

An Evaluation of the Minimal-MUMCUT Logic Criterion and Prime Path Coverage

Garrett Kaminski, Upsorn Praphamontripong, Paul Ammann, Jeff Offutt
Computer Science Department, George Mason University, Fairfax, VA USA
gkaminsk@gmu.edu, uprapham@gmu.edu, pammann@gmu.edu, offutt@gmu.edu – SERP 2010

Abstract

This paper presents comparisons of the Minimal-MUMCUT logic criterion and prime path coverage. A theoretical comparison of the two criteria is performed in terms of (1) how well tests satisfying one criterion satisfy the other and (2) fault detection. We then compare the criteria experimentally. For 22 programs, we develop tests to satisfy Minimal-MUMCUT and prime path coverage. We use these tests in two separate experiments. First we measure the effectiveness of the tests developed for one criterion in terms of the other. Next we investigate the ability of the test sets to find actual faults. Faults are seeded via a mutation tool and then supplemented with mutants created by DNF logic mutation operators. We then measure the number of non-equivalent mutants killed by each test set. Results indicate that while prime path-adequate test sets are closer to satisfying Minimal-MUMCUT than vice versa, the criteria had similar fault detection and Minimal-MUMCUT required fewer tests.

KEY WORDS: Software Testing, Logic Criteria, MUMCUT, Prime Path Coverage, Graph Coverage

1. Introduction

The Minimal-MUMCUT logic test criterion and the prime path graph coverage test criterion are known to provide a high level of testing. Both subsume other criteria. For example, Minimal-MUMCUT subsumes certain versions of Modified Condition Decision Coverage (MCDC) [8] and prime path coverage subsumes data flow coverage [2]. However, little research has been performed comparing logic testing approaches with graph coverage approaches and no research has specifically compared Minimal-MUMCUT and prime path coverage. Test engineers and test managers need objective, factual studies to make well-informed decisions about testing. This paper tries to establish some idea of the practical cost to benefit tradeoffs between Minimal-MUMCUT and prime path coverage.

Although experience has led us to believe that there is significant overlap between the two criteria, they have not previously been compared on either an analytical or experimental basis. We attempt to compare how well each criterion covers the other and how well each criterion does at fault detection. This comparison has both an analytic part and an experimental part. In the analytical part, we examine the degree to which each criterion covers the other (in the sense of how close tests that satisfy one criterion come to satisfying the other) depending on properties of the program under test. Also in the analytic part, we examine certain

fault types and show how each criterion is guaranteed to detect the fault type, is guaranteed to not detect the fault type, or is not guaranteed either way. In the first experiment, we compare the criteria to see to what degree each one covers the other. In the second experiment, we compare the two criteria by determining how many actual faults are detected by tests that satisfy each criterion.

Our results indicate that prime path requires more tests than Minimal-MUMCUT for most programs, and each criterion provides benefits that the other lacks. Our eventual goal is to find a way to test software that provides the advantages of both criteria, by combining them or by deriving a new criterion that offers the power of both.

The paper is organized as follows. The remainder of Section 1 reviews logic criteria, prime path coverage, and related work. Section 2 describes the hypotheses and comparison methods. Sections 3 and 4 discuss analytical and empirical comparisons, respectively. Section 5 concludes the paper.

The contributions of this paper are:

- 1) An analytical comparison of the degree to which prime path coverage satisfies Minimal-MUMCUT and vice versa
- 2) An analytical comparison of fault detection capabilities for prime path coverage and Minimal-MUMCUT.
- 3) An experimental evaluation of the degree to which prime path coverage satisfies Minimal-MUMCUT and vice versa.
- 4) An experimental evaluation of fault detection capabilities for prime path coverage and Minimal-MUMCUT.
- 5) An experimental evaluation of test set size for prime path coverage and Minimal-MUMCUT.

1.1 Logic Criteria

Table 1.1 lists definitions used in this paper.

Table 1.1 Basic Definitions

Term or Symbol	Definition
1	The Boolean value TRUE
0	The Boolean value FALSE
Literals	Variables representing clauses in a predicate
+	OR operator
Adjacency between literals	AND operator
Term	A set of literals connected by AND
~	Negation
Disjunctive Normal Form (DNF)	Predicate syntax where terms are separated by OR and literals are separated by AND
Implicant	A term that when TRUE, means the predicate is TRUE
Prime implicants	Implicants where removing a literal could potentially change the value of the predicate
Irredundant DNF	Predicate syntax where it is possible to make each term TRUE in turn while all other terms

Term or Symbol	Definition
	are FALSE
Minimal DNF	Predicate syntax in irredundant DNF where all implicants are prime implicants
Unique True Point (UTP)	An assignment of values such that only a single term is TRUE. In $ab + cd$, UTPs for ab are 1100, 1101, 1110.
Near False Point (NFP)	An assignment of values such that the predicate is FALSE but negating a single literal makes the predicate TRUE. In $ab + cd$, NFPs for a are 0100, 0101, 0110.
Corresponding NFP	A NFP that differs from an UTP for the literal's term only in the value of that literal. In $ab + cd$, 0100 is a corresponding NFP for a as it differs from the UTP 1100 for ab only in the value of a .
Feasible	A logic criterion is feasible if and only if it is possible to construct all required tests.

1.1.2 Logic Criteria

A predicate in n variables has 2^n possible tests. When n is large, the exhaustive test set is expensive, and thus logic criteria trade fault detection capability for reduced test set size. Chen, Lau and Yu [3] developed the MUMCUT criterion to guarantee detection of certain DNF logic faults [9]. Kaminski and Ammann developed the Minimal-MUMCUT criterion, which preserves fault detection with a smaller test set size [6]. Minimal-MUMCUT applies criterion feasibility to select UTPs and NFPs in the Boolean space to guarantee the DNF fault detection of MUMCUT but with a smaller test set size [6].

1.2 Prime Path Coverage

This study focuses on unit testing, where the most common graph is a control flow graph (CFG) [2]. A CFG consists of nodes and edges where the nodes are basic blocks (or sometimes program statements) and the edges represent flow of control between the blocks. A path is a sequence of nodes where each pair of adjacent nodes is in the set of edges. Any graph that has a loop has an infinite number of paths, thus researchers have invented graph coverage criteria that cover paths without requiring an infinite number of tests. One such criterion is *prime path coverage*. Prime paths are based on the concept of a simple path. A *simple path* is a path where no node appears more than once, except for possibly the first and last nodes. A *prime path* is a simple path that does not appear as a proper subpath of any other simple path. The prime path coverage criterion requires that all prime paths be covered. Prime path coverage subsumes many other graph coverage criteria, such as edge coverage, which demands that each edge in the graph be toured. Even small graphs can have a large number of prime paths. However, a test path (which begins at an initial node and ends at a final node to represent a complete execution trace through a program) often tours more than one prime path. Infeasibility poses problems for graph coverage if a path cannot be executed.

Both Minimal-MUMCUT and prime path coverage subsume edge coverage (each requires every logic statement to evaluate to TRUE and FALSE). However, Minimal-MUMCUT focuses on logic fault detection while prime path coverage focuses on control flow.

1.3 Related Work

Although there has been much discussion on the relative strengths of graph and logic coverage, we know of no attempt to compare the two. One problem is how to make the comparison. Analytical comparisons show theoretical relationships, and allow claims that are true in all situations. Empirical comparisons show relations that are based on specific studies. Although it is difficult to show that empirical results always hold, analytical comparisons cannot always be made, but empirical comparisons can. While logic and graph coverage have not been directly compared, mutation testing and graph coverage have.

Mutation testing was originally proposed by DeMillo et al. [4] and Hamlet [5] requiring testers to create tests to detect a specified set of faults. Mutant programs vary from the original program by one or more syntactic changes. If possible, testers find inputs to distinguish the mutants from the original program in that the output of the mutant and original program are different. Such a mutant is then considered to be *strongly killed*. For a mutant to be strongly killed, the mutated statement must be reached (*reachability*), execution of the mutated statement must cause mutant and the original program state to differ (*infection*), and the difference in program state must propagate to an output (*propagation*). Equivalent mutants are functionally (semantically) equivalent to the original program and thus cannot be killed. In other words, for every possible input, the output of the original program and the equivalent mutant are the same. Determination of equivalent mutants is undecidable [12]. A mutant is *strongly equivalent* if propagation does not occur. A mutant is *weakly equivalent* if infection does not occur. One use of mutation is to seed faults into programs, and this study used muJava [10] to seed faults to compare fault detection between Minimal-MUMCUT tests and prime path tests.

From an analytical view, Offutt and Voas [14] compared mutation testing with a form of graph coverage known as all-defs data flow. *All-defs* requires that each definition of a variable reaches at least one use. They showed that mutation subsumes all-defs. From an empirical view, we consider two relations that have been defined elsewhere. Weyuker, Weiss, and Hamlet [16] suggest a relation called PROB_BETTER. A criterion A is PROB_BETTER than criterion B for a program P if a randomly selected test set T that satisfies A is more likely to detect a fault than a randomly selected test set that satisfies B. Mathur and Wong [11] suggest a different relation called PROB_SUBSUMES. A criterion A PROB_SUBSUMES criterion B for a program P if a test set T that satisfies A is

likely to satisfy B. If A PROB_SUBSUMES B then A is said to be more difficult to satisfy than B.

The PROB_BETTER relation is defined in terms of fault detection capability of tests. The PROB_SUBSUMES relation is defined with respect to the difficulty of satisfying one criterion in terms of another. Both are probabilistic relations between criteria defined in terms of specific programs. It is difficult to draw general conclusions from one study, but as the number and variety of programs studied increases, confidence in these relations can increase.

Mathur and Wong [11] use the PROB_SUBSUMES relation in experimental comparisons of all-uses data flow testing with mutation testing by manually generating test data to satisfy both criteria. They used 4 programs and 30 sets of test cases per program and detected equivalent mutants and unexecutable paths by hand. This study indicated that mutation-adequate test sets were closer to being data flow-adequate than data flow-adequate tests were to being mutation-adequate.

Tewary and Harrold [15] devised algorithms for inserting faults into programs using the program dependence graph. The algorithms were demonstrated by inserting faults and comparing fault detection ability of mutation and data flow testing. They found that when faults involved changes to the control dependence relations in the program dependence graph, the mutation-adequate and data flow-adequate test sets were almost equally effective at fault detection.

Offutt et al. [13] performed a study similar to that of Mathur and Wong's study, but with a few differences. One difference is that Offutt et al automated the data generation process. Another difference was that Offutt et al. used 10 programs (rather than 4) and 5 sets of tests per program (rather than 30). Thus, Offutt et al. sacrificed number of test cases for number of programs. Also, Offutt et al. included fault detection in their study whereas Mathur and Wong did not. Offutt et al. found that mutation-adequate test sets were closer to being data flow-adequate than vice versa and mutation-adequate tests had better fault detection.

2 Hypotheses and Conduct

This paper presents the following comparisons:

- 1) Analytical comparison of PROB_SUBSUMES
- 2) Analytical comparison of PROB_BETTER
- 3) Experimental comparison of PROB_SUBSUMES
- 4) Experimental comparison of PROB_BETTER

If test sets created for one criterion also satisfy another, then the first criterion can be considered to be "superior" to the second. This is the essence of the PROB_SUBSUMES relationship. Thus, we try to determine if prime path-adequate test sets usually cover Minimal-MUMCUT and

vice versa. An independent and more practical question is whether tests find faults. This is the essence of the PROB_BETTER relationship.

For our comparison we formulated these hypotheses:

Hypothesis 1: Prime path coverage PROB_SUBSUMES Minimal-MUMCUT.

Hypothesis 2: Minimal-MUMCUT PROB_SUBSUMES prime path coverage.

Hypothesis 3: Prime path coverage is PROB_BETTER than Minimal-MUMCUT.

Hypothesis 4: Minimal-MUMCUT is PROB_BETTER than prime path coverage.

We selected 22 static utility methods in the Arrays and Collections Java classes (J2SE 1.7) and converted each into a separate program. To make manual test data generation and manual identification of equivalent mutants feasible, we limited our sample to methods containing 20 or less lines of code. The source files for the Collections and Arrays classes are accessible at:

<http://www.docjar.com/html/api/java/util/Collections.java.html>

<http://www.docjar.com/html/api/java/util/Arrays.java.html>

A total of 287 executable statements exist for the 22 programs, (low of 6, average of 13, and high of 20). A total of 274 prime paths exist (low of 2, average of 12, and high of 27). Of the 274 prime paths, 270 were feasible. The 270 feasible prime paths can be toured with a minimal test set of 104 tests (low of 2, average of 5, and high of 8). (The term *minimal* means that if even a single test is removed then prime path coverage is not achieved). A total of 137 Minimal-MUMCUT constraints exist (low of 4, average of 6, high of 11). (A *constraint* means that literals in a predicate must attain certain values.) Of the 137 Minimal-MUMCUT constraints, 136 were feasible. The 136 feasible Minimal-MUMCUT constraints can be satisfied by a minimal test set of 77 tests (low of 2, average of 3.5, and high of 6). The number of feasible prime paths is nearly twice the number of feasible Minimal-MUMCUT constraints and the number of tests needed to satisfy prime path coverage is 1.35 times the number of tests needed to satisfy Minimal-MUMCUT.

Below are data for the 65 predicates in the programs. All 65 predicates were in minimal DNF. A predicate is considered to be an "if" statement or loop condition ("for", "while", or "do while"). If a loop's condition is TRUE, we call it a TRUE loop. Otherwise we call it a non-TRUE loop.

2 predicates have a loop condition of TRUE

54 predicates have 1 literal

8 predicates have 2 literals

1 predicate has 3 literals

14 programs have at least one non-TRUE loop

8 programs do not have a loop or have only TRUE loops

MuJava (<http://cs.gmu.edu/~offutt/mujava/>) was used to seed faults for the empirical fault detection study. MuJava is a mutation tool that automates the process of generating and running mutants. Equivalent mutants were identified by hand and subsequently removed. muJava does not seed minimal DNF faults (as described in Table 3.1, so we added mutants that had faults corresponding to those in Table 3.1 without duplicating any strongly non-equivalent muJava mutants. We did duplicate strongly equivalent muJava mutants that corresponded to fault types in Table 3.1 since these mutants had been removed. For these mutants only we applied weak mutation testing (we considered a mutant killed if infection occurred). This reduced bias against Minimal-MUMCUT since a large amount of the strongly equivalent muJava mutants involved mutations to “*if*” and “*else if*” predicates. 910 mutants were generated, including all muJava-generated mutants and all mutants based on the faults in Table 3.1 not overlapping with muJava mutants. 126 mutants involved mutations to “*if*” or “*else if*” predicates, 60 of which were strongly equivalent (48%). Of the other 784 mutants, only 95 were strongly equivalent (8%). This difference is due to predicates used to speed up searching/sorting with no affect on the output. In general, we expect mutants to “*if*” or “*else if*” predicates to affect the output. Thus, we applied weak mutation testing to mutants that corresponded to a fault in Table 3.1.

We used automated tools for each program to determine a set of test paths that would tour all prime paths and to determine the values required by literals in predicates to satisfy Minimal-MUMCUT. We then used the outputs of these tools to manually generate tests for each program. To reduce bias, one author created test inputs that toured all the test paths needed to achieve prime path coverage while a different author independently created test inputs that satisfied Minimal-MUMCUT. Also, the author responsible for generating prime path tests was unfamiliar with Minimal-MUMCUT. The automated tools can be found online:

<http://cs.gmu.edu:8080/offutt/coverage/GraphCoverage>)

[http://cs.gmu.edu:8080/offutt/coverage/MinimalMUMCUT Coverage](http://cs.gmu.edu:8080/offutt/coverage/MinimalMUMCUTCoverage))

3 Analytical Evaluation

Logic criteria have mathematical properties that allow strong statements to be argued analytically. This section presents these arguments, for Minimal-MUMCUT, for coverage and fault detection.

3.1 Analytical Coverage

Coverage is the degree to which a test set that satisfies criterion A for a program also satisfies criterion B [2]. Coverage of criterion A by criterion B is 100% if and only if tests that are adequate for A are also adequate for B. Let $F(X, T, P)$ be the function that measures the degree to

which a test set T for a program P is adequate for criterion X. If T is adequate for X on P then $F(X, T, P) = 100\%$. The coverage measure for prime path testing is the number of prime paths toured divided by the number of feasible prime paths. The coverage measure for Minimal-MUMCUT is the number of constraints satisfied divided by the number of feasible constraints. If T is a Minimal-MUMCUT test set and PP is the prime path criterion, then $F(PP, T, P)$ gives the degree to which a Minimal-MUMCUT test set satisfies prime path coverage. If T is a prime path test set and MM is the Minimal-MUMCUT criterion, then $F(MM, T, P)$ gives us the degree to which a prime path test set satisfies Minimal-MUMCUT.

Let T be a prime path test set and MM be the Minimal-MUMCUT criterion. For any program that contains only single-literal predicates, $F(MM, T, P) = 100\%$. Thus, for a program containing only single-literal predicates, prime path coverage subsumes Minimal-MUMCUT. This was the case for 13 of the 22 programs. Minimal-MUMCUT demands that the literal evaluate to TRUE and FALSE, which any prime path test set guarantees since a prime path test set guarantees that each predicate evaluates to TRUE and FALSE. For any minimal DNF predicate with two literals, there are three MM constraints. Consider predicate $a + b$. MM demands that $a=0, b=0$, and that $a=0, b=1$, and that $a=1, b=0$. A minimal prime path test set is guaranteed to satisfy $a=0$ and $b=0$ but may or may not satisfy either of the other two constraints (the predicate can be made TRUE with $a=1, b=1$). If the predicate only needs to be reached twice to satisfy prime path coverage, then minimal prime path tests will at most satisfy two of three constraints. A similar analysis for the 3-literal predicate $a + bc$ yields that MM demands five constraints and that a minimal prime path test set is guaranteed to satisfy at least one but no more than two constraints (again assuming that the predicate needs to be reached twice to satisfy prime path coverage).

The 22 programs had 54 single literal predicates (each with two MM constraints), eight 2-literal predicates (each having three MM constraints) and one 3-literal predicate (having six MM constraints). For one of the 2-literal predicates, one of the MM constraints was infeasible. Thus, 137 feasible MM constraints exist. A prime path test set is guaranteed to satisfy both MM constraints for a single literal predicate, at least one MM constraint for a 2-literal predicate, and at least one MM constraint for a 3-literal predicate. Thus, a prime path test set will satisfy at least $54 \times 2 + 8 \times 1 + 1 \times 1 = 117$ of 137 (85%) feasible MM constraints. If T is a prime path test set for all 22 programs and P represents all 22 programs, $F(MM, T, P) \geq 85\%$.

Let T be a Minimal-MUMCUT test set and PP be the prime path criterion. For any program where every test set that satisfies edge coverage also satisfies prime path coverage, $F(PP, T, P) = 100\%$. This is because Minimal-MUMCUT subsumes edge coverage as Minimal-MUMCUT demands (amongst other requirements) that each predicate evaluate

to TRUE and FALSE. Thus, for any such program, Minimal-MUMCUT subsumes prime path coverage. This was the case for nine of the 22 programs. For any program containing a non-TRUE loop, a test set exists where edge coverage is satisfied but not prime path coverage. Prime path coverage demands that every non-TRUE loop execute zero times in one test and at least one time in another test. A test set can satisfy Minimal-MUMCUT (and edge coverage) yet have no test that executes the loop zero times. If a loop contains a single literal in its predicate, then when considering just this predicate, Minimal-MUMCUT (and edge coverage) demands only that the literal be TRUE and the literal be FALSE. Thus, any test that causes the loop condition to be TRUE in the first iteration and FALSE after the first iteration will satisfy Minimal-MUMCUT (and edge coverage). However, Minimal-MUMCUT does not require loops to be executed zero times.

The nine programs where every test set that satisfies edge coverage also satisfies prime path coverage have a total of 30 feasible prime paths, so Minimal-MUMCUT tests tour these. For the other programs, Minimal-MUMCUT tests are guaranteed to execute each *for* and *while* loop at least once and each *do while* loop at least twice. This translates to guaranteeing touring another 54 prime paths. Thus, a Minimal-MUMCUT test set is guaranteed to tour 84 of 270 feasible prime paths (31%). Thus, 186 of 270 feasible prime paths (69%) may or may not be toured. If T is a Minimal-MUMCUT test set for all 22 programs and P represents all 22 programs, $F(PP, T, P) \geq 31\%$, the lower bound in the empirical study. These results show that while PROB_SUBSUMES is not strong in either direction, it is stronger for prime path coverage subsuming Minimal-MUMCUT than vice versa.

3.2 Analytical Fault Detection

Any fault that can only be detected by executing a *for* or *while* loop zero times (or a *do while* loop one time) can go undetected by Minimal-MUMCUT tests. If a fault exists such that any test that executes a *for* or *while* loop zero times (or executes a *do-while* loop one time) can detect the fault, but no other test can detect the fault, then prime path tests are guaranteed to detect the fault but Minimal-MUMCUT tests are not. On the other hand, Minimal-MUMCUT test are guaranteed to detect other faults that prime path tests are not. One way to evaluate tests is to determine how many of nine minimal DNF faults in Table 3.1 a test set is guaranteed to detect [8].

Table 3.1 Typical Minimal DNF Logic Faults

Fault	Description
Expression Negation Fault (ENF)	Predicate implemented as its negation: $ab + c$ implemented as $\sim(ab + c)$.
Term Negation Fault (TNF)	A term is negated: $ab + c$ implemented as $\sim(ab) + c$.
Operator Reference Fault + (ORF+)	Replacing OR with AND: $a + b$ implemented as ab .
Operator Reference Fault . (ORF.)	Replacing AND with OR: ab

Fault	Description
	implemented as $a + b$.
Literal Negation Fault (LNF)	A literal is negated: ab implemented as $a\sim b$.
Literal Reference Fault (LRF)	A literal is replaced by a literal or the negation of a literal not in the term: $ab + cd$ implemented as $cb + cd$ or as $\sim cb + cd$.
Term Omission Fault (TOF)	A term is omitted: $ab + cd$ implemented as ab .
Literal Omission Fault (LOF)	A literal is omitted: ab implemented as a .
Literal Insertion Fault (LIF)	A literal not in a term is inserted as itself or as its negation: $ab + cd$ implemented as $abc + cd$ or as $ab\sim c + cd$.

The faults in Table 3.1 should be tested for based on the competent programmer hypothesis should be analyzed [1]. (The *competent programmer hypothesis* states that competent programmers write programs that differ from a correct version by a few simple faults.) Minimal-MUMCUT tests are guaranteed to detect all faults in Table 3.1 [6]. When a predicate contains a single literal, all faults in Table 3.1 reduce to the ENF, which prime path tests are guaranteed to detect. In general, a prime path test set is only guaranteed to detect two fault types in Table 3.1 (ENF and TNF). When considering a predicate in isolation prime path coverage demands only that the predicate evaluate to TRUE and FALSE. Thus, neither an NFP nor UTP needs to be in a prime path test set for any predicate with at least two unique literals. For example, consider the predicate $a + b$. Six minimal DNF faults exist as described in Table 3.2.

Table 3.2 Minimal DNF Logic Faults for $a + b$

Fault	Description
Expression Negation Fault (ENF)	$\!(a + b)$
Term Negation Fault (TNF)	$\!a + b$
Term Negation Fault (TNF)	$a + \!b$
Operator Reference Fault + (ORF+)	ab
Term Omission Fault (TOF)	b
Term Omission Fault (TOF)	a

A prime path test set might include tests that make $a=1, b=1$ and $a=0, b=0$. These points miss detecting the ORF and both TOFs as neither test is a UTP. Prime path tests are guaranteed to detect 3 of 6 minimal DNF faults for $a + b$. Prime path tests are also guaranteed to detect three of six minimal DNF faults for the predicate ab . For a predicate with three unique literals, prime path tests can miss more faults. A single predicate with three unique literals existed in the programs: $a + bc$. For this predicate, 24 non-equivalent minimal DNF faults exist. Prime path tests will detect at least five of them because such tests do not require a UTP and the only NFP that is required is an NFP for literal a .

The 22 programs have a total of 126 minimal DNF faults. For the 54 predicates with one unique literal, prime path tests will detect all 54 ENFs. For the eight predicates with

two unique literals, prime path tests will detect at least 24 of 48 minimal DNF faults. For the one predicate with three unique literals, prime path tests will detect at least five of 24 minimal DNF faults. Thus, minimal prime paths tests will detect at least 83 of 126 (66%) minimal DNF faults.

If all predicates in a program contain a single literal and it is possible to satisfy edge coverage without satisfying prime path coverage (such as when at least one non-TRUE loop exists), then prime path coverage is `PROB_BETTER` than Minimal-MUMCUT. This was true for nine programs. If every test set that satisfies edge coverage also satisfies prime path coverage and at least one multi-literal predicate exists, then Minimal-MUMCUT is `PROB_BETTER` than prime path coverage. This was true for four programs. If all predicates contain a single literal and every test set that satisfies edge coverage also satisfies prime path coverage, then neither criterion is `PROB_BETTER` than the other. This was true for four programs. If at least one multi-literal predicate exists and it is possible to satisfy edge coverage without satisfying prime path coverage (such as when at least one non-TRUE loop exists), then we need empirical evaluation. This was true for five programs. As the ratio of the number of multi-literal predicates to the number of non-TRUE loops increases, we expect Minimal-MUMCUT to be `PROB_BETTER` than prime path coverage. As this ratio decreases, we expect prime path coverage to be `PROB_BETTER` than Minimal-MUMCUT. Our results show that while `PROB_BETTER` is not strong in either direction, it is stronger for prime path coverage being better than Minimal-MUMCUT than the inverse.

4 Empirical Evaluation

A total of 77 tests were designed to satisfy Minimal-MUMCUT for all 22 programs, with a low of two, a high of six, and an average of 3.50 per program (one test per 3.73 LOC). A minimal test set of 104 tests satisfies prime path coverage for all 22 programs, with a low of two, a high of eight, and an average of 4.73 per program (one test per 2.76 LOC). The rest of this section specifies (1) the degree to which each test set covers the other and (2) the fault detection capability of each.

The earlier analysis showed that for the programs studied, at least 117 (85%) of the 137 feasible Minimal-MUMCUT constraints would be satisfied by prime path tests. The actual number satisfied by prime path tests was 126 (92%), with a low of 1/3 (33%) and a high of 6/6 (100%). Of the 20 Minimal-MUMCUT constraints that might or might not have been satisfied, nine (45%) were actually satisfied. For all 22 programs, 11 additional tests need to be added to the prime path tests to satisfy Minimal-MUMCUT, with a low of 0, a high of 2, and an average of 0.5 per program. For 13 of the 22 programs, the prime path tests satisfied Minimal-MUMCUT.

The analysis also showed that for the programs studied, at least 84 of 270 (31%) feasible prime paths would be toured

by Minimal-MUMCUT tests. The actual number of prime path tests toured by Minimal-MUMCUT tests was 196 (73%) with a low of 5/13 (38%) and a high of 4/4 (100%). Of the 186 prime paths that may or may not be toured, 112 (60%) were actually toured by Minimal-MUMCUT tests. For all 22 programs, 39 additional tests need to be added to the Minimal-MUMCUT tests to achieve prime path coverage, with a low of 0, a high of 4, and an average of 1.77 per program. For 9 of the 22 programs, Minimal-MUMCUT tests achieved prime path coverage. These results show that while the `PROB_SUBSUMES` relation is not strong either way, it is stronger for prime path tests subsuming Minimal-MUMCUT than vice versa.

814 non-equivalent faults were seeded via mutation through muJava and additional minimal DNF mutation operators corresponding to faults in Table 3.1. Minimal-MUMCUT tests detected 762 of these faults (94%) with a low of 85% and high of 100%. Minimal-MUMCUT tests detected all faults in 12 programs. Prime path tests detected 773 faults (95%), with a low of 33% and a high of 100%. Prime path tests detected all faults in five programs. Minimal-MUMCUT tests detected more faults than prime path tests for nine programs, while the opposite was true for six programs. From a percentage standpoint, the `PROB_BETTER` relation is not strong in either direction, but is slightly stronger for prime path tests being `PROB_BETTER` than Minimal-MUMCUT tests than vice versa. This agrees with the analytical conclusion from Section 3.

A test set formed by the union of the Minimal-MUMCUT and prime path test sets detected 796 (98%) of the faults with a low of 91%. For 19 of the 22 programs, a union test set detected all or all but one fault.

Of the 814 seeded faults, 126 were minimal DNF faults. Minimal-MUMCUT is guaranteed to detect these faults, but prime path coverage is not. The analytical analysis showed that for the programs studied, prime path tests would detect at least 83 of these faults (66%) and prime path tests actually detected 109 of them (87%). Of the 43 DNF faults that may or may not be detected, 26 of them (60%) were actually detected by prime path tests.

5 Conclusion

This paper presents results from two analytical and two empirical studies that compared prime path (PP) coverage and Minimal-MUMCUT (MM). Analytical and empirical studies were also performed to assess fault detection for each. Finally, test set size for each was compared.

For 59% of the programs, PP coverage will always subsume MM. For 41% of the programs, MM always subsumes PP coverage. For 18% of the programs, PP coverage and MM always subsume each other. For 23% of the programs, neither criterion always subsumes the other. Thus, `PROB_SUBSUMES` is not strong either way, but it is

stronger for PP coverage subsuming MM than vice versa. PP tests were guaranteed to satisfy a higher percentage of MM constraints than vice versa (85% to 31%).

We showed that any fault that can only be detected by executing a loop zero times is guaranteed to be detected by a PP test set but not a MM test set. However, PP tests can fail to detect seven of nine DNF logic fault types that a MM test set is guaranteed to detect. For 41% of the programs, PP coverage was PROB_BETTER than MM. For 18% of the programs, MM was PROB_BETTER than PP coverage. For 18% of the programs, neither criterion was PROB_BETTER than the other. For 23% of the programs, empirical analysis is needed to determine the PROB_BETTER relation. The empirical study on subsumption showed that MM tests toured 73% of the feasible prime paths and of the feasible prime paths that may or may not be toured by a MM test set, 60% were actually toured. PP tests satisfied 92% of feasible MM constraints and of the feasible MM constraints that may or may not be satisfied by PP tests, 45% were actually satisfied. The empirical study on fault detection showed that MM tests detected 94% of seeded faults whereas PP tests detected 95% of seeded faults. Thus, neither was found to be PROB_BETTER than the other.

The experiments have limitations as we cannot claim that the programs are representative samples from a population. Thus, claims of significance cannot be made. Also, the concern remains of how results scale to larger programs. However, since MM and PP coverage are useful in unit testing, this concern is somewhat negated. Our results indicate that in general, PP coverage demands more tests than MM and is more likely to subsume (or come close to subsuming) it than vice versa, but fault detection for each is similar. For software with many loops and few multi-literal predicates, fault detection can be expected to be better for PP tests while for software with few loops and many multi-literal predicates, fault detection may be better for MM tests.

6 References

- [1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Saywood. "Mutation Analysis," Technical Report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, September 1979.
- [2] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press. 2008, ISBN 0-52188-038-1.
- [3] T. Y. Chen, M. F. Lau, and Y. T. Yu. MUMCUT: A Fault-Based Criterion for Testing Predicates. *Proceedings of the Sixth Asia Pacific Software Engineering Conference*. Pages 606-613. Takamatsu, Japan. December 1999.
- [4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34-41, April 1978.
- [5] R. G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*, 8(4):371-379, July 1982.
- [6] G. Kaminski and P. Ammann. Using Logic Criterion Feasibility to Reduce Test Set Size While Guaranteeing Fault Detection. *Proceedings of the Second International Conference on Software Testing*. Denver, CO. April 2009.
- [7] G. Kaminski and P. Ammann. Reducing Logic Test Set Size While Preserving Fault Detection. In progress.
- [8] G. Kaminski, G. Williams, and P. Ammann. Reconciling Perspectives of Logic Testing for Software. *Software Testing, Verification, and Reliability*, 18(3):149-188, September 2008.
- [9] M. F. Lau and Y. T. Yu. An Extended Fault Class Hierarchy for Predicate-Based Testing. *ACM Transactions on Software Engineering and Methodology*, 14(3): 247-276, July 2005.
- [10] Yu-Seung Ma, Jeff Offutt and Yong-Rae Kwon. MuJava : An Automated Class Mutation System. *Software Testing, Verification and Reliability*, 15(2):97-133, June 2005.
- [11] A. P. Mathur and W. E. Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification, and Reliability*, 4(1): 9-31, March 1994.
- [12] J. Offutt and J. Pan. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Software Testing, Verification, and Reliability*, 7(3):165-192, September 1997.
- [13] J. Offutt, J. Pan, T. Zhang, and K. Tewary. An Experimental Evaluation of Data Flow and Mutation Testing. *Software Practice and Experience*, 26(2): 165-176, February 1996.
- [14] J. Offutt and J. M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical Report ISSE-TR-96-01. January 1996.
- [15] K. Tewary and M. J. Harrold. Fault modeling using the program dependence graph. *Proceedings of the Fifth International Symposium on Software Reliability Engineering*. Pages 1-10. Monterey, CA. November 1994
- [16] E. J. Weyuker, S. N. Weiss and R. G. Hamlet. Comparison of program testing strategies. *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*. Pages 1-10. Victoria, British Columbia, Canada. October 1991.