The Theory of Composite Faults

Abstract

Fault masking is an important problem in software testing, where the effect of one fault serves to mask that of another fault for particular test inputs. The *coupling effect* is relied upon by testing practitioners to ensure that fault masking is rare. It states that *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.*.

While this effect has been empirically evaluated, our theoretical understanding of *the coupling effect* is as yet incomplete. Wah proposed a theory of the coupling effect on finite bijective (or near bijective) functions with the same *domain* and *co-domain*, and assuming uniform distribution for candidate functions. This model however, was criticized as being too simple to model real systems, as it did not account for differing *domain* and *co-domain* in real programs, or for syntactic neighborhood (it assumed all faults equally likely).

We propose a new theory of fault coupling that is applicable in the realm of general functions (with certain constraints). We show that there are two kinds of fault interactions, of which only the weak interaction can be modeled by the theory of the coupling effect. The strong interaction can produce faults that are semantically different from the original faults. These faults should hence be considered as independent atomic faults. Our analysis show that the theory holds even when the effect of syntactical neighborhood of the program is considered.

We also propose a modified hypothesis that is stronger than the traditional definition — the *composite fault hypothesis*:

Tests detecting a fault in isolation will (with high probability κ) continue to detect the fault even when it occurs in combination with other faults.

We analyze numerous real-world programs with real faults to validate and empirically approximate the *composite coupling ratio* κ. We find that κ is approximately 99% with 95% confidence, and very close to the general coupling ratio C which was also found to be greater than 99% with 95% confidence.

Categories and Subject Descriptors D [2]: 4

General Terms faults, coupling

Keywords test frameworks, evaluation of coverage, coverage criteria, statistical analysis

1. Introduction

Fault masking occurs when interactions between component faults in a complex fault result in expected (non-faulty) value being produced for particular test inputs. This can result in faults being missed by test cases, even though the complex fault may produce faulty outputs for other values, hence negatively impacting the testability of a software system. It can also result in undeserved overconfidence in the reliability of a software system.

The *coupling effect* [10] hypothesis studies the semiotics[1] of fault masking. It asserts that *"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."* [31, 30, 18].

This is relied upon by software testers to assert that fault masking is indeed rare. Further, Mutation analysis [23, 7], the premier technique for evaluating test suite quality relies on the *coupling effect* to assert that detecting simple faults is sufficient to guarantee the quality of a test suite against more complex faults.

However, our understanding of the *coupling effect* is woefully inadequate. We do not know when, and how often fault coupling can happen, whether multiple faults will always result in fault coupling, and the effect of increase in number of faults in the number of faults masked.

Further, the formal statement of coupling effect itself is ambiguous and inadequate as it covers only the case where all simple faults have been found. Even worse, it has no unambiguous definition of what a simple (or atomic) fault is. We propose a stronger version of the *coupling effect* (called the *composite fault hypothesis* to avoid confusion):

Tests detecting a fault in isolation will (with high probability κ) continue to detect the fault even when it occurs in combination with other faults.

We investigate our hypothesis both theoretically and empirically. We describe how faults interact, categorize fault interaction to weak and strong interaction, and provide an unambiguous description of atomic faults. We further validate our findings using real faults from numerous software systems. The terms used in this paper are given in Note 1.

1.1 Theory

Theoretical aspects of the coupling effect on functions was investigated by Wah et al. [36, 37, 39]. Wah assumes that any software is built by composition of q independent functions, with a few restrictions. This model is called the standard q-function model. The q-functions have the following restrictions:

- Functions have same *domain* and *range* (order n) so that any two functions can be composed together, and the functions are *bijective* (one-to-one). The non-*bijective* functions are modeled as *degenerate* functions which are close to *bijective*, but with a few duplicate inputs that map to same location in image.

[1] The relation between syntax and semantics of faults.

(Semantic) Separability of faults: Two faults present in a function are said to be separable *if and only if* the smallest possible chunk containing both faults can be decomposed into two functions g and h such that each fault is isolated within a single function (providing g_a and h_b as faulty functions), the behavior of composition $h \circ g$ equals the behavior of the original chunk in terms of input and output , and composition $h_b \circ g_a$ equals the behavior of the chunk with both faults.
Simple fault: A fault that cannot be *lexically* separated into other independent smaller faults. Also called a *first order* fault.
Complex fault: A fault that can be *lexically* separated into independent smaller faults. Also called a *higher order* fault, or a *combined fault*.
Constituent fault: A fault that is *lexically* contained in another fault.
Atomic fault: A fault that cannot be *semantically* separated into other independent smaller faults.
Composite fault: A fault that can be *semantically* separated into independent smaller faults.
Traditional coupling ratio (C): The ratio between the percentage of complex faults detected and the percentage of simple faults that were detected by a test suite.
Composite coupling ratio (κ): The ratio between the percentage of complex faults detected by the same set of test cases that detected the constituent simple faults, and the percentage of constituent simple faults detected.
Domain *of a function*: The *domain* of a function is the set of all values a function can take as inputs (this is practically the input type of a function).
Co-Domain *of a function*: The *co-domain* of a function is the set of all values that a function can produce when it is provided with a valid input from its *domain* (this is practically the output type of a function).
Range of a function: The *range* of a function is the set of all values in *co-domain* that directly maps to a value in the *domain*.
Finite domain *and* co-domain: We assume that the set of possible input values, and the set of output values corresponding to them, are finite.
Total functions: Functions where an output is defined for all elements in its *domain*.
Syntactic neighborhood: The set of functions that can be reached from a given function by modifying its *syntactical representation in a given language* a given number of times.
Degenerate function: A function which is not *bijective* from *domain* to *range*. If n values in *domain* maps to duplicate values in *range*, it is called an n-degenerate function.

Note 1. Terms used in this paper

- A program with two faults can be split into two programs, with one program containing one fault, and the other program the other[2].
- All applicable functions have equal probability of being chosen as a faulty representation of another (ignoring the syntactical neighborhood of a function) – the *democratic assumption*.
- The number of functions considered q is much smaller than the size of the domain. That is, $q \ll n$. Wah suggests that as q nears n, the coupling effect weakens.

Wah showed that for q functions, the survival ratio of first and second order test sets are $\frac{1}{n}$ and $\frac{1}{n^2}$ respectively, where n is the order of the *domain*. Further, Wah finds a general observation, and a heuristic, that the survival ratio of a multi-fault alternate is $\frac{p+1}{n}$ if there are p fault free functions at the end after a faulty function. This allows Wah to solve the q-function model since there are $2^{p-1}-1$ multi fault alternates with last faulty function at p. Hence, the expected number of survivors for a q-function composition is given by:

$$\frac{1}{n^r}\sum_{p=1}^{q}(2^{p-1}-1)(q-p+1)^r$$

for test sets of order r, q-functions, and n the order of the domain.

Wah's analysis, however, lacks wider applicability due to the constraints placed on functions. Functions in typical programs vary widely in their *domain* and *co-domain*. Second, the assumption that distribution of mathematical functions can represent the distribution of programs with same *domain* and *co-domain* is rather strong because there can be an infinite number of alternative programs that represent the same mathematical function. Third, the assumption that all applicable functions are equally probable as faulty representation of a function ignores the impact of syntactical neighborhood, because functions that are in the syntactical neighborhood (those functions that can be reached by a small modification of the original function) are more probable as faulty replacements than others further away. That is, it is possible that a *quick sort* implementation can have a small bug, resulting in an incorrect sort. However, it is quite improbable that it is replaced by an algorithm for — say — *random shuffle*, which has the same *domain* and *co-domain* as that of a sorting function. While syntactical nearness does not completely capture semantic nearness, it is closer than assuming any function is a plausible fault for any other function. Next, Wah assumed that all complex faults can be semantically separated to component faults with the component faults appearing in independent functions. However, as we show in Section 3, this assumption is valid only in certain cases. Further, Wah's analysis does not account for recursion and iteration. If one assumes that different functions in the q-function model may be considered as different steps in an iteration, it falls short due to the specification that q be much less than the domain of the function considered (at which point, Wah suggests that the approximations no longer hold true). Finally, Wah's analysis suggested that the survival ratio of mutants is dependent on the *domain* of the function. However, this result was due to the assumption that both *domain* and *range* of the functions examined were similar. We show that the survival ratio of a mutant is actually dependent on the *co-domain* of the function examined, when *domain* and *co-domain* are different (we note that the size of *co-domain* is bounded by the size of *domain* of the function).

We propose a simpler theory of fault coupling that uses a similar model as that of Wah, but with relaxed constraints. Our analysis incorporates functions with differing *domain* and *co-domain*. We clarify the semantic separability of complex faults, and show how it affects the coupling effect. More importantly, we show that certain common classes of complex faults may not semantically separable. This provides us with a definition of an *atomic fault*, as a fault that cannot be semantically separated into simpler faults. This is important because two faults that may be lexically separate but inseparable can be expected to produce a different faulty behavior than either fault considered independently. Further, we consider the impact of syntactic neighborhood. Using both case analysis and statistical argument, we show that our analysis remains valid even when the syntactic neighborhood is considered.

1.2 Empirical Validation

The first empirical study of the coupling effect was conducted by Lipton et al. [24, 10]. They used a sample of $22,000$ mutants for the program *find* up to the fourth order, and observed that test cases for first order mutants were adequate for samples up to fourth

[2] This was assumed by Wah to be true for general functions. However, we show in Section 3 that it is not applicable to all functions.

order. More empirical evidence for the coupling effect was supplied by Offutt [31, 30] using two more programs, *mid* and *tryp*, who observed that the tests for first order mutants were sufficient to kill up to 99% of all 2^{nd} order mutants, and 99% of 3^{rd} order mutants sampled.

The validity of mutation analysis itself — that mutations are coupled to real faults — has been demonstrated many times in the past. DeMillo et al. [11] showed that TEX errors could be coupled to simple mutants, and Daran et al. [9] found that 85% of error traces produced by mutants were similar to those produced by real faults. Andrews et al. [2, 3], found that faults generated by mutation analysis are similar in detectability to real world faults (in contrast to hand seeded faults). Do et al. [13] showed that using mutation faults for test prioritization result in higher detection rates than using hand seeded faults. Comparing four adequacy criteria — mutation, edge pair, all uses, and prime path — Li et al. [22] showed that mutation adequate testing could detect hand seeded faults better (85%) than other criteria (65%). Finally, Just et al. [19] empirically showed that the effectiveness of a test suite in detecting mutants mirrors its ability to detect real faults, and the faults introduced by mutation were coupled to 73% of real faults.

We note that, as Offutt suggests [31, 30], there are two distinct definitions of coupling involved. The *general coupling effect* suggests that simple faults are coupled to more complex faults such that test data adequate for simple faults will be able to kill a majority of more complex faults. The *mutation coupling effect* states that test data adequate for simple first order mutants will be able to detect a majority of more complex mutants. While the *mutation coupling effect* has been demonstrated, the *general coupling effect* has not been empirically validated yet.

Our empirical analysis aims to accomplish the following goals: First, we empirically evaluate the value of the composite coupling ratio κ for numerous real-world projects. This gives us confidence in the assumptions made in the theoretical analysis, and serves to validate the *composite fault hypothesis*. Second, we empirically evaluate the general coupling effect for faults, and compute the traditional coupling ratio C.

What is the relation between the composite coupling ratio κ and the traditional coupling ratio C? We can regard the composite coupling ratio as a lower limit of the traditional coupling ratio. As we explain further, the general coupling ratio does not discount the effect of strong fault interactions, which can produce complex faults semantically independent from the constituent faults. Hence, C is not bounded by any number, and will often be larger than κ, which will be strictly less than one.

Contributions:

- We identify the vagueness of formal statement of the *coupling effect* in dealing with non-adequate mutation scores, and propose the *composite fault hypothesis* to resolve this inadequacy.
- Our theoretical analysis results in the *composite fault hypothesis* for general functions. We find the composite coupling ratio to be $1 - \frac{1}{n}$, where n is the *co-domain* of function considered.
- We show that our analysis remains valid even when considering recursion and loops — common features in programming.
- We provide empirical validation for the *composite fault hypothesis*, and compute the composite coupling ratio κ to be greater than 0.99, with 95% confidence using 25 projects. This helps substantiate the impact of composite coupling.
- We provide the first empirical validation of the *general coupling effect*, which states that simple faults and more complex faults are coupled.

Our full data set is available for replication[3].

2. Related Work

Fault masking in digital circuits was studied before it was studied in software. Dias [12] studies the problem of fault masking, and derives an algebraic expression that details the number of faults to be considered for detection of all multiple faults.

Coupling effect was first studied under the aegis of *mutation analysis*. Hence we review the general mutation analysis research first. The idea of mutation analysis was first proposed by Lipton [23], and its main concepts were formalized by DeMillo et al. in the “Hints” [10] paper. The first implementation of mutation analysis was provided in the PhD thesis of Budd [6] in 1980. Previous research in mutation analysis [5, 25, 33] suggests that it subsumes different coverage measures, including *statement*, *branch*, and *all-defs* dataflow coverage [5, 25, 33]. There is also some evidence that the faults produced by mutation analysis are similar to real faults in terms of error trace produced [9] and the ease of detection [2, 3]. Recent research by Just et al. [19] using 357 real bugs suggests that the mutation score increases with test effectiveness for 75% of the cases, which was better than the 46% reported for structural coverage.

The validity of mutation analysis rests upon two fundamental assumptions: *the competent programmer hypothesis* — which states that programmers tend to make simple mistakes, and the *coupling effect* — which states that test cases capable of detecting faults in isolation continue to be effective even when faults appear in combination with other faults [10]. The coupling effect was first validated by Lipton et al. [24, 10] up to fourth order by sampling. Offutt [30, 31] investigated second order exhaustively, and sampled until 3^{rd} order mutants. Langdon et al. [21] using 85 first order mutants of *triangle* program and $16,383$ tests observed that tests that kill more first order mutants kill proportionately more second and third order mutants. Morell [28] provided a theoretical treatment of fault based testing. Morell suggested that the competent programmer hypothesis is same as *alternate sufficiency* in fault based testing which suggests that at least one of the alternates in the arena being considered is the correct version, and also provided a formal treatment of the coupling effect [27] (i.e. the assumption is that fault masking has low probability of occurrence). Morell [26] also provides a proof that no general algorithm exists to identify existence of coupling between faults. Wah et al. [36, 37, 38, 39] using a simple model of finite functions (the *q-function* model, where q represents the number of functions thus composed) showed that the survival ratio of first and second order test sets are respectively $\frac{1}{n}$ and $\frac{1}{(n^2-n)}$ where n is the order of the *domain* [18]. A major finding of Wah is that *the coupling effect weakens as the system size (in terms of number of functions in an execution path) increases (i.e. q increases), and it becomes unreliable when the system size nears the domain of functions*. Another important finding was that *minimization of test sets has a detrimental effect*. That is, for n faults, one should use n test cases, with each test case able to detect $n-1$ faults (rather than a single fault) to ensure that the test suite minimizes the risk of missing higher order faults due to fault masking. Kapoor [20] proved the existence of the coupling effect on logical faults. The competent programmer hypothesis was quantified recently by Gopinath et al. [14]. Voas et al. [35] and later Woodward et al. [40] suggests that the *domain* to *range* ratio has an impact in hiding faults. Specifically, functions with a high *DRR* tend to mask faults. A similar measure is Dynamic Range to Domain ratio studied by Al-Khanjari et al. [1]. They found that for some programs

[3] blinded for review.

there is a strong relationship between the ratio and testability, but it is weak for others.

Androutsopoulos et al. [4] suggested an approach using information theoretic measures to study failed error propagation. It was also found that one in ten tests suffered from failed error propagation. Clark et al. [8] found that likelihood of collisions were strongly correlated with an information theoretic measure called *squeeziness*, which is related to the amount of information destroyed after applying the function to its input. Hence choosing a path for tests that minimize squeeziness would reduce fault masking.

A fruitful area of research has been reducing the cost of mutation analysis, broadly categorized as *do smarter*, *do faster*, and *do fewer* by Offutt et al. [32]. The *do smarter* approaches include space-time trade-offs, weak mutation analysis, and parallelization of mutation analysis. The *do faster* approaches include mutant schema generation, code patching, and other methods to make the mutation analysis faster as a whole. Finally, the *do fewer* approaches try to reduce the number of mutants examined, and include selective mutation and mutant sampling.

An important area of research when considering the coupling effect is that of higher order mutants [16, 17, 29]. The important idea here is that of subsuming higher order mutants, which are mutants that are harder to kill than their constituent mutants. These are mutants where there is at least a partial masking of effect of the first mutant by the second mutant. While incidence of such mutants are low (as our results show), they are important for the simple reason that they represent the hard to find bugs that tend to interact, and hence represent a different class of bugs.

Our research is an extension of the theoretical work of Wah [37] and Offutt [30, 31]. The major theoretical difference from Wah [37] is that, given a pair of faulty functions that compose, we try to find the probability that, for given test data, the second function masks the error produced by the first one. On the other hand, Wah [37] tries to show that the coupling effect exists considering the *entire* program composed of q functions, each having a single fault (given by q in the q-function model). Next, Wah [37] assumes semantic separability of all complex faults. However, as we show, there exist a class of complex faults that are not semantically separable. We make this restriction clear. Further, our analysis shows that the probability of coupling is related to the *co-domain*, not the *domain*, as Wah [37] suggests. In fact, Wah [37] considers only functions which have exactly same *domain* and *range*, and hence are more restricted than our analysis. Finally, we show that even if syntax is considered, our analysis results remains valid.

With regard to the work by Offutt [31], the primary difference is in the empirical relation we attempt to demonstrate. While Offutt [31] evaluates the traditional coupling effect, and shows the empirical relation with respect to *all* simple faults and their combinations, we aim to demonstrate the *composite fault hypothesis* and evaluate the relation between any pair of faults, and the combined fault including both.

3. Theory of Fault Coupling

We start with a functional view of programs where programs are constructed by composition of different functions (this an assumption that was also adopted by Wah [39]). While Wah considered composition of q functions, which may have as many as q faults, we consider only pairs of faulty functions. This is because any faulty program with a number of separable faults can be modeled as composition of two faulty functions, which may themselves contain complex separable faults. Hence, a theoretical evaluation of faulty function pairs is sufficient to model higher order faults.

We note that our analysis is subject to these major assumptions, also made by Wah [37]: in the initial phase, we assume that we can model programs as mathematical functions. This is the biggest assumption that we make, since there can be an infinite number of syntactical equivalent alternatives to any given program. We assume finite *domain* and *co-domain* for functions considered, and only total functions. We assume that faulty versions of a function have same *domain* and *co-domain* as that of the non-faulty version[4]. Further, we model single parameter functions. We note that any function can be regarded as a single parameter function by considering their input as composed of a tuple of all the original parameters. That is, we consider a function with two inputs – say a and b as a function taking a tuple: (a, b) as input, with size of *domain* given by $dom(a) \times dom(b)$.

A significant difference from the standard q-function model of Wah is that, while Wah considers how a known number of test inputs (1, 2, 3 and higher) some of which can detect some of the component faulty functions (there can be as many as q faulty functions) can together detect the composite faulty function, we consider the probability of any single test input that can detect a fault being masked by the addition of a new fault. This allow us to significantly simplify our analysis.

In order for our theory to be applied, the fault pair needs to be separated out such that first function and second function can be clearly demarcated. The theory relies on the *co-domain* of first function being the *domain* of the second. This means that the influence of first should be limited to the input parameter for the second, and the first completely independent of the second (we show later how this condition may be relaxed). There are various places where this assumption may not hold. Note that the theory *does not* rely on the constituent faults being considered to be simple.

A major idea in our analysis is the semantic separability of faults. Two faults present in a function are said to be separable *if and only if* the smallest possible chunk containing both faults can be decomposed into two functions g and h such that each fault is isolated within a single function (providing g_a and h_b as faulty functions), the behavior of composition $h \circ g$ equals the behavior of the original chunk in terms of input and output , and composition $h_b \circ g_a$ equals the behavior of the function with both faults. A *chunk* here is any small section of the program that can be replaced by an independent function preserving the behavior.

That is, given a function:

```
def functionX(x, y, n)
  for i in (1..n):
      y = faultyA(x)              (1)
      if odd(i): x = faultyB(y)   (2)
      x += 1
```

The lines (1) and (2) together form a chunk. The interaction between the faults and their separability is discussed next.

3.1 Interaction Between Faults

We define two kinds of interaction between faults here: *weak* interaction, and *strong* interaction. *Weak* interactions occur when faults can be isolated into different functions such that the output of the function containing the first fault is the input of a function containing the second fault.

That is, given two faults $\hat{a}$ and $\hat{b}$ in a function f, which can be split into $f_{ab} = h_b \circ g_a$, where g_a and h_b are faulty functions, the only interaction between $\hat{a}$ and $\hat{b}$ is because the fault $\hat{a}$ modifies the input of h (or h_b) from $g(i_0)$ to $g_a(i_0)$ (where i_0 is an input for

[4] In practical terms, the (*domain*,*co-domain*) of a function is its type signature. It is possible that the *range* (or *image*) of a faulty version may differ from the non-faulty version. However, we assume that a valid alternate that survives the compiler will have same type signature, and hence same *co-domain*.

f). That is, the interaction can be represented by a modified input value.

Strong interactions happen when the interpretation of the second fault is affected by the first fault. In this case, we cannot separate them out into independent faulty functions. For a practical example, consider the following function in Python:

```
def swap(x,y): x,y=y,x
```

Say this was mutated into

```
def swap(x,y): x,y=x,y
```

Clearly, there were two independent lexical changes: $x \to y$ and $y \to x$. However, the disassembly of the original program is given by

```
>>> dis.dis(swap)
  1      0 LOAD_FAST          1 (y)
         3 LOAD_FAST          0 (x)
         6 ROT_TWO
         7 STORE_FAST         0 (x)
        10 STORE_FAST         1 (y)
        13 LOAD_CONST         0 (None)
        16 RETURN_VALUE
```

It can be seen that the changes in Python resulted in intertwined changes, and hence cannot be separated out. Since the faults cannot be separated out, *strong* interactions produce faults that have a different characteristic from the simple faults from which they are built, and hence should be considered as independent atomic faults[5]. Why should we consider the semantically inseparable faults as independent faults? An intuitive argument is to consider two functions that implement id (these are not strongly interacting). That is, given any value x, we have $g(x) = h(x) = x$. If two faults $\hat{a}$, and $\hat{b}$ occur as we suggest above in g and h, causing inputs i to g_a and inputs j to h_b to fail, then the faulty inputs for $h_b \circ g_a$ are bounded by $i \cup j$, where i represents inputs to f that result in faulty outputs due to faults in g and j represents inputs to f resulting in faulty outputs due to faults in h.

What about fault masking? Any input i that failed for g_a could possibly result in an input value that would cause a failure for h_b. For any element outside of i, there is no possibility of two faults acting on it, and hence no possibility of fault masking. However, if the faults are not semantically separable, one cannot make these guarantees, as the faulty inputs may be larger than $i \cup j$ or even completely different. In the general case, when the interaction is weak, we expect the faulty output for up to $i \cup j$. These are represented in Figure 1.

This is simple enough to prove formally. Consider a function f that has *domain* x. Say that it can be represented as $h \circ g$ using two independent functions. Replacing g with g_a causes $i \in x$ inputs to result in faults. Similarly, replacing h with h_b causes $j \in x$ inputs to f to result in faults. Joining together to form f_{ab}, we know that any of $i \in x$ has a potential to produce a faulty output unless it was masked by h_b. Similarly, any of $j \in x$ also has the possibility of producing a faulty output. Now, consider any element k not in either i or j. It will not result in a faulty output while passing through g_a because it is not in i, further, the value $g_a(k) = g(k) = k_1$. We already know that k_1 would not result in a faulty output from h_b because $k \notin j$. Hence, any element $k \notin i \cup j$ will not be affected by faults $\hat{a}$ and $\hat{b}$.

Remember that we can make this assertion only because we can replace g and h separately. Suppose it is a strong interaction between $\hat{a}$ and $\hat{b}$. In this case, any function could potentially be a replacement for f. Hence, any element in x may potentially result in a fault when f_{ab} is applied. Indeed, we are not the first ones to make this observation. These kinds of interactions where the higher order faults have a different behavior (and result in different inputs with faulty outputs) than the component faults are called decoupled higher order mutants by Harman et al. [15].

Depending on the syntax and semantics of the language used, there may be different language features (another common example is the order of arguments to a function) that cause faults to be strongly interacting.

3.2 Analysis

Say we have a program f, where we have two simple faults $\hat{a}$, and $\hat{b}$, which can be applied to f to produce two functions f_a and f_b containing one fault each, and f_{ab} containing both faults (Figure 3).

Say such a program can be partitioned into two functions g and h such that

$$f = h \circ g$$

with added restriction that the fault $\hat{a}$ lies in function g, producing an alternative function g_a, and the fault $\hat{b}$ lies in function h producing h_b, such that the new faulty version of f containing both is given by

$$f_{ab} = h_b \circ g_a$$

We note that the particular kind of fault depends on the syntax and semantics of the programming language used, and there can be fault pairs that cannot be separated cleanly. As stated previously, we ignore these kinds of fault pairs as they are syntax dependent and strongly interacting. Hence, no general solution is possible for these faults.

The basic question we are trying to resolve is, given that we can distinguish a fault in isolation using a given input, what is the probability that another fault would not result in the masking of that fault for the same input?

That is, we are given a test input i_0 for f, that can distinguish between (f, f_a). The question is, what is the probability that (f, f_{ab}) can be distinguished by the same input?

Since we know that f_a is distinguished from f, we know that $g_a(i_0) \neq g(i_0)$. Hence, the function h_b will have a different input than h. Thus, the question simplifies to: given an alternate input for function h (or anything that can be substituted in its place), what is the probability that a faulty h, with the new input $g_a(i_0)$ will result in same output as the old h, with the old input $g(i_0)$?

Let us assume for simplicity[6] that functions g and h have fixed *domain* and a *co-domain* given by

$$g \in G : \mathbb{L} \to \mathbb{M}$$
$$h \in H : \mathbb{M} \to \mathbb{N}$$

That is, h belongs to a set of functions H, which has a *domain* M, and a *co-domain* N such that $m = |M|$ and $n = |N|$. Considering all possible functions in H, with the given *domain* and *co-domain*, there will be n^m unique functions in H (separated by at least one different $\{input, output\}$ pair).

[5] We note that Wah's analysis [37] assumes that all complex faults can be separated, and does not take into consideration such strongly interacting faults.

[6] There are two factors that make this assumption reasonable: first, for real world functions, there is almost always a limit on the *domain* and *co-domain* whether it be due to the data type used, or due to constraints such as the available resources such as memory. Second, as we show later, the size of the *co-domain* is actually inversely proportional to the masking effect, which helps our case as the *co-domain* increases (note that *domain* will always be larger than *co-domain*). We also note that the assumption of finite *domain* was also made by Wah [37].

The only constraint on h_b we have is that $h_b(g_a(i_0))$ should result in the exact same output as $h(g(i_0))$. That is, for some particular input for h_b, the output is fixed at a particular output. We are looking for functions that can vary in every other $\{input, output\}$ pair except for the pair given by $\{g_a(i_0), h(g(i_0))\}$. There are n^{m-1} functions that can do that out of $|H| = n^m$ functions. That is, the composite coupling ratio is given by

$$\kappa = 1 - \frac{n^{m-1}}{n^m}$$

This is simplified to $1 - \frac{1}{n}$ of the total number of eligible functions where m is the size of *domain*, and n is the size of *co-domain* of the function. That is, given any test input, the probability of the composite coupling effect where the fault in one constituent is not masked by the fault in another is $1 - \frac{1}{n}$, and $\frac{1}{n}$ tends to be very small when the *co-domain* of the function (n) is large.

A symmetric argument can be made when the function fixed is h, and g varies. There are m^l functions in G, of which m^{l-1} can be used as a replacement without affecting $\{input, output\}$, in which case, the probability of composite coupling effect is $1 - \frac{1}{m}$ where m is the *co-domain*[7].

For example, in the case of a function with its *domain* and *co-domain* restricted to boolean, with *domain* 2 and *co-domain* 2, the probability of composite coupling effect is $1 - \frac{1}{2} = 0.5$, and in the case of an octet (8 bits), the probability of composite coupling effect is 0.9961 (given equal probability for all faults).

This is easy to verify in the case of booleans: There are 2^2 possible unique boolean functions. f_{id} is just the identity function, and f_{not} is the not function. f_{true} always returns $true$, and f_{false} always returns $false$. Consider a combination of functions such as $f = f_{id} \circ f_{not}$. We consider a test input $true$, which will be converted to $false$ by function f. Say there is an error in f_{id}, and we mistakenly used f_{false} instead (note that we would not be able to use f_{true} since we assume that the original input would have detected the error if it was taken individually). The question is, how many ways can you make a faulty f_{not} such that we get $false$? That is, which functions map $false$ to $false$? There are just two – f_{id}, and f_{false} which is 50% of the available functions.

3.3 Recursion and Iteration

A common feature of general purpose languages is *recursion* and *iteration*. These present challenges to our analysis. For example, consider the loop below.

```
while y > 0: y = h(g(y)
```

Here, the two functions g and h are otherwise independent. However, the input of h influences g, and vice versa. Here, we do not know when the loop will end, and any faults will be detected. The faults may be detected after a larger or smaller number of iterations than the non faulty version of the program. Hence, what we can do, is to consider the chances of propagation of the faulty value after each iteration. That is, if a faulty value is present after executing the function g_a once, what are the chances that it will be caught at the end of each iteration?

Let us use f to denote the program segment composed of g and h, as before. After the first iteration of f, we will have $\frac{1}{n}$ possibility for fault masking as we discussed before, and $\frac{n-1}{n}$ possibility for detectable faulty values. Now, consider the next iteration. In this case, of the original $\frac{1}{n}$ masked outputs, $\frac{1}{n}$ will again be masked, for a total of $\frac{1}{n^2}$, and the remaining $\frac{(n-1)}{n^2}$ will have a value that is

[7] We note that the logic of probability is very similar to Wah [37], and this is the same value derived by Wah for single test input, where n is the *domain* of the function as Wah does not consider functions that have a different *domain* and *co-domain*.

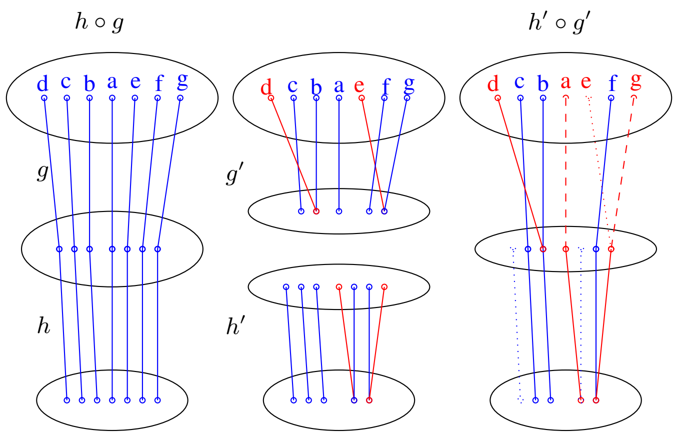
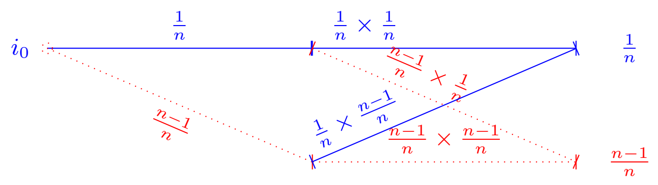


Figure 1. Weak fault interaction: The first function g is faulty at inputs d and e, while second function h is faulty at inputs a and g. Hence, we expect d, e, a, g to produce faulty outputs for the combined function $h' \circ g'$ if only weak interaction is considered. The input e has a possibility of being masked by the fault at g in the second function h.



i_0
$\frac{1}{n}$
$\frac{1}{n} \times \frac{1}{n}$
$\frac{1}{n}$
$\frac{n-1}{n} \times \frac{1}{n}$
$\frac{n-1}{n}$
$\frac{1}{n} \times \frac{n-1}{n}$
$\frac{n-1}{n} \times \frac{n-1}{n}$
$\frac{n-1}{n}$

Figure 2. Recursive interaction. The *blue solid* lines represent the masking values where the values are same as what would be expected before the fault was introduced, and the *red dotted* lines represent values that are different from the non-faulty version so that faults could be detected.

faulty. Consider the original $\frac{n-1}{n}$ that had faulty values in the first iteration. Out of that, $\frac{1}{n}$ will be masked in the second iteration (i.e. $\frac{n-1}{n^2}$). Similarly, $\frac{(n-1)^2}{n^2}$ of the original faulty outputs will remain faulty. That is, after second iteration, we will have $\frac{1}{n^2} + \frac{n-1}{n^2} = \frac{1}{n}$ masked output values. Similarly, we will have $\frac{n-1}{n^2} + \frac{(n-1)^2}{n^2} = \frac{1-n}{n}$ faulty output values. That is, after each iteration, we will have $\frac{1}{n}$ possibility of fault masking (See Figure 2). Hence, composite fault hypothesis will hold even for recursion and iteration.

A question may be asked, what happens for functions where the iteration may proceed in different paths during different executions. For example:

```
for i in 1..10:
  if odd(i): x = g(y)
  else: y = h(x)
```

In programs such as this, one may unroll the loop, i.e.

```
for i in 1..10:2:
   x = g(y)
   y = h(x)
```

which can make it amenable to the above treatment. Recursion can also be treated in a same fashion. We do not claim that this is exhaustive. There could exist other patterns of recursion or iteration

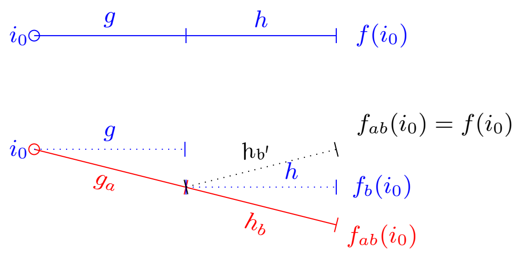


Figure 3. Fault interaction ($g_a(i_0)$ is masked by $h_{b'}$)

that do not fit this template. However, most common patterns of recursion and iteration could be captured in this pattern.

Can we extend the bounds we found ($i \cup j$ for faulty outputs) to recursion? Unfortunately, it is possible for a faulty function to interact with its own output during recursion, and hence mask a failure. Hence, we can not bound the failure causing inputs in a doubly faulty function that incorporates recursion.

3.4 Accounting for Multiple Faults

What happens when there are multiple faults? Say, we have a system modeled by $p \circ q \circ r \circ s \circ t \circ u$, where any of the functions may be faulty or not faulty, for example $p_a \circ q \circ r_b \circ s_c \circ t_d \circ u$. We can not directly apply the technique in recursion because there are non-faulty functions interspersed. The thing to remember here is that a non faulty function immediately adjacent to a faulty function can together be considered a faulty function. Hence, the above reduces to $(p_a \circ q) \circ r_b \circ s_c \circ (t_d \circ u)$, or equivalently $pq_a \circ r_b \circ s_c \circ tu_d$. This is now amenable to the treatment in Figure 2 because each function now can produce $\frac{1}{n}$ non-faulty and $\frac{n-1}{n}$ faulty outputs. An additional complication is that a general expression is not possible unless we simplify further, and assumes that *domain* and *co-domain* of all functions are same. With this simplification, even when we consider a number of faulty functions, the mean ratio of fault masking remains the same at $\frac{1}{n}$.

Indeed, we note that this is one of the significant differences from Wah's analysis. Wah does not attempt to collapse the non-faulty functions to their neighbours. Why do we do this? Because we know that each faulty function on their own was detected by the test suite. That is, we know that $p \circ q_a \circ r \circ s \circ t \circ u$ would have been detected. Hence, we can certainly consider $pq_a \circ r \circ s \circ t \circ u$ as the set of functions where the function pq_a is the faulty function with an atomic fault.

3.5 Dynamically Checked Languages

How can we apply the composite coupling hypothesis to *dynamically checked* languages, or *unityped* languages? In both cases, any single function can be taken to have the exact same *domain* and *domain*, the size of which may be taken to be extremely large (but finite due to constraints of the environment). Intuitively, this is because any function may be replaced by any other, and one may not identify a faulty input type until execution passes through that function. Hence in such languages, we can expect the composite coupling ratio to be large.

3.6 Impact of Syntax

In order to model composite coupling, we relied on a simplification — that all faults are equally probable. However, in the real world this is often not the case, with faults that are closer syntactically being more probable than faults which are not in the syntactic neighborhood of correctness. In fact, we have some reasonable estimate of the distribution of size of faults that programmers make [14].

Implementation of functions as code need not necessarily follow the same distribution as that of their mathematical counterparts. For example, for mathematical functions, there exist only 4 functions that map from a boolean to a boolean. However, there can be an infinite number of program implementations of that function. The way it can be made tractable is again to consider the human element. The *competent programmer hypothesis* suggests that faulty programs are close (syntactically) to the correct versions. So one need only consider a limited number of alternatives (the number of which is a function of the size of the correct version, if one assumes that each token may be legally replaced by another).

As soon as we speak about syntactic neighborhood, the syntax of a language can have a large influence on which faults can be considered to be in a neighborhood. However, we note that most languages seem to follow a similar distribution of faults with a size below 10 tokens for 90% of faults [14].

Let us call the original input to h, $g(i_0) = j_0$, and the changed value $g_a(i_0) = j_a$. Similarly, let $f(i_0) = k_0$, $f_a(i_0) = k_a$, $f_b(i_0) = k_b$, and $f_{ab}(i_0) = k_{ab}$. Given two inputs i_0, and i_1 for a function f, we call i_0, and i_1 *semantically close* if their execution paths in f follow equivalent profiles, e.g., taking the same branches of conditionals. We call i_0 and i_1 *semantically far* in terms of f if their execution profile is different.

Consider the possibility of masking the output of h_b by g_a ($g_{a'}$ in Figure 3). We already know that $h(j_a) = k_b$ was detected. That is, we know that j_a was sufficiently different from j_0, that it propagated through $h(j_a)$ to be caught by a test case. Say j_a was *semantically far* from j_0, and the difference in code path contained the fault $\hat{a}$. In that case, the fault $\hat{a}$ would not have been executed, and since $k_{ab} = k_b$, it will always be detected.

On the other hand, say j_a was *semantically close* to j_0 in terms of g and the fault $\hat{a}$ was executed. There are again three possibilities. The first is that $\hat{a}$ had no impact, in which case the analysis is the same as before. The second is that $\hat{a}$ caused a change in the output. It is possible that execution of $\hat{a}$ could be problematic enough to always cause an error, in which case we have $k_{ab} = k_a$, and detection. Thus masking requires k_{ab} to be equal to k_0.

Even if we assume that the function h_b is close syntactically to h, and that this implies semantic closeness of functions h and h_b, we expect the value k_{ab} to be near k_b, not k_0. This suggests that masking, even when considered in the light of syntactical neighborhood, is still unlikely, but this belief requires empirical verification since we are unable to assign probabilities to the cases above. Our empirical data (provided in the next section of this paper) should shed light on the actual incidence of masking when syntactic/semantic neighborhoods are taken into account, since real faults are likely in the syntactic and semantic neighborhood of the correct code.

A statistical observation can further buttress our argument. We know that if all functions were equally probable, fault masking has low probability. Now, consider the functions that are syntactically close to a given function. For most input values, we can assume that the syntactically close functions will have same output as that of the given function, more so than functions that are far away lexically. If h did not mask a value originally, (which we know since we were able to detect fault $h(g_a(i_0))$), then the syntactically close functions to h will with a higher probability than a uniform sample, produce the same value as $h(g_a(i_0))$, which will be detected as faulty.

3.7 Can Strong Interaction be Avoided?

Remember that the argument of mutation analysis is that if a test suite can find all atomic faults, then by composite fault hypothesis, a large percentage (κ) of complex faults will also be found by the same test suite. A problem with this is that one can in general never be sure that all atomic faults have been found, as any fault interaction has the potential to produce an atomic fault through strong interaction. Hence, it is interesting to ask, what features in a programming language contribute to strong fault interaction, and is there a representation of programs where strong fault interaction is guaranteed to be absent.

The first question is, given that the strong interaction is dependent on the execution, can runtime environment or compiler order computation so that strong interaction is no longer present?

Consider the function *swap (a,b) = (b,a)* that we examined earlier. We see how one may mistakenly use *id (a,b) = (a,b)* instead, and cause a strong interaction. Now, the question is, does there exist a way to split the two functions, so that the condition of separability can be satisfied? Given that there are only four possible functions that can operate on a tuple, (*swap (a,b) = (b,a)*, *id (a,b) = (a,b)*, *dupleft (a,b) = (a,a)*, *dupright (a,b) = (b,b)*) we could check it exhaustively. The condition is that the functions representing single faults should individually cause a detectable deviation on their own, and on composition, result in same behavior as *id*. Now, it can be seen that, neither of the single fault functions can behave like $swap$ since that represents *no* fault, so they can not behave like id, since that suggests that the other faulty function behaves like $swap$. Hence, no compiler or runtime environment can remove the strong interaction in *swap*.

The next question is, where can we expect strong interaction to appear? While we can not provide an exhaustive overview of possible language features, we can demonstrate that even very simple languages such as the λ-calculus are vulnerable to strong interaction. For example, consider the λ-calculus expression $\lambda x\, y\, .y\, x$, and its faulty version $\lambda x\, y\, .x\, y$. There are two lexical points where the faults have been injected $\{x \rightarrow y, y \rightarrow x\}$. However, they cannot be separated out. That is, even simple features such as variables can cause strong interaction.

4. Methodology for Assessment

Our methodology was guided by two principles [34]: We sought to minimize the number of variables that were in play, striving to remove the effects of those variables whose impact was unknown. Second, our aim was to be as broadly general as possible in conclusions. The programs from which we drew faults therefore had to be as realistic as possible, and the faults had to be real, and plausible enough to have happened in an environment where there was some standard of quality.

For these reasons, we selected the Java common library projects from the Apache open-source project. Apache has a reputation of having a well maintained standard of quality for their projects, and the commits that go into their projects. They are also well known for having reasonably well tested projects. By choosing projects from a single source, we minimize the variability due to coding practices, development culture, and other possible confounding factors. However, the tradeoff is that Apache common library projects may not be representative of the broader software ecosystem. Further research is necessary to ensure that our findings remain valid for the broader ecosystem.

For our set of projects, we iterated through their commit logs, and generated reverse patches for each commit. For each patch thus created, we applied the patch on the latest repository, and removed any changes to the test directory, thus ensuring that the test suite we tested with was always the latest. Any patch that resulted in a compilation error was removed. This resulted in a set of patches for each project that could be independently applied. The complete test suite for the project was executed on each of the patches left, and any patch that did not result in a test failure was removed. The failed test cases that corresponded to each patch were thus collected. At this point, we had a set of patches that introduce specific test case

	Projects	SLOC	TLOC	CPatches	Fails
1	commons-bcel	30,175	3,155	148	6
2	commons-beanutils	11,640	21,665	63	5
3	commons-cli	2,665	3,768	71	5
4	commons-codec	6,599	11,026	179	4
5	commons-collections	27,820	32,913	333	16
6	commons-compress	18,746	13,496	430	65
7	commons-configuration	26,793	37,806	322	78
8	commons-csv	1,421	3,168	150	8
9	commons-dbcp	11,259	8,487	98	18
10	commons-dbutils	3,064	3,699	43	1
11	commons-discovery	2,320	268	171	1
12	commons-exec	1,757	1,601	90	5
13	commons-fileupload	2,389	1,946	129	8
14	commons-imaging	31,152	6,525	174	4
15	commons-io	9,813	17,968	177	18
16	commons-jexl	10,921	9,509	54	10
17	commons-jxpath	18,773	6,137	10	2
18	commons-lang	25,468	43,981	571	49
19	commons-mail	2,720	3,869	48	5
20	commons-math	84,809	89,336	954	142
21	commons-net	19,749	7,465	454	21
22	commons-ognl	13,139	6,873	190	3
23	commons-pool	5,242	8,042	149	12
24	commons-scxml	9,524	5,119	74	7
25	commons-validator	6,681	7,926	126	17

Table 1. Apache Commons Libraries. SLOC is the program size in LOC, TLOC is the test suite size in LOC, CPatches is the number of compiled patches, and Fails is the number of test failures.

	count
All	1,126
Coupled	1,126
Subsuming	56
Strongly Subsuming	2
Strongly Subsuming and Coupled	2
Weakly Subsuming and Coupled	54
Weakly Subsuming and De-coupled	0
Non Subsuming and De-coupled	0
Non Subsuming and Coupled	1,070

Table 2. Higher Order Mutant Statistics

failures. The set of Apache projects, along with the set of reverse patches thus found, are given in Table 1.

We conducted our remaining analysis in two parts. For the first part, we generated patch pairs by joining together two random patches for any given project. For the projects where the total number of unique pairs were larger than 100, we randomly sampled 100 of the pairs produced. After removing patch combinations that resulted in compilation errors, we had 1,126 patch combinations. We evaluated the test suite of each project against the pair-patches thus generated, and collected the test cases which failed against these. Adopting the terminology of Jia et al. [17], out of 1,126, we had 1,126 coupled higher order mutants, and 56 subsuming mutants. Out of these, there were only 2 strongly subsuming mutants. The details are given in Table 2.

We tried to reduce the number of external variables further for the second part, and chose a single large project — *Apache commons-math*. We generated a set of combined patches by joining 2, 4, 8, 16, 32, and 64 patches at random, and evaluated the test suite for Apache commons-math against each of these combined

k^{th} order patches. We removed all patches that resulted in any compilation errors. This resulted in 342 patch combinations.

For both parts of our analysis, we generated two sets. The first set containing the unique failures from the constituent faults in isolation, and the second, containing failures from the joined patches.

5. Analysis

There are two questions that we tackle here. The first investigates the fraction of test cases that detect any of the constituent mutants that also detect the combined mutant. That is, evaluates the following prediction from the model: "Given two faults, and the test cases killing each, (assuming a sufficiently large *domain* and *co-domain*, and ignoring the effects of strong interaction), there is a high probability for the same test cases to kill the combined fault."

The second investigates the general coupling effect. Note that general coupling effect does not distinguish between strong and weak interaction. Since the general coupling ratio does not distinguish between strong and weak interaction, this also serves as an evaluation of the strong interaction between faults where inputs other than the original i and j – that is outside $i \cup j$ becomes faulty (where i represents faulty inputs to f due to faults in h, and j represents faulty inputs to f due to faults in g).

Indeed, we believe that strong interaction between different faults is rarer than weak interaction. While there is no easy way to verify it, one may look at the newer faults (new test failures) that are introduced by a combination of patches when compared to the original patches as instances of strong fault interaction (These are not the only instances, since some of the older tests could also be failing on new behaviors. Further, some of the instances where new tests are failing could also be due to too strict assertions failing only when multiple conditions are satisfied. However, it is a reasonable proxy).

Our empirical evaluation does not require individual patches to be simple faults. Rather, we are investigating the probability of fault masking between *any* two patches. Our theory suggests that irrespective of whether the faults are complex or not, we can expect the same fault masking probability.

5.1 All Projects

This section investigates fault pairs constructed from all projects.

5.1.1 The Composite Fault Model

Here, we try to answer the question: *what percentage of test cases detecting constituent faults can detect the complex faults?*

Figure 4 plots the set of test cases able to detect the faults when they were separate with the set of test cases able to detect the combined fault.

To analyze the fraction of test cases expected to detect the combined mutant, we evaluate the regression model given by:

$$\mu\{AfterT|BeforeT\} = \beta_0 + \beta_1 \times BeforeT \qquad (1)$$

where $BeforeT$ is the size of the test suite that includes all test cases that can detect both faults separately, and $AfterT$ is the size of the test suite which is a subset of $BeforeT$ that can detect the fault pair when combined. We force β_0 to zero to account for the fact that if no test cases detected the original mutant, then the question of their fraction does not arise. This linear regression model lets us predict the number of test fails for combined faults from the test fails for separated faults.

We note that we are interested in β_1 for another purpose. β_1 is also the composite coupling ratio κ. Thus this regression provides us with a model for prediction, its goodness of fit (R^2), and also the composite coupling ratio.

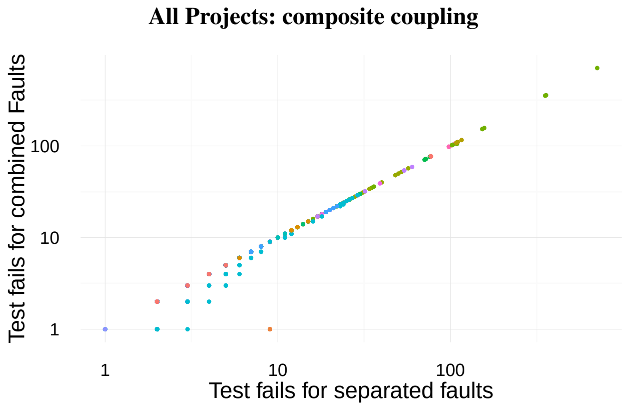


Figure 4. The size of set of the test cases able to detect the faults when they were separate is in the *x-axis*, and the *subset of the same test cases* able to detect the combined fault is in the *y-axis*. Different projects are indicated by different colors. The two projects that have a slightly below the mean relation between X and Y axis from other projects are *commons-io* and *commons-beanutils*.

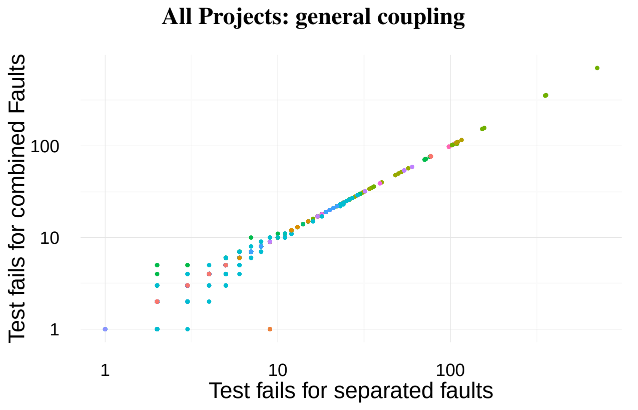


Figure 5. The size of the set of test cases able to detect the faults when they were separate is the *x-axis*, and the *set of all test cases* able to detect the combined fault is in the *y-axis*. Different projects are indicated by different colors. The three projects that have a slightly different relation between X and Y axis from other projects are *commons-io* and *commons-beanutils* (below the mean), and *commons-dbcp* (above the mean).

5.1.2 The General Coupling Model

Figure 5 plots the general coupling of faults. To answer this question, we evaluate the following regression model.

$$\mu\{NewT|BeforeT\} = \beta_0 + \beta_1 \times BeforeT \qquad (2)$$

where $BeforeT$ is the size of the test suite that includes all test cases that can detect both faults separately, and $NewT$ is the size of the test suite that can detect the fault pair when combined. Note that we do not set $\beta_0 = 0$ here as the combined fault pair may be detected by a new test case even if its constituents were not detected. In fact, β_0 represents the complex faults that became detectable due to interaction even though the constituent faults are not detectable.

However, if one wishes to investigate the general coupling ratio, we have to investigate a simpler regression model, because the

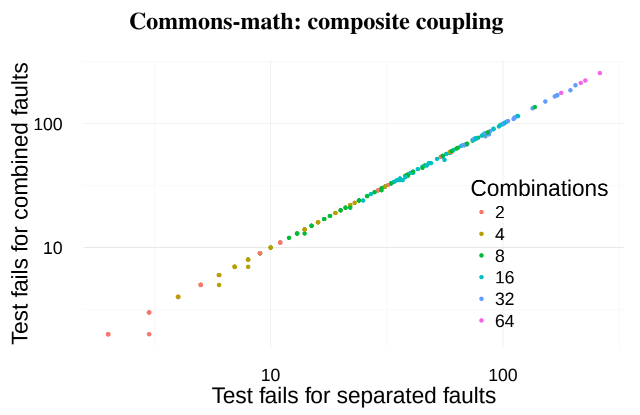


Figure 6. The set of test cases able to detect the faults when they were separate is in the *x-axis*, and the *subset of the same test cases* able to detect the combined fault is in the *y-axis*. The different number of patch combinations used are indicated by different colors.

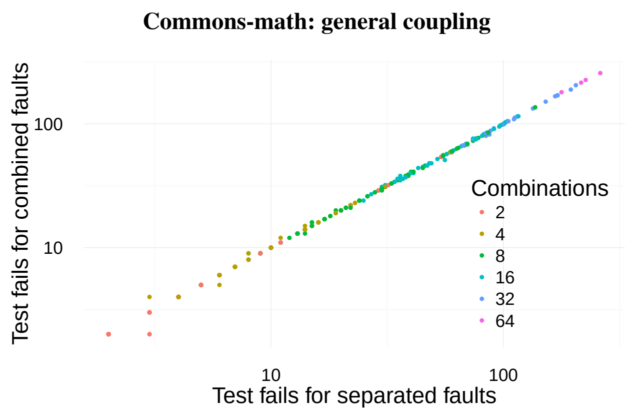


Figure 7. The set of test cases able to detect the faults when they were separate is in the *x-axis*, and the *set of all test cases* able to detect the combined fault is in the *y-axis*. The different number of patch combinations used are indicated by different colors.

general coupling ratio does not permit an intercept.

$$\mu\{NewT|BeforeT\} = \beta_1 \times BeforeT \tag{3}$$

Here, similar to the previous section, β_1 corresponds to the general coupling ratio C.

5.1.3 Strong fault interaction

The incidence of strong fault interaction may be ascertained by the average number of new test cases that failed for the combined patch. Note that this number is not exhaustive, as some of the original test cases may fail for new faulty behavior too, even if the behavior is not same as that of the component faults.

5.2 Apache Commons-math

5.2.1 The Composite Fault Model

We try to answer the question *what percentage of test cases detecting constituent faults can detect the complex faults?* for Apache Commons-math.

Figure 6 plots the set of test cases able to detect the faults when they were separate with the set of test cases able to detect the combined fault. We rely on the regression given by Equation 1 to answer this question.

5.2.2 The General Coupling Model

Figure 7 plots the general coupling of faults for *Apache commons math*. We rely on the regressions given by Equation 2 and Equation 3 to answer this question.

5.2.3 Strong fault interaction

The incidence of strong fault interaction may be ascertained by the average number of new test cases that failed for the combined patch. The difference of note here is that the number of patches are larger, and hence the chances of strong interaction are correspondingly larger.

6. Results

6.1 All Projects

The results for regression for Equation 1 for all projects is given in Table 3. The correlation between the dependent and independent variable is 0.99975. The composite coupling ratio was found to be 0.99916. The results for regression for Equation 2 for all projects is given in Table 4. The correlation between the dependent and independent variable is 0.99967. The results for regression for Equation 3 for all projects is given in Table 5. The general coupling ratio was found to be 0.99931. Further, the mean number of faulty test cases that were not present in the component faults were found to be 0.0417. See Table 6 for the summary.

	Estimate	Std. Error	t value	Pr(>|t|)
SeparateFaults	0.9992	0.0005	2,116.13	0.0000

Table 3. All projects regression for composite coupling ratio. R^2 =0.99975

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-0.0399	0.0189	-2.12	0.0343
SeparateFaults	0.9997	0.0005	1,847.83	0.0000

Table 4. All projects regression. R^2 =0.99967

	Estimate	Std. Error	t value	Pr(>|t|)
SeparateFaults	0.9993	0.0005	1,939.82	0.0000

Table 5. All projects regression for general coupling ratio. R^2 =0.9997

6.2 Apache commons-math

The results for regression for Equation 1 for all projects is given in Table 7. The correlation between the dependent and independent variable is 0.99983. The composite coupling ratio was found to be 0.98956. The results for regression for Equation 2 for commons-math is given in Table 8. The correlation between the dependent and independent variable is 0.99971. The results for regression for Equation 3 for commons-math is given in Table 9. The general coupling ratio was found to be 0.9944. Further, the mean number of faulty test cases that were not present in the component faults were found to be 0.137. See Table 10 for the summary.

7. Discussion

Fault masking is one of the key concerns in software testing. The *coupling effect* hypothesis asserts that fault masking is rare, and describes the general impact of interactions between faults. Unfortunately, very little is known about the theory behind fault coupling.

	SeparateFaults	JoinedFaults	RemovedFaults	AddedFaults
bcel	27.73	27.73	0.00	0.00
beanutils	4.80	1.60	3.20	0.00
cli	7.60	7.60	0.00	0.00
codec	2.50	2.50	0.00	0.00
collections	16.49	16.49	0.00	0.00
compress	11.60	11.60	0.00	0.00
configuration	37.19	37.16	0.04	0.02
csv	2.00	2.00	0.00	0.00
dbcp	10.60	10.91	0.01	0.32
exec	16.50	16.50	0.00	0.00
fileupload	4.64	4.64	0.00	0.00
imaging	6.50	6.50	0.00	0.00
io	7.93	7.55	0.54	0.17
jexl	3.58	3.56	0.02	0.00
jxpath	3.00	3.00	0.00	0.00
lang	4.46	4.46	0.00	0.00
mail	2.30	2.30	0.00	0.00
math	7.67	7.64	0.03	0.00
net	4.13	4.13	0.00	0.00
ognl	27.00	27.00	0.00	0.00
pool	4.48	4.45	0.05	0.02
scxml	36.62	36.62	0.00	0.00
validator	3.07	3.06	0.01	0.00

Table 6. Summary for all projects. The SeparateFaults column shows the number of faults when the patches were separately analyzed, and JoinedFaults column shows the number of faults when the patches were combined and analyzed. RemovedFaults indicate the number of faults from SeparateFaults that were removed in JoinedFaults, and AddedFaults indicate the number of newly added faults that were not in SeparateFaults.

	Estimate	Std. Error	t value	Pr(>|t|)
SeparateFaults	0.9896	0.0007	1,418.94	0.0000

Table 7. Commons-math regression for composite coupling ratio. R^2 =0.99983

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	0.0924	0.0482	1.92	0.0563
SeparateFaults	0.9933	0.0009	1,090.92	0.0000

Table 8. Commons-math regression. R^2 =0.99971

	Estimate	Std. Error	t value	Pr(>|t|)
SeparateFaults	0.9944	0.0007	1,401.55	0.0000

Table 9. Commons-math regression for general coupling ratio. R^2 =0.99983

	SeparateFaults	JoinedFaults	RemovedFaults	AddedFaults
2	7.67	7.64	0.03	0.00
4	14.67	14.70	0.05	0.07
8	30.53	30.42	0.17	0.06
16	59.25	59.09	0.42	0.25
32	109.85	108.96	1.37	0.48
64	220.25	219.50	3.00	2.25

Table 10. Summary for all Commons-math. The rownames indicate the number of patches involved. The SeparateFaults column shows the number of faults when the patches were separately analyzed, and JoinedFaults column shows the number of faults when the patches were combined and analyzed. RemovedFaults indicate the number of faults from SeparateFaults that were removed in JoinedFaults, and AddedFaults indicate the number of newly added faults that were not in SeparateFaults.

We study the *coupling effect* and fault masking using a theoretical model, and evaluate our findings empirically.

Since the formal statement of *coupling effect* is vague and ambiguous, we provide a more concrete definition (called the *composite fault hypothesis* to avoid confusion) — *Tests detecting a fault in isolation will (with high probability κ) continue to detect the fault even when it occurs in combination with other faults.*

We theoretically evaluate the *composite fault hypothesis*. Our analysis shows that for any pair of separable faults, the composite coupling effect exists. We find that composite coupling ratio $\kappa = 1 - \frac{1}{n}$, where n is the *co-domain* of the function being considered. We also show that the consideration of the syntactical neighborhood does not have an adverse impact on our result. Further, according to Wah, as system size increases, and number of functions in the execution path nears the domain of function, the coupling effect weakens exponentially. However, our results suggest that the mean coupling ratio remains the same at $\frac{1}{n}$.

Why is our prediction on fault masking so important? The reason is that basic testing relies on fault masking to a large extent. That is, say you are unit testing a function with multiple faults, and some of the faults are left undetected due to fault masking. Wah's analysis suggests that when we integrate the units to a larger system, the faults in larger system has a much larger (indeed exponential) tendency to self correct, and avoid failure due to masking. Our analysis suggests that even on larger systems composed of smaller systems, the rate of fault masking remains the same.

We next empirically validated the *composite fault hypothesis*. Our subjects were numerous and this allows us to make a strong general claim about the composite coupling ratio κ, and also about the traditional coupling ratio C.

We also proposed the existence of strongly interacting faults, which cannot be accounted for within the formal coupling theory. Our empirical analysis (see Table 6 and Table 10) indicates that strong interaction is possibly rare, and is of similar frequency as that of fault masking.

Examining Figure 4, we note that while there is some reduction in the combined faults for the faults with smaller semantic footprint (as given by the number of test cases that failed for that fault) with respect to the constituent faults, the difference vanishes when the size of the fault increases. This same effect is also seen in Figure 6.

The results for regression Equation 1 from all projects, and Apache commons-math also suggests a similar observation — that the test cases that are able to detect a fault in isolation will with very high probability detect the same fault when in combination with other faults.

A counter-intuitive result is suggested by the regression (Equation 3) comparing the semantic sizes of constituent faults and combined faults. Table 5 and Table 9 suggests that the coefficient for $SeparateFaults$ is less than unity, which means that when separate, the faults have a slightly larger semantic footprint than when combined — possibly due to fault interactions. However, we note that the effect is not statistically significant for all projects. The coefficient for separate faults for Table 5 lies between {0.99831.0003}, and the coefficient for separate faults for Table 9 lies between {0.993000.99579} — 95% confidence interval, $p < 0.0001$.

Overall, our statistical analysis suggests that there is a very high probability (between {0.998241.00009} for all projects, and {0.988180.99093} for math — 95% confidence interval with statistical significance $p < 0.0001$) that when two faults are paired to produce a combined fault, any test cases that detected either of the faults continue to detect the combined fault.

Our results for Table 5 suggests that between {0.99831.0003} of complex faults are caught (95% confidence interval, $p < 0.0001$). This is again confirmed by the deeper analysis of *Apache commons-*

math, using larger size faults in Table 9 which suggests that between $\{0.993000.99579\}$ fraction of complex faults are caught (95% confidence interval, $p < 0.0001$). We note that this is the first confirmation of the *general coupling effect* (unlike the *mutation coupling effect* which has been validated multiple times). Why is validating general coupling effect important? We already know that faults emulated by traditional mutants are only a subset of possible kinds of faults (Just et al. [19] found that up to 27% of faults were inadequately represented by mutants). Hence, it is important to verify *general coupling effect* using real faults so that our results are applicable for faults in general, and especially for possible future mutation operators. Indeed, the *mutation coupling effect* has been validated multiple times, and we do not attempt it again here.

How much difference is there between the traditional general coupling ratio C and the composite coupling ratio κ? Our analysis finds that in general, there is less than a percentage drop for the composite coupling ratio from the traditional coupling ratio.

There is another point that is of interest in our study. Research in software reliability, and software testing often relies on curated sets of faults [19, 11] that took manual effort to produce. The paucity of such fault databases has held back research in software testing to some extent. This also means that there is a high chance of skew either due to manual bias, or due to skew in the projects thus selected. Our technique is entirely automated. When a representative set of code repositories are available, our methodology can be used to produce faults with relatively less skew from external variables.

8. Threats to Validity

While we have taken every care to ensure that our results are unbiased, and have tried to eliminate the effects of random noise, our results are subject to the following threats to validity.

8.1 Threats to Theoretical Analysis

Our theoretical analysis relies on a main assumption, that all the faults of a function have an equal probability of being chosen. We discussed the effect of syntax on our model in Section 3, where we noted that syntax should not have a large impact so long as its impact is similar across functions with expected output, and functions which do not have the expected output. While there does not seem to be any reason to suspect otherwise, and the results of empirical analysis seem to bear out our theoretical result, there exists a possibility that this assumption is unwarranted in general, or may depend on the particular program being considered.

A second assumption that we made in the theoretical analysis is that the *co-domain* of function being considered is large. To indicate the impact of this assumption, for a function with binary *co-domain* (only true, and false are allowed as output), there is 50% chance of masking, and hence only 50% chance of coupling. However, the situation changes proportionally with larger *co-domain*, with faults in a function with a possibility of output of even 100 unique values have 99% chance of coupling. Hence, we believe that the assumption of large *co-domain* is warranted on average.

Our theoretical analysis is incomplete for cases of more complex execution paths where there may be cases where the solved templates for recursion and iteration may not apply. This may also have an impact on the actual fault masking for specific programs that do not fit into these templates.

Finally, we assumed that programs corresponded to mathematical functions when taking that there were n^m alternatives to a program $h \in: M \to N$. However, in real world, syntax plays a much larger role, and there are an infinite number of equivalent programs with exactly the same *domain* and *co-domain*.

We note that our empirical analysis does not verify the major difference that we suggested from Wah's analysis — that coupling effect does not weaken with system size. This is not taken up here due to the significant investment in finding and testing systems that are large enough. This is an avenue for future work.

8.2 Threats to Empirical Analysis

Threats due to sampling bias: To ensure that our results were representative of real world programs, with non-trivial faults, we opted for Java projects from the Apache commons library. We used all projects that we could build and complete testing. Further, we used all patches that could be applied in isolation and resulted in a valid build, with at least one test failure, which indicated that the patch fixed something in the project. This however, implies that our sample of programs could be biased by any factor that skews the kind of projects that the Apache community works on, or their development practices. The representativeness of our sample patches are also impacted by any factor that may skew the independent patches that we extracted.

Projects of differing maturity, and quality: Since we used real world projects from the Apache commons library, the maturity and quality of these projects is representative of the real world. However, not all projects have been in development for a long time, and hence have a different distribution of faults from what is typically expected in an industrial setting with a high standard of quality. To counter this threat we evaluated Apache commons-math — the largest, most mature, and highest quality project which had 90% statement coverage and 73.2% mutation score for our analysis for the second part. The results from Apache commons-math seem to indicate that our experiment is relatively unbiased in this direction.

9. Conclusion

The *coupling effect* hypothesis is a general theory of fault interaction, and is used to quantify *fault masking*. It also finds use in mutation analysis. While there is interesting empirical evidence for the *coupling effect*, theoretical understanding is lacking. The extant theory by Wah is too restrictive to be useful for real world systems. We correct this limitation, and provide a stronger, modified version of the theory called the *composite fault hypothesis*.

We provide a theory of fault coupling applicable to general functions (with some restrictions). It incorporates the syntactical neighborhood and clarify assumptions made. We show that even under real world conditions, the *composite fault hypothesis* holds.

Our theoretical analysis suggests that the composite fault hypothesis has a high probability of occurring ($1 - \frac{1}{n}$, where n is the *co-domain* of the function under consideration) under the assumptions of total functions, finite *domain*, and separability of faults, *irrespective of the size of the system*.

Our empirical study provides validation, and an empirical approximation of the composite coupling ratio κ (0.99), with 99% of the test cases that detected a fault in isolation continuing to detect it when it is combined with other faults.

Finally, the *general coupling effect* has never been demonstrated for real faults. Our empirical analysis also provides firm evidence in favor of the *general coupling effect* ($C = 99\%$) for real faults.

References

[1] Al-Khanjari, Z. A., and Woodward, M. R. Investigating the partial relationships between testability and the dynamic range-to-domain ratio. *Australasian Journal of Information Systems 11* (2003).

[2] Andrews, J. H., Briand, L. C., and Labiche, Y. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering* (2005), IEEE, pp. 402–411.

[3] Andrews, J. H., Briand, L. C., Labiche, Y., and Namin, A. S. Using mutation analysis for assessing and comparing testing

coverage criteria. *IEEE Transactions on Software Engineering 32*, 8 (2006), 608–624.

[4] ANDROUTSOPOULOS, K., CLARK, D., DAN, H., HIERONS, R. M., AND HARMAN, M. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In *International Conference on Software Engineering* (New York, NY, USA, 2014), ACM, pp. 573–583.

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

[6] BUDD, T. A., DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1980), ACM, pp. 220–233.

[7] BUDD, T. A., LIPTON, R. J., DEMILLO, R. A., AND SAYWARD, F. G. *Mutation analysis*. Yale University, Department of Computer Science, 1979.

[8] CLARK, D., AND HIERONS, R. M. Squeeziness: An information theoretic measure for avoiding fault masking. *Information Processing Letters 112*, 8 (2012), 335–340.

[9] DARAN, M., AND THÉVENOD-FOSSE, P. Software error analysis: A real case study involving real faults and mutations. In *ACM SIGSOFT International Symposium on Software Testing and Analysis* (1996), ACM, pp. 158–171.

[10] DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. *Computer 11*, 4 (1978), 34–41.

[11] DEMILLO, R. A., AND MATHUR, A. P. On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software. Tech. Rep. SERC-TR92-P, Software Engineering Research Center, Purdue University, West Lafayette, IN,, 1991.

[12] DIAS, F. J. O. Fault masking in combinational logic circuits. *IEEE Trans. Comput. 24*, 5 (May 1975), 476–482.

[13] DO, H., AND ROTHERMEL, G. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering 32*, 9 (2006), 733–752.

[14] GOPINATH, R., JENSEN, C., AND GROCE, A. Mutations: How close are they to real faults? In *International Symposium on Software Reliability Engineering* (Nov 2014), pp. 189–200.

[15] HARMAN, M., JIA, Y., AND LANGDON, W. B. A manifesto for higher order mutation testing. In *International Conference on Software Testing, Verification and Validation Workshops* (2010), IEEE, pp. 80–89.

[16] JIA, Y., AND HARMAN, M. Constructing subtle faults using higher order mutation testing. In *IEEE International Working Conference on Source Code Analysis and Manipulation* (2008), IEEE, pp. 249–258.

[17] JIA, Y., AND HARMAN, M. Higher order mutation testing. *Information and Software Technology 51*, 10 (Oct. 2009), 1379–1393.

[18] JIA, Y., AND HARMAN, M. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering 37*, 5 (2011), 649–678.

[19] JUST, R., JALALI, D., INOZEMTSEVA, L., ERNST, M. D., HOLMES, R., AND FRASER, G. Are mutants a valid substitute for real faults in software testing? In *ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Hong Kong, China, 2014), ACM, pp. 654–665.

[20] KAPOOR, K. Formal analysis of coupling hypothesis for logical faults. *Innovations in Systems and Software Engineering 2*, 2 (2006), 80–87.

[21] LANGDON, W. B., HARMAN, M., AND JIA, Y. Efficient multi-objective higher order mutation testing with genetic programming. *The Journal of Systems and Software 83*, 12 (2010), 2416–2430.

[22] LI, N., PRAPHAMONTRIPONG, U., AND OFFUTT, J. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *International Conference on Software Testing, Verification and Validation Workshops* (2009), IEEE, pp. 220–229.

[23] LIPTON, R. J. Fault diagnosis of computer programs. Tech. rep., Carnegie Mellon Univ., 1971.

[24] LIPTON, R. J., AND SAYWARD, F. G. The status of research on program mutation. In *Digest for the Workshop on Software Testing and Test Documentation* (December 1978), pp. 355–373.

[25] MATHUR, A. P., AND WONG, W. E. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability 4*, 1 (1994), 9–31.

[26] MORELL, L. J. *A Theory of Error-based Testing*. PhD thesis, University of Maryland at College Park, College Park, MD, USA, 1983.

[27] MORELL, L. J. A model for code-based testing schemes. In *Fifth Annual Pacific Northwest Software Quality Conf* (1987), p. 309.

[28] MORELL, L. J. A theory of fault-based testing. *IEEE Transactions on Software Engineering 16*, 8 (1990), 844–857.

[29] NGUYEN, Q. V., AND MADEYSKI, L. Problems of mutation testing and higher order mutation testing. In *Advanced Computational Methods for Knowledge Engineering*. Springer, 2014, pp. 157–172.

[30] OFFUTT, A. J. The Coupling Effect : Fact or Fiction? *ACM SIGSOFT Software Engineering Notes 14*, 8 (Nov. 1989), 131–140.

[31] OFFUTT, A. J. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology 1*, 1 (1992), 5–20.

[32] OFFUTT, A. J., AND UNTCH, R. H. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*. Springer, 2001, pp. 34–44.

[33] OFFUTT, A. J., AND VOAS, J. M. Subsumption of condition coverage techniques by mutation testing. Tech. rep., Technical Report ISSE-TR-96-01, Information and Software Systems Engineering, George Mason University, 1996.

[34] SIEGMUND, J., SIEGMUND, N., AND APEL, S. Views on internal and external validity in empirical software engineering. In *International Conference on Software Engineering* (2015).

[35] VOAS, J. M., AND MILLER, K. W. Semantic metrics for software testability. *The Journal of Systems and Software 20*, 3 (1993), 207 – 216.

[36] WAH, K. S. H. T. Fault coupling in finite bijective functions. *Software Testing, Verification and Reliability 5*, 1 (1995), 3–47.

[37] WAH, K. S. H. T. A theoretical study of fault coupling. *Software Testing, Verification and Reliability 10*, 1 (2000), 3–45.

[38] WAH, K. S. H. T. Theoretical insights into the coupling effect. In *Mutation testing for the new century*, W. E. Wong, Ed. Kluwer Academic Publishers, Norwell, MA, USA, 2001, pp. 62–70.

[39] WAH, K. S. H. T. An analysis of the coupling effect I: single test data. *Science of Computer Programming 48*, 2 (2003), 119–161.

[40] WOODWARD, M. R., AND AL-KHANJARI, Z. A. Testability, fault size and the domain-to-range ratio: An eternal triangle. *ACM SIGSOFT Software Engineering Notes 25*, 5 (2000), 168–172.