 *For all $k \geq 2$, the required (k+1)-tuples+ criterion universally properly covers the required k-tuples+ criterion. The required 2-tuples+ criterion universally properly covers all-uses.*

Proof:
This follows immediately from the fact that for $k \geq 2$,

$$\mathcal{SD}_{\text{all-uses}}(P,S) \subseteq \mathcal{SD}_{\text{req-2+}}(P,S) \subseteq \ldots \subseteq \mathcal{SD}_{\text{req-k+}}(P,S) \subseteq \mathcal{SD}_{\text{req-(k+1)+}}(P,S).$$

□

We next consider Laski and Korel's criteria [15], which have subsequently become known as *context-coverage* and *ordered-context-coverage*. These criteria consider paths through definitions of *all* variables used in a given statement. Let $X_1, \ldots, X_k$ be variables that are all used in node n. An *elementary data context* for n is a set $\{\delta_1, \ldots, \delta_k\}$ where δ_i is a definition of X_i and there is a path p from the start node to node n such that for $1 \leq i \leq k$, p has a suffix that is a definition-clear path with respect to X_i from δ_i to n. Thus, control can reach n with variables $X_1, \ldots, X_k$ having the values that were assigned to them in nodes $\delta_1, \ldots, \delta_k$, respectively. The *context-coverage* criterion requires execution of such a path for each context. Clarke *et al.* [2] defined the *context-coverage+* criterion by making the following modifications:

1. each subset of the set of variables used in node n gives rise to a context;

2. the execution of paths to the *successors* of node n is required.

They showed that context-coverage+ subsumes all-uses. Modification (1) was motivated by the fact that there may be a definition-clear path with respect to X_i from the start node to a use of X_i in n. Since we are assuming the NFA property, in the class of programs considered here, no such path can be feasible. Furthermore, since there are 2^k subsets of a set of k variables, modification (1) may lead to extremely large numbers of subdomains.

The Laski and Korel definitions associated uses occurring in decisions with the decision node, not with the edges leaving that node. Modification (2) was added in order to insure that context-coverage subsumed branch testing. Alternatively, this can be achieved by distinguishing p-uses from c-uses and associating p-uses with edges, as in [18, 19]. In the remainder of the paper, we will use the term *context-coverage* to refer to the original Laski-Korel criterion, with this minor modification. We use the notation $(\{\delta_1, \ldots, \delta_k\}, u)$ to denote the context arising from definitions of X_i in nodes δ_i and uses of $X_1, \ldots, X_k$ in node or edge u.

Laski and Korel also introduced the *ordered-context-coverage* criterion [15]. An *ordered elementary data context* for node n is a permutation of an elementary data context for n. The criterion requires that each ordered-context be exercised by a path that visits the definitions in the given order.

Lemma 3 *Ordered-context-coverage universally properly covers context-coverage. Context-coverage universally properly covers decision-coverage.*

Proof:
By Observation 1, the fact that ordered-context-coverage universally properly covers context-coverage follows immediately from the definitions. The proof that context-coverage universally properly covers decision-coverage is similar to that of Lemma 1. Let $\mathtt{v1}, \ldots, \mathtt{vk}$ be the variables occurring in the decision in node n, let m be a successor node of n, let D be the decision-coverage subdomain corresponding to (n, m), let $C = \{C_1, \ldots, C_l\}$ be the set of contexts corresponding to edge (n, m), and let $D_1, \ldots, D_l$ be the corresponding subdomains. Consider a test case $t \in D$. Let $\delta_1, \ldots, \delta_k$ be the last definitions of $\mathtt{v1}, \ldots, \mathtt{vk}$, respectively, before the first occurrence of (n, m) in the path executed by t. Clearly, $(\{\delta_1, \ldots, \delta_k, \}, (n, m))$ is one of the contexts C_i, so $t \in \cup D_i$.

Conversely, assume $t \in \cup D_i$. By the definition of context-coverage (as modified to associate p-uses with edges), t executes a path that includes (n, m), and hence $t \in D$.

Since each edge from each decision gives rise to a distinct set of contexts, the covering is proper. □

Note that if a statement s uses variables $v_1, \ldots, v_k$ and the number of definitions of v_i that reach s is a_i, then the number of executable contexts arising from this single statement may be as high as $\prod_{i=1}^{k} a_i$. Furthermore, the number of executable ordered contexts arising from each context may be as high as $k!$. Thus both context-coverage and ordered-context-coverage have the potential of being extremely expensive criteria. In contrast, the number of definition-use associations arising from statement s is $\sum_{i=1}^{k} a_i$.

One might expect context-coverage and ordered-context-coverage to be guaranteed to be better at exposing faults than simpler data flow testing criteria. Hamlet has argued that such criteria should be good at concentrating failure-causing input [13]. In fact, this is not always the case. We next show that these criteria are *not* guaranteed to be better at detecting faults than the all-p-uses criterion.

Lemma 4 *Context-coverage does not universally properly cover all-p-uses or all-uses. Ordered-context-coverage does not universally properly cover all-p-uses or all-uses.*

Proof:
Consider the following program:

```
P(x,y: integer);
{precondition: 1 <= x <= 10 and 1 <= y <= 10}
begin
   if (x = 1) then x := 1;
              else x := 0;
```

Figure 1: Program for which context-coverage does not properly cover all-p-uses.

```
    if (y = 1) then y := 1;
               else y := 0;
    if x + y <= 20 then  write(x*y)
                   else  ...
end;
```

A flow graph for this program is shown in Figure 1. Note that the decision in node 8 always evaluates to `true`, thus control always follows edge $(8,9)$. This is purely a convenience to simplify the descriptions of the subdomains and the calculations of M and E.

The only variable used on edges $(2,3)$ and $(2,4)$ is `x`. Consequently, the def-p-use subdomains arising from these p-uses are identical to the context subdomains arising from them. Similarly, the subdomains arising from def-p-use associations $(1,(5,6),\mathtt{y})$ and $(1,(5,7),\mathtt{y})$ are identical to the corresponding context subdomains.

The only difference between the two criteria comes from the decision involving *both* variables `x` and `y`. The def-p-use associations arising from this decision give rise to subdomains D_1, D_2, D_3, and D_4, shown in the first three columns of Table 1, and the contexts arising from this decision give rise to subdomains D_5, D_6, D_7, and D_8.

Observe that $D_2 = D_7 \cup D_8$ and $D_4 = D_6 \cup D_8$. Furthermore, D_8 must be used in *any* covering of D_2 or D_4, and hence context-coverage does not properly cover all-p-uses for this program.

The ordered-contexts of this program are identical to the contexts. This is because every path from the start node to edge (8,9) passes through a definition of `x` in node 4

id	*subdomain*	*dua/context*	*d*	*m*
D_1	$\{(1,y)\}$	$(3,(8,9),\texttt{x})$	10	0
D_2	$\{(x,y)\|x \neq 1\}$	$(4,(8,9),\texttt{x})$	90	45
D_3	$\{(x,1)\}$	$(6,(8,9),\texttt{y})$	10	0
D_4	$\{(x,y)\|y \neq 1\}$	$(7,(8,9),\texttt{y})$	90	45
D_5	$\{(1,1)\}$	$(\{3,6\},(8,9))$	1	0
D_6	$\{(1,y)\|y \neq 1\}$	$(\{3,7\},(8,9))$	9	0
D_7	$\{(x,1)x \neq 1\}$	$(\{4,6\},(8,9))$	9	0
D_8	$\{(x,y)\|x \neq 1 \text{ and } y \neq 1\}$	$(\{4,7\},(8,9))$	81	45

Table 1: Subdomains of all-p-uses, context-coverage, and ordered-context-coverage

or node 5 before it passes through a definition of `y` in node 6 or node 7. Thus ordered-context-coverage does not properly cover all-p-uses for this program.

The fact that context-coverage and ordered-context-coverage do not universally properly cover all-uses follows from the transitivity of the universally properly covers relation. □

Example 2:

Now consider the specification,

$$S(x,y) = \begin{cases} 1 & \text{if } x = y = 1 \\ 2 & \text{if } x + y > 11 \\ 0 & \text{otherwise} \end{cases}$$

Note that $P(x,y) = S(x,y)$ if and only if $x + y \leq 11$, so all 45 of the failure-causing inputs are in D_8, as well as in some other subdomains. Based on the values of d_i and m_i shown in Table 1,

$$M(\text{ordered-context-coverage}, P, S) = M(\text{context-coverage}, P, S) = 1-(1-\frac{45}{81})A = 1-0.44A,$$

$$M(\text{all-p-uses}, P, S) = 1 - (1 - \frac{45}{90})^2 A = 1 - 0.25A,$$

where A is the probability that no failure-causing input is selected from any of the subdomains arising from uses *other than the ones on edge* $(8,9)$, and

$$E(\text{ordered-context-coverage}, P, S) = E(\text{context-coverage}, P, S) = B + \frac{45}{81} = B + 0.56,$$

$$E(\text{all-p-uses}, P, S) = B + 2(\frac{45}{90}) = B + 1.0,$$

where B is the expected number of failure-causing input selected from any of the subdomains arising from uses *other than the ones on edge* $(8,9)$. Thus, for this program, all-p-uses is better than context-coverage or ordered-context-coverage according to the measures M and E. $\Box$

Note that this program also illustrates how 2^k contexts can arise from a statement using k variables, and thus even (unordered) context-coverage may be prohibitively expensive.

Summarizing the lemmas in this section, we have

Theorem 3 *All-uses, all-p-uses, required-k-tuples, context-coverage, and ordered-context-coverage all universally properly cover decision-coverage. For $k \geq 2$, Required-(k+1)-tuples+ universally properly covers required-k-tuples+. Required 2-tuples universally properly covers all-uses. Neither context-coverage nor ordered-context-coverage universally properly covers all-p-uses or all-uses.*

We conclude this section by mentioning two other data flow testing criteria, the *all-du-paths criterion* [19] and the *all-simple-OI-paths criterion* [20]. Roughly speaking, the all-du-paths criterion requires the execution of particular paths from variable definitions to uses, while the all-simple-OI-paths criterion requires the execution of particular types of paths that cover chains of definitions and uses leading from inputs to outputs. The restrictions on the kinds of paths considered arise from control flow considerations – in all-du-paths, attention is restricted to *simple* paths, i.e. paths in which all nodes, except possibly the first and last, are distinct, while in the all-simple-OI-paths criterion, attention is restricted to paths that traverse certain loops zero, one, or two times. Rapps and Weyuker showed that all-du-paths subsumes all-uses and Ural and Yang showed that all-simple-OI-paths subsumes all-du-paths. However, these results were based on the original (non-applicable) versions of the criteria. We have previously shown that when one considers instead the applicable analogs of the criteria, all-du-paths does not even *subsume* branch testing [7]. Similar problems arise with all-simple-OI-paths. Consequently, by careful placement of faults in executable portions of the code that are *not* included in any executable du-path or in any executable simple OI-path, it is possible to construct programs for which these criteria are less likely to expose a fault than branch testing.

4 Mutation Testing

We next consider the mutation testing criterion [3]. Unlike the other criteria examined so far, mutation testing is not a path-oriented criterion. Instead, it considers a test suite T adequate for testing program P if T distinguishes P from each of a set of variants of P called *mutants*. These mutants are formed by applying *mutation operators*, which are simple syntactic changes, to the program. In mutation testing, the subdomains are of the form $\{t|P(t) \neq P'(t)\}$ where P' is a particular mutant of P.

Obviously the number and nature of the subdomains will depend on exactly which mutation operators are used. It is well-known that if the mutation operators include the following two operators, then mutation testing subsumes branch testing [1]:

1. replace a decision by `true`
2. replace a decision by `false`.

Since these are the only mutation operators that are directly relevant to our comparison of mutation testing to branch testing, we will use the term *limited mutation testing* to refer to mutation testing with only these two operators.

To see that limited mutation testing subsumes decision-coverage, consider a decision d in program P. Let $D_{d=T}$ be the decision-coverage subdomain arising from the `true` outcome of this decision. Let P' be the mutant in which d is replaced by `false`, let $D_{P'}$ be the corresponding mutation testing subdomain, and let $t \in D_{P'}$, i.e. t is an input such that $P(t) \neq P'(t)$. Therefore, t must cause d to evaluate to `true` at least once – otherwise there would be no distinction between the computations of P and P' on t. Thus $D_{P'} \subseteq D_{d=T}$. However, as the proof of the following theorem shows, $D_{P'}$ and $D_{d=T}$ are not necessarily identical, because even though the "wrong" branch is taken in the mutant, the output may not be affected. This observation allows the construction of programs for which decision-coverage is more likely to detect a fault than limited mutation testing.

Theorem 4 *Limited mutation testing does not universally cover decision-coverage.*

Proof:
Consider the following program P:

```
var x: integer; y: real;
begin
   read(x);
   if C(x) then y := x div 2  {integer part of x/2 }
           else y := x/2;
   write(y);
end;
```

Note that when x is even, the output is $x/2$ regardless of whether or not $C(x)$ holds.

Let P' be the mutant in which `C(x)` is replaced by `true` and let P'' be the mutant in which `C(x)` is replaced by `false`. The subdomains arising from these mutants are

$$\begin{aligned} D_{P'} &= \{x|\text{not } C(x) \text{ and } Odd(x)\} \\ D_{P''} &= \{x|C(x) \text{ and } Odd(x)\} \end{aligned}$$

and the subdomains arising from decision-coverage are

$$\begin{aligned} D_{C=F} &= \{x | \text{not } C(x)\} \\ D_{C=T} &= \{x | C(x)\} \end{aligned}$$

Thus no even element of either $D_{C=F}$ or $D_{C=T}$ belongs to either of the limited mutation testing subdomains, so limited mutation testing does not cover decision-coverage for this program. □

Example 3:
Let P be the above program and let S be any specification such that there is at least one failure-causing input and all failure-causing inputs are even. Then

$$\begin{aligned} 0 &= M(mutation, P, S) < M(decision, P, S) \text{ and} \\ 0 &= E(mutation, P, S) < E(decision, P, S). \end{aligned}$$

□

At first glance, one might attribute this rather surprising result to the highly restricted form of mutation testing considered. We therefore now consider mutation testing with additional mutation operators. Each application of a mutation operator gives rise to an additional non-empty subdomain, provided that the mutated program is not equivalent to the original program. Thus, additional mutation operators can certainly increase the fault-detecting ability. However, most mutation operators proposed in the literature do not necessarily lead to proper coverage of branch testing. We therefore believe that, even with the addition of other mutation operators, it will be possible to construct programs for which mutation testing is less likely to detect faults than branch testing when assessed using either M or E.

5 The Condition Coverage Family

Several criteria based on considering the individual conditions that comprise a decision have been proposed. We will refer to these criteria as the condition-coverage family. Consider a conditional statement controlled by a compound predicate, such as `if (A and B) then S`. Branch testing would require the selection of a test case that makes the predicate `(A and B)` evaluate to `true` and a test case that makes `(A and B)` evaluate to `false`. Note that it is possible to adequately test this statement using the branch testing strategy, without ever having the sub-expression `B` be `false` simply by selecting one test case making both `A` `true` and `B` `true` and another making `A` `false` and `B` `true`. Similarly the statement `if (A or B) then S` can be adequately branch-tested without `B` ever evaluating to `true`. Myers [16] argued that this is a weakness of branch testing, since a fault

```
P(x,y: integer);
{precondition: 1 <= x <= 10 and 1 <= y <= 10}
begin
   if (x = 1) and (y = 1) then writeln('1')
   else writeln('0')
end;
```

Figure 2: A simple program.

Figure 3: Flow graph of program from Figure 2.

such as `B` being the wrong expression could go undetected when using branch testing. He therefore introduced three new criteria, condition-coverage, decision-condition-coverage, and multiple-condition-coverage intended to overcome this deficiency.

Recall that a *decision* is a maximal Boolean expression controlling the execution of a conditional statement or loop and that *decision-coverage* requires that every decision take on the value `true` at least once during testing and also take on the value `false` at least once. For example, consider the program shown in Figure 2, A flow-graph of this program is shown in Figure 3. In this program, `(x=1) and (y=1)` is a decision.

A *condition* is a Boolean variable, a relational expression, or a Boolean function occurring in a decision. For this example, `(x=1)` is a condition, as is `(y=1)`. Thus a decision is made up of conditions. *Condition-coverage* requires that every condition take on the value `true` at least once and take on the value `false` at least once. For this example, two test cases are sufficient to satisfy condition-coverage. Test suites of the form $\{(x = 1, y = 1), (x \neq 1, y \neq 1)\}$, among others, would satisfy the criterion.

Decision-condition-coverage requires that every decision take on the value `true` at least once and take on the value `false` at least once *and* that every condition take on the value `true` at least once during testing and take on the value `false` at least once. In this example, the same test suite that satisfied condition-coverage would satisfy decision-condition-coverage.

Multiple-condition-coverage requires that every *combination* of truth values of conditions occurs at least once during testing.[5] For the example of Figure 2, multiple-condition-coverage would require four test cases:

$$(x = 1, y = 1), (x = 1, y \neq 1), (x \neq 1, y = 1), \text{ and } (x \neq 1, y \neq 1).$$

Note that the number of subdomains arising from multiple condition coverage is $\sum_{i=1}^{n} 2^{c_i}$, where n is the number of decision statements in P, and c_i is the number of conditions in the i^{th} decision in P. Thus, in the worst case the number of test cases required by multiple-condition-coverage is exponential in the number of conditions in P, while the numbers of test cases required by decision-coverage, condition-coverage, and decision-condition-coverage are all at worst linear in the number of conditions.

Example 4:
Consider again the program shown in Figure 2. Recall that the input domain of this program is $D = \{(x, y) \in \mathsf{Z} \times \mathsf{Z} | 1 \leq x \leq 10 \text{ and } 1 \leq y \leq 10\}$, and that the decision-coverage subdomains are $\mathcal{SD}_{dc}(P, S) = \{D_1, D_2\}$, where

$$\begin{aligned} D_1 &= \{(1,1)\} \\ D_2 &= \{(x, y) | x \neq 1 \text{ or } y \neq 1\}. \end{aligned}$$

Let

$$\begin{aligned} D_3 &= \{(1, y)\} \\ D_4 &= \{(x, 1)\} \\ D_5 &= \{(x, y) | x \neq 1\} \\ D_6 &= \{(x, y) | y \neq 1\} \end{aligned}$$

[5] There is some variation in the meaning of the term multiple-condition-coverage in subsequent software testing literature. Our usage of the terms here is consistent with Myers' definitions.

These are the subdomains corresponding to the conditions and their negations, so the multi-set of subdomains arising from condition-coverage is

$$\mathcal{SD}_{cc}(P, S) = \{D_3, D_4, D_5, D_6\}.$$

The multi-set of subdomains arising from decision-condition-coverage is

$$\mathcal{SD}_{dcc}(P, S) = \mathcal{SD}_{dc}(P, S) \cup \mathcal{SD}_{cc}(P, S) = \{D_1, D_2, D_3, D_4, D_5, D_6\}.$$

Finally, let

$$\begin{aligned} D_7 &= \{(1, y) | y \neq 1\} \\ D_8 &= \{(x, 1) | x \neq 1\} \\ D_9 &= \{(x, y) | x \neq 1 \text{ and } y \neq 1\} \end{aligned}$$

These subdomains, along with D_1 correspond to the combinations of conditions, so the multi-set of subdomains arising from multiple-condition-coverage is

$$\mathcal{SD}_{mcc}(P, S) = \{D_1, D_7, D_8, D_9\}.$$

□

Notice that for this program the condition-coverage and decision-condition-coverage criteria give rise to non-disjoint subdomains. More generally, programs exist for which each of the criteria in this family have non-disjoint subdomains. There are several reasons for this. First, the conditions in a compound predicate need not be mutually exclusive. Second, each condition or negation of a condition overlaps with the decision in which it occurs or the negation of that decision. Furthermore, even the subdomains arising from a particular decision or condition and its own negation, or from different combinations of conditions and their negations may intersect; this can occur when the decision occurs within a loop – a single test case may cause the decision (or condition or combination of conditions) to evaluate to `true` on one iteration of the loop and to `false` on another.

Myers points out (though not using this terminology) that condition-coverage does not subsume decision-coverage but decision-condition-coverage does. He then points out what he considers to be a deficiency in decision-condition-coverage: it is possible to satisfy decision-condition-coverage without executing all the branches in the transformed flow graph in which compound decisions are broken into series of decisions containing only individual conditions. Myers introduced multiple-condition-coverage to overcome this deficiency. He indicates (correctly) that multiple-condition-coverage subsumes decision-condition-coverage and concludes that multiple-condition-coverage is superior to decision-condition-coverage. Summarizing the criteria, Myers says,

> For programs containing decisions having multiple conditions, the minimum criterion is a sufficient number of test cases to evoke all possible combinations of condition outcomes in each decision, and all points of entry to the program, at least once.([16], p. 44)

Given that Myers recommends multiple-condition-coverage as a minimal criterion, in spite of its expense, in a book that is widely used by testing practitioners, it is worth investigating whether multiple-condition-coverage is really good at detecting faults. We show that both decision-condition-coverage and multiple-condition-coverage universally properly cover decision-coverage, but that multiple-condition-coverage *does not* universally properly cover decision-condition-coverage. We exhibit a program for which decision-condition-coverage is better than multiple-condition-coverage according to both M and E.

Before examining the relationship between the criteria, we note that one could argue that decision-condition-coverage has an "unfair advantage" over multiple-condition-coverage in the program in Figure 2 because in this particular program there are more decision-condition-coverage subdomains than multiple-condition-coverage subdomains. In fact, the reason multiple-condition-coverage fails to properly cover decision-condition-coverage is much more fundamental than that. To illustrate this, we introduce a "pared-down" version of decision-condition-coverage, which we call *minimized-decision-condition-coverage*. Note that the decision-condition-coverage criterion has several redundant subdomains. Any test case satisfying `(A and B)` obviously satisfies both `A` and `B`. Similarly, any test case satisfying either `not A` or `not B` satisfies `not (A and B)`. So the subdomains corresponding to `A`, `B`, and `not (A and B)` can be eliminated from consideration. Redundant subdomains arising from a disjunctive clause can be eliminated in a similar manner.

More precisely, for the minimized-decision-condition-coverage criterion, the multi-set of subdomains $\mathcal{SD}_{mdcc}(P, S)$ is obtained as follows: for each decision `d` of the form `A and B`, include the subdomain $D_{d=T}$ consisting of those inputs that make `d` evaluate to `true`, the subdomain $D_{A=F}$ consisting of those inputs that make `A` evaluate to `false`, and the subdomain $D_{B=F}$ consisting of those inputs that make `B` evaluate to `false`; for each decision `d` of the form `A or B` include the subdomain $D_{d=F}$ consisting of those inputs that make `d` evaluate to `false`, the subdomain $D_{A=T}$ consisting of those inputs that make `A` evaluate to `true`, and the subdomain $D_{B=T}$ consisting of those inputs that make `B` evaluate to `true`; for all other decisions, include all of the corresponding decision-condition-coverage subdomains. Note that for any program, the number of subdomains arising from minimized-decision-condition coverage is less than or equal to the number arising from multiple-condition-coverage.

Example 5:
Consider the program shown in Figure 2. For any specification S, the subdomains arising

from minimized-decision-condition-coverage are

$$\mathcal{SD}_{mdcc}(P,S) = \{D_1, D_5, D_6\},$$

□

Theorem 5 *The relations among the condition-coverage family of criteria are as follows:*

1. *Decision-condition-coverage universally properly covers decision-coverage.*
2. *Multiple-condition-coverage universally properly covers decision-coverage.*
3. *Minimized-decision-condition-coverage universally properly covers decision-coverage.*
4. *Decision-condition-coverage universally properly covers minimized-decision-condition-coverage.*
5. *Multiple-condition-coverage* does not *universally properly cover minimized-decision-condition-coverage.*
6. *Multiple-condition-coverage* does not *universally properly cover decision-condition-coverage.*
7. *Multiple-condition-coverage* does not *universally properly cover condition-coverage.*

Proof:
We prove parts 3, 5 and 7. The rest of the proof is straight-forward and can be found in [9]. To see part 3, let $D \in \mathcal{SD}_{dc}(P,S)$. By definition of decision-coverage, D is either $D_{d=T}$ or $D_{d=F}$ for some decision `d`. Case 1: `d` is of the form `A and B` and D is $D_{d=T}$. Then $D \in \mathcal{SD}_{mdcc}$. Case 2: `d` is of the form `A and B` and D is $D_{d=F}$. Then $D_{A=F}$ and $D_{B=F} \in \mathcal{SD}_{mdcc}$ and $D = D_{A=F} \cup D_{B=F}$. Case 3: `d` is of the form `A or B` and D is $D_{d=T}$. Then $D_{A=T}$ and $D_{B=T} \in \mathcal{SD}_{mdcc}$ and $D = D_{A=T} \cup D_{B=T}$. Case 4: `d` is of the form `A or B` and D is $D_{d=F}$. Then $D \in \mathcal{SD}_{mdcc}$. If `d` is not a conjunct or disjunct of two conditions then $D \in \mathcal{SD}_{mdcc}$. In each of these cases, D is a union of minimized-decision-condition subdomains arising from this decision `d`. Since each decision gives rise to its own collection of minimized-decision-condition subdomains, the covering is proper.

To see part 5, consider again the program in Figure 2. Recall that $\mathcal{SD}_{mcc}(P,S) = \{D_1, D_7, D_8, D_9\}$ and $\mathcal{SD}_{mdcc}(P,S) = \{D_1, D_5, D_6\}$. It is possible to express each minimized-decision-condition subdomain as a union of multiple-condition-coverage subdomains, as follows:

$$\begin{aligned} D_1 &= D_1 \\ D_5 &= D_8 \cup D_9 \\ D_6 &= D_7 \cup D_9 \end{aligned}$$

id	$subdomain$	d	m
D_1	$\{(1,1)\}$	1	0
D_2	$\{(x,y)\|x \neq 1$ or $y \neq 1\}$	99	45
D_3	$\{(1,y)\}$	10	0
D_4	$\{(x,1)\}$	10	0
D_5	$\{(x,y)\|x \neq 1\}$	90	45
D_6	$\{(x,y)\|y \neq 1\}$	90	45
D_7	$\{(1,y)\|y \neq 1\}$	9	0
D_8	$\{(x,1)x \neq 1\}$	9	0
D_9	$\{(x,y)\|x \neq 1$ and $y \neq 1\}$	81	45

Table 2: Subdomain, sizes, and numbers of failures for program in Example 6

Notice that the subdomain D_9 only occurs once in $\mathcal{SD}_{mcc}(P,S)$ but occurs twice in the covering of $\mathcal{SD}_{mdcc}(P,S)$, and that both of these occurrences are necessary in the sense that any covering of $\mathcal{SD}_{mdcc}(P,S)$ must have at least two occurrences of D_9. Therefore, multiple-condition-coverage does not properly cover minimized-decision-condition-coverage for this program.

The proof of part 7 is almost identical. Since $D_5, D_6 \in \mathcal{SD}_{cc}(P,S)$, D_9 must be used more than once in any covering of $\mathcal{SD}_{cc}(P,S)$. $\square$

We now use this result to exhibit a program for which each of the criteria discussed in this section, except for decision-coverage, is better than multiple-condition-coverage according to measures M and E.

Example 6:
Let P be the program in Examples 4 and 5 and, as in Example 2 above, let S be a specification such that $P(x,y) = S(x,y)$ if and only if $x + y \leq 11$, and $1 \leq x \leq 10$ and $1 \leq y \leq 10$. The values for m and d for each subdomain are shown in Table 2. Note that since all of the 45 failure-causing inputs lie in subdomains D_5, D_6, and D_9, and no failure-causing inputs lie in D_1, D_7, or D_8, the minimized-decision-condition criterion has a greater chance of selecting a failure-causing input than the multiple-condition-coverage criterion, even though there are more subdomains (and hence more test cases) induced by the multiple-condition-coverage criterion than by the minimized-decision-condition criterion. In particular:

$$\begin{aligned} M(mdcc,P,S) &= 1-(1-\frac{m_1}{d_1})(1-\frac{m_5}{d_5})(1-\frac{m_6}{d_6}) \\ &= 1-(1-\frac{0}{1})(1-\frac{45}{90})(1-\frac{45}{90}) \\ &= 0.75, \end{aligned}$$

criterion	M	E
decision-coverage	0.45	0.45
condition-coverage	0.75	1.00
decision-condition-coverage	0.86	1.45
minimized decision-condition-coverage	0.75	1.00
multiple-condition-coverage	0.56	0.56

Table 3: Values of M and E for the program from Example 5.

whereas

$$\begin{aligned} M(mcc, P, S) &= 1-(1-\frac{m_1}{d_1})(1-\frac{m_7}{d_7})(1-\frac{m_8}{d_8})(1-\frac{m_9}{d_9}) \\ &= 1-(1-\frac{0}{1})(1-\frac{0}{9})(1-\frac{0}{9})(1-\frac{45}{81}) \\ &= 0.56, \end{aligned}$$

Similarly,

$$\begin{aligned} E(mdcc, P, S) &= \frac{m_1}{d_1}+\frac{m_5}{d_5}+\frac{m_6}{d_6} \\ &= \frac{0}{1}+\frac{45}{90}+\frac{45}{90} \\ &= 1.0, \end{aligned}$$

whereas

$$\begin{aligned} E(mcc, P, S) &= \frac{m_1}{d_1}+\frac{m_7}{d_7}+\frac{m_8}{d_8}+\frac{m_9}{d_9} \\ &= \frac{0}{1}+\frac{0}{9}+\frac{0}{9}+\frac{45}{81} \\ &= 0.56. \end{aligned}$$

The values of M and E for condition-coverage and decision-condition-coverage, shown in Table 3, illustrate that multiple-condition-coverage also performs worse than these criteria for this program and specification, according to these measures. As noted above, multiple-condition-coverage requires more test cases for this program than are required by minimized-decision-condition-coverage, so this example also illustrates that bigger test suites are not necessarily better. □

Myers' motivation for introducing multiple-condition-coverage was that it is possible to satisfy decision-condition-coverage without covering all of the edges in the flow graph arising from assembly code. This concern may have arisen from an analogy with hardware testing, where wires connecting gates are the analogs of branches in assembly code. Even though this is not an issue for software testing, there is some intuitive motivation for requiring that such edges be executed. The "low-level" flow graph has edges corresponding to the evaluations of subexpressions in a condition. If the program has an erroneous subexpression of the form `A and B`, the subdomain consisting of those inputs that make the subexpression evaluate to true may concentrate the failure-causing inputs more than the subdomains `A = true` and `B = true` corresponding to the individual conditions. Similarly, if the program has an erroneous subexpression of the form `A or B`, the subdomain consisting of those inputs that make the subexpression evaluate to false may concentrate the failure-causing inputs more than the subdomains `A = false` and `B = false` corresponding to the individual conditions. Consequently, selecting test cases from the subexpression subdomains can increase the likelihood of detecting that type of fault. Several criteria, including the *multi-value expressions* criterion [11, 21], that capture this intuition and do properly cover decision-condition-coverage, but whose cost is linear in the number of conditions, are discussed in [9].

6 Conclusion

We have compared the fault-detecting ability of several software testing criteria. This comparison was done according to carefully defined notions of what it means for criterion C_1 to be better at detecting faults than criterion C_2. In particular, we investigated whether test suites chosen by randomly selecting one element from each subdomain induced by C_1 are at least as likely to include at least one failure-causing input as are test suites chosen in the same manner to satisfy C_2.

We had previously shown that if C_1 *universally properly covers* C_2, then C_1 is better than C_2, in that sense. In this paper, we extended this result by using the expected number of failures exposed as the basis for comparison. We showed that if C_1 properly covers C_2, the expected number of failures exposed by test suites selected using this same strategy is guaranteed to be at least as large for C_1 as for C_2. Note that if C_1 universally properly covers C_2, then C_1 is guaranteed to be at least as good as C_2 according to these measures, for *any* program, *regardless* of what faults are in the program.

We showed that the all-p-uses, all-uses, required-k-tuples+, context-coverage, ordered-context-coverage, minimized-decision-condition-coverage, decision-condition-coverage, and multiple-condition-coverage, all universally properly cover decision-coverage. However, limited mutation testing subsumes but does not universally properly cover decision-coverage. Furthermore, multiple-condition-coverage subsumes but does not universally properly cover decision-condition-coverage; context-coverage subsumes but does not universally properly cover all-uses or all-p-uses; and ordered-context-coverage subsumes but does not universally properly cover all-uses or all-p-uses. These relations are summarized in Figure 4. In each case for which C_1 fails to universally properly cover C_2, we exhibited a simple program for which $M(C_1, P, S) < M(C_2, P, S)$ and $E(C_1, P, S) < E(C_2, P, S)$. Since multiple-condition-coverage, context-coverage, and ordered-context-coverage each potentially require a number of test cases which is exponential in some aspect of the program size, this calls into question their usefulness. We emphasize that the results

Figure 4: Summary of Relations between criteria. A solid arrow from C_1 to C_2 indicates that C_1 universally properly covers C_2, a dotted arrow from C_1 to C_2 indicates that C_1 subsumes but does not universally properly cover C_2; any relation that is not explicitly shown in the figure and that does not follow from transitivity along with the fact that universally properly covers implies subsumption, does not hold.

presented here show the *existence* of programs and specifications for which C_2 is better at detecting faults than C_1 in situations when C_1 subsumes but does not universally properly cover C_2. They are based on the fact that when C_1 does not properly cover C_2 for program P, it is often possible to find a specification S (or equivalently, to find a distribution of failure-causing inputs) for which $M(C_2, P, S) > M(C_1, P, S)$ and $E(C_2, P, S) > E(C_1, P, S)$. However, even for such a program P, there are other specifications S' (or equivalently, other distributions of failure-causing inputs) for which $M(C_2, P, S') \leq M(C_1, P, S')$ and $E(C_2, P, S') \leq E(C_1, P, S')$. Furthermore, even when C_1 does not *universally* properly cover C_2, it may be the case that C_1 properly covers C_2 for the particular program that is being tested, and thus C_1 is guaranteed to be at least as likely as C_2 to detect a fault *in that program*, but not necessarily in others.

There are several implications of these results for the testing practitioner. If C_1 universally properly covers C_2, the practitioner is guaranteed to be more likely to detect a fault using C_1 than using C_2, *provided* that tests are selected using the strategy described earlier. On the other hand, if C_1 does not properly cover C_2 for the program under test, the practitioner should be *warned* that even if C_1 requires many more test cases than C_2, it may be less likely to detect a fault. Thus, criteria which are relatively low in the hierarchy of criteria induced by the properly covers relation, may be of questionable value. On the other hand, it certainly could happen that such criteria are indeed good at detecting faults in "typical" programs. This paper does not address that question; in fact, in the absence of a precise notion of what constitutes a "typical" program, such questions can only be answered through the accumulation of anecdotal evidence.

We also note that it seems reasonable to conjecture that using test selection strategies that closely approximate the one described here will yield similar results in practice. However, more analytical and/or empirical research will be needed to bear this out for particular approximation strategies. One might argue that this approach is "making the real world fit the model", but in fact doing so may be justified. In software engineering, unlike the physical sciences, we can exercise some significant control over the real world when there is some benefit from doing so. For example, it may be reasonable to devise test data selection strategies that closely resemble the ones used in our model. The payoff for this would be that the tester has precise knowledge about the relative fault-detecting ability of criteria, without prior knowledge of the nature of the faults in a program.

Our investigation also sheds some light on how to develop new criteria that are provably better than an existing criterion C. One possibility is to take a criterion C' that universally covers but does not universally properly cover C, find those subdomains of C' that are used $k > 1$ times in the covering of C, and include k duplicates of each such subdomain, or equivalently, choose k test cases from each such subdomain.

Finally, we note again that our results are based on a test suite selection procedure that is an idealization of the way test suites are selected in a typical testing environment. It is assumed that test cases are selected only after the domain has been divided into subdomains. In practice, these testing criteria are generally used as the basis for evaluating the adequacy of test suites, not for selecting test cases. This suggests two open problems: develop practical test selection algorithms that approximate the strategy used here, and conduct theoretical studies similar to this one based on measures that more closely reflect existing test selection strategies.

Acknowledgments: The authors would like to thank the anonymous referees for carefully reading the manuscript and making several useful suggestions. Some of the results presented in this paper appeared in the *Proceedings of the IEEE 15th International Conference on Software Engineering*, May 1993.

References

[1] T. A. Budd. Mutation analysis: Ideas, examples, problems and prospects. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 129–148. North Holland, July 1981.

[2] L. Clarke, A. Podgurski, D. Richardson, and S. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, SE-15(11):244–251, Nov. 1989.

[3] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

[4] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10(7):438–444, July 1984.

[5] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305 – 330, Sept. 1976.

[6] P. G. Frankl. Test selection for analytical comparability of fault-detecting ability. Technical Report PUCS-100-93, Polytechnic University Computer Science Dept., Jan. 1993.

[7] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, SE-14(10):1483–1498, Oct. 1988.

[8] P. G. Frankl and E. J. Weyuker. Assessing the fault-detecting ability of testing methods. In *ACM SIGSOFT '91 Conference on Software for Critical Systems*, pages 77 –91. ACM Press, Dec. 1991.

[9] P. G. Frankl and E. J. Weyuker. Analytical comparison of several testing strategies. Technical Report PUCS-100-92, Polytechnic University Computer Science Dept., July 1992.

[10] P. G. Frankl and E. J. Weyuker. A formal analysis of the fault detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 1993. To appear.

[11] J. D. Gannon, P. McMullin, and R. G. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3:211–213, July 1981.

[12] J. S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, SE-9(6):686–709, Nov. 1983.

[13] D. Hamlet. Theoretical comparison of testing methods. In *Proceedings ACM SIGSOFT Third Symposium on Software Testing, Analysis, and Verification*, pages 28–37. ACM Press, Dec. 1989.

[14] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402 – 1411, Dec. 1990.

[15] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, May 1983.

[16] G. J. Myers. *The Art of Software Testing*. Wiley, New York, 1979.

[17] S. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, Nov. 1984.

[18] S. Rapps and E. J. Weyuker. Data flow analyisis techniques for program test data selection. In *Proceedings Sixth International Conference on Software Engineering*, pages 272–278, Sept. 1982. Tokyo, Japan.

[19] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-14(4):367–375, Apr. 1985.

[20] H. Ural and B. Yang. A structural test selection criterion. *Information Processing Letters*, 28(3):157–163, July 1988.

[21] M. Weiser, J. Gannon, and P. McMullin. Comparison of structural test coverage metrics. *IEEE Software*, 2(2):80–85, Mar. 1985.

[22] S. N. Weiss. Comparing test data adequacy criteria. *Software Engineering Notes*, 14(6):42–49, Oct. 1989.

[23] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[24] E. J. Weyuker, S. N. Weiss, and D. Hamlet. Comparison of program testing strategies. In *Proceedings Fourth Symposium on Software Testing, Analysis, and Verification*, pages 1–10. ACM Press, Oct. 1991.