Investigations of the Software Testing Coupling Effect

A. JEFFERSON OFFUTT
Clemson University

Fault-based testing strategies test software by focusing on specific, common types of faults. The *coupling effect* hypothesizes that test data sets that detect simple types of faults are sensitive enough to detect more complex types of faults. This paper describes empirical investigations into the coupling effect over a specific class of software faults. All of the results from this investigation support the validity of the coupling effect. The major conclusion from this investigation is the fact that by explicitly testing for simple faults, we are also implicitly testing for more complicated faults, giving us confidence that fault-based testing is an effective way to test software.

1 INTRODUCTION

Fault-based testing is a general strategy for testing software that has been widely studied in recent years [5, 6, 12, 17, 20, 21, 22]. Fault-based testing strategies are based on the notion of testing for specific kinds of faults and succeed because programmers tend to make certain types of faults that can be well defined. Since the number of possible faults for a given program can be large, fault-based testing strategies assume that by testing for certain restricted classes of faults, we can find a wide class of faults. The set of faults is commonly restricted by two principles, the *Competent Programmer Hypothesis* [1] and the *Coupling Effect* [6].

The Competent Programmer Hypothesis states that competent programmers tend to write programs that are "close" to being correct. In other words, a program written by a competent programmer may be incorrect, but it will differ from a correct version by relatively simple faults. The Coupling Effect states that a test data set that detects all simple faults in a program is so

Author's address: A. J. Offutt, Department of Computer Science, Clemson University, Clemson, SC 29634-1906.

sensitive that it will also detect more complex faults [6]. Although the term complex fault has never been formally defined, we introduce the following working definitions for this paper:

Definition. A *simple fault* is a fault that can be fixed by making a single change to a source statement. A *complex fault* is a fault that cannot be fixed by making a single change to a source statement.

Thus, mutations introduce simple faults into a program. If *higher-order mutants*, or *complex mutants*, are mutants that are generated by introducing multiple mutations into the program, then it is clear that many higher-order mutants are complex faults. We also expect that there are complex faults that cannot be generated by higher-order mutants (e.g., missing paths), thus higher-order mutants form a proper subset of the complex faults. With this definition of complex faults, we can define the coupling effect as follows:

Hypothesis. *Coupling Effect*: Complex faults are *coupled* to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults.

Since examples of complex faults that are not coupled to simple faults can be constructed, the coupling effect is probabilistic rather than absolute. On the other hand, we cannot show correctness through testing [10], software testing is an imperfect science and we see no reason for the coupling effect to be perfect. If it is usually true, it can help us to increase the reliability of our software. In the above form, the coupling effect is a very general statement that covers all faults, all programs, and all test data generation methods. For the empirical analysis of this paper, we restrict the coupling effect:

Hypothesis. *Mutation Coupling Effect*: Complex mutants are *coupled* to simple mutants in such a way that a test data set that detects all simple mutants in a program will detect a large percentage of the complex mutants.

This paper presents empirical analyses of the mutation coupling effect. The experiments used a modified version of the Mothra mutation system [5] to generate higher-order mutants for several programs. This study gives strong evidence that the mutation coupling effect is true, and although more evidence is certainly needed, we argue that this indicates that the more general form of the coupling effect is valid.

1.1 Mutation Testing

Mutation analysis is a fault-based testing technique that helps the tester create a set of test cases to detect specific, predetermined types of faults [6, 5]. Mutation analysis systems work by inducing a large number of simple faults, called *mutations*, into a test program to create a set of *mutant* programs. These mutants are created from the original program by applying *mutation operators*, which describe syntactic changes on the programming language. Test cases are then measured by determining how many of the mutant programs produce incorrect output when executed. Each live mutant is executed with the test cases and when a mutant produces incorrect output on a test case, that mutant is said to be "killed" by that test case and is not executed against subsequent test cases. This shows that the current test case set is able to find the faults represented by the dead mutants. Two programs are functionally equivalent if they always produce the same output on every input. Some mutants are functionally equivalent to the original program and cannot be killed. Despite recent work in automating detection of equivalent mutants [4], this is usually done manually, and is one of the greatest expenses of current mutation systems.

A mutation score of a test set is the percentage of non-equivalent mutants that are killed by a test case. More formally, if a program has M mutants, E of which are equivalent, and a test set T kills K mutants, the mutation score is defined to be:

$$MS(P,T) \equiv \frac{K}{(M-E)}.$$

A test set is mutation-adequate if its score is 100% (all non-equivalent mutants were killed). In practice, data sets that score above 95 percent on a mutation system tend to be difficult to create, but are effective at finding faults. This has been demonstrated both experimentally [8] and analytically [3].

The function `Sum` in figure 1 computes the sum of the integers from 1 to N. The original version of `Sum` is shown on the left, and `Sum` with six of the mutants that are generated by the Mothra mutation system [5] is shown on the right, with the lines preceded by a "Δ" representing mutations of the original function. $\Delta 1$ is a `constant replacement (crp)` mutation replacing the constant 0 with 1, $\Delta 2$ is a `scalar for constant replacement (scr)` mutation replacing the constant 0 with `N`, $\Delta 3$ is a `scalar variable replacement (svr)` mutation replacing the variable `Rslt` with `N`, $\Delta 4$ is a `statement deletion (sdl)` mutation replacing the entire statement with a `CONTINUE`, $\Delta 5$ is an `arithmetic operator replacement (aor)` mutation replacing the arithmetic operator $+$ with $*$, and $\Delta 6$ is another `svr` mutation replacing `I` with `N`. Note that each of the six mutants in figure 1 represents a separate program.

```
          Function Sum (N)
     C    Sum the integers from 1 to N.
          INTEGER N, I, Rslt
1         Rslt = 0
2         DO 10 I = 1, N
             Rslt = Rslt + I
4    10   CONTINUE
5         Sum = Rslt
6         RETURN
7         END
```

Original Program

```
          Function Sum (N)
     C    Sum the integers from 1 to N.
          INTEGER N, I, Rslt
1         Rslt = 0
     Δ1   Rslt = 1
     Δ2   Rslt = N
     Δ3   N = 0
     Δ4   CONTINUE
2         DO 10 I = 1, N
3            Rslt = Rslt + I
     Δ5      Rslt = Rslt * I
     Δ6     Rslt = Rslt + N
4    10   CONTINUE
5         Sum = Rslt
6         RETURN
7         END
```

Mutated statements are shown with Δ

Figure 1: Function SUM

After all mutants have been executed, the tester is left with two pieces of information. The number of dead mutants indicates how well the program has been tested. The live mutants indicate inadequacies in the current test set (and potential faults in the program). The tester then must add additional test cases to kill the remaining live mutants. This process of adding test cases, verifying the output of the test cases, and executing mutants continues until the tester is satisfied with the mutation score.

The expected output verification is an important step in this process that is often lost in descriptions of mutation testing. After executing a test case on the original program, the tester must decide whether the output is correct on that test case. This action is commonly referred to as the *oracle* function, and is common to all dynamic testing strategies, but unfortunately has not been successfully automated. In mutation, this is the point at which the tester finds bugs. The tester finds bugs because if the original program contains a fault, there will usually be a set of mutants that remain alive. This has been referred to as the fundamental premise of mutation analysis: **if the software contains a fault, it is likely that there is a mutant that can only be killed by a test case that also detects that fault**.

1.2 The Coupling Effect

The intuitive rationale of the coupling effect is that subtle faults are in some sense "harder" to detect than simple faults. As Morell [17] pointed out, the distinction between "simple" and "complex" faults is not always clear. In this paper, we use mutation to model complex faults by inducing multiple mutations into the program simultaneously. 2-order mutations are created by combining 2 mutations, 3-order mutations are created by combining 3 mutations, and in general, n-order mutations are mutations that are created by combining n mutations.

Mothra's mutation operators were derived so that they correspond to typical programming errors [2]. This particular set of 22 mutation operators [13] represents more than 10 years of refinement through several mutation systems. They explicitly require the test data to meet criteria such as statement and branch coverage, extremal values criteria, domain perturbation, and they also directly model many types of faults. Whereas other studies used only a few of the mutation operators (e.g., [8, 11, 15]), it is important that all available mutation operators be used. Not using all operators severely biases a study against mutation.

Although this study focuses on the validity of the mutation coupling effect, a more general underlying question is whether the general coupling effect is valid. The mutation coupling effect being valid implies the general coupling effect in one of two cases. For case one, if the number of complex mutants is large in relation to the number of complex faults, then by detecting complex mutants we at least detect most complex faults. Thus, in this case, demonstrating the mutation coupling effect will be enough to validate the general form of the coupling effect. For case two, it must be determined whether complex faults are easier to detect than complex mutants. This has always been the contention of mutation researchers [6, 14, 2], but only on an intuitive basis. The first case seems unlikely, and there is some negative evidence (e.g., Marick's study [15]). The second case seems more likely, but although anecdotal evidence abounds, it has never been carefully studied. If true, it seems likely that there is a definable class of complex faults that do not couple (perhaps missing paths or errors of omission); these faults would have to be targeted by a non fault-based testing technique.

1.3 Previous Work

Even though the coupling effect has been mentioned by numerous researchers [1, 2, 6, 16, 17], there has been little effort to verify or disprove the effect. Morell [16] presented several theoretical results on coupling, using a definition of coupling that differs slightly from earlier definitions. In Morell's terms, two mutations are *coupled* for a test set if the test set kills each of the mutants but does not kill the mutant composed from their combination. Note that though this definition is essentially the inverse of the definition used here, Morell's results hold for both. Morell's results include the fact that there is no algorithm for determining if two mutations couple [16]. He also gives a probabilistic argument to claim that for any given pair of non-equivalent 1-order mutations, there are relatively few test cases for which the two mutations couple [16].

An experiment presented by Lipton and Sayward [14] gives some evidence for the validity of the mutation coupling effect. In Lipton and Sayward's experiment, an adequate set of 49 test cases was derived for the program FIND [9]. This set was heuristically reduced to a set of 7 test cases that were also adequate. Then, random k-order mutations (where $k > 1$) were executed on the 7 test cases. 21,100 2-order mutants were created and 19 were determined to be equivalent to the original program. Impressively, every non-equivalent 2-order mutant was killed by the test cases. This experiment was extended to cover 3000 3-order and 3000 4-order randomly generated mutants. Here, each of the k-order mutants involved combinations of 1-order mutants where the 1-order mutants were all related (for example, affecting the same line). For this experiment, all of the k-order mutants were also killed.

Although their experiment presents some evidence for believing the mutation coupling effect, there are compelling reasons for additional experimentation. Specifically, only a very small percentage of the k-order mutants were generated (the experiments reported in this paper generated over 500,000 2-order mutants for FIND, so 21,100 mutants represent less than 5% of the total number). One is left wondering whether the perfect results from the earlier experiment are significant, or an artifact of the small percentage of mutants that were investigated. Also, the experimental data was not completely reported, making it difficult to evaluate the data or the process. Perhaps most importantly, some doubt remains about the validity of the coupling effect, and with the increased attention being given to testing strategies that depend on this effect, more evidence is needed to either verify or disprove this principle.

2 AN EXPERIMENT IN COUPLING

One difficulty with this type of experimentation is the sheer number of executions required. The number of mutations of a program is on the order of the square of the number of variable references in the program [2]. The number of 2-order mutations of a program is on the order of the square of the number of 1-order mutations—N^4 in the number of variable references. For this paper, 2-order mutants were constructed and executed with the Mothra mutation system [5] [1].

2.1 Experimental Procedure

As stated above, the mutation coupling effect claims that test cases that kill simple, or 1-order, mutants will also kill more complex, or n-order, mutants. Thus, if faults are coupled, then a test data set that is mutation-adequate for a set of 1-order mutants should also be adequate for a set of 2-order mutants. This assumption is the subject of the following procedure:

1. A set of test data was constructed that killed all non-equivalent 1-order mutants for a program. To remove any bias contributed by the experimenter, this test data was constructed by the automated test data generator Godzilla [7].
2. Ineffective test cases (test cases that killed no 1-order mutants) were eliminated from the set.
3. The Mothra interpreter [13] was modified to generate and execute all *connected* 2-order mutants. A pair of mutants are considered connected if they modify statements that can appear on the same execution path (as determined by a reachability test on the program control flow graph). Thus, mutation pairs that cannot both be executed simultaneously (such as appearing on different branches of a decision statement) are not considered.
4. Each 2-order mutant was executed against those test cases that caused both of the mutated statements to be executed. That is, test cases that did not execute both mutated statements were omitted.

The elimination of non-connected mutation pairs in step 3 is intended to remove a potential for experimental bias. Mutation pairs that modified statements that could never both be reached during the same execution would always be killed. Assume that the 1-order mutation m_i modifies statement s_i and is killed by test case t_i. Similarly, the 1-order mutation m_j modifies statement s_j

[1]Information on acquiring the current version of the Mothra system can be obtained from the Software Engineering Research Center, Purdue University, West Lafayette, IN 47907.

and is killed by test case t_j. If s_i and s_j are not connected, then when we generate both mutations simultaneously, t_i will execute mutation m_i on statement s_i but will never execute statement s_j, and the 2-order mutant will be killed by t_i. Mutants that are created by mutations that appear on non-connected statements will be called *no-reach* mutants in the remainder of this paper.

Since killing no-reach mutants artificially inflates the 2-order mutation score, mutations on non-connected statements were not paired to create 2-order mutants. This bias was also avoided in step 4 by not executing 2-order mutants against test cases unless the test case executed both mutated statements. By eliminating these executions, test cases were only executed against mutants when there was a possibility for mutation interaction[2].

2.2 2-Order Mutants

The technical problem of generating 2-order mutants was not difficult and required minor modifications to Mothra's interpreter. Normally, the interpreter operates in a loop over all test cases and within that loop, over all live mutants. For each mutant, the internal form of the test program is modified to induce the mutation into the code and the mutant is executed. Then the internal form is returned to its original state before executing the next mutant. To generate and execute 2-order mutants, the interpreter was executed once for each mutant m_i. It was modified so that it immediately induced m_i into the internal form, and then proceeded through the mutant execution loop to execute m_i with other mutants.

In the remainder of this paper, the notation $(m_i{:}m_j)$ will refer to a 2-order mutant. If m_i and m_j are both 1-order mutants, then the 2-order mutant $(m_i{:}m_j)$ is the mutant created by inducing both m_i and m_j into the program simultaneously. The number of 2-order mutants for even a small program is quite large. If there are N 1-order mutants of a program, then there are N^2 combinations of those mutants. We can reduce this number by a constant factor by noting that the 2-order mutant $(m_i{:}m_j)$ is usually equivalent to the 2-order mutant $(m_j{:}m_i)$ (except when

[2]A difference between this and an earlier study [19] is that these biases were **not** eliminated in the earlier experiment.

m_i and m_j affect the same intermediate code instruction, as discussed below). Also, the mutant $(m_i{:}m_i)$ produces the same program as the 1-order mutant m_i and does not need to be considered. Thus, if there are NR no-reach mutants, there are $\frac{N^2-N}{2} - NR$ distinct 2-order mutants for a program (still on the order of N^2). For this reason, for each mutant m_i, the mutant execution loop described above generated and executed every mutant m_j, where $i < j \leq N$ and N is the total number of 1-order mutants. In addition, the no-reach mutants were skipped, and test cases that did not execute both mutated statements were skipped.

Note that sometimes the second mutation that is induced into the program, m_j, will change the same intermediate code instruction as the first mutation, m_i. This has the effect of only inducing one mutation into the program, and the fact that m_j is induced rather than m_i is an artifact of the ordering of the mutations. This situation is fairly rare and the affect is very small, falling into one of three cases. First, if both m_i and m_j were killed as 1-order mutants, then either $(m_i{:}m_j)$ or $(m_j{:}m_i)$ will be killed as 2-order mutants. Secondly, if both m_i and m_j were equivalent as 1-order mutants, then either $(m_i{:}m_j)$ or $(m_j{:}m_i)$ will be equivalent as 2-order mutants. Thus, in these two cases, the order the mutations are induced is irrelevant. The third case occurs if m_i was killed and m_j was equivalent. In this case, $(m_i{:}m_j)$ is an equivalent 2-order mutant while $(m_j{:}m_i)$ will be killed as a 2-order mutant. The bias introduced by this arbitrary ordering of the application of the mutation operators is negligible, since there is very little difference in the mutation score in killing a mutant as opposed to marking it equivalent. Marking a killable mutant equivalent has the effect of reducing both the numerator and denominator by one; which alters the result by only a small amount. In no case was a killable 2-order mutant left alive or a 2-order mutant that should have lived killed.

2.3 Experimental Programs

The experiment described above was performed with the three programs MID, TRITYP, and FIND. Because of the computational expense of executing 2-order mutants, the programs must be small.

MID returns the middle value of three integers. TRITYP has been widely studied in software testing and inputs three integers that represent the relative lengths of the sides of a triangle and classifies the triangle as equilateral, isosceles, scalene or illegal. FIND was studied by Hoare [9] and by DeMillo, Lipton and Sayward [6] and accepts an array A of integers and an index F. It returns the array with every element to the left of *A(F)* less than or equal to *A(F)* and every element to the right of *A(F)* greater than or equal to *A(F)*. Some of the characteristics of these programs are shown in Table 1.

PROGRAM	LINES	M^1	M^2	NR	T
MID	16	183	10,388	6265	22
FIND	28	1022	521,731	0	19
TRITYP	28	951	350,982	100,743	53

Table 1: Experimental Programs

In Table 1, the M^1 column is the number of 1-order mutants generated by Mothra and the M^2 column is the number of 2-order mutants generated for this experiment (not counting the no-reach mutants). The number of no-reach mutants is in the NR column. The T column refers to the number of test cases that were generated to kill all non-equivalent 1-order mutants. FIND has no no-reach mutants because the body of the function is a loop, allowing each statement to be reached from each other statement. This table graphically illustrates why the coupling effect is so important to the success of fault-based testing strategies such as mutation analysis. Even with small program modules like FIND and TRITYP, there are around a half million 2-order mutants. If we need to perform this many executions to construct an adequate set of test data, then fault-based techniques would be prohibitively expensive.

2.4 Experimental Results

The 2-order mutants for the three programs were run on diskless Sun 3/50s, using a Sun 4/280 file server. Since performance was not a critical factor in this experiment, execution times were not carefully kept, however, each program typically required over a week to execute. Table 2 summarizes the results of running the 2-order mutants for the three programs above.

	M^2	K	Eq	Live	MS^2
MID	10,388	10,287	95	6	.9994
FIND	521,731	518,910	2775	46	.9999
TRITYP	350,982	346,134	4838	10	.9999

Table 2: 2-Order Results

In Table 2, the columns labeled M^2, K, Eq and Live refer to the number of 2-order mutants that were created, killed, equivalent, and that remained alive after execution of the test cases. The equivalent mutants were determined by an exhaustive hand analysis. The MS^2 column is the resultant mutation score, or the ratio of dead over non-equivalent 2-order mutants.

Even if one already had confidence in the coupling effect, the 2-order mutation scores are surprisingly high. The fact that so few 2-order mutants remained alive gives great credence to the validity of the mutation coupling effect. However, the remaining 2-order mutants are worth examining in depth. These mutants could have remained alive for one of two reasons. If the 2-order mutant has some characteristic that makes it unable to be killed by test data generated for 1-order mutants, we call the mutant *strongly* uncoupled. On the other hand, if the 2-order mutant just happened to be "missed" by the test cases, then we call the mutant *weakly* uncoupled. Strongly uncoupled mutants would indicate that the mutation coupling effect is seriously flawed as a rationale for developing test data, whereas weakly uncoupled mutants merely mean that the mutation coupling effect does not always hold, a less serious restriction for practical applications of fault-based testing. Since the first program, MID, is the smaller of the three programs, we shall examine its live 2-order mutants. MID is shown in Figure 2 with seven 1-order mutants. The live 2-order mutants are all combinations of these seven 1-order mutants [3].

There were only six live 2-order mutants for MID, (33:46), (33:154), (36:46), (36:154), (71:179), and (171:179). Interestingly, the four 1-order mutants 46, 71, 154, and 171 are all equivalent to the original program. Although this is certainly a surprising coincidence there is no obvious reason for this pattern. It does, however, imply that we cannot ignore equivalent mutants when

[3]The mutation numbers shown beside the mutated lines are the numbers used by Mothra to reference the mutants. The actual numbers are unimportant, except that they serve as convenient labels for this discussion.

```
                INTEGER FUNCTION MID (X, Y, Z)
                INTEGER X, Y, Z
1               MID = Z
2               IF (Y .LT. Z) THEN
        Δ46     IF (Y .LE. Z) THEN
        Δ154    IF (−−Y .LT. Z) THEN
3                  IF (X .LT. Y) THEN
4                     MID = Y
5                  ELSE IF (X .LT. Z) THEN
6                     MID = X
7                  ENDIF
8               ELSE
9                  IF (X .GT. Y) THEN
        Δ33        IF (X .GE. ZPUSH(Y)) THEN
        Δ71        IF (X .GE. Y) THEN
        Δ171       IF (++X .GT. Y) THEN
10                    MID = Y
        Δ36           MID = ZPUSH(Y)
11                 ELSE IF (X .GT. Z) THEN
        Δ179       ELSE IF (−X .GT. ZPUSH(Z)) THEN
12                    MID = X
13                 ENDIF
14              ENDIF
15              RETURN
```

Figure 2: Function MID

going to higher order mutants, as Morell did [16].

Mutant 33 has a **ZPUSH** unary operator mutation that generates a failure if the expression is zero and returns the value of the expression otherwise. Thus, mutant 33 will only be killed when $Y = 0$ at statement 9. In addition, for statement 9 to be executed, the test case must have $(Y \geq Z)$ at statement 2. In the 1-order case, mutant 33 was killed by the test case $(X = 8,\ Y = 0,\ Z = 0)$. When mutant 33 is combined with mutant 46, however, the test case $(X = 8,\ Y = 0,\ Z = 0)$ executes a different path and never reaches statement 9. The other test cases in this set for which $(Y = 0)$ are $(X = 97,\ Y = 0,\ Z = 1)$ and $(X = -29,\ Y = 0,\ Z = 39)$. The first test case executes the path $(1, 2, 3, 5, 7, 14, 15)$, and the other executes the path $(1, 2, 3, 4, 7, 14, 15)$; neither of which executes line 9.

This analysis explains why the 2-order mutant (33:46) did not die with this particular set of test cases, but not why the coupling effect did not hold in this case. In fact, there seems to be no subtle or generalizable reason why the test set did not kill mutant (33:46), and we must hypothesize that a different test case set developed for the same 1-order mutants may well kill the mutant.

When analyzing the other three 2-order mutants, we see a similar pattern to that of mutant (33:46). In fact, mutant (33:154) is equivalent to mutant (33:46) and both would be killed by the same test case, mutant (36:46) is equivalent to mutant (36:46), and mutant (71:179) is equivalent to (171:179). All six 2-order mutants could be killed with the two test cases ($X = 1,\ Y = 0,\ Z = -1$) and ($X = 2,\ Y = 4,\ Z = 1$).

The remaining live 2-order mutants for FIND and TRITYP follow a similar analysis. There is no reason to believe that any of the live 2-order mutants for these programs are in any way "special" and are particularly unlikely to be killed by a set of test cases that is mutation-adequate for 1-order mutants. The remaining mutants for FIND and TRITYP do not seem to be strongly uncoupled in any sense.

2.5 A Different Test Case Set

If it is true that the remaining live mutants were weakly uncoupled, then we would expect a different test case set to leave a different set of 2-order mutants alive. To verify this, three new sets of test data were generated for MID. By using the fact that Godzilla incorporates some amount of randomness in its test case generation algorithm [18], different sets of test data were generated that killed all non-equivalent 1-order mutants for MID. The results of this test data is shown in Table 3.

	M^2	K	Eq	Live	MS^2
MID 1	10,388	10,287	95	6	.9994
MID 2	10,388	10,292	95	1	.9999
MID 3	10,388	10,286	95	7	.9993
MID 4	10,388	10,287	95	6	.9994

Table 3: Repeated Results for MID

The results of repeating the 2-order experiment for MID are comparable to that of the first experiment. The sets of mutants that remained alive were, however, different. In the second experiment, for example, the only 2-order mutant that remained alive was (46:154). Not only was this mutant killed by all three of the other test case sets, but the mutants that remained alive in the other experiments were killed by this test case set. In fact, the four experiments in table 3 left disjoint sets of mutants alive, lending support to the hypothesis that the six 2-order mutants left alive by the first test case set and the mutant left alive by this test case set are only weakly uncoupled.

An interesting characteristic of this data is that all of the 2-order mutants that lived were composed of at least one equivalent 1-order mutant. However the live 2-order mutants for FIND and TRITYP did not involve combinations of any equivalent 1-order mutants, so it seems that no general conclusions can be drawn from this coincidence.

3 1-ORDER VERSUS 2-ORDER MUTATION SCORE

If the mutation coupling effect is valid, we would expect that the more complex a fault is, the more likely we are to detect that fault with test data sets derived for simple faults. In other words, when $j > i$, j-order mutants are easier to detect than i-order mutants. This implies that for the same test data set, the 2-order mutation score should be higher than the 1-order mutation score. We rephrase this in a modified form of the mutation coupling effect:

Hypothesis. *Mutation Coupling Effect, Modified Form*: Complex mutants are *coupled* to simple mutants in such a way that a test data set will detect a higher percentage of complex mutants than of simple mutants.

To test this hypothesis, sets of test data were generated for MID and TRITYP that were not fully mutation-adequate. To avoid any bias, test cases were automatically generated and as in the experiments in section 2, no-reach mutants were not considered. To get a weaker set of test cases, the data was generated randomly until the desired mutation score was reached. This test data was executed against the 1-order mutants and then the 2-order mutants. The results of this experiment are shown in Table 4.

		M	K	Eq	Live	MS
MID	1-order	183	148	16	19	.89
MID	2-order	10,388	9,955	95	338	.97
MID	1-order	183	126	16	41	.75
MID	2-order	10,388	9,450	95	843	.92
TRITYP	1-order	951	746	108	97	.88
TRITYP	2-order	350,982	332,628	4838	13516	.96
TRITYP	1-order	951	622	108	221	.74
TRITYP	2-order	350,982	304,935	4838	41209	.88

Table 4: Coupling With a Weaker Test Case Set

All four sets of test data resulted in a significant increase in mutation score from the 1-order case to the 2-order case. These increases in mutation scores indicate that the above hypothesis is true, giving us some confidence that repeating the above experiments with k-order mutants, where $k > 2$, would yield even higher mutation scores.

From the data in the previous section, one might conclude that as we go from 1-order to 2-order mutants, the mutation score actually decreases, so these two experiments may superficially seem to contradict each other. In reality, when applying a test data set to 2-order mutants, we expect some of the 2-order mutants to live, and some to die. The first experiment shows that many more live than die; the second experiment implies that the mutation score for 2-order mutants in the first experiment should be very close to 1.0, rather than exactly 1.0. The 2-order mutants that lived in the first experiment (a very small percentage, ranging from .0001 to .0007) are an artifact of the probabilistic nature of the coupling effect.

4 SOME 3-ORDER RESULTS

If test data developed for 1-order mutants kills a higher percentage of 2-order mutants, we would expect it to kill a still higher percentage of 3-order mutants. In fact, as N gets large, we would expect the percentage of N-order mutants killed to tend towards 1.0. To investigate this, two 3-order experiments were performed. Since the number of 3-order mutants is so huge, the first program was a two line function that returned the maximum of two integers; the other program was MID.

There were 1,004,731 3-order mutants for MID, 401,890 of which were no-reach mutants. Since MAX is a straight-line program, there were no no-reach mutants. This experiment was not performed for a larger function such as TRITYP or FIND because of the sheer numbers of 3-order mutants. If there are M 1-order mutants for a program, then there are:

$$\binom{M}{3} = \frac{M^3 - 3M^2 + 2M}{6}$$

3-order mutants. For example, FIND has 177,388,540 3-order mutants!

		M	Live	MS
MAX	1-order	43	10	.75
MAX	2-order	903	53	.94
MAX	3-order	12,341	64	.99
MID	1-order	183	41	.75
MID	2-order	10,388	1459	.92
MID	3-order	602,841	4097	.99

Table 5: 3-Order Mutants

The results shown in Table 5 indicates that the strength of the mutation coupling effect increases with 3-order mutants. This is strong evidence that as the number of faults in a program that contribute to a failure increases, our test data is more likely to detect the failures.

5 CONCLUSIONS

This paper presents several results about the coupling effect as measured over the domain of mutation analysis. First, we found that test data developed to kill 1-order mutants are very successful at killing 2-order mutants. Of course, the fact that two mutants are executed on the same path does not necessarily mean that they interact in any meaningful way, but such interactions are difficult to determine analytically. By including all 2-order mutations on the same paths, we include all those that can interact. Second, we observed that the 2-order mutants that remained alive exhibited no characteristics that would lead one to believe that they were impossible or difficult to kill using test data developed for 1-order mutants. Third, we found that the set of test data developed for 1-order mutants actually killed a higher percentage of mutants when applied to 2-order mutants. Finally, test data generated for 1-order mutants killed a higher percentage of mutants when applied to 3-order mutants.

The important result is that when using mutation testing we can focus on 1-order mutants, and ignore n-order mutants. Whether killing n-order mutants means we can detect complex faults is yet to be determined. As discussed in the introduction, the set of higher-order mutants is a subset of the complex faults. The ability to kill higher-order mutants implies the ability to detect complex faults if either the higher-order mutants are a large percentage of the complex faults or if the complex faults are easier to detect than higher-order mutants. Although it was argued in section 1.2 that both seem likely, we cannot be sure without further evidence. Regardless of the relationship between complex mutants and complex faults, it is encouraging that the coupling effect manifested itself quite emphatically when restricted to mutation analysis.

The fact that such a tiny number of second order mutants survived the test cases is pleasantly surprising. This leads us to conjecture that the mutation coupling effect holds true a very large percentage of the time, agreeing with the probabilistic analysis given by Morell [16]. These results are very encouraging for software testing researchers and practitioners. This positive empirical evidence about the coupling effect indicates that fault-based strategies are based on a firm foundation. The important practical implication of this result is that when we test software by focusing on a small restricted class of faults we can expect to detect more complicated faults as well, giving us confidence that fault-based testing strategies can provide effective ways to test software.

6 ACKNOWLEDGEMENTS

I would like to thank the members of the Software Engineering Research Center at Purdue University for ongoing support and maintenance of Mothra, without which this study would not have been possible. I would also like to thank Peter Ware for suggesting the experiment in section 3 and various attendees of TAV-3 for pointing out the bias concerning the no-reach mutants that was eliminated in the experiments. The reviewers for ACM TOSEM have also provided valuable insights and criticisms, particularly in helping develop the definitions of *complex fault*, *mutation coupling effect* and *coupling effect* in the Introduction.

7 BIBLIOGRAPHY

References

[1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, September 1979.

[2] T. A. Budd. *Mutation Analysis of Program Test Data.* PhD thesis, Yale University, New Haven CT, 1980.

[3] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, November 1982.

[4] W. M. Craft. Detecting equivalent mutants using compiler optimization techniques. Master's thesis, Department of Computer Science, Clemson University, Clemson SC, 1989. Technical Report 91-128.

[5] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop*

on Software Testing, Verification, and Analysis, pages 142–151, Banff Alberta, July 1988. IEEE Computer Society Press.

[6] 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.

[7] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[8] M. R. Girgis and M. R. Woodward. An experimental comparison of the error exposing ability of program testing criteria. In *Proceedings of the Workshop on Software Testing*, pages 64–73. IEEE Computer Society Press, July 1986.

[9] C. A. R. Hoare. Proof of a program: Find. *Communications of the ACM*, 14(1):39–45, January 1971.

[10] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, September 1976.

[11] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[12] W. E. Howden. *Functional Programing Testing and Analysis*. McGraw-Hill Book Company, New York NY, 1987.

[13] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software–Practice and Experience*, 21(7):685–718, July 1991.

[14] R. J. Lipton and F. G. Sayward. The status of research on program mutation. In *Digest for the Workshop on Software Testing and Test Documentation*, pages 355–373, December 1978.

[15] B. Marick. Two experiments in software testing. Technical report UIUCDCS-R-90-1644, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign Illinois, November 1990.

[16] L. J. Morell. *A Theory of Error-Based Testing.* PhD thesis, University of Maryland, College Park MD, 1984. Technical Report TR-1395.

[17] L. J. Morell. Theoretical insights into fault-based testing. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 45–62, Banff Alberta, July 1988. IEEE Computer Society Press.

[18] A. J. Offutt. An integrated automatic test data generation system. *Journal of Systems Integration*, 1(3):391–409, November 1991.

[19] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.

[20] D. J. Richardson and M. C. Thompson. The relay model for error detection and its application. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 223–230, Banff Alberta, July 1988. IEEE Computer Society Press.

[21] E. J. Weyuker. Assessing test data adequacy through program inference. *ACM Transactions on Programming Languages and Systems*, 5(4):641–655, October 1983.

[22] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3):236–246, May 1980.