A Formal Analysis of the Fault-detecting Ability of Testing Methods

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

Abstract

This paper examines several relations between software testing criteria, exploring whether for each relation R and each pair of criteria, C_1 and C_2, $R(C_1, C_2)$ guarantees that C_1 is better at detecting faults than C_2 according to various probabilistic measures of fault-detecting ability. It is shown that the fact that C_1 subsumes C_2 does not guarantee that C_1 is better at detecting faults. Relations that strengthen the subsumption relation and that have more bearing on fault-detecting ability are introduced.

1 Introduction

Over the last several years, researchers have defined a wide variety of software testing techniques, studied their properties, and built tools based on some of them. Recently, there has been increasing interest in the question of how techniques compare to one another in terms of their ability to expose faults.

Aspects of this problem have been addressed through experiments [2], simulations [1, 7], and analysis [9, 14]. While these approaches offer certain insights, the results lack generality, either because of the inherent nature of experimentation and simulation, or because of the assumptions underlying the simulations and analyses. In particular those simulations and analyses [1, 7, 14] investigated criteria that *partition* the input domain of the program into disjoint subdomains. In this paper, we analyze the more realistic situation in which the criteria divide the input domain into overlapping subdomains. We

*Author's address: Computer Science Dept., Polytechnic University, 333 Jay St., Brooklyn, N.Y. 11201. Supported by NSF Grant CCR-8810287 and by the New York State Science and Technology Foundation Center for Advanced Technology program.

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

characterize various relationships between criteria, and show circumstances under which we can conclude that test suites chosen using one criterion are more likely to expose faults than test suites chosen using another. We also apply these results to compare well-known control flow based and data flow based techniques.

Subsumption has frequently been used to compare criteria. Criterion C_1 *subsumes* criterion C_2 if every test suite that satisfies C_1 also satisfies C_2. In this paper, we show that according to three probabilistic measures of fault-detecting ability, the fact that C_1 subsumes C_2 does not guarantee that C_1 is better at detecting faults than C_2. These measures are related to two different test data selection strategies. We also explore the question of how the subsumes relation can be strengthened in order to obtain a relation that has more bearing on fault-detecting ability. We define two natural relations, *covers* and *partitions*, each of which is stronger than subsumes, and show that, in the worst case, neither of them provide much additional insight into fault-detecting ability. Finally, we define two variations of these relations, *properly covers* and *properly partitions*, and prove that when C_1 properly covers or properly partitions C_2, C_1 is guaranteed to be better at detecting faults than C_2 when assessed using certain reasonable probabilistic measures.

1.1 Preliminary Definitions

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. 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. Therefore, $\{0,1,1\} \subseteq \{0,1,1,1\}$, but it is not the case that $\{0,1,1\} \subseteq \{0,1,2\}$.

A *test suite* is a multi-set of test cases, each of which is a possible input to the program. 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. These techniques are sometimes referred to as *partition testing*, but in fact, most of them subdivide the input domain into overlapping subdomains, and thus do not form a true partition of the input domain. In this paper, we will refer to such strategies as *subdomain-based testing*. In subdomain strategies based on the program's structure, each subdomain consists of all elements that cause a particular code element to be executed. For example, in branch testing, each subdomain consists of all inputs that cause execution of a particular branch; in path testing, each subdomain consists of all inputs that cause execution of a particular path; in data flow testing using the all-uses criterion, each subdomain consists of all inputs that execute any path from a particular definition of a variable `v` to a particular use of `v` without any intervening redefinition of `v`. Other subdomain-based testing strategies include mutation testing, in which each subdomain consists of all inputs that kill a particular mutant; specification-based testing techniques, in which each subdomain consists of inputs that satisfy a particular condition or combination of conditions mentioned in the specification; and exhaustive testing, in which each subdomain consists of a single point. In most of these strategies, most programs and specifications give rise to overlapping and duplicate subdomains.

A test data adequacy criterion is a relation $C \subseteq$ Test Suites$\times$Programs$\times$Specifications, that, in practice, is used to determine whether a given test set T does a "thorough" job of testing program P for specification S. If $C(T, P, S)$ holds, we will say "T is adequate for testing P for S according to C", or, more simply, "T is C-adequate for P and S".

A testing criterion C is *subdomain-based* if, for each program P and specification S, there is a non-empty multi-set $\mathcal{SD}_C(P, S)$ of subdomains (subsets of the input domain), such that C requires the selection of one or more test cases from each subdomain in $\mathcal{SD}_C(P, S)$. If C requires the selection of more than one test case from a given subdomain, they need not be distinct. 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, the same set of test cases might cause the execution of two different statements, and hence appear twice in the multi-set of subdomains arising from the statement testing criterion. Depending on the details of the criterion and the test selection strategy, it may then be necessary to choose test cases from such a subdomain more than once. Note that since $\mathcal{SD}_C$ 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 criterion.

Subdomain-based criterion C is said to be *applicable to (P,S)* if and only if there exists a test suite T such that $C(T, P, S)$ holds, and 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 specifically noted. In fact, many of the criteria that have been defined and discussed in the software testing literature are *not* universally applicable. In path-based or structural criteria, this occurs because infeasible paths through the program can lead to testing requirements that can never be fulfilled (represented by empty subdomains belonging to $\mathcal{SD}_C(P, S)$), while in mutation testing this occurs due to mutants that are equivalent to the original program. However, it is the *universally applicable analogs* of those criteria (obtained by removing the empty subdomain from $\mathcal{SD}_C(P, S)$) that are actually *usable in practice*. It is important to note that the relationship between the universally applicable analogs of criteria may be different than that between the original criteria. For example, in [3] we defined universally applicable analogs of the data flow testing criteria defined in [12, 13], and showed that while the all-du-paths criterion subsumes the all-uses criterion, the universally applicable analog of all-du-paths does not subsume the universally applicable analog of all-uses.

In order to illustrate some of our points with realistic examples, we review the definitions of several well-known criteria. Since some of these criteria are based on program structure, it is necessary at this point to make certain assumptions about the programs under test. We will limit attention to programs that are written in Pascal with no **goto** statements.[1] 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. 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 NFA have the possibility of referencing an undefined variable. Although the question of whether a given program satisfies NFA is undecidable, it is possible to check the stronger, *no anomalies property (NA)*, which requires that every path from the start node to a use of a variable `v` to pass through a node having a definition of `v`. Furthermore, NA can be enforced by adding a dummy definition of each variable to the start node.

The *all-edges* criterion (also known as branch testing) requires that every edge in the program's flow graph be executed by at least one test case. Data flow testing criteria [8, 10, 11, 13] require that the test data exercise paths from points at which variables are defined to points at which their values are subsequently used. Occurrences of variables in the program under test are classified as being either *definitions*, in which values are stored, or *uses*, in which values are fetched. In [13] variable uses are further classified as being either *p-uses*, i.e., uses occurring in the Boolean expression in a conditional statement, or *c-uses*, i.e., uses occurring in assignment statements or arguments in a procedure call. The *all-uses* criterion [13] requires that the test data cover every *definition-use association* (dua) in the program, where a dua is a triple (d,u,`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.[2] A test case t *covers* dua (d,u,`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 [13] is a restricted version of all-uses that only requires that the test data cover every (d,u,`v`) in which u is an edge with a p-use of variable `v`. Precise definitions of the criteria for the subset of Pascal in question are given in [3].

Notice that these criteria are not universally applicable; in fact, for many programs the all-p-uses and all-uses criteria are not applicable because some dua is not executable. For example, any program having a **for** loop whose bounds are non-equal constants has an unexecutable dua involving the initialization of the counter variable and the edge exiting the loop. Such a dua is unexecutable because the only executable path from the definition to the use traverses the loop at least once, and in so doing, redefines the counter variable [3]. In the sequel we will use the terms all-edges, all-p-uses, and all-uses to refer to the universally applicable analogs of these criteria, unless otherwise specifically noted.

[1]This assumption is used in the proof of Theorem 5, below.

[2]A definition-clear path with respect to `v` is a path in which none of the nodes, except perhaps the first and last, have a definition of variable `v`.

A program satisfies the *no syntactic undefined p-uses* property (NSUP) if for every p-use of a variable `v`, there is a path from program entry to the p-use along which `v` is defined. Rapps and Weyuker showed that for the class of programs that satisfies NSUP, for the original (not universally applicable) criteria, all-p-uses subsumes all-edges. In [3], we showed that a stronger restriction on the class of programs considered is needed to make subsumption hold for the universally applicable analogs of the criteria. Namely, we required the *no feasible undefined p-uses* property (NFUP), which requires that for every p-use of a variable `v`, there is an *executable* path from program entry to the p-use along which `v` is defined. Since any program that satisfies NFA also satisfies NFUP, for the class of programs we are considering, all-uses subsumes all-edges.

2 Fault Detecting Ability of Subdomain Criteria

In this section, we define three probabilistic measures of the fault-detecting ability of a testing criterion. In [1, 7, 14] the probability that a test set generated to satisfy some subdomain-based testing strategy would expose a fault was compared to the probability that a randomly generated test set of the same size would expose a fault. In contrast, we are interested in comparing the probability that test sets generated to satisfy different subdomain-based strategies will detect at least one fault.

We will focus on two conceptually simple test selection strategies. The first requires the tester to independently randomly select test cases from the domain until the adequacy criterion has been satisfied. The second strategy assumes that the domain has first been divided into subdomains and then requires independent random selection of a predetermined number, n, of test cases from each subdomain. We assume throughout that the random selection is done using a uniform distribution. These strategies, Sel_1 and Sel_2, are shown in Figure 1.

Our analytical results on the fault-detecting ability of subdomain criteria are based on the assumption that test suites are selected using Sel_1 or Sel_2. In practice it is rare to use either of these strategies, but rather something closer to a combination of the two. Typically a tester begins by selecting some test cases. Sometimes these test cases are chosen because the tester believes they have some particular significance, and sometimes they are chosen in a more *ad hoc* fashion. After some amount of testing, the tester checks to see which features of the program still need to be exercised in order to satisfy the criterion and tries to select test cases accordingly. This corresponds to selecting test cases from the appropriate subdomains in a manner similar to the second strategy. Thus, for example, one might select an all-edges adequate test suite by generating an initial test suite, checking to see which edges had not yet been covered, and then selecting test cases specifically aimed at covering each of the edges that remain unexercised.

Note that when Sel_1 is used, a test case that lies in the intersection of two or more subdomains may "count" toward each subdomain, while when using Sel_2 it will not.

```
Sel_1:
    T := empty test suite;
    repeat
        t := a uniformly randomly selected element of the input domain;
        add t to T
    until T is C-adequate

Sel_2(n):
    T := empty test suite;
    for each subdomain D ∈ SD_C(P, S) do
    begin
        for i := 1 to n do
        begin
            t := a uniformly randomly selected element of D;
            add t to T
        end
    end
```

Figure 1: Two test suite selection strategies

Thus, Sel_1 may lead to selection of substantially smaller test suites than $Sel_2(1)$.

Also note that if the multi-set $SD_C(P, S)$ contains m copies of some subdomain D, then $Sel_2(n)$ requires that n test cases be selected independently from D m different times, for a total of $m \times n$ test cases from D. This can make Sel_2 sensitive to details of the definitions of the testing criterion. For example, the number of all-edges subdomains for a given program P will depend on details of the construction of the P's flow graph, such as whether or not an extra exit node is added to the graph. One could avoid such problems by assuming that duplicate elements of $SD_C(P, S)$ are removed before test case selection. However, in practice one usually works with predicates describing the subdomains, rather than with explicit lists of their elements. Since it is undecidable whether two predicates are equivalent, it is not always possible to identify duplicate subdomains. Consequently, we do not assume that duplicates have been eliminated.

Obviously there are many reasonable measures of fault-detecting ability that could be defined. Our measures are related to the probability that a test suite will expose a fault. Following the notation used in [14], given a program P, specification S, and criterion C, we let $d_i = |D_i|$ and $m_i =$ the number of failure-causing inputs in D_i, where $\mathcal{SD}_C(P, S) = \{D_1, ..., D_k\}$.

As Weyuker and Jeng [14] have noted, the fault-detecting ability of a criterion is related to the extent to which failure-causing inputs are concentrated by its subdomains. The first measure we consider is

$$M_1(C, P, S) = max_{1 \leq i \leq k}(\frac{m_i}{d_i}).$$

Let D_{max} denote a subdomain that has the highest concentration of failure-causing inputs, i.e., a subdomain for which m_i/d_i is maximal. Since we are considering subdomain-based criteria, any C-adequate test set contains at least one test case from D_{max}. When using Sel_1 or Sel_2, each element of D_{max} is equally likely to be selected to "represent" D_{max}, and therefore M_1 gives a crude lower bound on the probability that a C-adequate test suite selected using Sel_1 or Sel_2 will expose at least one fault.

The second measure we consider is

$$M_2(C, P, S) = 1 - \prod_{i=1}^{k}(1 - \frac{m_i}{d_i}).$$

M_2 gives the exact probability that a test set chosen using $Sel_2(1)$ will expose at least one fault. Duran and Ntafos [1] and Hamlet and Taylor [7] performed simulations investigating whether hypothetical partition testing strategies (i.e., subdomain-based testing strategies in which the subdomains are pairwise disjoint) are more likely to expose faults than random testing according to this measure. Weyuker and Jeng investigated similar questions analytically [14]. In this paper, we investigate the more realistic case of subdomain-based criteria in which the subdomains may intersect, and compare subdomain-based criteria to one another rather than to random testing.

One problem with using M_2 as a measure is that one subdomain-based criterion C_1 may divide the domain D into k_1 subdomains while another criterion C_2 divides D into k_2 subdomains where $k_1 > k_2$. Then, in a sense, M_2 gives C_1 an unfair advantage since C_1 will require k_1 test cases while C_2 only requires k_2, and hence M_2 is comparing the fault-detecting qualities of different sized test suites. In order to correct this imbalance, we first define a measure that is a generalization of M_2. Instead of always selecting one test case per subdomain, n test cases are chosen per subdomain. We call this measure M_3:

$$M_3(C, P, S, n) = 1 - \prod_{i=1}^{k} (1 - (\frac{m_i}{d_i}))^n,$$

Of course this measure still has the deficiency cited above that different numbers of test cases might be needed to satisfy C_1 and C_2. The reason M_3 is a useful generalization of M_2 is that it allows us to adjust for the test suite size differences. Letting k_1 and k_2 denote the number of subdomains of C_1 and C_2, respectively, for program P and specification S, and letting $r = (k_1/k_2)$, we can compensate for the inequality in test set size by comparing $M_2(C_1, P, S) = M_3(C_1, P, S, 1)$ to $M_3(C_2, P, S, r)$.

M_3 is actually a special case of the measure called P_p in [1, 7, 14]. In those papers, a strategy could select a different number of test cases for each subdomain, whereas M_3 requires the same number to be selected from each subdomain.

3 Relations between criteria

In this section, we define five relations between criteria: the *narrows*, *covers*, *partitions*, *properly covers*, and *properly partitions* relations. We compare these relations to one another and to the subsumes relation, illustrate the relations with examples, and examine what knowing that C_1 narrows, covers, partitions, properly covers, or properly partitions C_2 for program P and specification S tells us about the relative values of each measure of fault-detecting ability. Specifically, we investigate whether for each relationship R, for every program P and specification S,

- $R(C_1, C_2) \stackrel{?}{\Rightarrow} M_1(C_1, P, S) \geq M_1(C_2, P, S)$
- $R(C_1, C_2) \stackrel{?}{\Rightarrow} M_2(C_1, P, S) \geq M_2(C_2, P, S)$
- $R(C_1, C_2) \stackrel{?}{\Rightarrow} M_3(C_1, P, S, 1) \geq M_3(C_2, P, S, r)$ where $r = |\mathcal{SD}_{C1}(P, S)|/|\mathcal{SD}_{C2}(P, S)|$.

The resulting theorems are summarized in Table 1.

Many relations that have been previously studied in connection with fault-detecting ability depend on the distribution of failure-causing inputs. These include Gourlay's

	$M_1(C_1) \geq M_1(C_2)$?	$M_2(C_1) \geq M_2(C_2)$?	$M_3(C_1, 1) \geq M_3(C_2, r)$?
C_1 narrows C_2	sometimes	sometimes	sometimes
C_1 covers C_2	sometimes	sometimes	sometimes
C_1 partitions C_2	always	sometimes	sometimes
C_1 properly covers C_2	sometimes	always	sometimes
C_1 properly partitions C_2	always	always	sometimes

Table 1: Summary of Results

power relation [6] and the *better* and *probbetter* relations proposed by Weyuker, Weiss, and Hamlet [15]. In contrast, each of the relations between criteria introduced here is induced by a relation between the corresponding multi-sets of subdomains. Thus, for these relations the question of whether $R(C_1, C_2)$ holds for program P and specification S depends purely on the way the criteria divide the input domain into subdomains, and *not* on such factors as the way failure-causing inputs are distributed in the input domain. Consequently, the results here are very general. If we prove that $R(C_1, C_2)$ guarantees that C_1 is at least as good as C_2 according to some probabilistic measure and if we prove that $R(C_1, C_2)$ holds for all programs and specifications in some class, then we are guaranteed that C_1 is at least as good as C_2 for testing programs in that class, *regardless* of which particular faults occur in the program.

3.1 The narrows relation

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 $D' \subseteq D$. This notion is known as *refinement* in point set topology. C_1 *universally narrows* C_2 if for every program, specification pair (P,S), C_1 narrows C_2 for (P,S).

Observation 1 *For each program P and specification S, the narrows relation is reflexive and transitive. If $\mathcal{SD}_{C2}(P, S) \subseteq \mathcal{SD}_{C1}(P, S)$ then C_1 narrows C_2.*

The next theorem shows that the narrows relation is closely related to the subsumes relation. However, the question of whether C_1 narrows C_2 deals only with the relationships between the multi-sets of subdomains induced by the two criteria, while the question of whether C_1 subsumes C_2 may involve additional considerations, such as the number of test cases from each subdomain required by each criterion. For example, consider criteria C_1 and C_2 such that for any (P,S), C_1 and C_2 give rise to the same set of subdomains, but C_2 requires selection of two test cases from each subdomain whereas C_1 only requires selection of one test case from each subdomain. Trivially, C_1 universally narrows C_2.

However, C_1 does not subsume C_2, since a test suite consisting of one element from each subdomain is C_1-adequate but not C_2-adequate. As the next theorem shows, as long as the criteria require selection of at least *one* element from each subdomain (as opposed to requiring selection of at least k elements for some $k > 1$) subsumption is equivalent to the universally narrows relation.

Theorem 1 *Let C_1 and C_2 be subdomain-based criteria, each of which explicitly requires selection of at least one test case from each subdomain. Then C_1 subsumes C_2 if and only if C_1 universally narrows C_2.*

Proof:
Assume C_1 universally narrows C_2. Let T be a test suite that is C_1-adequate for some program P and specification S. T contains at least one element from each subdomain in $\mathcal{SD}_{C1}(P,S)$. Thus, since each subdomain in $\mathcal{SD}_{C2}(P,S)$ is a superset of some subdomain belonging to $\mathcal{SD}_{C1}(P,S)$, T contains at least one element from each subdomain in $\mathcal{SD}_{C2}(P,S)$. So T is C_2-adequate for (P,S).

Conversely, assume C_1 does not universally narrow C_2. There exists a program P and a specification S such that some subdomain $D \in \mathcal{SD}_{C2}(P,S)$ is not a superset of any subdomain of $\mathcal{SD}_{C1}(P,S)$. Thus, for each $D' \in \mathcal{SD}_{C1}(P,S)$, $D' - D \neq \emptyset$. Let T be a test suite obtained by selecting one element from $D' - D$ for each $D' \in \mathcal{SD}_{C1}$. Then T is C_1-adequate but not C_2-adequate for (P,S). So C_1 does not subsume C_2. $\Box$

The next group of theorems establish that the fact that a criterion C_1 narrows criterion C_2 does not guarantee that C_1 is better able to detect faults than C_2, according to any of our three measures.

Theorem 2 *There exists a program P, specification S, and criteria C_1 and C_2 such that C_1 narrows C_2 but $M_1(C_1,P,S) < M_1(C_2,P,S)$.*

Proof:
Let P be a program whose input domain is the integers between $-N$ and N, where $N > 1$. Let C_1 be the criterion that requires selection of at least one test case that is zero and at least one test case that is non-zero. Let C_2 be the criterion that requires selection of at least one test case that is greater than or equal to zero and at least one test case that is less than or equal to zero. Suppose

$$P(x) = \begin{cases} S(x)+1 & \text{if } x = 1 \\ S(x) & \text{otherwise} \end{cases}$$

That is, suppose P is correct on all inputs except $x = 1$. Then $\mathcal{SD}_{C1} = \{D_1, D_2\}$ and $\mathcal{SD}_{C2} = \{D_3, D_4\}$ where

$$\begin{array}{lll} D_1 = \{0\} & d_1 = 1 & m_1 = 0 \\ D_2 = \{x \mid -N \leq x \leq N \text{ and } x \neq 0\} & d_2 = 2N & m_2 = 1 \\ D_3 = \{x \mid 0 \leq x \leq N\} & d_3 = N+1 & m_3 = 1 \\ D_4 = \{x \mid -N \leq x \leq 0\} & d_4 = N+1 & m_4 = 0. \end{array}$$

So C_1 narrows C_2, but

$$M_1(C_1, P, S) = \frac{1}{2N} < \frac{1}{N+1} = M_1(C_2, P, S)$$

□

Theorem 3 *There exists a program P, specification S, and criteria C_1 and C_2 such that C_1 narrows C_2 but $M_2(C_1, P, S) < M_2(C_2, P, S)$.*

Proof:
Let P, C_1 and C_2 be as in the proof of Theorem 2. Let

$$P(x) = \begin{cases} S(x) & \text{if } x \in \{0,1\} \\ S(x)+1 & \text{otherwise} \end{cases}$$

That is, P is *incorrect* on all inputs except $x = 0$ and $x = 1$. Then $\mathcal{SD}_{C1}$, $\mathcal{SD}_{C2}$, and each d_i, and m_1 are as above, but $m_2 = 2N - 1$, $m_3 = N - 1$, $m_4 = N$. So

$$M_2(C_2, P, S) = 1 - (\frac{2}{N+1})(\frac{1}{N+1}) > 1 - 1(\frac{1}{2N}) = M_2(C_1, P, S).$$

□

Theorem 4 *There exists a program P, specification S, and criteria C_1 and C_2 such that C_1 narrows C_2 but $M_3(C_1, P, S, 1) < M_3(C_2, P, S, r)$, where $r = |\mathcal{SD}_{C1}(P, S)|/|\mathcal{SD}_{C2}(P, S)|$.*

Proof:
Let P, S, C_1, and C_2 be as in Theorem 3. Since $|\mathcal{SD}_{C1}(P, S)| = |\mathcal{SD}_{C2}(P, S)| = 2$, $r = 1$, so

$$M_3(C_2, P, S, r) = M_2(C_2, P, S) > M_2(C_1, P, S) = M_3(C_1, P, S, 1).$$

□

Corollary 1 *For each measure M_i, there exists criteria C_1 and C_2, program P and specification S such that C_1 subsumes C_2 but $M_i(C_2, P, S) > M_i(C_1, P, S)$.*

3.2 The covers relation

The next relation we examine, the *covers* relation, strengthens the narrows relation in a natural way. The covers relation is interesting to examine because many criteria that had previously been shown to subsume the all-edges criterion actually also cover all-edges.

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 non-empty collection of subdomains $\{D_1, \ldots, D_n\}$, 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).

Observation 2 *For each program P and specification S, the covers relation is reflexive and transitive. If C_1 covers C_2 for (P,S) then C_1 narrows C_2 for (P,S). If $\mathcal{SD}_{C2}(P,S) \subseteq \mathcal{SD}_{C1}(P,S)$ then C_1 covers C_2 for (P,S).*

The following example illustrates the distinction between the narrows and covers relations.

Example 1:
Consider the criteria in the proof of Theorem 2. Since $D_1 \subseteq D_3$ and $D_1 \subseteq D_4$, C_1 narrows C_2. However, since $D_3 \neq D_1$, $D_3 \neq D_2$, and $D_3 \neq (D_1 \cup D_2)$, C_1 does not cover C_2. □

We next show that the all-p-uses and all-uses criteria universally cover the all-edges criterion.

Theorem 5 *The all-p-uses criterion universally covers the all-edges criterion.*

Proof:
Let P be a program, let e be an executable edge in P, and let D_e be the subdomain corresponding to e, that is, the set of inputs that execute paths that cover edge e.
Case 1: e has a p-use of some variable `v`. 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 $\{t | t \text{ covers dua } (\delta_i, e, \mathtt{v})\}$. Since the program satisfies the NFA property, every feasible path from the start node to e passes through at least one of the δ_i, so $D_e = D_1 \cup \ldots \cup D_n$.
Case 2: e does not have a p-use of any variable. Then, since P has no **goto** statements, there are edges $e_1, \ldots, e_m$ that do have p-uses of variables, s.t. $D_e = D_{e_1} \cup \ldots \cup D_{e_m}$. To see this, let s be the innermost nested conditional or repetitive statement such that e is in the subgraph corresponding to s. If s is an **if-then**, **if-then-else**, or **case** statement then $m = 1$ and e_1 is the edge going from the decision node toward e; if s is a **while** or **for** statement then $m = 1$ and e_1 is the edge entering the loop; if s is a **repeat** statement then $m = 2$ and e_1 and e_2 are the back-edge of the loop and the edge exiting the loop. If e is not contained in any conditional or repetitive statement, $e_1, \ldots, e_m$ are the edges out of the first decision node encountered on a path beginning at the start node. By case 1, each D_{e_i} is a union of all-p-uses subdomains; thus the union of all of these all-p-uses subdomains equals D_e. □

Corollary 2 *The all-uses criterion universally covers the all-edges criterion.*

Proof:
For any program P and specification S, $\mathcal{SD}_{all-p-uses}(P,S) \subseteq \mathcal{SD}_{all-uses}(P,S)$, so by Observation 2, all-uses covers all-p-uses. Since *covers* is a transitive relation (Observation 2), all-uses covers all-edges. □

The next group of theorems shows that the fact that C_1 covers C_2 does not guarantee that C_1 is better than C_2 according to any of the measures.

Theorem 6 *There exists a program P, specification S, and criteria C_1 and C_2 such that C_1 covers C_2 but $M_1(C_1, P, S) < M_1(C_2, P, S)$.*

Proof:
Let P be a program whose input domain is the integers between $-N$ and N, where $N \geq 1$, let $D_1 = \{x | -N \leq x \leq N\}$, $D_2 = \{x | -N+1 \leq x \leq N\}$, $D_3 = \{x | -N \leq x \leq N-1\}$, and let C_1 and C_2 be the criteria whose subdomains are $\mathcal{SD}_{C1} = \{D_2, D_3\}$ and $\mathcal{SD}_{C2} = \{D_1\}$. $D_1 = D_2 \cup D_3$, and so C_1 covers C_2 for P. Let S be a specification such that the only failure-causing inputs are $-N$ and N. Then

$$M_1(C_1, P, S) = \frac{1}{2N} < \frac{2}{2N+1} = M_1(C_2, P, S).$$

□

Note that in the proof of Theorem 6, the fact that the failure-causing inputs lie outside the intersection of D_2 and D_3 contributes to the fact that D_1 has a higher concentration of failure-causing inputs than D_2 or D_3. We next show that this phenomenon can occur with "real" adequacy criteria, by exhibiting a program and specification for which $M_1(\text{all-p-uses}, P, S) < M_1(\text{all-edges}, P, S)$.

Example 2:
Consider the program shown in Figure 2.[3] Assume that the inputs x and y lie in the range $[-N+1 \ldots N]$, and let $n = 2N$. The executable edges and executable definition-p-use associations are shown in column 2 of Table 2. Their corresponding subdomains are shown in column 3 of Table 2.

Assume that $3 \leq MAX \leq N$. Notice that

$$P(x, y) = \begin{cases} y+1 & \text{if } y \geq \text{MAX} \\ \text{MAX} & \textit{otherwise} \end{cases}$$

On any input (x, y) such that $y \leq MAX - 2$, the path taken will go through either node 4 or node 5 on the first traversal of the loop (depending on the parity of $x + y$), and then will alternate between going through node 4 and going through node 5 on subsequent traversals of the loop. Consequently, among all the inputs that cause execution of edge (6,3), all of them *except* those in which $y = MAX - 2$ will cover *both* of the definition-use associations (4,(6,3),y) and (5,(6,3),y).

We now manufacture a specification for which the program will fail on precisely those inputs that lie within the subdomain corresponding to edge (6,3), but outside the intersection of subdomains corresponding to (4,(6,3),y) and (5,(6,3),y), i.e., in $D_g - (D_n \cap$

[3]Note that the fact that the program has the same statement in both branches of the **if-then-else** statement is merely a convenience which makes it easy to achieve the desired behavior. It is certainly possible to devise a program without this characteristic which also has the desired behavior.

id	edge/ dua	D_i	d_i	m_i
a	(1,2)	whole input domain	n^2	n
b	(2,3)	whole input domain	n^2	n
c	(3,4)	$(y \leq MAX - 2)$ or $(odd(y + x))$	$(N + MAX - 2)n$ $+(N - MAX + 2)n/2$	n
d	(3,5)	$(y \leq MAX - 2)$ or $(even(y + x))$	$(N + MAX - 2)n$ $+(N - MAX + 2)n/2$	n
e	(4,6)	$(y \leq MAX - 2)$ or $(odd(y + x))$	$(N + MAX - 2)n$ $+(N - MAX + 2)n/2$	n
f	(5,6)	$(y \leq MAX - 2)$ or $(even(y + x))$	$(N + MAX - 2)n$ $+(N - MAX + 2)n/2$	n
g	(6,3)	$(y \leq MAX - 2)$	$(N + MAX - 2)n$	n
h	(6,7)	whole input domain	n^2	n
i	(2,(3,4),x)	$odd(y + x)$	$n^2/2$	$n/2$
j	(2,(3,5),x)	$even(y + x)$	$n^2/2$	$n/2$
k	(2,(3,4),y)	$odd(y + x)$	$n^2/2$	$n/2$
l	(2,(3,5),y)	$even(y + x)$	$n^2/2$	$n/2$
m	(4,(3,5),y)	$(y < MAX - 2)$ or $((y = MAX - 2)$ and $odd(y + x))$	$n(N + MAX - 3) + n/2$	$n/2$
n	(4,(6,3),y)	$(y < MAX - 2)$ or $((y = MAX - 2)$ and $odd(y + x))$	$n(N + MAX - 3) + n/2$	$n/2$
o	(5,(6,3),y)	$(y < MAX - 2)$ or $((y = MAX - 2)$ and $even(y + x))$	$n(N + MAX - 3) + n/2$	$n/2$
p	(5,(3,4),y)	$(y < MAX - 2)$ or $((y = MAX - 2)$ and $even(y + x))$	$n(N + MAX - 3) + n/2$	$n/2$
q	(4,(6,7),y)	$(odd(MAX - 1 + x)$ and $(y < MAX))$ or $((y \geq MAX)$ and $odd(y + x))$	$n^2/2$	$n/2$
r	(5,(6,7),y)	$(even(MAX - 1 + x)$ and $(y < MAX))$ or $((y \geq MAX)$ and $even(y + x))$	$n^2/2$	$n/2$

Table 2: Subdomains of Program shown in Figure 2

```
program P(input,output);
const MAX = ...;
var x, y : integer;
begin
{1}
{2}     read(x,y);
{3}     repeat if odd(y+x)
{4}             then y := y +1
{5}             else y := y +1;
{6}     until (y >= MAX);
{7}     writeln(y)
end.
```

Figure 2: Program for which M_1(all-p-uses) $<$ M_1(all-edges)

D_o). Let

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

Then $P(x,y) = S(x,y)$ unless $y = \text{MAX} - 2$, as desired.

The subdomains for the all-edges criterion are shown in the first eight rows of Table 2, while those for the all-p-uses criterion are shown in the last ten rows. Thus,

$$\begin{aligned} M_1(\text{all-edges}, P, S) &= \frac{m_g}{d_g} = \frac{1}{N + MAX - 2} \\ &> \frac{m_i}{d_i} = \frac{1}{2N} = M_1(\text{all-p-uses}, P, S) \\ &> \frac{m_n}{d_n} = \frac{m_o}{d_o} = \frac{1}{2N + 2MAX - 5}. \end{aligned}$$

Note that this last inequality holds since $MAX \geq 3$ by assumption.

□

We next investigate the relationship between covering and measure M_2.

Theorem 7 *There exists a program P, specification S, and criteria C_1 and C_2 such that C_1 covers C_2 but $M_2(C_1, P, S) < M_2(C_2, P, S)$.*

Figure 3: Program illustrating $M_2(\text{all-uses}, P, S) < M_2(\text{all-edges}, P, S)$

Proof:

By Corollary 2, it suffices to exhibit a program P and specification S for which $M_2(\text{all-uses}, P, S) < M_2(\text{all-edges}, P, S)$. Consider program P whose flow graph is shown in Figure 3. Assume that the input domain is $\{x | 0 \leq x < 200\}$, that c is a constant, and that the specification S is such that the set of failure-causing inputs is $\{0, 197, 198, 199\}$. The subdomains arising from the all-edges and all-uses criteria and the corresponding values of m and d are as shown in Table 3. Notice that each criterion induces nine subdomains for this program, and in each case two of the subdomains contain no failure-causing inputs.

Since

$$M_2(\text{all-uses}, P, S) = 1 - (1 - \frac{1}{100})^2(1 - \frac{3}{100})^2(1 - \frac{1}{50})^3 = 0.13$$

and

$$M_2(\text{all-edges}, P, S) = 1 - (1 - \frac{4}{200})^2(1 - \frac{3}{100})^2(1 - \frac{1}{100})(1 - \frac{1}{50})^2 = 0.14$$

$M_2(\text{all-uses}, P, S) < M_2(\text{all-edges}, P, S)$, as desired. $\square$

edge/ dua	D	d	m
(1,2)	$0 \leq x < 200$	200	4
(2,3)	$0 \leq x < 100$	100	1
(2,4)	$100 \leq x < 200$	100	3
(3,5)	$0 \leq x < 50$	50	1
(3,6)	$50 \leq x < 100$	50	0
(4,7)	$100 \leq x < 200$	100	3
(5,7)	$0 \leq x < 50$	50	1
(6,7)	$50 \leq x < 100$	50	0
(7,8)	$0 \leq x < 200$	200	4
(1,(2,3),x)	$0 \leq x < 100$	100	1
(1,3,x)	$0 \leq x < 100$	100	1
(1,(2,4),x)	$100 \leq x < 200$	100	3
(1,8,x)	$100 \leq x < 200$	100	3
(3,(3,5),x)	$0 \leq x < 50$	50	1
(3,5,x)	$0 \leq x < 50$	50	1
(3,(3,6),x)	$50 \leq x < 100$	50	0
(3,8,x)	$50 \leq x < 100$	50	0
(5,8,x)	$0 \leq x < 50$	50	1

Table 3: Subdomains of Program shown in Figure 3

The following corollary follows from the fact that the number of all-edges subdomains is equal to the number of all-uses subdomains in the program in Figure 3.

Corollary 3 *There exists a program P, specification S, and criteria C_1 and C_2 such that C_1 covers C_2 but $M_2(C_1, P, S) = M_3(C_1, P, S, 1) < M_3(C_2, P, S, r)$, where $r = |\mathcal{SD}_{C1}(P, S)|/|\mathcal{SD}_{C2}(P, S)|$.*

3.3 The partitions relation

Definition: Let C_1 and C_2 be criteria. C_1 *partitions* C_2 *for (P,S)* if for every subdomain $D \in \mathcal{SD}_{C2}(P, S)$ there is a non-empty collection $\{D_1, \ldots, D_n\}$ of *pairwise disjoint* subdomains belonging to $\mathcal{SD}_{C1}(P, S)$ such that $D_1 \cup \ldots \cup D_n = D$. C_1 *universally partitions* C_2 if for every program/specification pair (P,S), C_1 partitions C_2 for (P,S). Note that the term partition is used here in the strict mathematical sense.

Observation 3 *For each program P and specification S, the partitions relation is reflexive and transitive. If C_1 partitions C_2 for (P,S) then C_1 covers C_2 for (P,S). If $\mathcal{SD}_{C2}(P, S) \subseteq \mathcal{SD}_{C1}(P, S)$ then C_1 partitions C_2 for (P,S).*

The next example illustrates the distinction between the covers and partitions relations.

Example 3:
Consider again the program whose flow graph is shown in Figure 2. The executable edges and executable definition-p-use associations and the corresponding subdomains are shown in Table 2.

Recall that the all-p-uses criterion universally covers the all-edges criterion. To see that all-p-uses does not partition all-edges for this program, consider edge $(6, 3)$ (subdomain g). The only elements of $\mathcal{SD}_{all-p-uses}$ that are contained in D_g are D_m=D_n and D_p=D_o, since all the other subdomains contain inputs in which $y \geq MAX - 1$. Since D_n and D_o have a non-empty intersection, and neither is equal to D_g, all-p-uses does not partition all-edges for this program.

The executable definition-c-use associations for this program are shown in Table 4. Note that for each definition-c-use association except (1,2,input), there is a definition-p-use association with the same subdomain. Consequently, the above argument also shows that for this program all-uses does not partition all-edges since the set of definition-use associations for the all-uses criterion is simply the union of the sets of associations for all-p-uses and all-c-uses. □

Next, we examine the relationship between partitioning and fault exposing ability, as assessed by the measures M_1, M_2, and M_3.

id	def-c-use assoc.	equiv. def-p-use assoc.
s	(1,2,input)	—
t	(2,4,y)	(2,(3,4),y) [k]
u	(2,5,y)	(2,(3,5),y) [l]
v	(4,5,y)	(4,(3,5),y) [m]
w	(5,4,y)	(5,(3,4),y) [p]
x	(4,7),y)	(4,(6,7),y) [q]
y	(5,7,y)	(5,(6,7),y) [r]

Table 4: Definition-c-use associations of Program shown in Figure 2

Theorem 8 *If C_1 partitions C_2 for program P and specification S then $M_1(C_1, P, S) \geq M_1(C_2, P, S)$.*

Proof:
Let $D_0 \in \mathcal{SD}_{C2}(P, S)$. Let $D_1, \ldots, D_n$ be disjoint subdomains belonging to $\mathcal{SD}_{C1}(P, S)$ such that

$$D_0 = D_1 \cup \ldots \cup D_n$$

Then $m_0 = m_1 + \ldots + m_n$ and $d_0 = d_1 + \ldots + d_n$. Thus $max_{i=1}^{n}(\frac{m_i}{d_i})$ is minimized when each $\frac{m_i}{d_i} = \frac{m_0}{d_0}$, so $max_{i=1}^{n}(\frac{m_i}{d_i}) \geq \frac{m_0}{d_0}$, and therefore $M_1(C_1, P, S) \geq M_1(C_2, P, S)$. $\Box$

Theorem 9 *There exists a program P, specification S, and criteria C_1 and C_2 such that C_1 partitions C_2 for P and S, but $M_2(C_1, P, S) < M_2(C_2, P, S)$.*

Proof:
Consider again the program and specification in the proof of Theorem 7, for which M_2(all-uses, P, S) was shown to be less than M_2(all-edges, P, S). All-uses partitions all-edges for this program, since the only element of $\mathcal{SD}_{\text{all-edges}}$ that does not belong to $\mathcal{SD}_{\text{all-uses}}$ is $\{x|0 \leq x < 200\}$, which is equal to a union of disjoint elements of $\mathcal{SD}_{all-uses}$, $\{x|0 \leq x < 100\} \cup \{x|100 \leq x < 200\}$. $\Box$

Corollary 4 *There exists a program P, specification S, and criteria C_1 and C_2 such that C_1 partitions C_2 for P and S, but $M_2(C_1, P, S) = M_3(C_1, P, S, 1) < M_3(C_2, P, S, r)$, where $r = |\mathcal{SD}_{C1}(P, S)|/|\mathcal{SD}_{C2}(P, S)|$.*

3.4 The properly covers relation

We have seen that the fact that C_1 covers C_2 does not guarantee that C_1 is better at detecting faults according to M_2. However, when an additional constraint on the nature of the covering is added, the situation improves.

Definition: Let $\mathcal{SD}_{C1}(P,S) = \{D^1_1, \ldots, D^1_m\}$, and let $\mathcal{SD}_{C2}(P,S) = \{D^2_1, \ldots, D^2_n\}$. C_1 *properly covers* C_2 *for (P,S)* if there is a multi-set

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

such that $\mathcal{M} \subseteq \mathcal{SD}_{C1}(P,S)$ and

$$\begin{aligned} D^2_1 &= D^1_{1,1} \cup \ldots \cup D^1_{1,k_1} \\ &\vdots \\ D^2_n &= D^1_{n,1} \cup \ldots \cup D^1_{n,k_n} \end{aligned}$$

Note that the number of occurrences of any subdomain $D^1_{i,j}$ in the above expression is less than or equal to the number of occurrences of that subdomain in the multi-set $\mathcal{SD}_{C1}$. C_1 *universally properly covers* C_2 if for every program P and specification S, C_1 properly covers C_2 for (P,S).

Observation 4 *The properly covers relation is reflexive and transitive. If C_1 properly covers C_2 for (P,S) then C_1 covers C_2 for (P,S). If $\mathcal{SD}_{C2}(P,S) \subseteq \mathcal{SD}_{C1}(P,S)$ then C_1 properly covers C_2 for (P,S). If the elements of $\mathcal{SD}_{C2}(P,S)$ are disjoint (i.e., if C_2 induces a true partition of the input domain), then C_1 covers C_2 if and only if C_1 properly covers C_2.*

The next example illustrates the distinction between the covers and properly covers relations.

Example 4:
Consider a program P with integer input domain $\{x|0 \leq x \leq 3\}$, and criteria C_1 and C_2 such that $\mathcal{SD}_{C1} = \{D_a, D_b, D_c\}$, and $\mathcal{SD}_{C2} = \{D_d, D_e\}$ where $D_a = \{0\}$, $D_b = \{1,2\}$, $D_c = \{3\}$, $D_d = \{0,1,2\}$, and $D_e = \{1,2,3\}$. Then $D_d = D_a \cup D_b$ and $D_e = D_c \cup D_b$, so C_1 covers (in fact partitions) C_2. However C_1 does not *properly* cover C_2 because the subdomain D_b is needed in the coverings of both D_d and D_e, but only occurs once in the multi-set $\mathcal{SD}_{C1}$.

On the other hand consider criterion C_3 where $\mathcal{SD}_{C3} = \{D_a, D_b, D_b, D_c\}$. C_3 does properly cover C_2; it is legitimate to use D_b twice in the covering, since it occurs twice in $\mathcal{SD}_{C3}$. Note that such duplication of subdomains frequently occurs in real subdomain-based criteria; for example, in Table 2, $D_a = D_b = D_h$, $D_c = D_e$, and $D_d = D_f$.

Also consider C_4 with subdomains $D_f = \{0,1\}$, $D_g = \{2,3\}$, $D_h = \{1\}$, and $D_i = \{2\}$. Then C_4 properly covers C_2 because $D_d = D_f \cup D_i$ and $D_e = D_g \cup D_h$. Since no C_4 subdomain is contained in the intersection of C_2 subdomains, there is no danger that one of them will need to be used more than once in covering all the C_2 subdomains. $\square$

Example 5:
The next example also illustrates the the properly covers relation. We show that for the

program P in Figure 2, all-uses properly covers all-edges, but all-p-uses does not properly cover all-edges.

Consider the subdomains arising from the edges and definition-p-use associations of P, shown in Table 2 and the definition-c-use associations shown in Table 4. Recall that $\mathcal{SD}_{\text{all-uses}} = \mathcal{SD}_{\text{all-p-uses}} \cup \mathcal{SD}_{\text{all-c-uses}}$. Note that

$$\begin{aligned}
D_a &= D_s \\
D_b &= D_i \cup D_j \\
D_c &= D_k \cup D_p \\
D_d &= D_l \cup D_m \\
D_e &= D_t \cup D_w \\
D_f &= D_u \cup D_v \\
D_g &= D_n \cup D_o \\
D_h &= D_q \cup D_r
\end{aligned}$$

Since the multi-set

$$\{D_i, D_j, D_k, D_l, D_m, D_n, D_o, D_p, D_q, D_r, D_s, D_t, D_u, D_v, D_w\} \subseteq \mathcal{SD}_{\text{all-uses}}(P, S),$$

this is a proper covering. On the other hand, when the subdomains arising just from the all-p-uses criterion are used to cover the all-edges subdomains, it is necessary to use some subdomains more often than they occur in the multi-set $\mathcal{SD}_{\text{all-p-uses}}(P, S)$. Consequently, all-p-uses does not properly cover all-edges for P and S. □

The next three theorems show that if C_1 properly covers C_2, then C_1 is guaranteed to be better according to M_2, but not necessarily according to M_1 or M_3.

Theorem 10 *There exists a program P, specification S, and criteria C_1 and C_2 such that C_1 properly covers C_2 for P and S, but $M_1(C_1, P, S) < M_1(C_2, P, S)$.*

Proof:
Consider again the program P and specification S in Figure 2. As shown in Example 5, all-uses *properly covers* all-edges for P and S. Since for this program $M_1(\text{all-p-uses}, P, S) = M_1(\text{all-uses}, P, S)$, Example 2 shows that $M_1(\text{all-edges}, P, S) > M_1(\text{all-uses}, P, S)$.[4] □

On the other hand, the fact that C_1 properly covers C_2 does guarantee that it is better according to M_2.

Theorem 11 *If C_1 properly covers C_2 for program P and specification S, then $M_2(C_1, P, S) \geq M_2(C_2, P, S)$.*

[4]In fact, given program P, specification S, and criteria C_1 and C_2 such that C_1 covers C_2 for P and S, one can always manufacture a criterion C_1' such that C_1' properly covers C_2 for P and S, and $M_1(C_1', P, S) = M_1(C_1, P, S)$, by simply adding duplicates of certain C_1 subdomains to $\mathcal{SD}_{C1}(P, S)$.

The following lemma is proved in the Appendix:

Lemma 1 *Let $D = D_1 \cup \ldots \cup D_n$. Then $f(D) \geq f(D_1, \ldots, D_n)$, where*

$$f(D_1, \ldots, D_n) = \prod_{i=1}^{n} (1 - \frac{m_i}{d_i}).$$

We now prove Theorem 11.

Proof:
Assume C_1 properly covers C_2 for program P and specification S. Let $\mathcal{SD}_{C2}(P,S) = \{D_1^2, \ldots, D_n^2\}$, let $\mathcal{SD}_{C1}(P,S) = \{D_1^1, \ldots, D_m^1\}$, 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 multi-set such that $\mathcal{M} \subseteq \mathcal{SD}_{C1}(P,S)$ and

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

Then

$$\begin{aligned} f(D_1^2, \ldots, D_n^2) &= \prod_i f(D_i^2) && (1) \\ &\geq \prod_{i,j} f(D_{i,j}^1) && (2) \\ &\geq f(D_1^1, \ldots, D_m^1) && (3) \end{aligned}$$

Line (2) follows from Lemma 1, and line (3) holds because no $D_{i,j}^1$ occurs more times in the product in line (2) than in the product in line (3). Consequently, $M_2(C_1, P, S) \geq M_2(C_2, P, S)$. $\Box$

Theorem 12 *There exists a program P, specification S, and criteria C_1 and C_2 such that C_1 properly covers C_2 for P and S, but $M_2(C_1, P, S) = M_3(C_1, P, S, 1) < M_3(C_2, P, S, r)$, where $r = |\mathcal{SD}_{C1}(P,S)|/|\mathcal{SD}_{C2}(P,S)|$.*

Proof:
Let $\mathcal{SD}_{C1} = \{D_a, D_b, D_b, D_c\}$, and $\mathcal{SD}_{C2} = \{D_d, D_e\}$ where $D_a = \{0\}$, $D_b = \{1,2\}$, $D_c = \{3\}$, $D_d = \{0,1,2\}$, and $D_e = \{1,2,3\}$. As shown in Example 4, (where C_1 was called C_3), C_1 properly covers (in fact properly partitions) C_2. Let P be a program with an integer input domain $\{x | 0 \leq x \leq 3\}$ that is correct with respect to its specification

for every input except $x = 2$, so that $d_a = d_c = 1$, $d_b = 2$, $d_d = d_e = 3$, and $m_a = m_c = 0$, $m_b = m_d = m_e = 1$. Then

$$M_3(C_2, P, S, 2) = 1 - (1 - \frac{1}{3})^2 (1 - \frac{1}{3})^2 = \frac{65}{81}$$

and

$$M_3(C_1, P, S, 1) = 1 - (1 - \frac{1}{2})^2 (1 - 0)(1 - 0) = \frac{3}{4},$$

so $M_3(C_2, P, S, 2) > M_3(C_1, P, S, 1)$. □

3.5 The properly partitions relation

Definition: Let $\mathcal{SD}_{C1}(P, S) = \{D^1_1, \ldots, D^1_m\}$, and let $\mathcal{SD}_{C2}(P, S) = \{D^2_1, \ldots, D^2_n\}$. C_1 *properly partitions* C_2 *for (P,S)* if there is a multi-set

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

such that $\mathcal{M} \subseteq \mathcal{SD}_{C1}(P, S)$,

$$\begin{aligned} D^2_1 &= D^1_{1,1} \cup \ldots \cup D^1_{1,k_1} \\ &\vdots \\ D^2_n &= D^1_{n,1} \cup \ldots \cup D^1_{n,k_n} \end{aligned}$$

and for each i, the collection $\{D^1_{i,1}, \ldots, D^1_{i,k_i}\}$ is pairwise disjoint. C_1 *universally properly partitions* C_2 if for every program P and specification S, C_1 properly partitions C_2 for (P,S).

Observation 5 *The properly partitions relation is reflexive and transitive. If C_1 properly partitions C_2 for (P,S) then C_1 partitions C_2 for (P,S) and C_1 properly covers C_2 for (P,S). If $\mathcal{SD}_{C2}(P, S) \subseteq \mathcal{SD}_{C1}(P, S)$ then C_1 properly partitions C_2 for (P,S). If the elements of $\mathcal{SD}_{C2}(P, S)$ are pairwise disjoint (i.e., if C_2 induces a true partition of the input domain) then C_1 partitions C_2 if and only if C_1 properly partitions C_2.*

Theorem 13 *If C_1 properly partitions C_2 for program P and specification S then $M_1(C_1, P, S) \geq M_1(C_2, P, S)$.*

Proof:
If C_1 properly partitions C_2 for (P,S) then C_1 partitions C_2 for (P,S), so the result follows from Theorem 8. □

Theorem 14 *If C_1 properly partitions C_2 for program P and specification S then $M_2(C_1, P, S) \geq M_2(C_2, P, S)$.*

Proof:
If C_1 properly partitions C_2 for (P,S) then C_1 properly covers C_2 for (P,S), so the result follows from Theorem 11. $\Box$

Theorem 15 *There exists a program P, specification S, and criteria C_1 and C_2 such that C_1 properly partitions C_2 but $M_2(C_1, P, S) = M_3(C_1, P, S, 1) < M_3(C_2, P, S, r)$, where $r = |\mathcal{SD}_{C1}(P,S)|/|\mathcal{SD}_{C2}(P,S)|$.*

Proof:
In the proof of Theorem 12, C_1 properly partitions C_2. $\Box$

4 Conclusion

In this paper, we have defined several relationships between software testing criteria, each induced by a relation between the corresponding multi-sets of subdomains. We have also investigated whether for each of these relations R, $R(C_1, C_2)$ guarantees that C_1 is better at detecting faults than C_2, according to various measures. Each of our three measures of fault-detecting ability is related to the probability that a test suite selected according to a particular strategy will detect a fault.

The first relation examined was the *narrows* relation, which was shown to be closely related to a commonly used means of comparing criteria called subsumption. We showed that the fact that criterion C_1 narrows criterion C_2 does not guarantee that C_1 is better at detecting faults than C_2 according to any of the measures. The next relation examined was the *covers* relation, which strengthens the narrows relation in a natural way. Some well known pairs of criteria, such as all-uses and all-edges are related to one another according to this relation. We also showed that the fact that criterion C_1 covers criterion C_2 does not guarantee that C_1 is better at detecting faults than C_2 according to any of the measures. The *partitions* relation further strengthens the covers relation. While the fact that C_1 partitions C_2 does guarantee that C_1 is better according to one of the measures, M_1, it does not guarantee that C_1 is better according to the other measures.

The last two relations between criteria, the *properly covers* and *properly partitions* relations were designed to overcome some of the deficiencies of the covers and partitions relations. We proved that the fact that C_1 properly covers C_2 does guarantee that C_1 is better than C_2 according to measure M_2, and that the fact that C_1 properly partitions C_2 guarantees that C_1 is better than C_2 according to measures M_1 and M_2.

Since for most criteria of interest the universally narrows relation is equivalent to the subsumes relation, one could interpret our results as saying that subsumption is a poor basis for comparing criteria. However, it is important to note that the results here are worst case results in the sense that we consider only whether or not the fact that one criterion subsumes another *guarantees* improved fault-detecting ability. The question of what C_1 subsuming (or narrowing, covering, or partitioning) C_2 tells us about their relative ability to detect faults in "typical" programs remains open.

We have recently shown that the all-p-uses and all-uses criteria *do* properly cover a variant of branch testing known as *decision coverage* and investigated the relationships among several other subdomain-based criteria, including other data flow testing criteria, mutation-testing, and multiple-condition coverage [5]. Other directions for future analytical research include finding conditions on programs that guarantee that C_1 properly covers C_2 for various well-known criteria C_1 and C_2; finding conditions under which C_1 is guaranteed to be better than C_2 according to M_3, and finding weaker conditions that guarantee that C_1 is better than C_2 according to M_2. Our results also suggest several more pragmatic research problems, including the design of new criteria that are guaranteed to properly cover commonly used criteria such as all-edges, and the design of test data selection tools that approximate the selection strategies upon which our results are based.

Acknowledgments

Some of the results in this paper appeared in the *Proceedings of the ACM SIGSOFT '91 Conference on Software for Critical Systems* [4].

References

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

[2] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings Fourth Symposium on Software Testing, Analysis, and Verification*, pages 154–164. ACM Press, Oct. 1991.

[3] 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.

[4] 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.

[5] 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.

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

[7] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):206–215, Dec. 1990.

[8] P. Herman. A data flow analysis approach to program testing. *The Australian Computer Journal*, 8(3):92–96, Nov. 1976.

[9] B. Jeng and E. J. Weyuker. Some observations on partition testing. In *Proceedings ACM SIGSOFT Third Symposium on Software Testing, Analysis, and Verification*, pages 38–47. ACM Press, Dec. 1989.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

A Appendix

In this appendix we prove Lemma 1. We begin with a few preliminary lemmas.

Lemma 2 *Assume $d_1, d_2 > 0$, $0 \le x \le d_1$, $0 \le x \le d_2$, $0 \le m_1 \le d_1 - x$, $0 \le m_2 \le d_2 - x$. Then*

$$\frac{m_1 + m_2}{d_1 + d_2 - x} \le (1 - (1 - \frac{m_1}{d_1})(1 - \frac{m_2}{d_2}))$$

Proof:
Since

$$(1-(1-\frac{m_1}{d_1})(1-\frac{m_2}{d_2})) = \frac{m_1d_2+m_2d_1-m_1m_2}{d_1d_2},$$

it suffices to show that

$$\begin{aligned} 0 &\leq (m_1d_2+m_2d_1-m_1m_2)(d_1+d_2-x)-(m_1+m_2)(d_1d_2) \\ &= m_2d_1(d_1-m_1-x)+m_1d_2(d_2-m_2-x)+m_1m_2x \end{aligned}$$

But this follows immediately from the assumptions that (d_1-m_1-x) , (d_2-m_2-x), d_i, m_i, and x are all non-negative. □

Lemma 3 *Let* $D_3 = D_1 \cup D_2$. *Then*

$$\frac{m_3}{d_3} < (1-(1-\frac{m_1}{d_1})(1-\frac{m_2}{d_2}))$$

Proof:

We want to show that

$$1-f(D_3) < 1-f(D_1,D_2),$$

or that

$$f(D_3) > f(D_1,D_2),$$

where

$$f(D_1,\ldots,D_n) = \prod_{i=1}^{n}(1-\frac{m_i}{d_i}).$$

We begin by showing that for given values d_3 and m_3, the value of $f(D_1,D_2)$ is maximized when $D_1 \cap D_2$ contains as few failure-causing inputs as possible. This is clear intuitively, since points in the intersection are more likely to be selected; thus when as many of them as possible are *non*-failure-causing, the probability that no failure-causing input is selected is maximal.

More formally, let $D_a = D_3 - D_2$, $D_b = D_3 - D_1$, and $D_c = D_1 \cap D_2$, and let d_a, d_b, d_c and x_a, x_b, x_c be their sizes and number of *non*-failure-causing inputs, respectively. Then

$$f(D_1,D_2) = (\frac{x_a+x_c}{d_a+d_c})(\frac{x_b+x_c}{d_b+d_c}).$$

Suppose that it is possible to swap one non-failure-causing input from D_a with one failure-causing input from D_c. Doing so leaves the values of d_a, d_b, d_c, and x_b unchanged, but decrements x_a and increments x_c, yielding

$$f(D_1',D_2') = (\frac{x_a-1+x_c+1}{d_a+d_c})(\frac{x_b+x_c+1}{d_b+d_c}) > f(D_1,D_2).$$

Similarly, swapping a non-failure-causing input from D_b with a failure-causing input from D_c increases f.

Thus, to prove the lemma, it suffices to consider the following two cases:
Case 1: D_c consists entirely of non-failure-causing inputs.
In this case, letting $x = d_c = |D_1 \cap D_2|$, the hypotheses of Lemma 2 are satisfied, so

$$\frac{m_1 + m_2}{d_1 + d_2 - d_c} \leq (1 - (1 - \frac{m_1}{d_1})(1 - \frac{m_2}{d_2})).$$

Since $m_3 = m_1 + m_2$ and $d_3 = d_1 + d_2 - d_c$, this gives the desired result.
Case 2: D_a and D_b consist entirely of failure-causing inputs. We want to show that $f(D_3) - f(D_1, D_2) \geq 0$, where

$$f(D_3) = \frac{x_c}{d_a + d_b + d_c} \text{and } f(D_1, D_2) = (\frac{x_c}{d_a + d_c})(\frac{x_c}{d_b + d_c})$$

It suffices to show

$$\begin{aligned} 0 &\leq x_c[(d_a + d_c)(d_b + d_c) - x_c(d_a + d_b + d_c)] \\ &= x_c[d_a(d_b + d_c - x_c) + d_b(d_c - x_c) + d_c(d_c - x_c)] \end{aligned}$$

But this follows immediately from the fact that $0 \leq x_c \leq d_c$. □

We now prove Lemma 1.

Lemma 1 *Let $D = D_1 \cup \ldots \cup D_n$. Then $f(D) \geq f(D_1, \ldots, D_n)$.*

Proof:
Proof by induction on n. The base case, $n = 1$ is trivial. Now assuming that

$$D = D_1 \cup \ldots \cup D_k \Rightarrow f(D) \geq f(D_1, \ldots D_k),$$

we want to show that

$$D' = D_1 \cup \ldots \cup D_{k+1} \Rightarrow f(D') \geq f(D_1, \ldots D_{k+1}).$$

Since $D' = D \cup D_{k+1}$,

$$\begin{aligned} f(D') &\geq f(D, D_{k+1}) && \text{(lemma 3)} \\ &= f(D)f(D_{k+1}) && \text{(definition of f)} \\ &\geq f(D_1, \ldots, D_k)f(D_{k+1}) && \text{(inductive hyp.)} \\ &= f(D_1, \ldots, D_k, D_{k+1}) && \text{(definition of f)} \end{aligned}$$

□

List of Figures

List of Tables