An Analytical Comparison of the Fault-detecting Ability of Data Flow Testing Techniques

Phyllis G. Frankl
Computer Science Department
Polytechnic University
6 Metrotech Center
Brooklyn, N.Y. 11201

Elaine J. Weyuker
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

Abstract

This paper compares several data flow based software testing criteria to one another and to branch testing. It has previously been shown that the facts that criterion $C_1$ subsumes criterion $C_2$, does not guarantee that $C_1$ is better at detecting faults than $C_2$. However, if a certain stronger relation between the criteria holds, then for any program and any specification, $C_1$ 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 $C_1$ subdomain is at least as likely to detect a fault as a test suite similarly selected using $C_2$. This paper shows that under those conditions, the expected number of failure-causing inputs in the $C_1$ test suite is also at least as great as that of the $C_2$ test suite. These results are used to compare a number of data flow testing criteria to one another and to branch testing.

1 Introduction

Over the last two decades, a wide variety of software testing techniques, including several based on data flow analysis, have been proposed and investigated. These criteria have generally been justified by appeals to intuition. However, surprisingly little concrete information about the fault-detecting ability of these criteria has been gathered.

One factor that makes it difficult to compare the fault-detecting ability of testing techniques is that typically a large number of different test suites satisfy the criterion for a given program and specification. Often, some of these test suites expose a fault, while others do not. Consequently, a reasonable question to ask is whether test suites developed for a given program and/or specification using one technique are more likely to detect a fault than test suites developed using another technique.

Most previous analytical comparisons of testing criteria have been based on the subsumes relation and its variants. Criterion $C_1$ subsumes criterion $C_2$ if for every program $P$ and specification $S$, every test suite that satisfies $C_1$ for $(P, S)$ also satisfies $C_2$. Hamlet pointed out that the subsumes relation can be "misleading" since even when $C_1$ subsumes $C_2$, it is possible that a $C_2$-adequate test suite for some $(P, S)$ detects a fault while a $C_1$-adequate test suite fails to do so [8]. We subsequently proved that it is possible for $C_1$ to subsume $C_2$ but for $C_2$ to be more likely to detect a fault than $C_1$, in a sense to be described below. In addition, we defined a more demanding relation between criteria, the universally properly covers relation, and showed that if $C_1$ universally properly covers $C_2$ then for any $(P, S)$, $C_1$ is always at least as likely to detect a fault as $C_2$ [5, 7].

This paper applies these results to compare several data flow testing criteria [13, 11, 10] to one another and to branch testing in terms of fault-detecting ability.

Notice that the results in this paper are probabilistic results. When we show that criterion $C_1$ universally properly covers criterion $C_2$, we show that $C_1$-adequate test suites (selected according to a particular strategy described below) are guaranteed to be more likely to detect a fault than $C_2$-adequate test suites, for every program, no matter what bugs occur in the program. On the other hand, when $C_1$ does not universally properly cover $C_2$, we only show the existence of programs for which $C_2$ is likely to do better than

---

*Supported in part by NSF Grant CCR-9206910 and by the New York State Science and Technology Foundation Center for Advanced Technology program.

†Supported in part by NASA grant NASA-8920701 and by NASA grant NASA-1-1238.
Thus positive results of the form $C_1$ universally properly covers $C_2$ give practitioners an important reason to use $C_1$ rather than $C_2$ (provided that our abstract model of the testing process behaves in a sufficiently similar manner to the practitioners' actual testing process), and negative results of the form $C_1$ does not universally properly cover $C_2$, give practitioners a warning that $C_1$ may not be a good choice.

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 which indicate 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 $S_1$.

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, 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. 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 $P$ for $S$ 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 software testing are based on the idea of dividing the input domain of a program into subsets, called subdomains, then selecting at least one test case from each subdomain. These techniques are sometimes 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.

We will say that a testing criterion $C$ is subdomain-based if, for each program $P$ and specification $S$, there is a non-empty multi-set of subdomains, $SD_C(P, S)$, such that $C$ requires the selection of one or more test cases from each subdomain in $SD_C(P, S)$.

In general, $SD_C(P, S)$ 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-edges criterion, in which each subdomain corresponds to the set of inputs that cause the execution of a particular edge in the program’s flow graph. If two different edges are traversed 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 edge also covers another or because the semantics of the program force two seemingly independent edges to be traversed by exactly the same test cases. Of course, given a criterion $C$ for which the multi-set $SD_C(P, S)$ contains duplicates, one can define a new criterion $C'$ in which the duplicates are eliminated, but as we shall discuss below, it is frequently more efficient to keep the duplicates (and hence select “extra” test cases) than to check to see whether duplicates exist. None of the results in this paper are dependent on this convention.

Note that since $SD_C(P, S)$ 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 $SD_C(P, S)$.

Throughout this paper, all testing criteria will be universally applicable subdomain-based criteria, unless otherwise noted. In fact, many well-known criteria are not universally applicable [4]. 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 is not executable by any input. However, it is the universality applicable analogs of those criteria that are actually usable in practice. Formally, these are obtained by removing the empty subdomains from $SD_C(P, S)$. Although in general, determining whether a subdomain is empty is undecidable, in practice, it is often easy for testers to determine whether a subdomain is empty by inspecting the program code.

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 do not address the question of determining the nature of the fault in $P$ that caused the failure, only with determining that some fault exists.

Intuitively, a testing strategy is good if it is likely to require the selection of at least one failure-causing input, if any exist, or to detect many faults. It is with this intuition that we present two measures of effectiveness of testing criteria, $M$ and $E$.

Let $\mathcal{SD}_C(P, S) = \{D_1, \ldots, D_n\}$. Let $d_i = |D_i|$, the size of subdomain $D_i$, let $m_i = \text{the number of failure-cause inputs in } D_i$, 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.

Let $\mathcal{SD}_C(P, S) = \{D_1, \ldots, D_n\}$. Again we assume independent random selection of one test case from each subdomain, using a uniform distribution, and let

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

This represents the expected number of failures detected.

It could be argued that the test selection model upon which $M$ and $E$ are based 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 testing 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 scheme. Manually crafting test cases is generally 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.

The measures $M$ and $E$ have an additional property which makes them reasonable bases for comparing criteria. In practice, most proposed adequacy criteria are monotonic. That is, if a test suite is adequate for the criterion, then any test suite formed by adding test cases to the suite is also adequate. The test suite selection strategy upon which $M$ and $E$ are based does not allow these "extra" test cases, thus $M$ and $E$ compare test suites containing only elements required by the criterion. Finally, notice that, $M$ has been used to compare criteria in previous investigations [2, 9, 15], as has $E$ [2].

In [7], 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. We showed that the fact that $C_1$ subsumes $C_2$ does not guarantee that $M(C_1, P, S) \geq M(C_2, P, S)$. We then introduced in [7], a stronger relation among criteria, the properly covers relation, which we showed does capture the relative 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 narrow and covers relations, also introduced in [7].

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

In [7], we showed that for most criteria of interest, $C_1$ subsumes $C_2$ if and only if $C_1$ universally narrow $C_2$.

Definition: Let $C_1$ and $C_2$ be criteria. $C_1$ covers $C_2$ for $(P, S)$ if for every subdomain $D \in \mathcal{SD}_{C_2}(P, S)$ there is a collection $\{D_1, \ldots, D_n\}$ of subdomains belonging to $\mathcal{SD}_{C_1}(P, S)$ 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 [7] 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 also showed that the properly covers relation overcomes this problem.

Definition: Let $\mathcal{SD}_{C_1}(P, S) = \{D_1, \ldots, D_n\}$, and
let $\mathcal{SD}_{C_2}(P, S) = \{D^1_1, \ldots, D^2_n\}$. $C_1$ 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 $\mathcal{M}$ is a sub-multi-set of $\mathcal{SD}_{C_1}(P, S)$ and

$$D^2_1 = D^1_{1,1} \cup \ldots \cup D^1_{1,k_1}$$

$$\vdots$$

$$D^2_n = D^1_{n,1} \cup \ldots \cup D^1_{n,k_n}.$$ 

Note that the number of occurrences of any subdomain $D^1_{i,j}$ in $\mathcal{M}$ is less than or equal to the number of occurrences of that subdomain in the multi-set $\mathcal{SD}_{C_1}$. 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 $\mathcal{SD}_{C_1}$. $C_1$ 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 proper 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 $D^1_1 = \{x|1 \leq x \leq 6\}$ and $D^2_1 = \{x|4 \leq x \leq 10\}$. Let $C_1$ be the criterion whose subdomains are $D^1_1 = \{x|1 \leq x \leq 4\}$, $D^1_2 = \{x|3 \leq x \leq 6\}$, $D^1_3 = \{x|3 \leq x \leq 8\}$, $D^1_4 = \{x|4 \leq x \leq 10\}$, $D^1_5 = \{x|6 \leq x \leq 10\}$. Then $C_1$ properly covers $C_2$ for $P$ since $D^1_2 = D^1_4 \cup D^1_5$ and $D^2_1 = D^1_1 \cup D^1_3$. In this case $\mathcal{M} = \{D^1_{1,1}, D^1_{1,2}, D^1_{2,1}, D^1_{2,2}\}$ with $D^1_{1,1} = D^1_1$, $D^1_{1,2} = D^1_2$, $D^1_{2,1} = D^1_4$, $D^1_{2,2} = D^1_5$.

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 $C_2$. $\Box$

**Observation 1.** For any $(P, S)$, the narrower, 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 $\mathcal{SD}_{C_2}(P, S) \subseteq \mathcal{SD}_{C_1}(P, S)$ then $C_1$ properly covers $C_2$ for $(P, S)$.

The following theorem is proven in [7]:

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

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 $C_2$.

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 $E$. Example 2 below, shows that this is not necessarily the case if $C_1$ subsumes or covers $C_2$, 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

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

(1)

where $d$ is the size of $D$ and $m$ is the number of failure-causing inputs in $D$. Let $\mathcal{SD}_{C_1}(P, S) = \{D^1_1, \ldots, D^1_p\}$, let $\mathcal{SD}_{C_2}(P, S) = \{D^2_1, \ldots, D^2_n\}$ and let

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

be a sub-multi-set of $\mathcal{SD}_{C_1}(P, S)$ such that

$$D^2_1 = D^1_{1,1} \cup \ldots \cup D^1_{1,k_1}$$

$$\vdots$$

$$D^2_n = D^1_{n,1} \cup \ldots \cup D^1_{n,k_n}.$$ 

For $k = 1, 2$ and for each $i$, let $d^{(k)}_i$ be the size of $D^k_i$ and $m^{(k)}_i$ be the number of failure-causing inputs in $D^k_i$. For each $i, j$, let $d^{(1)}_{i,j}$ be the size of $D^1_{i,j}$ and $m^{(1)}_{i,j}$ be the number of failure-causing inputs in $D^1_{i,j}$. Then

$$E(C_2, P, S) = \sum_{i=1}^{n} \frac{m^{(2)}_i}{d^{(2)}_i}$$

(2)

$$\leq \sum_{i=1}^{n} \sum_{j=1}^{k_i} \frac{m^{(1)}_{i,j}}{d^{(1)}_{i,j}}$$

(3)

$$\leq \sum_{i=1}^{n} \sum_{j=1}^{k_i} \frac{m^{(1)}_{i,j}}{d^{(1)}_{i,j}}$$

(4)

$$= E(C_1, P, S).$$

(5)

Note that (3) follows from (1) and (2), since for each $i$, $D^2_i = D^1_{1,i} \cup \ldots \cup D^1_{n,i,k_n}$. Since $\mathcal{M}$ is a sub-multi-set of $\mathcal{SD}_{C_1}(P, S)$, 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. $\Box$
Test selection criteria that are based solely on the structure of the program being tested are known as program-based, structural, or white-box techniques. For such criteria, the multi-set $SD_C(\mathcal{P}, \mathcal{S})$ 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. Thus, black-box criteria, i.e., those that are independent of the structure of the program, could also be compared using techniques similar to those used in this paper. All that is required is that the strategy divide the domain into subdomains, with one element being selected from each in an appropriate way.

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 $(n_1, n_2, \ldots, n_k)$ of nodes such that for each $i < k$, $(n_i, n_{i+1})$ is an edge. A suffix of a path $(n_1, n_2, \ldots, n_k)$ is a subpath $(n_j, n_{j+1}, \ldots, n_k)$ for some $j \geq 1$. A path is feasible whenever there exists some input that causes it to be executed and infinite 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. We sometimes associate a use of a variable occurring in the Boolean expression controlling a conditional or loop statement with each of the edges leaving that node, and call such a use 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. 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 always evaluate to false, and the other branch will be unexecutable.

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. Although there is no algorithm to check whether or not the NFA property holds, it is possible 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 $v$. 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 Boolean expression controlling the execution of a conditional statement or loop. 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. Note that these subdomains are not necessarily disjoint since the decision is within a loop, a single test case may cause the decision to evaluate to true on one iteration and to false on another iteration.

In the remainder of the paper, we use the properly covers relation to compare various data flow criteria to the decision-coverage version of branch testing and to each other. We thereby exhibit criteria that are guaranteed to be at least as good as decision-coverage according to the measures $M$ and $E$.

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.

\footnote{This variable occurrence may be implicit, as in the use of the input file variable in the statement, while not of do S.}

\footnote{Clarke et al. [1] 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 $p$ will attempt to reference an undefined variable.}

\footnote{In [7] we investigated a different variant of branch testing, called all-edges. The distinction between all-edges and decision coverage is discussed in [6].}
In this section we compare several of these criteria to branch testing and to one another.

The all-uses criterion [12, 13] requires that test data cover every definition-use association in the program, where a definition-use association is a triple \((d, u, v)\) such that \(d\) is a node in the program flow graph in which \(v\) 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 \(u\). We will frequently refer to a definition-use association as an association.

A test case \(t\) covers association \((d, u, v)\) if \(t\) causes a decision-clear path with respect to \(v\) from \(d\) to \(u\) to be executed. Similarly, the all-uses criterion [12, 13] is a restricted version of all-uses that requires test data cover every association \((d, u, v)\) 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 [4].

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

**Proof:**

Since \(SD_{all-uses}(P, S) \subseteq SD_{all-uses}(P, S)\), all-uses universally properly covers all-uses.

The proof that all-uses universally properly covers decision-coverage is similar to the proof in [7] that all-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 \text{true} (alternatively we could let \(D_d\) be the subdomain consisting of all inputs that cause decision \(d\) to evaluate to \text{false}). Let \(e\) be the edge that is executed if and only if \(d\) evaluates to \text{true} (or to \text{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 \(\delta_i\) to \(e\). For each \(i, 1 \leq i \leq n\), let \(D_{\delta_i}\) be the subdomain consisting of all inputs that cause \(\delta_i\) to \text{true} (alternatively we could let \(D_{\delta_i}\) be the subdomain consisting of all inputs that cause \(\delta_i\) to \text{false}). 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 \(\delta_i\), so \(D_d \subseteq D_{\delta_1} \cup \ldots \cup D_{\delta_n}\). Since any test case covering one of the \((\delta_i, e, v)\) must exercise edge \(e\), \(D_{\delta_1} \cup \ldots \cup D_{\delta_n} \subseteq D_d\). Since each outcome of each decision in \(P\) gives rise to a distinct set of associations, all-uses universally properly covers decision-coverage.

We next consider Ntafos’ required k-tuples criteria [11]. 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 \(X_i\) in \(s_i\) can influence the value of \(X_{i-1}\) which is used at \(s_{i+1}\). The required k-tuples criterion requires that each \(k\)-dr interaction be exercised, i.e., that a path \(s_1p_1\ldots s_{k-1}p_{k-1}s_k\) 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. [1] 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 \(l \leq k\),

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.

**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 \(k \geq 2\),

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

We next consider Laiki and Korel's criteria [10], 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 \( \delta_i \) 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 \( \delta_i \) 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. [1] 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 subsets.

The Laiki 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 [12, 13]. In the remainder of the paper, we will use the term context-coverage to refer to the original Laiki-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 \( \delta_i \) and uses of \( X_1, \ldots, X_k \) in node or edge \( u \).

Laiki and Korel also introduced the ordered-context-coverage criterion [10]. 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. Since there may be as many as \( k! \) ordered contexts for each context \( \{(\delta_1, \ldots, \delta_k), u\} \) and as many as \( 2^k \) contexts for each statement that uses \( k \) variables, this has the potential of being an extremely expensive criterion.

**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 \( v_1, \ldots, v_k \) 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 \( v_1, \ldots, v_k \), 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.

One might expect context-coverage and, a fortiori, 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 [8]. 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:
\[
P(x, y: \text{integer});
\text{begin}
\text{if } (x = 1) \text{ then } x := 1;
\text{else } x := 0;
\text{if } (y = 1) \text{ then } y := 1;
\text{else } y := 0;
\text{if } x + y \leq 20 \text{ then write}(x*y)
\text{else } ...
\text{end;}
\]

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),y) and (1,(5,7),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 context arising from this decision give rise to subdomains D_5, D_6, D_7, and D_8.

<table>
<thead>
<tr>
<th>id</th>
<th>subdomain</th>
<th>dua/context</th>
<th>d</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>D_1</td>
<td>{(1,y)}</td>
<td>(3, (8,9), x)</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>D_2</td>
<td>{(x, y)</td>
<td>x \neq 1}</td>
<td>(4, (8,9), x)</td>
<td>90</td>
</tr>
<tr>
<td>D_3</td>
<td>{(x, 1)}</td>
<td>(6, (8,9), x)</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>D_4</td>
<td>{(x, y)</td>
<td>y \neq 1}</td>
<td>(7, (8,9), x)</td>
<td>90</td>
</tr>
<tr>
<td>D_5</td>
<td>{(1,1)}</td>
<td>(3,6, (8,9))</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D_6</td>
<td>{(1,y)</td>
<td>y \neq 1}</td>
<td>(3,7, (8,9))</td>
<td>9</td>
</tr>
<tr>
<td>D_7</td>
<td>{(x,1)</td>
<td>x \neq 1}</td>
<td>(4,6, (8,9))</td>
<td>9</td>
</tr>
<tr>
<td>D_8</td>
<td>{(x, y)</td>
<td>x \neq 1 \text{ and } y \neq 1}</td>
<td>(4,7, (8,9))</td>
<td>81</td>
</tr>
</tbody>
</table>

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

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 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, \( M(\text{context-coverage}, P, S) = M(\text{all-p-uses}, P, S) = 1 - (1 - \frac{45}{41})A = 1 - 0.44A \) and \( E(\text{context-coverage}, P, S) = E(\text{all-p-uses}, P, S) = 1 - (1 - \frac{45}{41})A = 1 - 0.25A \), 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), \( E(\text{context-coverage}, P, S) = E(\text{all-p-uses}, P, S) = B + 45/81 = B + 0.56 \) and \( E(\text{all-p-uses}, P, S) = B + 2(45/90) = B + 1.0 \), 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 $E$. ⊓⊔

Note that this program also illustrates that $2^k$ contexts can arise from a statement using $k$ variables, 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 $k$-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 [13] and the all-simple-OL-paths criterion [14]. Roughly speaking, the all-du-paths criterion requires the execution of particular paths from variable definitions to uses, while the all-simple-OL-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, while in the all-simple-OL-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-OL-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 [4]. Similar problems arise with all-simple-OL-paths. Consequently, by careful placement of faults in executable portions of the code that are not included in any executable du-path or executable simple OL-path, it is possible to construct programs for which these criteria are less likely to expose a fault than branch testing.

4 Conclusion

We have compared the fault-detecting ability of several data flow testing criteria to that of the decision-coverage variant of branch testing and to one another. This comparison was done using two related notions of what it means for criterion $C_1$ to be better at detecting faults than criterion $C_2$. We considered $C_1$ to be better at detecting faults than $C_2$ if for every program $P$ and specification $S$, $M(C_1, P, S) \geq M(C_2, P, S)$. $M(C, P, S)$ was defined to be the probability that a test suite consisting of one test case selected from each of $C$'s subdomains by independent random selection using a uniform distribution includes at least one failure-causing input. This measure of fault-detecting ability has been used in several other investigations. We also considered the measure $E(C, P, S)$, the expected number of failure-causing inputs in a test suite selected for $C$.

We had previously shown that the fact that $C_1$ subsumes $C_2$ does not imply that $C_1$ is better at detecting faults than $C_2$, but that if $C_1$ universally properly covers $C_2$, then $C_1$ is better at detecting faults than $C_2$, when assessed using the measure $M$. This paper applied that result, by investigating whether the universally properly covers relation holds between various pairs of data flow testing criteria and between these criteria and decision-coverage. We also proved and applied an analogous result for the measure $E$.

We showed that the all-uses, all-p-uses, required-k-tuples+, context-coverage, ordered-context-coverage criteria all universally properly cover decision-coverage, and thus are guaranteed to be better, in the above sense, at detecting faults. In addition, the required-k-tuples+ criteria universally properly cover all-uses and all-p-uses. However, surprisingly, 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. We exhibited a simple program $P$ and specification $S$ for which $M(ordered-context-coverage, P, S) = M(\text{context-coverage}, P, S) < M(\text{all-uses}, P, S) \leq M(\text{all-p-uses}, P, S)$ and $E(\text{ordered-context-coverage}, P, S) = E(\text{context-coverage}, P, S) < E(\text{all-uses}, P, S) \leq E(\text{all-p-uses}, P, S)$. This calls into question the usefulness of context-coverage and ordered-context-coverage, especially since those criteria may require an exponential number of test cases.

It is important to remember that our results are based on a very demanding notion of what it means for one criterion to be better at detecting faults than another. Thus, when we say that $C_1$ is better at detecting faults than $C_2$, we are making a very strong statement about all programs and specifications; when we say $C_1$ is not necessarily better than $C_2$, we are only
asserting the existence of a program and specification for which \( M(C_2, P, S) > M(C_1, P, S) \). Our results 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 \( P \) 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. This would imply that \( C_1 \) might be a more appropriate choice of testing criterion for \( P \) than \( C_2 \).

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 a division into subdomains has been made. In practice, these criteria are generally used as the basis for evaluating test suites, not selecting them. This suggests several additional areas for further research. The first is to develop practical test selection algorithms that approximate the scheme used here. The second is to conduct theoretical studies similar to this one based on measures that more closely reflect existing test selection strategies. In that direction, Theorem 1 has recently been generalized to a larger class of probability distributions and the implications for test selection have been explored [3]. Finally, it should be useful to compare other testing techniques using measures similar to those described in this paper. Some results of that nature are reported in [6].

References


